

Pratik Derin Öğrenme — fast.ai (Jeremy Howard)'dan

ML Builder için Türkçe Notlar

Phase 2

2026-06-07

İçindekiler

1	Önsöz	1
2	Bu kitap nedir?	3
3	Nasıl Okumalı	5
4	27 Ders	7
5	Notasyon	9
6	Builder Eksen — fast.ai (top-down) ↔ Karpathy (bottom-up)	11
7	Yazım Kuralları	13
8	Başlangıç — İlk Modelini Eğit (Is it a bird?)	15
8.1	Bu Derste Ne Var?	15
8.2	xkcd 2015: İmkânsız Şakadan Trivial’e	16
8.3	“Önce Çalıştır” — Kuşu İki Dakikada Eğitelim	16
8.4	Görüntüler Sayılardan Oluşur	17
8.5	Sonuç: Bu Bir Kuş	18
8.6	Derin Öğrenme Bugün Ne Yapabiliyor?	19
8.7	Kurs Nasıl İşlenecek — Top-Down Öğrenme	19
8.8	Neden Artık Yapabiliyoruz? 2012 vs Sinir Ağları	20
8.9	Bir Sinir Ağı Ne Öğrenir?	21
8.10	Görü Dışında: Ses, Zaman Serisi, Tablo	22
8.11	Araçlar: PyTorch + fastai	23
8.12	Jupyter ve Bulut	24
8.13	DataBlock — Veriyi Modele Sokmak	24
8.14	DataLoaders, Batch ve Validation Set	25
8.15	Learner + Pretrained Model	25
8.16	predict ve Deploy	28
8.17	Diğer Uygulamalar ve Kapanış	28
8.18	Bu Dersin Özeti	29
8.19	Kontrol Soruları	29
8.20	Egzersizler	30
8.21	Sonraki Ders İçin Hazırlık	31
8.22	Anahtar Kavramlar (Cheat Sheet)	31
8.23	ML Builder Bağlantıları	32
9	Dağıtım — Modeli Dünyaya Aç (Deployment)	33
9.1	Bu Derste Ne Var?	33

9.2	Geri Dönüş: Öğrenciler Neler Yaptı?	34
9.3	Hedef: Modeli Production'a Almak	34
9.4	Veriyi Toplamak	35
9.5	Ters Sezgi: Önce Eğit, Sonra Temizle	35
9.6	Yeniden Boyutlandırma Yöntemleri	36
9.7	Veri Artırımı (Data Augmentation)	37
9.8	Confusion Matrix (Karışıklık Matrisi)	38
9.9	plot_top_losses	40
9.10	ImageClassifierCleaner	41
9.11	CPU RAM vs GPU RAM (VRAM)	41
9.12	Production'a Alma: Hugging Face Spaces + Gradio	41
9.13	learn.export — Modeli Dondurmak	42
9.14	load_learner + predict	42
9.15	Gradio Arayüzü (app.py)	44
9.16	#!export ile Notebook'tan Script	45
9.17	Kaç Epoch Eğitmeli? + Kapanış	45
9.18	Bu Dersin Özeti	46
9.19	Kontrol Soruları	46
9.20	Egzersizler	47
9.21	Sonraki Ders İçin Hazırlık	47
9.22	Anahtar Kavramlar (Cheat Sheet)	48
9.23	ML Builder Bağlantıları	49
10	Sinir Ağı Temelleri — Derin Öğrenmeyi Türetmek (Neural net foundations)	51
10.1	Bu Derste Ne Var?	51
10.2	Bir fast.ai Dersi Nasıl Çalışılır?	53
10.3	Daha İyi Bir Pet Detektörü	53
10.4	timm ile Model Karşılaştırması	54
10.5	Bulut ve JupyterLab	54
10.6	model.pkl'in İçinde Ne Var?	54
10.7	Bir Model = Parametrelili Fonksiyon	55
10.8	Fonksiyonu Elle Uydurmak	55
10.9	Loss: Modelin İyiliğini Ölçmek	57
10.10	Türev ve Gradient	57
10.11	Gradient Descent	58
10.12	ReLU: Rectified Linear Unit	59
10.13	Sonsuz Karmaşık Fonksiyon	60
10.14	“Deep Learning”i Türettik	61
10.15	Learning Rate	62
10.16	Pratik Kararlar: Yeterli Veri ve Model Seçimi	63
10.17	Kapanış	64
10.18	Bu Dersin Özeti	64
10.19	Kontrol Soruları	65
10.20	Egzersizler	65
10.21	Sonraki Ders İçin Hazırlık	66
10.22	Anahtar Kavramlar (Cheat Sheet)	66
10.23	ML Builder Bağlantıları	67

11 Doğal Dil İşleme — Metin, Tokenization ve Validation Disiplini (Natural Language Processing)	69
11.1 Bu Derste Ne Var?	69
11.2 Neden NLP için Farklı Kütüphane?	70
11.3 Fine-tuning ve Transfer Learning	71
11.4 Problem: US Patent Phrase Matching	71
11.5 Veriyi Hazırlama	72
11.6 Tokenization	72
11.7 Vocabulary ve Hugging Face Model Hub	73
11.8 Numericalization	74
11.9 ULMFiT	74
11.10 Overfitting ve Underfitting	75
11.11 Validation Set Neden Zorunlu?	77
11.12 İyi Bir Validation Set Kurmak	77
11.13 Test Set	78
11.14 Metric vs Loss	80
11.15 Hugging Face Trainer ile Eğitim	81
11.16 Kaggle İş Akışı ve Pratik Özet	81
11.17 Kapanış	82
11.18 Bu Dersin Özeti	82
11.19 Kontrol Soruları	83
11.20 Egzersizler	83
11.21 Sonraki Ders İçin Hazırlık	84
11.22 Anahtar Kavramlar (Cheat Sheet)	84
11.23 ML Builder Bağlantıları	85
12 Sıfırdan Model — Titanic'te Lineer Modelden Derin Ağa (From-scratch model)	87
12.1 Bu Derste Ne Var?	87
12.2 Sıfırdan Tabular Model	88
12.3 Titanic Verisi ve Eksik Değerler	88
12.4 Kategorik Veri: Dummy Variable	89
12.5 Ölçekleri Eşitlemek (Normalizasyon)	90
12.6 Lineer Modeli Kurmak	90
12.7 Broadcasting	91
12.8 Gradient Descent Adımı	91
12.9 Eğitim Döngüsü	91
12.10 Doğruluğu Ölçmek	93
12.11 Sigmoid	93
12.12 Kaggle'a Gönderme	95
12.13 Matris Çarpımı (@)	95
12.14 Bir Sinir Ağı	95
12.15 Derin Ağa Genişletme	98
12.16 "Sıfırdan Derin Öğrenme" ve Framework	98
12.17 Kapanış	99
12.18 Bu Dersin Özeti	100
12.19 Kontrol Soruları	101
12.20 Egzersizler	101
12.21 Sonraki Ders İçin Hazırlık	102
12.22 Anahtar Kavramlar (Cheat Sheet)	102

12.23ML Builder Bağlantıları	103
13 Rastgele Ormanlar — Karar Ağacından Random Forest’a (Random forests)	105
13.1 Bu Derste Ne Var?	105
13.2 Karar Ağaçları: Derin Öğrenme Olmayan ML	107
13.3 İlk Bölme: Bir Kuralla Başlamak	108
13.4 Bölme Skoru	108
13.5 Karar Ağacı Kurma	109
13.6 Gini Saflık Ölçüsü	109
13.7 Az Ön İşleme: Karar Ağacının Avantajı	111
13.8 Bagging	111
13.9 Random Forest Fikri	111
13.10Random Forest Kurma	113
13.11Feature Importance	114
13.12Kaç Ağaç?	116
13.13OOB: Out-of-Bag Hatası	117
13.14Model Yorumlama	117
13.15Random Forest Overfit Eder mi?	117
13.16Gradient Boosting	118
13.17Kapanış	118
13.18Bu Dersin Özeti	120
13.19Kontrol Soruları	120
13.20Egzersizler	121
13.21Sonraki Ders İçin Hazırlık	121
13.22Anahtar Kavramlar (Cheat Sheet)	122
13.23ML Builder Bağlantıları	123
14 Ortak Filtreleme — Embedding, Latent Faktör ve Dot Product (Collaborative filtering)	125
14.1 Bu Derste Ne Var?	125
14.2 Daha Büyük Modeller ve GPU Belleği	126
14.3 Gradient Accumulation	126
14.4 Ensembling	128
14.5 Multi-Target Modeller	129
14.6 Softmax	129
14.7 Cross-Entropy Loss	130
14.8 Collaborative Filtering: Öneri Problemi	132
14.9 Latent Faktörler	133
14.10Dot Product	133
14.11Embedding = Arama Tablosu	135
14.12Sıfırdan Dot Product Modeli	136
14.13Bias Terimleri	136
14.14Weight Decay (Regularization)	137
14.15collab_learner ve Embedding Boyutu	137
14.16Derin Öğrenme ile Collab ve Latent Uzay	138
14.17Kapanış	138
14.18Bu Dersin Özeti	140
14.19Kontrol Soruları	140
14.20Egzersizler	141

14.21	Sonraki Ders İçin Hazırlık	142
14.22	Anahtar Kavramlar (Cheat Sheet)	142
14.23	ML Builder Bağlantıları	143
15	Evrişimler / CNN — Convolution ve Embedding'in Her Yerdeliği (Convolutions)	145
15.1	Bu Derste Ne Var?	145
15.2	Part 1 Finali ve Embedding'i Sıfırdan Kurma	147
15.3	Embedding Yorumlama	147
15.4	fastai'de Collab: EmbeddingDotBias	148
15.5	Embedding Mesafesi	148
15.6	Derin Öğrenme ile Collaborative Filtering	148
15.7	NLP için Embedding	149
15.8	Tablo İçin Embedding (Entity Embeddings)	149
15.9	Embedding Her Yerde	150
15.10	Convolution: Kenar Bulma	150
15.11	apply_kernel: Filtreyi Kaydırma	151
15.12	F.conv2d: Verimli Convolution	153
15.13	Neden CNN? Parametre Paylaşımı	153
15.14	simple_cnn Mimarisi	153
15.15	Pooling	155
15.16	BatchNorm	157
15.17	Serinin Kapanışı: Part 1 → Part 2	158
15.18	Kapanış	158
15.19	Bu Dersin Özeti	159
15.20	Kontrol Soruları	159
15.21	Egzersizler	160
15.22	Sonraki: PART 2 İçin Hazırlık	160
15.23	Anahtar Kavramlar (Cheat Sheet)	161
15.24	ML Builder Bağlantıları	162
16	Ders 9 — Stable Diffusion (Part 2 Başlangıcı)	163
16.1	Bu Derste Ne Var?	163
16.2	Stable Diffusion'la Oynayalım	164
16.3	Kaynaklar ve Compute	165
16.4	StableDiffusionPipeline	165
16.5	num_inference_steps	166
16.6	guidance_scale (Classifier-Free Guidance)	167
16.7	negative_prompt	167
16.8	img2img: Görüntüden Görüntü	168
16.9	Textual Inversion ve DreamBooth	168
16.10	Stable Diffusion Nasıl Çalışır: Sihirli API	169
16.11	Pixel Gradyanı → Gürültü Giderme	169
16.12	U-Net: Gürültü Tahmincisi	171
16.13	Autoencoder / VAE: Latent Sıkıştırma	171
16.14	CLIP: Metin Koşullama	172
16.15	Latent + Scheduler + Noise Schedule	173
16.16	Düşük Seviye Pipeline	174
16.17	Kapanış	175

16.18Bu Dersin Özeti	176
16.19Kontrol Soruları	176
16.20Egzersizler	177
16.21Sonraki: Ders 9A İçin Hazırlık	178
16.22Anahtar Kavramlar (Cheat Sheet)	178
16.23ML Bağlantıları Özeti	179
17 Ders 9A — Stable Diffusion Derinlemesine (Whitaker)	181
17.1 Bu Derste Ne Var?	181
17.2 Sampling Loop’u Replike Etmek	182
17.3 Auto-Encoder (VAE) ve Latent Uzay	183
17.4 Latent’i Kodlama ve Çözme	183
17.5 Gürültü Ekleme ve Noise Schedule	184
17.6 img2img: Gürültüleyip Yeniden Çizme	185
17.7 Text Encoding: Tokenizer → CLIP	186
17.8 Token Embeddings ile Oynama	187
17.9 Textual Inversion	187
17.10U-Net ve Classifier-Free Guidance	188
17.11Sampling Döngüsü	188
17.12Özel Guidance: Kendi Guidance Fonksiyonun	190
17.13Kapanış	191
17.14Bu Dersin Özeti	192
17.15Kontrol Soruları	193
17.16Egzersizler	194
17.17Sonraki: Ders 9B İçin Hazırlık	194
17.18Anahtar Kavramlar (Cheat Sheet)	195
17.19ML Bağlantıları Özeti	195
18 Ders 9B — Diffusion’ın Matematiği (Waseem + Tanishq)	197
18.1 Bu Derste Ne Var?	197
18.2 Notasyondan Korkmamak	198
18.3 $q()$ Bir Olasılık Yoğunluk Fonksiyonudur	198
18.4 Data Distribution: $q(x^0)$	200
18.5 Koşullu Yoğunluk: $q(x_t x_{t-1})$	200
18.6 Normal (Gaussian) Dağılım	200
18.7 Isotropic Kovaryans: $\beta_t \cdot I$	201
18.8 Forward Process Formülü	202
18.9 Markov Process with Gaussian Transitions	203
18.10Reparameterization: Gaussian’dan Örnekleme	204
18.11Üretken Problem: Forward Kolay, Reverse Zor	205
18.12Lesson 9’un “Magic API”si ile Bağlantı	206
18.13Kapanış	207
18.14Bu Dersin Özeti	207
18.15Kontrol Soruları	208
18.16Egzersizler	209
18.17Sonraki: Ders 10 İçin Hazırlık	209
18.18Anahtar Kavramlar (Cheat Sheet)	210
18.19ML Bağlantıları Özeti	211

19 Ders 10 — Daha Derine (Diving Deeper)	213
19.1 Bu Derste Ne Var?	213
19.2 1. Ders 9 Özeti ve Öğrenci İşleri	214
19.3 2. Stable Diffusion'ı Hızlandırma: Distillation	215
19.4 3. Progressive Distillation: Adımları Yarıya İndirme	216
19.5 4. CFG ve Empty Prompt (Recap)	217
19.6 5. Tam SD Pipeline Kod Yürüyüşü	217
19.7 6. Rastgele Gürültüyü Ölçekleme	218
19.8 7. Sampling Loop ve VAE Decode	218
19.9 8. « From the Foundations » Felsefesi	219
19.109. Kuralın İncelikleri: Yeniden Kur, Sonra Kullan	220
19.1110. MNIST Veri Seti	220
19.1211. Matris Çarpımı: İlk Foundation	221
19.1312. Üç İç İç Döngüyle Matmul	221
19.1413. Python Döngüsü Çok Yavaş	222
19.1514. Numba ile Hızlandırma	223
19.1615. İleriye: Broadcasting	224
19.1716. Kapanış	224
19.18Bu Dersin Özeti	224
19.19Kontrol Soruları	226
19.20Egzersizler	227
19.21Sonraki: Ders 11 İçin Hazırlık	227
19.22Anahtar Kavramlar (Cheat Sheet)	227
19.23ML Bağlantıları Özeti	228
20 Ders 11 — Matris Çarpımı (Matrix multiplication)	231
20.1 Bu Derste Ne Var?	231
20.2 1. Ders 10 Özeti ve Öğrenci İşleri	233
20.3 2. DiffEdit: Diffusion ile Akıllı Düzenleme	233
20.4 3. Akademik Makale Okuma İş Akışı	234
20.5 4. “From the Foundations”a Dönüş	234
20.6 5. Tensör Rank'ı	235
20.7 6. Elementwise İşlemler	235
20.8 7. Frobenius Norm	236
20.9 8. Broadcasting: Skaler ile	237
20.109. Boyut Ekleme: unsqueeze ve None	237
20.1110. Vektör-Matris Broadcasting	238
20.1211. Broadcasting Kuralları	238
20.1312. Broadcasting ile Matmul: Döngüleri Kaldırma	239
20.1413. Broadcasting Matmul: Hız	239
20.1514. Neden Bu Kadar Hızlı?	241
20.1615. İleriye: Einsum ve @	241
20.1716. Kapanış	242
20.18Bu Dersin Özeti	242
20.19Kontrol Soruları	243
20.20Egzersizler	244
20.21Sonraki: Ders 12 İçin Hazırlık	244
20.22Anahtar Kavramlar (Cheat Sheet)	245

20.23ML Bağlantıları Özeti	245
21 Ders 12 — Mean Shift Kümeleme (Mean shift clustering)	247
21.1 Bu Derste Ne Var?	247
21.2 1. CLIP Interrogator: Görüntüden Prompt	248
21.3 2. Einstein Toplamı (einsum)	248
21.4 3. einsum ile Matmul	250
21.5 4. PyTorch @ ve Hız	251
21.6 5. CUDA Kernel: GPU'da Matmul	251
21.7 6. Kümeleme (Clustering) Nedir?	252
21.8 7. Sentetik Veri: Centroid'ler	252
21.9 8. Mean Shift: Temel Fikir	253
21.109. Mesafe ve Gaussian Kernel	253
21.1110. Tek Güncelleme Adımı	253
21.1211. Mean Shift Döngüsü	256
21.1312. Triangular Kernel (Alternatif)	257
21.1413. Kaç Küme?	257
21.1514. Batched Mean Shift (GPU)	258
21.1615. İleriye: Foundations Devam	258
21.1716. Kapanış	258
21.18Bu Dersin Özeti	259
21.19Kontrol Soruları	260
21.20Egzersizler	261
21.21Sonraki: Ders 13 İçin Hazırlık	261
21.22Anahtar Kavramlar (Cheat Sheet)	262
21.23ML Bağlantıları Özeti	262
22 Ders 13 — Geri Yayılım ve MLP (Backpropagation & MLP)	265
22.1 Bu Derste Ne Var?	265
22.2 1. Notebook 3 ve ReLU ile Esneklik	266
22.3 2. Lineer Modeller ve Rectified Lines	266
22.4 3. ReLU Toplamı ile Yüzeyler	268
22.5 4. MLP from Scratch: Mimari	269
22.6 5. Lineer Katman	269
22.7 6. ReLU Katmanı	269
22.8 7. MLP Forward Pass	270
22.9 8. MSE Loss from Scratch	271
22.109. Gradyanlar ve Backpropagation	272
22.1110. Chain Rule: Zincir Kuralı	272
22.1211. Lineer Katmanın Gradyanı	272
22.1312. forward_and_backward: Tam Manuel Backprop	273
22.1413. Karpathy Köprüsü	274
22.1514. Refactoring'e Doğru	275
22.1615. Neden Elle Kurmak?	276
22.1716. Kapanış	276
22.18Bu Dersin Özeti	277
22.19Kontrol Soruları	277
22.20Egzersizler	278

22.21	Sonraki: Ders 14 İçin Hazırlık	279
22.22	Anahtar Kavramlar (Cheat Sheet)	279
22.23	ML Bağlantıları Özeti	280
23	Ders 14 — Geri Yayılım: nn.Module'ü Sıfırdan (Backpropagation)	281
23.1	Bu Derste Ne Var?	281
23.2	1. Ders 13'ün Gözden Geçirilmesi	282
23.3	2. Chain Rule $\leftrightarrow$ Kod Eşlemesi	282
23.4	3. Refactoring Neden Gerekli?	284
23.5	4. Katmanları Sınıflara Çevirmek	284
23.6	5. Lineer Katman Sınıfı	285
23.7	6. Model Sınıfı	286
23.8	7. Module Taban Sınıfı	286
23.9	8. Temiz Alt Sınıflar	287
23.10	9. PyTorch nn.Module	289
23.11	10. nn.Module ile Linear ve Model	289
23.12	11. super().__init__() {#sec-super}	290
23.13	12. requires_grad_ ve Autograd	291
23.14	13. Hazır Parçalar: F.mse_loss, nn.ReLU	291
23.15	14. Minibatch Eğitime Doğru	292
23.16	15. Karpathy Köprüsü: nn.Module	293
23.17	16. Kapanış	293
23.18	Bu Dersin Özeti	293
23.19	Kontrol Soruları	295
23.20	Egzersizler	296
23.21	Sonraki: Ders 15 İçin Hazırlık	296
23.22	Anahtar Kavramlar (Cheat Sheet)	296
23.23	ML Bağlantıları Özeti	297
24	Ders 15 — Autoencoder'lar (Autoencoders)	299
24.1	Bu Derste Ne Var?	299
24.2	1. Hedef: Convolutional Autoencoder	300
24.3	2. Convolution Nedir, Ne İşe Yarar?	300
24.4	3. Kernel ve Sliding Window	302
24.5	4. apply_kernel: Sıfırdan Convolution	303
24.6	5. Convolution = Dot Product	303
24.7	6. im2col: Convolution'ı Matmul'a Çevirme	304
24.8	7. F.unfold + Matris Çarpımı	304
24.9	8. F.conv2d ve Hız	305
24.10	9. nn.Conv2d ile CNN	306
24.11	10. Eğitim Döngüsü ve Dataset	307
24.12	11. Autoencoder Nedir?	307
24.13	12. Encoder: Conv ile Küçültme	308
24.14	13. Decoder: Deconv ile Büyütme	309
24.15	14. Autoencoder'ı Eğitmek	309
24.16	15. VAE ve Stable Diffusion'a Köprü	310
24.17	16. Kapanış	310
24.18	Bu Dersin Özeti	311

24.19	Kontrol Soruları	312
24.20	Egzersizler	313
24.21	Sonraki: Ders 16 İçin Hazırlık	313
24.22	Anahtar Kavramlar (Cheat Sheet)	313
24.23	ML Bağlantıları Özeti	314
25	Ders 16 — Learner Çerçevesi (The Learner framework)	317
25.1	Bu Derste Ne Var?	317
25.2	1. Hedef: Esnek Eğitim Çerçevesi	318
25.3	2. Basit (Esnek Olmayan) Learner	318
25.4	3. Neden Esnek Değil?	320
25.5	4. Callback Fikri	321
25.6	5. Callback Taban Sınıfı ve order	322
25.7	6. run_cbs: Callback'leri Çağırarak	322
25.8	7. CompletionCB: İlk Callback Örneği	323
25.9	8. Basic Callback Learner	324
25.10	9. Cancel Exceptions	324
25.11	10. DeviceCB: Pratik Bir Callback	325
25.12	11. Esnek Learner: Context Manager	325
25.13	12. Metrics Callback	326
25.14	13. miniai'nin Doğuşu	326
25.15	14. fast.ai Learner ile Köprü	327
25.16	15. Neden Callback Deseni Güçlü?	327
25.17	16. Kapanış	328
25.18	Bu Dersin Özeti	329
25.19	Kontrol Soruları	330
25.20	Egzersizler	331
25.21	Sonraki: Ders 17 İçin Hazırlık	331
25.22	Anahtar Kavramlar (Cheat Sheet)	332
25.23	ML Bağlantıları Özeti	332
26	Ders 17 — Başlatma / Normalizasyon (Initialization/normalization)	335
26.1	Bu Derste Ne Var?	335
26.2	1. Telemetri: Ağın İçine Bakmak	337
26.3	2. Hook Nedir?	338
26.4	3. ActivationStats: Renkli Boyut ve Ölü Üniteler	338
26.5	4. Sorun: Aktivasyonların Patlaması/Sönmesi	339
26.6	5. Varyans ve Standart Sapma	339
26.7	6. Kovaryans	340
26.8	7. Xavier / Glorot Başlatma	341
26.9	8. ReLU Sorunu ve Kaiming (He) Başlatma	342
26.10	9. init Fonksiyonunu Uygulamak	344
26.11	10. Girdi Normalizasyonu	344
26.12	11. GeneralReLU: 0-Ortalamalı Aktivasyon	345
26.13	12. LSUV: Veriye Bağlı Başlatma	345
26.14	13. LayerNorm: En Basit Normalizasyon	346
26.15	14. BatchNorm: Çalışan İstatistikler	346
26.16	15. BatchNorm vs LayerNorm — Gümüş Kurşun Değil	348

26.17	16. Kapanış	348
26.18	Bu Dersin Özeti	349
26.19	Kontrol Soruları	350
26.20	Egzersizler	351
26.21	Sonraki: Ders 18 İçin Hazırlık	351
26.22	Anahtar Kavramlar (Cheat Sheet)	352
26.23	ML Bağlantıları Özeti	352
27	Ders 18 — Hızlandırılmış SGD ve ResNet (Accelerated SGD & ResNet)	355
27.1	Bu Derste Ne Var?	355
27.2	1. Excel’de Gradient Descent	357
27.3	2. Basit SGD (Sıfırdan)	357
27.4	3. Momentum	358
27.5	4. RMSProp	359
27.6	5. Adam: Momentum + RMSProp	360
27.7	6. Learning Rate Annealing	361
27.8	7. PyTorch Optimizer ve Scheduler Nasıl Çalışır?	362
27.9	8. Scheduler Callback	362
27.10	9. Cosine Annealing	363
27.11	10. 1-Cycle Learning Rate	363
27.12	11. ResNet Sorunu: Derin Ağ Neden Kötülebilir?	364
27.13	12. Skip Connection: $out = F(x) + x$	364
27.14	13. ResBlock Sıfırdan	366
27.15	14. ResNet Modeli ve Sonuç	367
27.16	15. ETAP Kapanışı: Temeller Tamamlandı	367
27.17	16. Kapanış	368
27.18	Bu Dersin Özeti	369
27.19	Kontrol Soruları	369
27.20	Egzersizler	370
27.21	Sonraki: Ders 19 İçin Hazırlık	371
27.22	Anahtar Kavramlar (Cheat Sheet)	371
27.23	ML Bağlantıları Özeti	372
28	Ders 19 — DDPM ve Dropout (DDPM and Dropout)	373
28.1	Bu Derste Ne Var?	373
28.2	1. Dropout	374
28.3	2. DDPM Nedir? (Saf Piksel Diffusion)	376
28.4	3. Forward Process: Gürültü Ekleme (Kolay Yön)	376
28.5	4. β Çizelgesi ve	378
28.6	5. Reverse Process: Gürültüyü Kaldırmak (Zor Yön)	379
28.7	6. Anahtar İçgörü: Kayıp Sadece MSE	379
28.8	7. noisify: Forward’ı Koda Dökmek	380
28.9	8. U-Net: Gürültü Tahmincisi	380
28.10	9. Eğitim Döngüsü: DDPMCB	381
28.11	10. Sampling: Reverse’ü Koda Dökmek	382
28.12	11. DDPM’in Üç Adımlı Eğitim Reçetesi	383
28.13	12. 9B Matematiğinin Koda Dökülmesi	384
28.14	13. NYU Köprüsü: Üretken Paradigmalar	384

28.15	14. Neden DDPM Çalışır? (Yinelemeli Arıtma)	385
28.16	15. Kapanış	386
28.17	Bu Dersin Özeti	387
28.18	Kontrol Soruları	387
28.19	Egzersizler	388
28.20	Sonraki: Ders 20 İçin Hazırlık	389
28.21	Anahtar Kavramlar (Cheat Sheet)	389
28.22	ML Bağlantıları Özeti	390
29	Ders 20 — Mixed Precision ve Style Transfer (Mixed Precision and Style Transfer)	391
29.1	Bu Derste Ne Var?	391
29.2	1. Mixed Precision Nedir?	393
29.3	2. MixedPrecision Callback (Sıfırdan)	394
29.4	3. HuggingFace Accelerate	395
29.5	4. Sneaky Trick: GPU'yu Meşgul Tutmak	396
29.6	5. Style Transfer'a Giriş	396
29.7	6. Görüntüyü Optimize Etmek (Ağırlık Değil)	396
29.8	7. Optimizasyon Döngüsü: Dummy Learner	398
29.9	8. VGG16 ile Özellik Çıkarma	398
29.10	9. Content Loss: İçeriği Korumak	399
29.11	10. Gram Matrisi: Stili Ölçmek	400
29.12	11. Style Loss ve Birleşim	401
29.13	12. Builder İçgörüsü: Optimizasyon Hedefi Değişebilir	401
29.14	13. miniai Köprüsü	402
29.15	14. NYU Köprüsü: Özellik Uzayında Çalışmak	403
29.16	15. Kapanış	403
29.17	Bu Dersin Özeti	404
29.18	Kontrol Soruları	405
29.19	Egzersizler	406
29.20	Sonraki: Ders 21 İçin Hazırlık	406
29.21	Anahtar Kavramlar (Cheat Sheet)	407
29.22	ML Bağlantıları Özeti	407
30	Ders 21 — Deney Takibi, FID/KID ve DDIM (Experiment Tracking, FID/KID and DDIM)	409
30.1	Bu Derste Ne Var?	409
30.2	1. Deney Takibi: Neden Gerekli?	411
30.3	2. Weights & Biases Callback	411
30.4	3. Üretilen Görüntü Kalitesini Ölçme Sorunu	412
30.5	4. Inception Özellikleri	413
30.6	5. FID: Fréchet Inception Distance	413
30.7	6. Matris Karekökü	414
30.8	7. KID: Kernel Inception Distance	415
30.9	8. FID/KID Uyarıları	417
30.10	9. DDPM Yavaş: 1000 Adım Sorunu	417
30.11	10. DDIM: Daha Hızlı, Deterministik Örnekleme	417
30.12	11. Adım Atlama: 1000 → 50	418
30.13	12. eta: Stokastiklikten Determinizme	419
30.14	13. miniai Köprüsü	420

30.15	14. Builder İçgörüsü: Üretkeni Nasıl Ölçeriz?	420
30.16	15. Kapanış	422
30.17	Bu Dersin Özeti	422
30.18	Kontrol Soruları	423
30.19	Egzersizler	424
30.20	Sonraki: Ders 22 İçin Hazırlık	424
30.21	Anahtar Kavramlar (Cheat Sheet)	425
30.22	ML Bağlantıları Özeti	425
31	Ders 22 — Cosine Schedule, Noise Prediction ve Karras (Cosine Schedule, Noise Prediction and Karras)	427
31.1	Bu Derste Ne Var?	427
31.2	1. Cosine Schedule	428
31.3	2. Sürekli Zaman	430
31.4	3. Gürültü Seviyesini Tahmin Etme	431
31.5	4. Neden Gürültü Seviyesi Tahmini Önemli?	431
31.6	5. Karras: Elucidating the Design Space	432
31.7	6. Sürekli Sigma (Gürültü Seviyesi)	433
31.8	7. Preconditioning: scalings	434
31.9	8. denoise: Temiz Görüntüyü Tahmin Etme	434
31.10	9. Sigma Schedule ve ρ	435
31.11	10. Daha İyi Örnekleyiciler: Euler	436
31.12	11. Heun's Method (2. Derece)	437
31.13	12. LMS Sampler (Lineer Çok-Adımlı)	437
31.14	13. Sonuç: Az Adımda Yüksek Kalite	439
31.15	14. ETAP Kapanışı: Sıfırdan Diffusion A Tamamlandı	439
31.16	15. Kapanış	440
31.17	Bu Dersin Özeti	440
31.18	Kontrol Soruları	441
31.19	Egzersizler	442
31.20	Sonraki: Ders 23 İçin Hazırlık	443
31.21	Anahtar Kavramlar (Cheat Sheet)	443
31.22	ML Bağlantıları Özeti	444
32	Ders 23 — Super-Resolution (Süper Çözünürlük)	445
32.1	Bu Derste Ne Var?	445
32.2	1. Super-Resolution Nedir?	446
32.3	2. Tiny ImageNet'e Geçiş	448
32.4	3. Super-Res U-Net (TinyUnet)	448
32.5	4. Cross Connections (U-Net Skip)	449
32.6	5. Pixel Loss'un Yetersizliği	450
32.7	6. Perceptual (Feature) Loss	450
32.8	7. cmodel: Özellik Modeli	452
32.9	8. comb_loss: Pixel + Feature	453
32.10	9. Eğitim ve Keskinlik	454
32.11	10. Super-Resolution = Koşullu Üretim	454
32.12	11. Diffusion'a Köprü: Koşullu Diffusion Super-Res	455
32.13	12. Builder İçgörüsü: Loss Tasarımı	456

32.1413. miniai Köprüsü	456
32.1514. NYU/Foundations Köprüsü	456
32.1615. Kapanış	457
32.17Bu Dersin Özeti	458
32.18Kontrol Soruları	458
32.19Egzersizler	459
32.20Sonraki: Ders 24 İçin Hazırlık	460
32.21Anahtar Kavramlar (Cheat Sheet)	460
32.22ML Bağlantıları Özeti	461
33 Ders 24 — Attention ve Transformer Blokları (Attention and Transformer Blocks)	463
33.1 Bu Derste Ne Var?	463
33.2 1. Convolution’ın Sınırı: Yerel Bakış	464
33.3 2. Attention Fikri: Ağırlıklı Ortalama	466
33.4 3. Query, Key, Value	466
33.5 4. Dot Product = Benzerlik	467
33.6 5. $\text{softmax}(QK^T/\sqrt{d})V$	467
33.7 6. SelfAttention Sıfırdan	468
33.8 7. Residual: Attention + Skip	469
33.9 8. Multi-Head Attention	470
33.109. einops rearrange	471
33.1110. nn.MultiheadAttention (PyTorch)	472
33.1211. Attention’ı U-Net’e Ekleme	472
33.1312. Transformer Bloğu = Residual + Attention	472
33.1413. Karpathy nanoGPT Köprüsü	473
33.1514. miniai/Builder: Yeniden Kur, Sonra Kullan	474
33.1615. Kapanış	475
33.17Bu Dersin Özeti	475
33.18Kontrol Soruları	476
33.19Egzersizler	477
33.20Sonraki: Ders 25 İçin Hazırlık	477
33.21Anahtar Kavramlar (Cheat Sheet)	478
33.22ML Bağlantıları Özeti	478
34 Ders 25 — Latent Diffusion: Stable Diffusion’ı Sıfırdan (Latent Diffusion — SON DERS)	481
34.1 Bu Derste Ne Var?	481
34.2 1. Hatırlatma: Ders 9’da SD’yi Kullandık	482
34.3 2. VAE: Encoder → Latent → Decoder	482
34.4 3. Reparameterization	485
34.5 4. VAE Loss: KLD + Yeniden Kurma	485
34.6 5. Latent Uzak: Neden?	486
34.7 6. AutoencoderKL: SD’nin VAE’si	488
34.8 7. Latent Diffusion: Diffusion Latent Uzakta	488
34.9 8. Latent Diffusion Eğitimi	489
34.109. Text Conditioning (CLIP — Kısmen Sonraki Kursu)	489
34.1110. Stable Diffusion’ın Tam Mimarisi	490
34.1211. “Yeniden Kur, Sonra Kullan” Döngüsü Kapanır	491
34.1312. Tüm Parçaların Birleşimi: Diffusion Evrimi	492

34.1413. Phase 2 Buluşması: Üç Kurs Tek Nuktada	492
34.1514. Builder İçgörüsü: Foundations'ın Meyvesi	492
34.1615. Kapanış: Yolculuğun Sonu	493
34.17Bu Dersin Özeti	493
34.18Kontrol Soruları	494
34.19Egzersizler	495
34.20Sonraki Adımlar (Kurs Sonrası)	495
34.21Anahtar Kavramlar (Cheat Sheet)	496
34.22ML Bağlantıları Özeti	496

1 Önsöz

2 Bu kitap nedir?

Bu, **Jeremy Howard — Practical Deep Learning for Coders (fast.ai)** ders serisinin Türkçe ders notlarıdır. Hedef, videoları izlerken paralel okunabilecek; sonradan tek başına da yeterli olabilecek bir referans seti üretmek.

Serinin tek bir pedagojik iddiası var: derin öğrenme **önce çalıştırılarak** öğrenilir. İlk derste lineer cebir veya calculus gözden geçirmeyiz — doğrudan çalışan bir görüntü sınıflandırıcı eğitiriz, “neden” ve “nasıl”ı sonra, ihtiyaç duydukça açarız. Howard’ın sözüyle: *“First you’ll learn to be very very good at actually building and deploying models. And you will learn why and how things work as you need.”* Bu **top-down** yaklaşım, sporu öğrenme biçimimizi taklit eder: önce bütün oyunu göster, sonra parçaları bir araya getir.

Her bölüm bir **Builder Notu** katmanı taşır: kavramın hem **geriye** (Karpathy’nin sıfırdan kurduğu çekirdek, 6.S191, Stat 110, 18.06) hem de **ileriye** (production, deployment, modern diffusion) köprüsü. fast.ai’nin fine_tune’unun altında Karpathy’nin elle kurduğu autograd/backprop yatar; bu seriyi onunla birlikte okuyoruz: **fast.ai “ne mümkün”ü gösterir, Karpathy “altında ne var”ı kazır.**

Kaynak

- **Seri:** [Practical Deep Learning for Coders \(course.fast.ai\)](https://course.fast.ai) — Jeremy Howard, 2022
- **Yazar:** Jeremy Howard — fast.ai kurucu ortağı, eski Kaggle başkanı, Enlitic kurucusu
- **Kitap:** [fastbook \(Deep Learning for Coders with fastai & PyTorch\)](#)
- **Ders depoları:** [course22](#) (Part 1) · [course22p2 / miniai](#) (Part 2)
- **Çeviri ve genişletme:** Phase 2 (TR + ML Builder köprüleri)

3 Nasıl Okumalı

Sıralı oku. Seri kümülatiftir — özellikle **Part 2** (Ders 9–25) “Stable Diffusion’ı sıfırdan kur” hedefine doğru her dersi bir öncekinin üstüne yığar. Ders 1’de hazır kütüphaneyle (`vision_learner`, `fine_tune`) kullandığımız her şeyi, Part 2’de katman katman kendi elimizle yeniden kurarız: matmul, backprop, Learner, optimizer, ResNet, U-Net, attention, VAE — ta ki latent diffusion (gerçek Stable Diffusion) çıkana dek.

Howard’ın önerdiği akış: önce videoyu izle, sonra ilgili dersi oku, en sonunda **kodu kendin çalıştır**. Bu set videoyu **destekler**, ikame etmez.

Pratik bir tavsiye

Her bölüm sonundaki **egzersizleri** atlama. Howard’ın felsefesi: bir şeye başla, ilginç bulduğun bir şey yap, paylaş. Ders 1’den sonra kendi iki sınıfını seçip (kedi/köpek, elma/armut) notebook’u baştan çalıştırmak, dersin tamamını içselleştirmenin en hızlı yoludur. “Önce çalıştır, sonra anla.”

4 27 Ders

Seri iki parçadır. **Part 1 (Ders 1–8)** uygulamalı derin öğrenme: görüntü, NLP, tablo, öneri sistemleri. **Part 2 (Ders 9–25)** “Foundations to Stable Diffusion”: her parçayı sıfırdan kurarak modern diffusion’a tırmanır.

#	Ders	Ana Fikir
1	Başlangıç (Is it a bird?)	Çalışan görüntü sınıflandırıcı; top-down öğrenme + transfer learning
2	Dağıtım (Deployment)	Veri temizleme, export, Hugging Face Spaces + Gradio
3	Sinir ağı nasıl çalışır	Gradient descent; bir model = parametrelili fonksiyon
4	NLP & Hugging Face	Transformer’larla metin sınıflandırma; ULMFiT mirası
5	Sıfırdan model	Doğrusal model + sinir ağını elle kur (Titanic)
6	Random Forests	Karar ağaçları, bagging; klasik ML hâlâ güçlü
7	Collaborative Filtering	Gömmü (embedding) tabanlı öneri sistemleri
8	Convolutions (CNN)	Konvolüsyon, kanal, stride; görü mimarisinin temeli
9	Stable Diffusion	Part 2 başlar: pipeline’ı yüksek seviyede çalıştır
9A	SD Deep Dive (Whitaker)	CLIP, VAE, U-Net, scheduler parça parça
9B	Diffusion Matematiği	Forward process, reparameterization (ilk LaTeX-yoğun ders)
10	Matmul Temelleri	01_matmul: 3-döngüden broadcasting’e
11	Broadcasting & Makale Okuma	Broadcasting kuralları; arXiv/DiffEdit okuma
12	einsum / CUDA / meanshift	einsum, @, GPU; clustering
13	Backprop (MLP)	03_backprop: ileri/geri geçiş elle (Karpathy köprüsü)
14	Backprop (nn.Module)	Refactor: Module deseni doğar
15	Convolutions & Autoencoder	im2col $\rightarrow$ F.conv2d; encoder/decoder
16	Learner & Callbacks (miniai)	09_learner: eğitim döngüsü + callback mimarisi
17	Aktivasyonlar & Başlatma	Xavier/Kaiming init, BatchNorm (Karpathy makemore 3)
18	Hızlandırılmış SGD & ResNet	Momentum $\rightarrow$ Adam, 1-cycle, ResBlock skip
19	DDPM (Sıfırdan Diffusion)	15_DDPM: forward/reverse, ϵ -tahmini (math zirvesi)
20	Mixed Precision & Style Transfer	autocast/GradScaler; içerik/stil kaybı
21	CIFAR / W&B / FID / DDIM	Deney takibi, FID metriği, DDIM örnekleyici
22	Cosine & Karras Scheduler	cosine $\bar{\alpha}(t)$, Karras sürekli σ
23	Super-Resolution	25_superres: koşullu üretim, perceptual loss
24	Attention	27_attention: self/cross-attention (Karpathy nanoGPT köprüsü)
25	VAE & Latent Diffusion	29_vae + latent diffusion = gerçek Stable Diffusion (halka kapanır)

4 27 Ders

Not: Phase 2 pilotu Ders 1 (Başlangıç) ile başlar. Dersler 2–25 aynı şablonla eklenecektir.

5 Notasyon

Part 1 büyük ölçüde matematik-hafiftir; ağırlık Part 2’de gelir (özellikle Ders 9B ve 19+). Sık geçen birkaç gösterim:

- **Tensör:** çok boyutlu sayı dizisi — bir RGB görüntü $H \times W \times 3$ boyutludur (Ders 1).
- **Olasılık çıktısı:** softmax’ın ürettiği sınıf dağılımı; toplamı 1 (kuş + orman = 1).
- **Pretrained ağ + fine_tune:** ImageNet’te eğitilmiş ağırlıklardan başlayıp kendi verine uyarlama (transfer learning).
- **Gradient descent (Ders 3+):** $p \leftarrow p - \eta \partial L / \partial p$ — kaybın gradyanı yönünde küçük adım.
- **ε -tahmini (Ders 19):** diffusion modelinin eklenen gürültüyü tahmin etmesi; kayıp $\text{MSE}(\varepsilon, \varepsilon_\theta)$.

Tüm matematik [MathJax 3](#) ile render ediliyor.

6 Builder Eksen — fast.ai (top-down) ↔ Karpathy (bottom-up)

💡 Her ders bu bağlantı katmanını taşır

fast.ai (kullan, sonra kur)	Altında ne var (Karpathy / temel kurslar)
vision_learner + fine_tune	nn.Module + autograd + optimizer (Karpathy micrograd→makemore)
Görüntü = tensör	matris-vektör yapısı (18.06), CNN girdisi (6.S191 Ders 3)
Pretrained öznetelik dedektörleri	öğrenilen ağırlıklar (Zeiler-Fergus); fonksiyon bileşkesi (zincir kuralı)
Olasılık çıktısı (softmax)	kategorik dağılım (Stat 110)
DataBlock / DataLoaders	veri pipeline disiplini (production gerçeği)
fine_tune'un altındaki gradient	backprop = calculus zincir kuralı (Part 2'de elle kurulur)
Stable Diffusion'ı <i>kullan</i> (Ders 9)	Stable Diffusion'ı <i>kur</i> (Ders 11–25: U-Net, attention, VAE, latent)

❗ Bir tek şey

Derin öğrenmenin gücü artık özel matematik veya pahalı donanımda değil; **veriyi doğru biçimde modele sokup hazır bir ağı kendi problemine ince ayarlamakta** yatar. fast.ai bunu önce çalıştırarak öğretir; perdenin arkasındaki saf PyTorch ve matematik, kurs derinleştikçe — ve Karpathy serisinde satır satır — açılır.

7 Yazım Kuralları

- **Türkçe terminoloji + parantez içinde İngilizce orijinal** ilk geçtiğinde: “ince ayar (fine-tuning)”, “öznitelik (feature)”, “doğrulama kümesi (validation set)”.
- **Howard’dan alıntılar** İngilizce orijinal hâliyle, blockquote içinde, zaman damgasıyla verilir.
- **Builder Notu** callout’ları her ana bölüm sonunda; ML köprüsünü (geriye + ileriye) buraya yazıyoruz.
- **fast.ai/PyTorch kodu** (Howard’ın notebook’ta yazdığı `search_images`, `DataBlock`, `vision_learner`, `fine_tune`) görünür python bloklarındadır — pedagojik olarak okunmak içindir, render sırasında çalıştırılmaz. **Figürler** ise sentetik/illüstratif verilerle matplotlib üreten executable hücrelerdir.
- **Kontrol Soruları** collapse’lu — cevap kapalı başlar, okur kendi düşündükten sonra açar.
- **Egzersizler** cevapsız — Howard’ın “bir şeye başla” çağrısını izler.

 Bu kitap Howard’ın yerine geçmez

Tek başına bu set yetmez — Howard’ın canlı kodlama anlatımının ve sezgisinin yerine geçemez. Önce videoyu izle, sonra ilgili dersi oku, son olarak kodu **kendin çalıştır**. Set videoyu **destekler**, ikame etmez.

8 Başlangıç — İlk Modelini Eğit (Is it a bird?)

Önce çalıştır, sonra anla: DuckDuckGo'dan kuş fotoğrafı indir, DataBlock ile veriyi modele sok, pretrained resnet18'i fine_tune et ve iki dakikada bir kuş sınıflandırıcısı kur

i Bölüm bilgisi

- **Howard'ın videosu:** [YouTube — Lesson 1: Getting started](#) (~83 dk)
- **Seri:** Practical Deep Learning for Coders — Part 1, Ders 1
- **Hoca:** Jeremy Howard
- **Kurs sayfası:** [course.fast.ai — Lesson 1](#)
- **Notebook:** [course22 — Is it a bird?](#)
- **Okuma süresi:** ~30 dk

8.1 Bu Derste Ne Var?

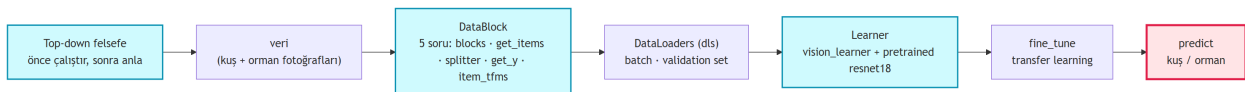
fast.ai serisinin ilk dersi, diğer her kurstan farklı bir yerden başlar: **önce çalışan bir model eğitiriz**, kavramları sonra geriye açarız. Howard 2015'te “bir fotoğraf kuş mu?” sorusunu yanıtlamanın imkânsız sayıldığı bir xkcd şakasını gösterir — sonra tam o sistemi dizüstü bilgisayarında iki dakikada kurar. Ders, derin öğrenmenin artık matematik veya devasa donanım gerektirmeden erişilebilir olduğunu **çalıştırarak** kanıtlar.

Üç temel fikir:

1. **Top-down öğrenme** — sporu nasıl öğreniyorsak (önce oyna, sonra teoriyi derinleştir) derin öğrenmeyi de öyle öğreniriz.
2. **fastai / DataBlock / Learner** — veriyi modele sokan ve modeli eğiten yüksek-seviye araçlar; “matrix multiplication ve gradient” pratikte nadiren karşımıza çıkar.
3. **Pretrained model + fine_tune** — sıfırdan değil, ImageNet üstünde eğitilmiş bir ağdan başlar, onu kendi verimize uyarlarız (transfer learning).

“We’re now going to build exactly that system for free in about two minutes!” — Howard, 1:06

Şekil 34.1 bu üç fikri, veriden tahmine uzanan tek bir akışta birleştirir.



Şekil 8.1: Ders 1'in kavram haritası: top-down felsefeden, veriyi modele sokan pipeline'a — veri → DataBlock → DataLoaders → Learner (pretrained resnet18) → fine_tune → predict (kuş / orman).

💡 Builder Notu — Top-down ↔ Bottom-up

- **Geriye (Karpathy):** Bu dersin “büyü” gibi görünen `vision_learner / fine_tune` satırlarının altında, Karpathy serisinde sıfırdan kurduğun `micrograd` → `makemore` çekirdeği durur. `fast.ai` top-down, Karpathy bottom-up — aynı madalyonun iki yüzü. Howard bunu kendisi söyler: Tesla’da AI ekibine katılan herkesin bu kursu yapması beklenir (21:59).
- **Geriye (6.S191):** “Görüntü = sayı tensörü” fikri 6.S191 Ders 3’teki (CNN) girdi temsiliyle aynıdır.
- **İleriye (production):** Ders sonunda model `learn.predict()` ile tek satırda çalışır hale gelir; Hugging Face Spaces + Gradio ile web uygulamasına dönüşür (Ders 2).
- **Tek cümle:** Doğru araçlarla, önce çalıştır-sonra-anla yaklaşımı derin öğrenmeyi herkese açar.

8.2 xkcd 2015: İmkânsız Şakadan Trivial’e

Howard derse 2015 sonundan bir xkcd çizgi-romanıyla başlar. Şakanın özü: bir kullanıcı “fotoğrafın bir kuş içerip içermediğini kontrol eden bir uygulama” ister; mühendis “bunun için bir araştırma ekibi ve beş yıl gerekir” der. 2015’te bir fotoğrafın kuş içerip içermediğini anlamak **neredeyse imkânsız** kabul ediliyordu — o kadar ki bir şakanın temeli olabiliyordu.

Howard’ın iddiası şu: aradan geçen sürede bu, imkânsızdan **trivial’e** dönüştü. Aynı sistemi birkaç dakikada, ücretsiz, bir dizüstü bilgisayarda kuracak.

“In 2015 the idea of checking whether something is a photo of a bird was considered nearly impossible. So impossible, it was the basic idea of a joke.” — Howard, 0:51

Bu çerçeve dersin tonunu belirler: teoriyle değil, **çalışan bir sonuçla** başlarız.

💡 Builder Notu — İmkânsızdan Trivial’e

- **İleriye (bu dersin kanıtı):** 2015’te imkânsız sayılan “kuş mu?” problemi, bu dersin sonunda [predict bölümünde](#) tek satırlık `learn.predict()` ile çözülür — imkânsızdan trivial’e geçişin kanıtı dersin kendisidir.
- **Geriye (alan tarihi):** 2015 öncesi bu problem [elle öznitelik mühendisliği](#) gerektiriyordu; trivial’e dönüşmesini sağlayan tam olarak modelin [kendi öğrendiği öznitelikler](#).
- **Tek cümle:** Bir şakanın temeli olan problem, doğru araçlarla bir dizüstü demosu hâline geldi.

8.3 “Önce Çalıştır” — Kuşu İki Dakikada Eğitelim

Howard hiç beklemeden kodu çalıştırır. Önce DuckDuckGo’dan kuş fotoğrafları aranır ve bir tanesi indirilip gösterilir — her adımda veriye bakmak bir alışkanlıktır:

```
from duckduckgo_search import DDGS
from fastcore.all import *

def search_images(keywords, max_images=200):
    # keyword için görsel URL listesi dondur
```

```

return L(DDGS().images(keywords, max_results=max_images)).itemgot('image')

# Once tek bir kus fotografi bul ve indir, gozunle dogrula
urls = search_images('bird photos', max_images=1)
from fastdownload import download_url
download_url(urls[0], 'bird.jpg', show_progress=False)

from fastai.vision.all import *
Image.open('bird.jpg').to_thumb(256,256)

```

Sonra iki sınıf için (kuş ve orman) yüzlerce fotoğraf indirilir ve küçültülür. “Kuş olmayan” diye arama yapılamayacağı için Howard zıt sınıf olarak “orman”ı seçer:

```

searches = 'forest','bird'
path = Path('bird_or_not')
for o in searches:
    dest = (path/o)
    dest.mkdir(exist_ok=True, parents=True)
    download_images(dest, urls=search_images(f'{o} photo'))
    # Buyuk gorsel gerekmez; 400px GPU icin yeterli ve hizli
    resize_images(path/o, max_size=400, dest=path/o)

```

“You can’t really search Google images or DuckDuckGo images for ‘not a bird’, it just doesn’t work that way. So I just decided to use forest.” — Howard, 3:31

💡 Builder Notu — Veri Toplama Döngüsü

- **İleriye (veri toplama):** Gerçek projelerde veri nadiren hazır gelir. Buradaki “ara-indir-temizle” döngüsü, production ML pipeline’ının minyatürüdür.
- **Top-down disiplini:** Henüz DataBlock’un ne olduğunu bilmiyoruz — önemli değil. Önce çalışıyor, kavram [DataBlock bölümünde](#) geri açılacak.

8.4 Görüntüler Sayılardan Oluşur

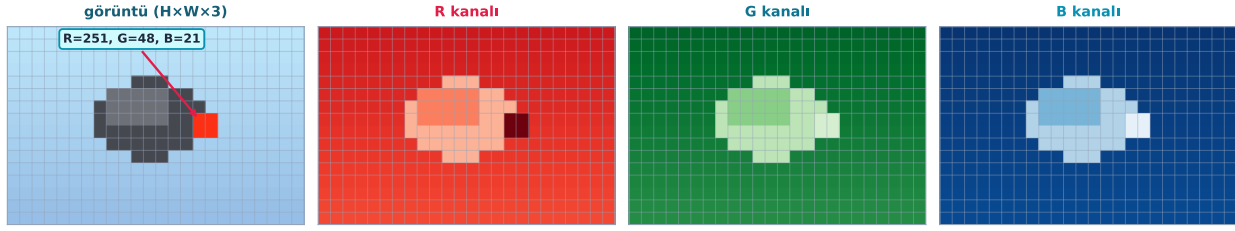
Modelin fotoğrafı “görmesi” için onu sayıya çevirmek gerekir — ve şansa bakın ki görüntüler zaten sayılardan oluşur. Her piksel kırmızı, yeşil ve mavi için 0–255 arası bir parlaklık değeri taşır. Howard örnek bir fotoğrafın gaga bölgesinde kırmızı 251, yeşil 48, mavi 21 olduğunu gösterir.

Böylece tanıdığımız bir resim, aslında bir sayı tensörüdür: örnek görüntü $256 \times 171 \times 3 \approx 131.000$ sayı. Bu tensör modelin girdisidir; çıktısı “kuş” veya “orman” olasılığıdır.

Girdi → model → çıktı akışı şöyledir: **piksel tensörü ($256 \times 171 \times 3$)** → **sinir ağı** → **olasılık (kuş / orman)**. Şekil 8.2 bu tensörü R/G/B kanallarına ayırarak somutlaştırır.

“Computers need numbers to work with, but luckily images are made of numbers.” — Howard, 2:04

8 Başlangıç — İlk Modelini Eğit (Is it a bird?)



Howard örneği: $256 \times 171 \times 3 \approx 131.000$ sayı

Şekil 8.2: Görüntü = sayı tensörü (illüstrasyon): her piksel R/G/B kanalı için 0-255 arası bir sayıdır. Turuncu gaga örneğinde R=251, G=48, B=21 — R kanalında parlak, G/B kanalında koyu. Howard’ın fotosu $256 \times 171 \times 3 \approx 131.000$ sayı.

💡 Builder Notu — Her Şey Tensör

- **Geriye (6.S191 / 18.06):** Bu çok boyutlu sayı dizisi bir **tensördür**. CNN’in girdi temsili (6.S191 Ders 3) ve matris-vektör yapısı (18.06) tam buradan başlar.
- **İleriye:** “Her şey sayıdır” ilkesi ses (spektrogram), zaman serisi ve metin (token) için de geçerlidir — [Ses bölümünde](#) göreceğiz.

8.5 Sonuç: Bu Bir Kuş

Model birkaç saniyede eğitilir. Başta indirdiğimiz kuş fotoğrafını ona sorduğumuzda cevap nettir:

```
is_bird,_,probs = learn.predict(PILImage.create('bird.jpg'))
print(f"This is a: {is_bird}.")
print(f"Probability it's a bird: {probs[0]:.4f}")
# This is a: bird.
# Probability it's a bird: 1.0000
```

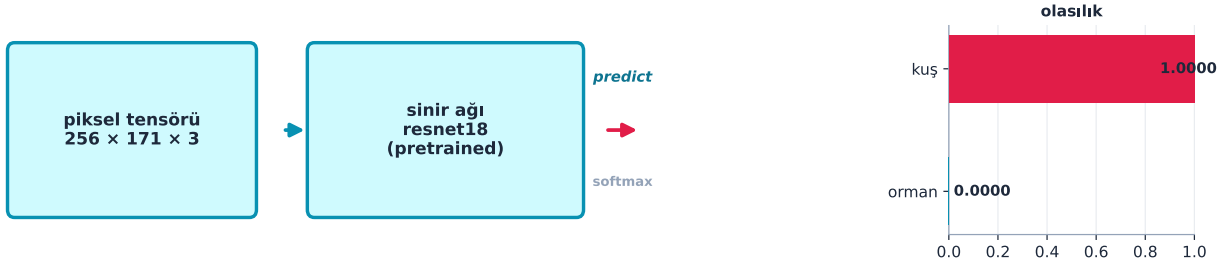
Howard’ın vurgusu: bütün bu süreç 30 saniyenin altında sürdü ve 2015’te imkânsız sayılan şeyi tamamladı.

“Something that has gone from so impossible it’s a joke to so easy that I can run it on my laptop computer in about two minutes. It didn’t take any math, didn’t take more than my laptop computer. It’s pretty accessible in fact.” — Howard, 6:23

Şekil 8.3 baştan sona bu akışı özetler: piksel tensöründen pretrained resnet18’e, oradan kuş/orman olasılığına.

💡 Builder Notu — predict ve Olasılık Dağılımı

- **İleriye (production):** `learn.predict()` modelin deploy yüzüdür — üç şey döndürür: tahmin (string), tahmin indeksi (integer) ve olasılıklar. Bunu [predict ve Deploy bölümünde](#) deploy bağlamında tekrar göreceğiz.



Şekil 8.3: Uçtan uca akış (şematik): piksel tensörü ($256 \times 171 \times 3$) → pretrained resnet18 → olasılık dağılımı (kuş / orman). predict çıktısı softmax üzerinden okunur; burada kuş 1.0000.

- **Geriye (Stat 110):** Çıktı bir olasılık dağılımıdır (kuş vs orman); softmax'ın ürettiği bu olasılıklar Stat 110'daki kategorik dağılımla aynı şeydir.

8.6 Derin Öğrenme Bugün Ne Yapabiliyor?

Howard alanın ne kadar hızlı ilerlediğini birkaç örnekle gösterir. **DALL·E 2** ve **MidJourney** metin girdisinden yeni görüntüler üretir; bir kullanıcı arkadaşlarının Twitter biyografilerini girip portreler oluşturmuştur. Google'ın **PaLM** dil modeli ise bir soruyu yanıtlamakla kalmaz, akıl yürütmesini açıklar — hatta bir esprinin neden komik olduğunu anlatabilir.

“When I look at these I still get pretty blown away that this is a computer algorithm using nothing but this text input to generate these arbitrary pictures.” — Howard, 8:22

Howard bunun etik ve pratik sonuçları olduğunu da ekler ve Rachel Thomas'ın veri etiği dersine (ethics.fast.ai) yönlendirir.

💡 Builder Notu — Diffusion'a Giden Yol

- **İleriye (bu kursun varış noktası):** DALL·E ve MidJourney'in arkasındaki teknoloji **diffusion**'dır. Bu serinin Part 2'si (Ders 9–25) tam olarak Stable Diffusion'ı sıfırdan kurar. Bugün “büyü” gibi görünen şey, dersin sonunda anlayacağın bir mekanizmaya dönüşecek.
- **Geriye (6.S191):** Üretken modeller (VAE, GAN, diffusion) 6.S191 Ders 4'te kavramsal olarak işlendi; fast.ai bunları kodla kuracak.

8.7 Kurs Nasıl İşlenecek — Top-Down Öğrenme

Howard'ın pedagojisi kasıtlı olarak terstir: derse lineer cebir ve calculus'un derinlemesine gözden geçirilmesiyle değil, **doğrudan bir model eğiterek** başladık. Gerekçesi eğitim araştırmalarına dayanır — insanlar bir

8 Başlangıç — İlk Modelini Eğit (Is it a bird?)

bağlam içinde çok daha iyi öğrenir.

“We started by training a model. We didn’t start by doing an in-depth review of linear algebra and calculus.” — Howard, 15:39

Analojisi spordur: kimseye önce yıllarca fizik ve fizyoloji öğretmeyiz; önce oyunun tamamını gösterir, basit versiyonlarını oynatır, sonra giderek daha fazla parçayı bir araya getiririz.

“The way most people learn effectively is from the way we teach sports, where we show you a whole game of sports. We show you how much fun it is. You go and start playing sports, simple versions of them, and then you gradually put more and more pieces together.” — Howard, 16:24

Derinlik gelecek — ama ihtiyaç duydukça. Önce model kurup deploy etmekte çok iyi olacak, “neden” ve “nasıl”ı gerektikçe öğreneceğiz.

“First you’ll learn to be very very good at actually building and deploying models. And you will learn why and how things work as you need, to get to the next level.” — Howard, 17:01

💡 Builder Notu — fast.ai ↔ Karpathy Yön Zıtlığı

- **Geriye (Karpathy — yön zıtlığı):** Karpathy serisi tam tersini yapar: önce sıfırdan kurar (bottom-up), soyutlamaya en son varır. İki yaklaşım çelişmez, tamamlar — fast.ai sana “ne mümkün”ü hızla gösterir, Karpathy “altında ne var”ı kazır. İkisini birlikte yapan, hem hızlı üretir hem derinlemesine anlar.
- **PhD uyarısı:** Howard, teknik eğitim almış kişilerin “her şeyi baştan anlamak” isteyeceğini, bunun bu yöntemde rahatsız edici olacağını söyler — “just do your best to go along with it” (17:10).

8.8 Neden Artık Yapabiliyoruz? 2012 vs Sinir Ağları

Howard 2015 öncesine bakar. 2012’de görüntü tanıma şöyle yapılıyordu: Stanford’daki “computational pathologist” projesi, meme kanseri sağkalımını tahmin etmek için matematikçi, bilgisayar bilimci ve patoloğlardan oluşan büyük bir ekibin **elle binlerce öznitelik** (epitel çekirdek komşulukları gibi) tasarlamasını gerektirdi. Sonra bu öznitelikler bir lojistik regresyona beslendi.

“This project took years, and a lot of people, and a lot of code, and a lot of math.” — Howard, 23:48

Sinir ağlarının farkı tam buradadır: öznitelikleri **bizim** kurmamızı gerektirmezler, kendileri öğrenirler.

“Neural networks don’t require us to build these features. They build them for us! We don’t give it features, we ask it to learn features.” — Howard, 24:06

Şekil 8.4 bu iki yaklaşımı yan yana koyar.

💡 Builder Notu — Elle vs Öğrenilen Öznitelik

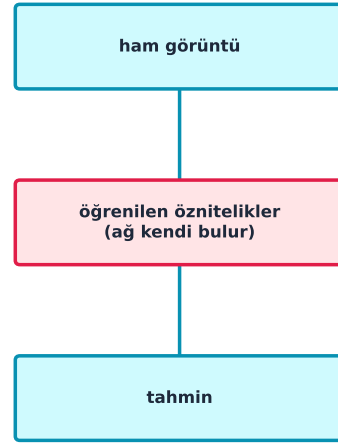
- **Geriye (klasik ML):** 2012 yaklaşımı = elle öznitelik mühendisliği + sığ model. Bu serinin Part 1 Ders 6’sında (random forest) hâlâ güçlü olan bu klasik yolu göreceğiz — ama farkı anlamak için.

2012: elle öznitelik mühendisliği



yavaş, çok emek

Sinir ağı: öğrenilen öznitelikler



öznitelikleri model öğrenir

Şekil 8.4: 2012 elle öznitelik mühendisliği (yıllar, ekip) vs sinir ağı (öznitelikleri kendi öğrenir) — şematik kontrast.

- **İleriye:** “Öznitelikleri model öğrensin” fikri, modern temel modellerin (foundation models) tüm temelidir.

8.9 Bir Sinir Ağı Ne Öğrenir?

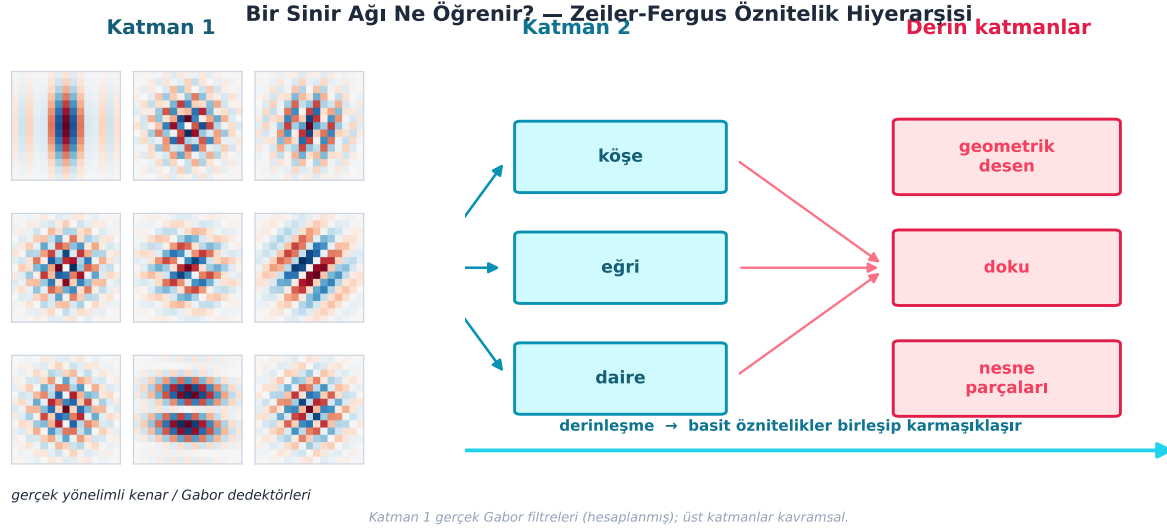
Peki ağ neyi öğrenir? 2013’te Matt Zeiler ve Rob Fergus eğitilmiş bir ağın içine bakıp ağırlıkların resmini çizdiler (*Visualizing and Understanding Convolutional Networks*; arXiv 1311.2901, ECCV 2014). İlk katmanda dokuz öznitelik dedektörü buldular: biri çapraz kenarları, biri sarı-mavi geçişleri, biri kırmızı-yeşil geçişleri buluyordu. Bunların hiçbiri elle kodlanmadı — ağ rastgele başlayıp örneklerden kendi öğrendi.

“When Zeiler and Fergus looked inside a neural network, they looked at the actual weights in the model and they drew a picture of them. We don’t have to hand code any of these.” — Howard, 24:40

“Derin” sıfatı buradan gelir: ağ basit öznitelikleri birleştirip daha karmaşık olanları kurar. İkinci katmanda köşe, eğri ve daire bulan dedektörler; daha derinde geometrik desenler, çiçek kenarları, hatta metin bulan dedektörler oluşur.

“Deep learning is deep because we can then take these features and combine them to create more advanced features.” — Howard, 25:13

Şekil 8.5 bu hiyerarşiyi gösterir: ilk katmanda gerçek kenar dedektörleri, üstte giderek karmaşıklaşan kavramlar.



Şekil 8.5: Zeiler-Fergus: Katman 1 = gerçek yönelimli kenar/Gabor dedektörleri (hesaplanmış); üst katmanlar kavramsal (köşe → desen → nesne). ‘Derin’ = basit öznitelikler birleşip karmaşılaşır. (Katman 1 gerçek hesaplama; üst katmanlar kavramsal)

💡 Builder Notu — Hiyerarşi = Fonksiyon Bileşkesi

- **Geriye (Calculus / 18.06):** Bu hiyerarşik birleştirme, fonksiyon bileşkesidir — Calculus zincir kuralının (geri yayılım) ve katmanlı doğrusal dönüşümlerin (18.06) çalışma biçimi. Karpathy serisinde bunu satır satır kuracaksın.
- **İleriye:** Katman görselleştirmesi, yorumlanabilirlik (interpretability) araştırmasının başlangıç noktasıdır.

8.10 Görü Dışında: Ses, Zaman Serisi, Tablo

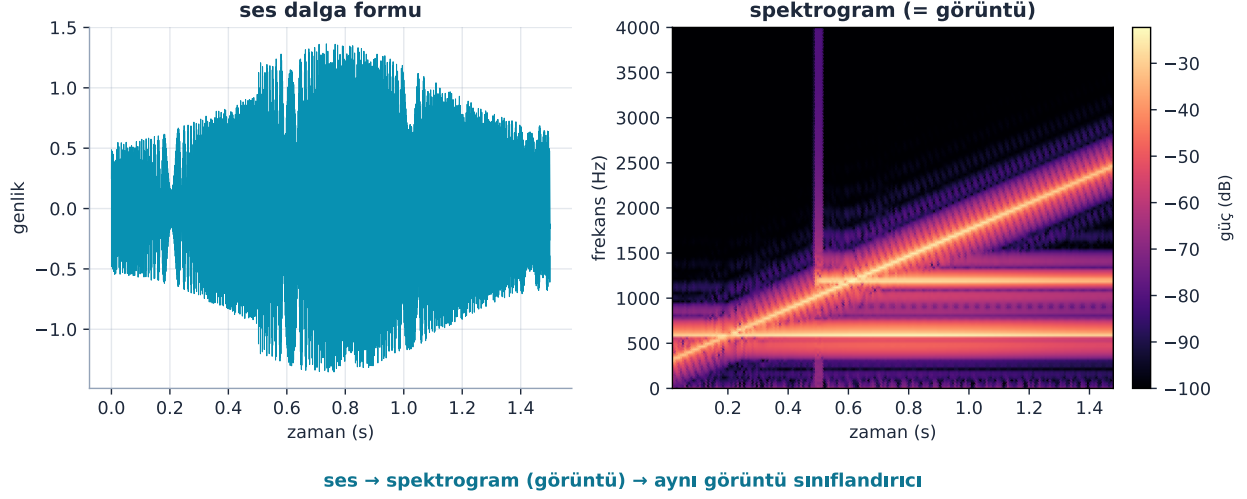
Howard kritik bir genelleme yapar: görüntü tabanlı algoritmalar yalnızca görüntü için değildir. Bir öğrenci, sesleri dalga formlarından **görüntüye** (spektrograma) çevirip aynı görüntü sınıflandırıcısıyla sınıflandırmıştır. Aynı teknik zaman serisi ve dolandırıcılık tespiti gibi problemlere de uygulanır.

“An image recognizer can also be used to classify sounds. They basically took sounds and created pictures from their waveforms.” — Howard, 27:29

Yaratıcılıkla, az sayıda temel teknik çok geniş bir alana yayılır — bu kursun ana temalarından biri budur. Şekil 8.6 bunu somutlaştırır: bir ses dalga formu spektrograma (görüntüye) dönüşür ve aynı sınıflandırıcıya girer.

💡 Builder Notu — Modaliteler Arası Transfer

- **Geriye (“her şey sayıdır”):** **Görüntü = tensör bölümündeki** fikir burada meyvesini verir — ses, metin, tablo hepsi sayıya çevrilince aynı makineye girer.
- **İleriye:** “Bir modaliteyi başkasına çevirip mevcut güçlü modeli kullanma” stratejisi, çok-modal



Şekil 8.6: Her şey sayıdır (gerçek hesaplama — `scipy.signal.spectrogram`): ses dalga formu → spektrogram (görüntü) → aynı görüntü sınıflandırıcı.

(multi-modal) öğrenmenin ve CLIP’in (Ders 9A) habercisidir.

8.11 Araçlar: PyTorch + fastai

Bu kurs **PyTorch** kullanır. Howard, TensorFlow’un araştırma dünyasında hızla gerilediğini, PyTorch’un ise büyüdüğünü belirtir — ve araştırmada kullanılan şeyin endüstrinin geleceğinin güçlü bir göstergesi olduğunu ekler.

“What people use in research is a very strong leading indicator of what’s going to happen in industry.” — Howard, 31:06

Ama PyTorch basit şeyler için bile çok kod gerektirir. Howard, AdamW optimizer’ını saf PyTorch’ta uygulayan uzun kodu, fast.ai’deki tek satırlık karşılığıyla yan yana gösterir. Fark PyTorch kötü olduğundan değil — PyTorch üstüne inşa edilecek sağlam bir temel olsun diye tasarlandığından.

“PyTorch is designed to be a strong foundation to build things on top of, like fast.ai. Particularly with deep learning: less code is better.” — Howard, 32:06

Az kod = daha az hata, daha az bakım, gömülü en iyi pratikler. Kurs derinleştikçe perdenin arkasındaki saf PyTorch giderek daha çok görünecek.

💡 Builder Notu — fast.ai’nin Altındaki PyTorch

- **Geriye (Karpathy):** Karpathy serisi tam o “saf PyTorch / sıfırdan” katmanıdır. fast.ai’nin gizlediği AdamW, autograd, nn.Module — Karpathy’de elle kurulur. fast.ai’nin `fine_tune`’u altında onun kodu yatar.
- **İleriye:** transfer learning ([Learner bölümü](#)) Howard’a göre “çok az kişinin değerinin farkında olduğu” şeydir (29:50) ve sıfırdan eğitim gereğini büyük ölçüde ortadan kaldırır.

8.12 Jupyter ve Bulut

Howard'ın slaytları aslında PowerPoint değil, **Jupyter notebook**'tur — kod ve prose'u birlikte tutan, endüstride ve akademide yaygın bir ortam. Çoğu öğrenci notebook'ları kendi bilgisayarında değil, bir bulut sunucusunda çalıştırır.

“Most people, at least most students, run Jupyter notebooks not on their own computers, but on a cloud server.” — Howard, 34:30

Howard örnek olarak **Kaggle**'ı kullanır: ücretsiz bulut notebook sunucusu + GPU. Bir hücrede ! ile başlayan satırlar Python değil, bash kabuk komutudur (örneğin `!pip install -Uqq fastai` en güncel sürümü kurar).

“Sometimes you'll see me use cells with an exclamation mark at the start. That's not Python, that's a bash shell command.” — Howard, 37:46

💡 Builder Notu — Reproducibility Disiplini

- **İleriye (reproducibility):** Notebook'un başına daima `!pip install -Uqq fastai` koymak, “neden çalışmıyor?” forum mesajlarının çoğunu önler — sürüm sabitleme, production ML'in temel disiplini.
- **Pratik:** Kaggle, internet ve GPU için telefon doğrulaması ister; notebook'un ilk hücresi bunu kontrol eder.

8.13 DataBlock — Veriyi Modele Sokmak

Howard'a göre yolculuğun başında öğrenilecek **en kritik** şey, mimari değil, veriyi modele nasıl soktuğundur. Bu şaşırtıcı gelebilir:

“You might be thinking we should be spending all of our time talking about neural network architectures, and matrix multiplication and gradients. The truth is very little of that comes up in practice.” — Howard, 42:04

Sebebi: derin öğrenme topluluğu, neredeyse tüm uygulamalar için işe yarayan az sayıda model tipi bulmuştur ve fast.ai çoğu zaman doğru modeli senin için seçer. Bu yüzden kursun adı *Practical Deep Learning*'dir — pratikte önemli olana odaklanır.

İşte **önce-çalıştır bölümünde** “büyü” gibi geçen DataBlock artık açılıyor. Yüzlerce projeden damıtılan beş soruyu yanıtlar:

```
dls = DataBlock(
    blocks=(ImageBlock, CategoryBlock),      # girdi = goruntu, cikti = kategori
    get_items=get_image_files,               # ogeleri nasil bul
    splitter=RandomSplitter(valid_pct=0.2, seed=42), # %20 dogrulama ayir
    get_y=parent_label,                     # etiket = ust klasor adi (bird/forest)
    item_tfms=[Resize(192, method='squish')] # her goruntuyu 192x192'ye getir
).dataloaders(path)
```

```
dls.show_batch(max_n=6)
```

“This course is called practical deep learning, and so we’re going to focus on the stuff that is practically important.” — Howard, 43:14

💡 Builder Notu — Veri Pipeline Önceliği

- **İleriye (production gerçeği):** “Veri hazırlama, mimariden daha çok zaman alır” — bu, her ML mühendisinin tecrübeyle öğrendiği gerçektir. fast.ai bunu en başa koyar.
- **Geriye (Karpathy / 18.06):** Howard “matrix multiplication ve gradient nadiren gelir” derken pratiği kasteder; ama altında onlar vardır. Karpathy serisi ve 18.06 tam o gizli katmanı açar.

8.14 DataLoaders, Batch ve Validation Set

DataBlock’tan **DataLoaders** (kısaca dls) üretilir. PyTorch bunların üzerinden veriyi parça parça çeker. GPU aynı anda binlerce işlem yapabildiği için, ona aynı anda binlerce iş vermek gerekir — bu yüzden veri tek tek değil, **batch** (veya mini-batch) hâlinde beslenir.

“A data loader will feed the training algorithm with a batch of your images at once; in fact we don’t call it a bunch, we call it a batch or a mini batch.” — Howard, 47:24

DataBlock’taki en kritik parçalardan biri **validation set**’tir: modelin doğruluğunu ölçmek için ayrılan, eğitimde görülmeyen veri. fast.ai bunu o kadar önemser ki, validation set olmadan model eğitmene izin vermez.

“It’s critical that you put aside some data for testing the accuracy of your model — that’s called a validation set. It’s so critical that fast.ai won’t let you train a model without one.” — Howard, 45:26

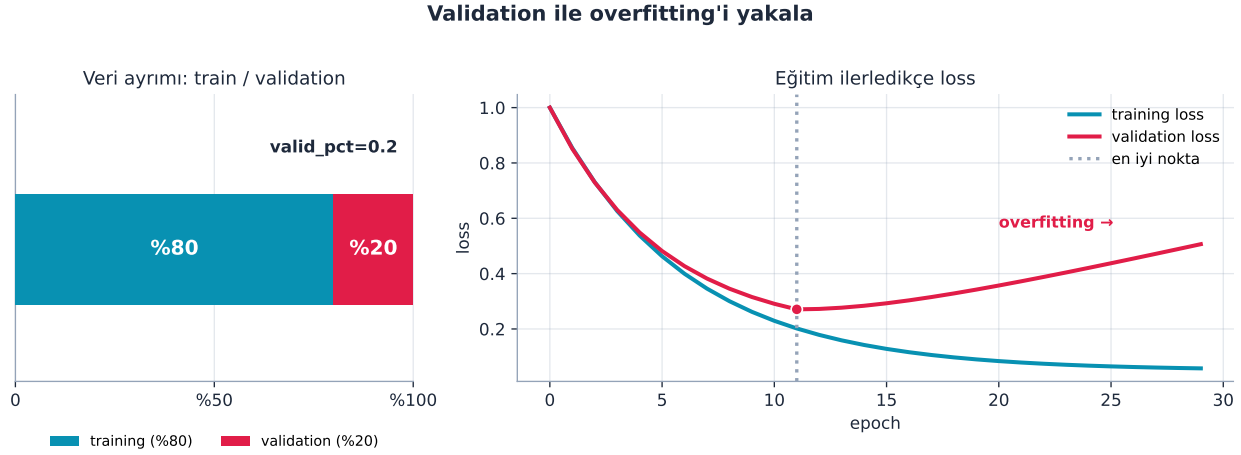
show_batch() ile bir batch’i (görüntü + etiket) gözle doğrulayabiliriz — Howard her adımda veriye bakmayı bir alışkanlık olarak öğütler. Şekil 8.7 train/validation ayrımını ve overfitting’i şematik olarak gösterir.

💡 Builder Notu — Validation ve Overfitting

- **Geriye (Stat 110):** validation set, aşırı öğrenmeyi (overfitting) ölçmenin yoludur; istatistikteki train/test ayrımının ML karşılığı. valid_pct=0.2 rastgele %20 örneklemdir.
- **İleriye:** batch boyutu, GPU bellek/throughput dengesini belirler — Part 2’de (Ders 20, mixed precision) bunu derinleştireceğiz.

8.15 Learner + Pretrained Model

Şimdi modele ihtiyaç var. fast.ai’nin merkez kavramı **Learner**’dır: modeli (sinir ağı fonksiyonu) ve onu eğiteceğimiz veriyi tek nesnede birleştirir. Bu yüzden iki şey verilir — dls ve bir model mimarisi:



Şekil 8.7: Train/validation ayrımı (%80/%20) + overfitting (şematik): training loss düşmeye devam eder, validation loss bir noktadan sonra yükselir. Modeli, eğitimde görmediği veride ölç — “en iyi nokta” validation loss’un en düşük olduğu epoch’tur.

```
learn = vision_learner(dls, resnet18, metrics=error_rate)
learn.fine_tune(3)
```

“The critical concept here in fastai is called a learner. A learner is something which combines a model and the data we use to train it with.” — Howard, 49:39

Burada asıl sihir resnet18’in **pretrained** (önceden eğitilmiş) olmasıdır. vision_learner çağrıldığında fast.ai resnet18.pth ağırlıklarını indirir — bu ağ daha önce **ImageNet** üstünde, 1000’den fazla kategoride, 1 milyondan fazla görüntüyle eğitilmiştir.

“You don’t start with a random network that can’t do anything. You actually start with a network that can do an awful lot.” — Howard, 52:26

fine_tune ise bu hazır ağırlıkları alıp, dikkatlice kontrol edilmiş bir biçimde **yalnızca** bizim veri setimizle ImageNet arasındaki farkı öğretir. Adı buradan gelir: ince ayar (transfer learning).

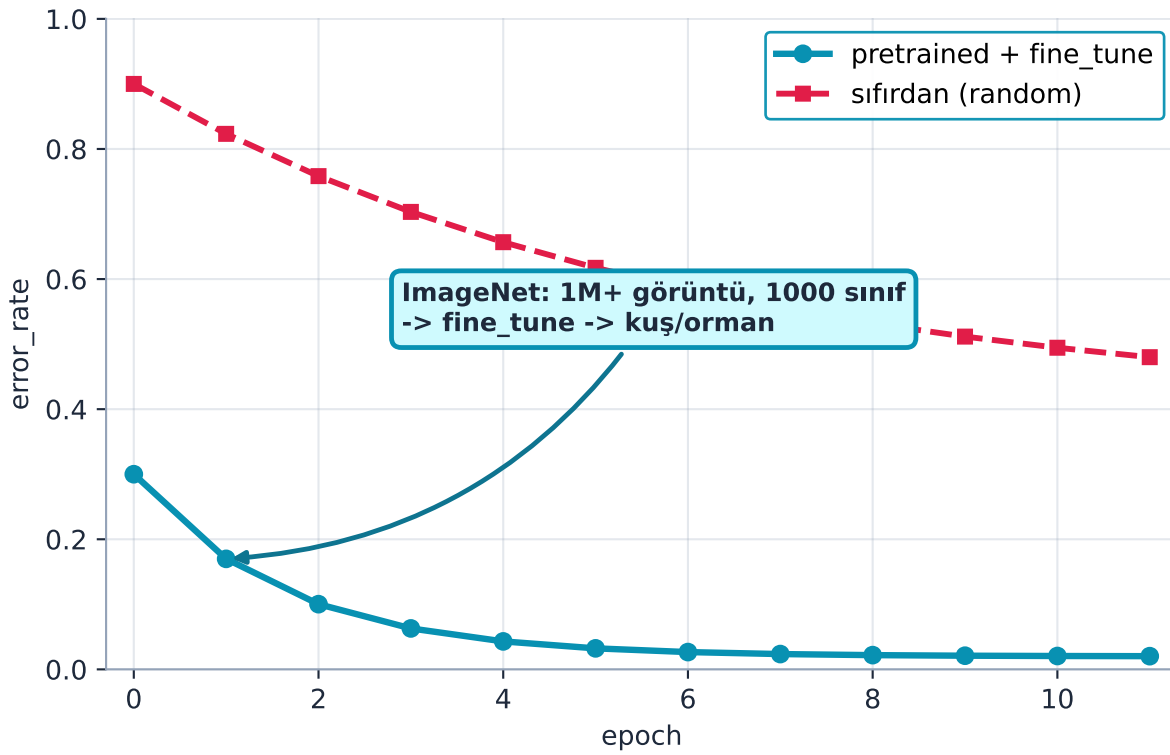
“fine_tune takes those pre-trained weights and adjusts them in a really carefully controlled way to just teach the model the differences between your data set and what it was originally trained for. That’s called fine tuning.” — Howard, 52:38

(fast.ai, Ross Wightman’ın timm kütüphanesini entegre eder — dünyanın en büyük gözü modeli koleksiyonu; ama resnet ailesi çoğu iş için yeterlidir.)

Şekil 8.8, pretrained bir ağı neden sıfırdan eğitime kıyasla çok daha hızlı yüksek doğruluğa ulaştığını gösterir.

💡 Builder Notu — Transfer Learning’in Gücü

- **Geriye (Karpathy / 6.S191):** resnet18 bir CNN’dir (6.S191 Ders 3); pretrained ağırlıklar, [Bir Sinir Ağı Ne Öğrenir](#) bölümündeki öğrenilmiş öznetelik dedektörleridir. fine_tune’un altındaki gradient/backprop, Karpathy serisinde elle kurulur.



Şekil 8.8: Transfer learning (illüstratif eğriler — gerçek eğitim değil): pretrained + fine_tune düşük başlar, hızlı düşer; sıfırdan (random) yüksek başlar, yavaş iner. ImageNet on-eğitimi -> fine_tune -> kuş/orman görevine adaptasyon.

- **İleriye (production):** Transfer learning, modern ML'in kalbidir — sıfırdan eğitim nadirdir. Aynı mantık LLM'lerde "base model + fine-tune" (Ders 4 ULMFiT, Part 2) olarak devam eder.

8.16 predict ve Deploy

Artık elimizde, kuşları ormanlardan ayırmak için ince ayar yapılmış bir Learner var. [Sonuç bölümünde](#) gördüğümüz `learn.predict()` tam da modelin **deploy** (kullanıma alma) yüzüdür. Üç şey döndürür: tahmin (string, örn. "bird"), tahmin indeksi (integer) ve her sınıfın olasılığı.

"learn.predict() is going to return whether it's a bird or not as a string, whether it's a bird or not as an integer, and the probability that it's a non-bird or a bird." — Howard, 54:14

Gerçek bir uygulamada (örneğin bir milli parkta kuş fotoğrafı doğrulayan bir app), gereken tek şey bu tek satırdır.

💡 Builder Notu — Deploy: Üretimin Asıl Zorluğu

- **İleriye (Ders 2):** Bir sonraki ders bu modeli Hugging Face Spaces + Gradio ile gerçek bir web uygulamasına dönüştürür. Eğitim değil, **deploy** üretimin asıl zorluğudur.
- **Geriye (Stat 110):** Olasılık çıktısı softmax'tan gelir; modelin güveni (calibration) Stat 110 kavramıdır.

8.17 Diğer Uygulamalar ve Kapanış

Howard, aynı birkaç satırın bilgisayarlı görünüm ötesine nasıl yayıldığını gösterir. **Segmentation**'da her piksel, ait olduğu nesneye göre renklendirilir (yol sahnelerinde araba, bina, çit). Model yaklaşık 20 saniyede, çok az veriyle eğitilir ve adımlar tanıdık.

"Segmentation is where we take photos and we color in every pixel according to what it is. The steps are actually going to look quite familiar." — Howard, 55:03

Tabular analiz (tablolardan, veritabanlarından sütun tahmini) endüstride en yaygın model türüdür ve o da daha önce gördüğümüze çok benzer: yine `data loaders`, yine `show_batch`, sadece "hangi sütunlar kategorik, hangileri sürekli" bilgisi eklenir.

"Perhaps the most widely used kind of model used in industry is tabular analysis. It really looks very similar to what we've seen already." — Howard, 58:19

Howard dersi bir çağrıyla kapatır: bir şeye başla, ilginç veya eğlenceli bulduğun bir şey yap, forumda paylaş — ve kitabın sonundaki quiz sorularını cevapla.

"Have a go at starting something. Create something you think would be fun or interesting and share it in the forum. And don't forget to look at the quiz questions at the end of the book." — Howard, 1:22:11

💡 Builder Notu — Aynı Omurga, Farklı Baş

- **İleriye:** Segmentation’da kullanılan **U-Net**, Part 2’de diffusion’ın gürültü tahmincisi olarak geri dönecek (Ders 19, 23). Aynı omurga, farklı baş.
- **Geriye (Part 1 yol haritası):** Tabular ve collaborative filtering Ders 5–7’de, CNN Ders 8’de derinleşir; “aynı adımlar” sözü tüm Part 1’in iskeletidir.

8.18 Bu Dersin Özeti

1. Derin öğrenme, 2015’te imkânsız sayılan görüntü tanımayı bugün dakikalar içinde, matematik veya pahalı donanım olmadan erişilebilir kılar.
2. fast.ai **top-down** öğretir: önce çalışan model, sonra “neden/nasıl” — sporu öğrenir gibi.
3. Görüntüler sayı tensörleridir ($H \times W \times 3$); “her şey sayıdır” ilkesi ses, zaman serisi ve tabloyu da kapsar.
4. Sinir ağları öznetelikleri **kendileri öğrenir** (Zeiler-Fergus); “derin”, basit özneteliklerin birleşerek karmaşılaşmasıdır.
5. DataBlock → DataLoaders veriyi modele sokar; en kritik pratik beceri budur, mimari değil.
6. **Learner** model + veriyi birleştirir; vision_learner pretrained bir ağ (ImageNet) yükler, fine_tune onu transfer learning ile kendi verimize uyarlar.
7. learn.predict() modelin deploy yüzüdür; tahmin + indeks + olasılık döndürür.
8. Aynı birkaç satır segmentation ve tabular gibi çok farklı problemlere genişler.

❗ Tek Bir Cümle

Derin öğrenmenin gücü artık özel matematik veya donanımda değil, **veriyi doğru biçimde modele sokup hazır bir ağı kendi problemine ince ayarlamakta** yatar — ve bunu önce çalıştırıp sonra anlayarak öğrenirsin.

8.19 Kontrol Soruları

i Soru 1: DataBlock, veriyi modele sokmak için hangi beş soruyu yanıtlar?

Cevap:

1. **blocks** — girdi ve çıktı tipi nedir? (örn. ImageBlock, CategoryBlock)
2. **get_items** — öğeleri nasıl buluruz? (örn. get_image_files)
3. **splitter** — validation set’i nasıl ayırırız? (örn. RandomSplitter(valid_pct=0.2))
4. **get_y** — etiket nereden gelir? (örn. parent_label = üst klasör adı)
5. **item_tfms** — her öğeye hangi dönüşüm uygulanır? (örn. Resize(192))

Bu beş parça, yüzlerce projeden damıtılmış “veriyi doğru şekle sokma” reçetesidir.

i Soru 2: Pretrained bir modelle başlamak, sıfırdan (rastgele) başlamaktan neden çok daha iyidir? fine_tune tam olarak ne yapar?

Cevap: Sıfırdan rastgele bir ağ hiçbir şey bilmez. Pretrained resnet18 ise ImageNet'te 1000+ kategoride, 1M+ görüntüyle eğitilmiştir — kenar, doku, şekil bulan öznetelik dedektörlerini zaten öğrenmiştir (**Bir Sinir Ağı Ne Öğrenir**). fine_tune bu hazır ağırlıkları alır ve dikkatlice kontrol edilmiş biçimde **yalnızca** kendi veri setimizle ImageNet arasındaki farkı öğretir. Sonuç: birkaç saniyede yüksek doğruluk. Bu transfer learning'dir ve Howard'a göre “çok az kişinin değerinin farkında olduğu” şeydir.

i Soru 3: “Kuş olmayan” diye arama yapılamayınca Howard ne yaptı? Ayrıca fast.ai neden validation set olmadan eğitime izin vermez?

Cevap: Howard zıt sınıf olarak “orman” (forest) fotoğraflarını seçti — “not a bird” araması işe yaramaz, ama “bird vs forest” iki net sınıf verir. Validation set zorunluluğu ise **aşırı öğrenmeyi** (overfitting) yakalamak içindir: model eğitim verisini ezberleyip yeni veride başarısız olabilir. Doğruluğu yalnızca eğitimde görülmeyen veride ölçmek dürüst tek yoldur; fast.ai bunu o kadar kritik görür ki validation set olmadan eğitime izin vermez.

i Soru 4: fast.ai'nin top-down yaklaşımı ile Karpathy'nin bottom-up yaklaşımı nasıl farklıdır? Hangisi ne zaman işe yarar?

Cevap: fast.ai **top-down**: önce çalışan model (vision_learner + fine_tune), kavram sonra geriye açılır — “ne mümkün”ü hızla gösterir, hızlı üretim sağlar. Karpathy **bottom-up**: micrograd → makemore → GPT'yi sıfırdan kurar — “altında ne var”ı kazır, derin anlama verir. İkisi çelişmez, tamamlar: fast.ai'nin fine_tune'unun altında Karpathy'nin elle kurduğu autograd/backprop yatar. Önce fast.ai ile üretmeyi öğren, sonra Karpathy ile büyüüyü çöz — tam builder böyle oluşur.

8.20 Egzersizler

Egzersiz 1 (Direkt uygulama). Kendi iki sınıfını seç (örn. “kedi” vs “köpek” veya “elma” vs “armut”), notebook'u baştan çalıştır ve error_rate'i gözle.

Egzersiz 2 (İki-aşamalı). Üçüncü bir sınıf ekle (örn. üç farklı kuş türü). DataBlock ve searches'i güncelle, modelin üç sınıfta doğruluğunu kıyasla.

Egzersiz 3 (Edge case). item_tfms'de Resize(192, method='squish') yerine method='crop' ve method='pad' dene. show_batch ile görsel farkı ve error_rate etkisini karşılaştır.

Egzersiz 4 (Python ile doğrulama). learn.predict()'in döndürdüğü probs'u ve dls.vocab'ı yazdır. Hangi sınıfın hangi indekste olduğunu ve olasılıkların toplamının neden 1 olduğunu açıkla.

Egzersiz 5 (Sonraki dersin habercisi). Eğittiğin modeli nasıl bir web uygulamasına çevirebileceğini araştır (Hugging Face Spaces + Gradio). Ders 2'de bunu yapacağız.

8.21 Sonraki Ders İçin Hazırlık

Ders 2: Dağıtım (Deployment)

Ders 1’de bir model eğittik; Ders 2’de onu gerçek dünyaya çıkarıyoruz. Eğitim değil, **deploy** üretimin asıl zorluğudur. Veri temizleme, model export ve bir web uygulaması olarak yayınlamayı göreceğiz.

Ana konular:

- Veri temizleme ve modelin iyileştirilmesi
- Modeli dışa aktarma (`learn.export`) ve yükleme
- Hugging Face Spaces + Gradio ile ücretsiz web uygulaması
- GitHub Pages / JavaScript ile bağlama



Ders 2 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 5).
- Kendi modelini Kaggle veya Colab’da en az bir kez baştan sona çalıştır.
- Ana cümleyi tekrar oku: “Önce çalıştır, sonra anla.”

8.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard’da
Top-down öğrenme	Önce çalışan model, teori sonra (spor analojisi)	15:45
Görüntü = tensör	$H \times W \times 3$ boyutlu, 0-255 arası sayı dizisi	2:04
DataBlock	Veriyi modele sokan 5-parçalı API	42:04
DataLoaders (dls)	Batch’leri modele besleyen iterator	47:24
Validation set	Doğruluk ölçen, eğitimde görülmeyen ayrılmış veri (zorunlu)	45:26
batch / mini-batch	GPU’ya aynı anda verilen veri grubu	47:24
Learner	Model + veriyi tek nesnede birleştiren çekirdek	49:39
vision_learner	Görü için hazır Learner üreticisi (resnet18)	49:58
Pretrained model	ImageNet’te (1M+ görüntü) eğitilmiş başlangıç ağı	52:04
fine_tune	Hazır ağırlıkları kendi veriye uyarlama (transfer learning)	52:38
predict	Tahmin + indeks + olasılık döndürür (deploy yüzü)	54:14

Kavram	Tanım	Howard'da
Segmentation	Her pikseli ait olduğu sınıfa göre etiketleme	55:17

8.23 ML Builder Bağlantıları

💡 Builder Notu — 6 ML Köprüsü

Bu ders, modern derin öğrenme pratiğinin top-down giriş kapısıdır — köprülerin özeti:

1. **Tensör** → görüntü, ses, metin hepsi sayı dizisidir; tüm derin öğrenmenin ortak girdi temsili (6.S191, 18.06).
2. **Transfer learning** → LLM'lerde “base model + fine-tune” olarak devam eder; modern ML'in standart reçetesi (Ders 4 ULMFiT, Part 2).
3. **DataBlock disiplini** → production ML'de veri pipeline'ı mimariden çok daha fazla zaman alır; en kritik pratik beceri.
4. **Pretrained CNN** → foundation models'ın atasıdır: bir kez pahalı eğit, sayısız yerde ucuza kullan.
5. **Top-down + bottom-up** → fast.ai (üretmeyi öğret) + Karpathy (anlamayı öğret) birlikte tam builder yapar.
6. **U-Net** → segmentation'dan diffusion'a (Part 2 Ders 19, 23) taşınan omurga; aynı yapı, farklı baş.

! Bu dersten tek bir şey alıp gideceksen

Bu dersten tek bir şey alıp gideceksen: derin öğrenme artık bir büyü değil, erişilebilir bir araçtır. Onu önce çalıştır, sonra altındaki mekanizmayı kazı. fast.ai sana “ne mümkün”ü gösterir; Karpathy serisi ve Part 2 “altında ne var”ı öğretir — ikisi birlikte seni hem üreten hem anlayan bir builder yapar.

9 Dağıtım — Modeli Dünyaya Aç (Deployment)

Önce eğit-sonra temizle, modeli `learn.export` ile dondur, `load_learner + predict` ile her yerde çalıştır ve Gradio + Hugging Face Spaces ile ücretsiz bir canlı web uygulaması yayınla

Bölüm bilgisi

- **Howard'ın videosu:** [course.fast.ai — Lesson 2: Deployment](https://course.fast.ai/lesson2-deployment) (~77 dk)
- **Seri:** Practical Deep Learning for Coders — Part 1, Ders 2
- **Hoca:** Jeremy Howard
- **Playlist:** [Practical Deep Learning Part 1 \(2022\)](#)
- **Notebook:** [course22 — 02-saving-a-basic-fastai-model](#)
- **Okuma süresi:** ~30 dk

9.1 Bu Derste Ne Var?

Ders 1'de bir model eğittik. Ders 2 onu **gerçek dünyaya çıkarır** — çünkü eğitim değil, **deploy** üretimin asıl zorluğudur. Howard yolu baştan sona yürür: veri toplama, modeli eğiterek **veriyi temizleme** (ters sezgi), modeli `learn.export` ile dondurma ve Hugging Face Spaces + Gradio ile ücretsiz bir web uygulaması yayınlama.

Üç temel fikir:

1. **Önce eğit, sonra temizle** — “önce veri hazırla, sonra eğit” sezgisinin tersi; eğitilmiş model en kötü verileri (yanlış etiket, bozuk görsel) senin yerine bulur ([Ters Sezgi bölümü](#)).
2. **Veri artırımı (data augmentation)** — aynı görüntüden her epoch'ta biraz farklı bir kopya üretip modeli sağlamlaştırma ([Veri Artırımı bölümü](#)).
3. **export → load → predict** — eğitilen model bir `pk1` dosyasına dondurulur, GPU'suz herhangi bir yerde yüklenip milisaniyelerde tahmin yapar ([export → load + predict](#)).

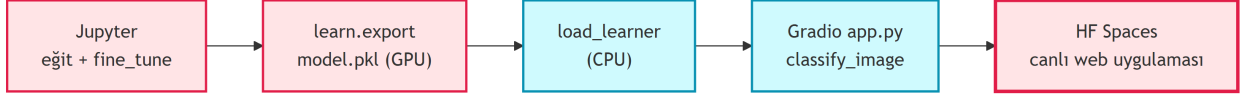
“It feels like I'm really really pumped about this lesson.” — Howard, 0:31

Şekil 9.1 bu üç fikri, Jupyter'da eğitimden Hugging Face Spaces'te canlı web uygulamasına uzanan tek bir deploy hattında birleştirir.

Builder Notu — Deploy = Üretimin Asıl İş

- **İleriye (production):** Bu ders tamamen MLOps'un giriş kapısıdır — model bir notebook'ta kalmaz, internette çalışan bir servise dönüşür. “Deploy = asıl zor kısım” tezi tüm production

9 Dağıtım — Modeli Dünyaya Aç (Deployment)



Şekil 9.1: Deploy hattı (şematik): Jupyter’da eğit + fine_tune (GPU) → learn.export model.pkl üretir (GPU gereken tek adım) → load_learner modeli CPU’da geri yükler → Gradio app.py tahmin fonksiyonunu web arayüzüne sarar → Hugging Face Spaces canlı web uygulaması olarak yayınlar.

ML’in gerçeğidir.

- **Geriye (Ders 1):** vision_learner + fine_tune aynı; yeni olan tek şey export / load_learner / predict zinciri ve veri temizleme araçları.
- **Tek cümle:** Bir model ancak başkalarının kullanabildiği anda değer üretir; bu ders modeli notebook’tan çıkarıp dünyaya verir.

9.2 Geri Dönüş: Öğrenciler Neler Yaptı?

Howard derse, geçen haftanın ödevinden (“yaptığımı paylaş”) gelen öğrenci projeleriyle başlar — bir hasarlı araç sınıflandırıcısı dahil. Mesaj açık: Ders 1’in birkaç satırıyla bile insanlar gerçek, çalışan uygulamalar kurabiliyor.

Howard ayrıca kitabı (Deep Learning for Coders) ve forumları (forums.fast.ai) hatırlatır: aynı bilgiyi farklı yollardan duymak öğrenmeyi pekiştirir.

“As promised, I put up the ‘show us what you’ve made’ post, and already a lot of people have posted.” — Howard, 4:16

💡 Builder Notu — Önce Üret, Sonra Paylaş

- **Geriye (Ders 1):** Bu, Ders 1 sonundaki “yaptığımı paylaş” ödevinin karşılığıdır — derste gösterilen öğrenci projeleri (hasarlı araç sınıflandırıcısı dahil) tam o ödevden geldi.
- **İleriye:** “Önce bir şey üret, paylaş” — fast.ai felsefesinin merkezi; portfolyo, ilk işi bulmanın en hızlı yoludur.

9.3 Hedef: Modeli Production’a Almak

Dersin omurgası net bir hedeftir: eğittiğimiz modeli **production**’a (kullanıma) almak. Howard adımları sayar — veriyi bul, modeli eğit, temizle, dışa aktar, bir web uygulaması olarak yayınla.

“Step one is, well, you’ve kind of done it: you train a model. Now we have to figure out how to put it into production.” — Howard, 6:49

Ders 1’in top-down ruhu burada da geçerli: önce çalışan bir deploy görürüz, ayrıntılar yolda açılır.

💡 Builder Notu — Eğitim Bir Kez, Deploy Sürekli

- **Geriye (Ders 1):** Ders 1’de modeli `fine_tune` ile bir kez eğitip orada bırakmıştık; o “bir kez”lik eğitim adımı bu dersin başlangıç noktası, asıl iş ise sonrasındır.
- **İleriye (MLOps):** Eğitim bir kez olur; deploy sürekli çalışmalıdır. Üretimde zamanın çoğu modeli değil, modelin **çevresini** (servis, izleme, güncelleme) kurmaya gider.

9.4 Veriyi Toplamak

Howard bir ayı sınıflandırıcısı (`grizzly / black / teddy`) kurar. Veri yine arama motorundan (Bing/DuckDuckGo) toplanır ve Ders 1’deki `DataBlock` ile yüklenir. Yardım bulmanın yollarını da gösterir: bir fonksiyon adının yanına tek ? kısa bilgi, çift ?? tüm kaynak kodu verir.

“If you put a double question mark next to any function name, you’ll actually get the whole source code for it.” — Howard, 11:26

💡 Builder Notu — ?? ile Kaynağa İnme

- **Geriye (Ders 1):** `DataBlock`’un beş parçası (`blocks`, `get_items`, `splitter`, `get_y`, `item_tfms`) burada aynen kullanılır.
- **İleriye:** ?? ile kaynağa inme alışkanlığı, kütüphane içini okuyup öğrenmenin en hızlı yoludur — Karpathy’nin “from scratch” ruhunun pratik versiyonu.

9.5 Ters Sezgi: Önce Eğit, Sonra Temizle

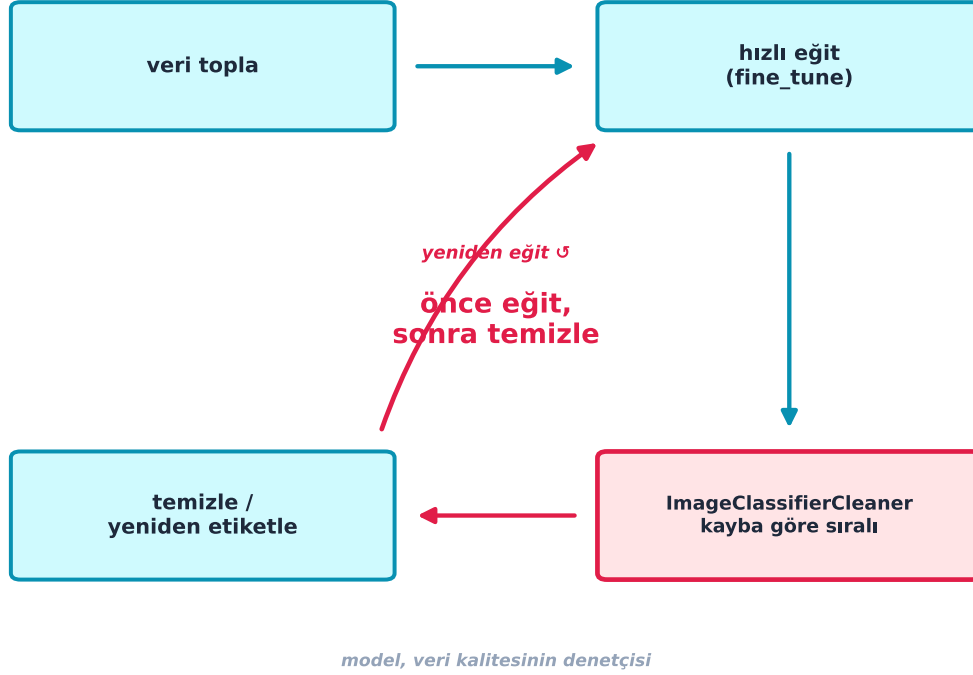
Howard kasıtlı olarak şaşırtıcı bir şey söyler: veriyi temizlemeden önce **bir model eğitiriz**. “Önce veriyi hazırla, sonra eğit” sezgisinin tam tersidir.

“I’m going to tell you something that you might find really surprising. You clean the data, you train a model. I know that’s going to sound really backwards. First we’re going to train a model and you’ll see why in a moment.” — Howard, 12:47

Sebebi sonraki bölümlerde açılacak: eğitilmiş model, hangi örneklerin yanlış etiketli veya bozuk olduğunu (en yüksek kayıplı örnekler) bizim için işaret eder — somut araç [ImageClassifierCleaner bölümünde](#). Şekil 9.2 bu ters sezgili döngüyü — eğit → temizle → yeniden eğit — bir bütün olarak gösterir. Önce hızlıca eğitiriz:

```
bears = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    get_y=parent_label,
    item_tfms=RandomResizedCrop(224, min_scale=0.5),
    batch_tfms=aug_transforms())
dls = bears.dataloaders(path)
```

```
learn = vision_learner(dls, resnet18, metrics=error_rate)
learn.fine_tune(4)
```



Şekil 9.2: Veri-merkezli temizleme döngüsü (şematik): önce hızlı eğit (fine_tune), ImageClassifierCleaner ile kayba göre sıralı şüpheli örnekleri temizle / yeniden etiketle, sonra yeniden eğit. Model, veri kalitesinin denetçisi — döngü kendini besler.

💡 Builder Notu — Veri-Merkezli ML’in Çekirdeği

- **Geriye (Ders 1):** Döngüdeki “hızlı eğit” adımı, Ders 1’in vision_learner + fine_tune zincirinin aynısıdır; yeni olan, o modeli veri kalitesini ölçmek için kullanmaktır.
- **İleriye (veri-merkezli ML):** “Model, veri kalitesini denetler” fikri modern veri-merkezli (data-centric) ML’in temelidir; etiket gürültüsünü modelle bulmak yaygın bir production tekniğidir.

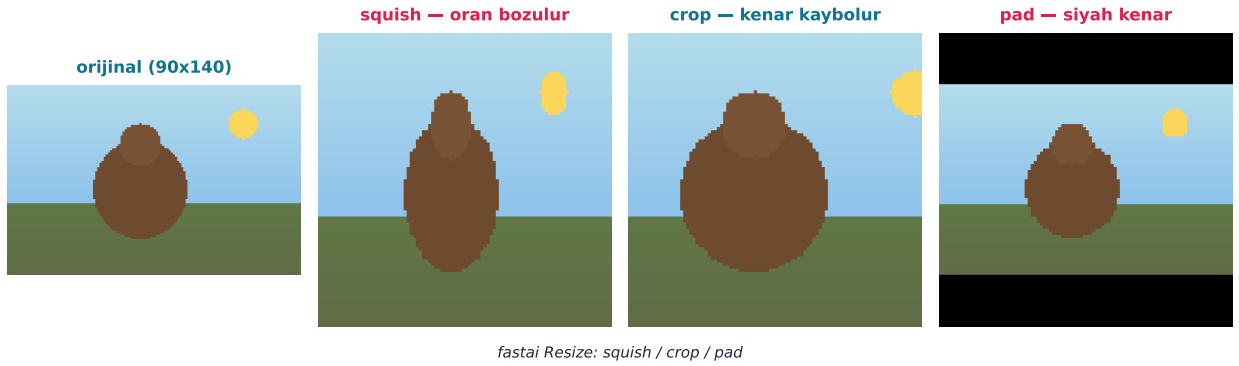
9.6 Yeniden Boyutlandırma Yöntemleri

Çoğu görü modeli tüm girdilerin aynı boyutta olmasını ister. fastai üç yol sunar: **squish** (sıkıştır — en-boy oranı bozulur, bütün görünür), **crop** (ortadan kırp — oran korunur, kenarlar kaybolur), **pad** (kenarlara siyah ekle — hem oran hem bütünlük korunur).

```
bears = bears.new(item_tfms=Resize(128, ResizeMethod.Squish)) # sıkıştır
bears = bears.new(item_tfms=Resize(128, ResizeMethod.Pad, pad_mode='zeros')) # doldur
bears = bears.new(item_tfms=RandomResizedCrop(128, min_scale=0.3)) # rastgele kirp
```

Şekil 9.3 bu üç yöntemi aynı sentetik foto üstünde yan yana gösterir — squish ezer, crop kenarı atar, pad siyah kenar ekler.

“Different situations result in different quality models; you can try them all, it doesn’t normally make too big a difference, so I wouldn’t worry about it too much.” — Howard, 14:37



Şekil 9.3: Üç yeniden boyutlandırma yöntemi (gerçek dönüşüm — sentetik foto): squish oran bozar (her şey görünür ama ezilir), crop kenarı atar (oran korunur ama bilgi kaybolur), pad siyah kenar ekler (oran ve bilgi korunur).

💡 Builder Notu — Squish/Crop/Pad Tercihi

- **Geriye (Ders 1):** Ders 1’de `Resize(192, method='squish')` kullanmıştık; burada üç seçeneğin sonuçlara etkisini görüyoruz.
- **İleriye:** Howard’a göre fark çoğu zaman küçüktür (14:37); pratikte üçünü de deneyip [confusion matrix](#) ile karşılaştırmak en sağlam yoldur.

9.7 Veri Artırımı (Data Augmentation)

En ilginç yöntem **RandomResizedCrop**’tur: her epoch’ta görüntünün farklı bir parçası seçilir. Bu, aynı görüntüden her seferinde biraz farklı bir kopya üretme fikrine, yani **veri artırımı**na (data augmentation) açılır — döndürme, eğme, renk kaydırma. `aug_transforms()` bunları otomatik uygular.

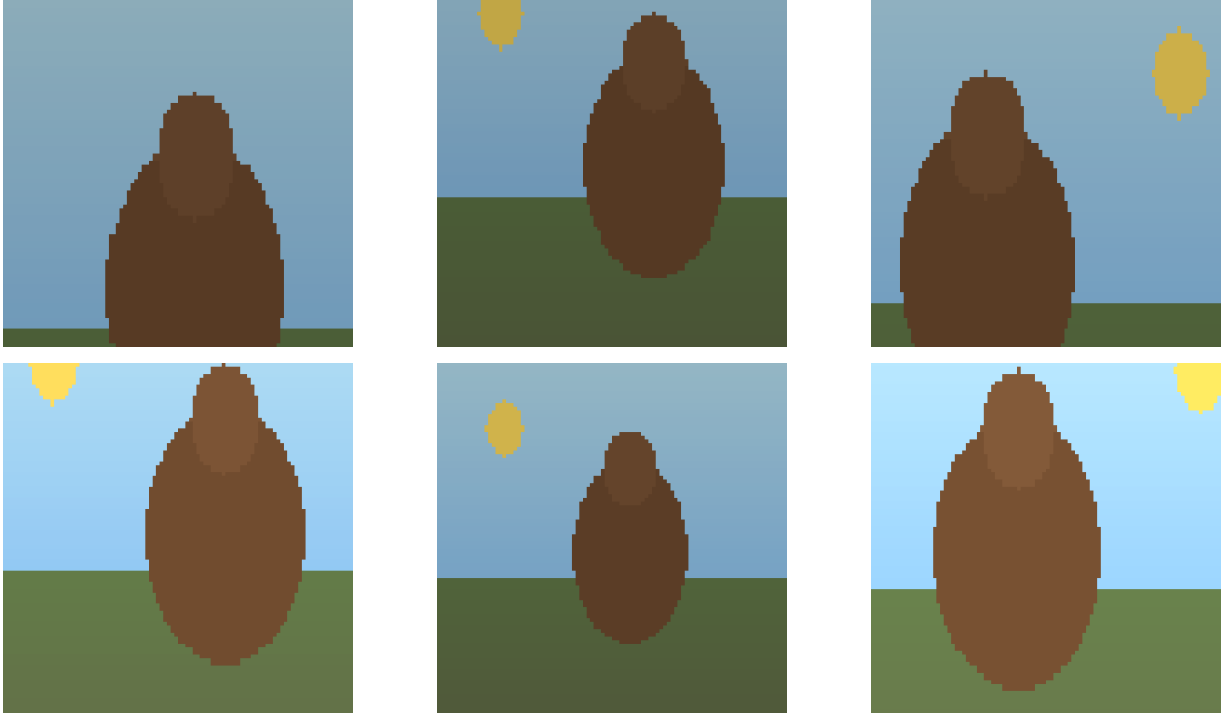
“This idea of getting different pictures each time from the same image is called data augmentation.”
— Howard, 15:53

Önemli nüans: kopyalar diske kaydedilmez; her epoch’ta görüntü **bellekte** gerçek zamanlı dönüştürülür. Şekil 9.4 aynı fotodan üretilen altı farklı kopyayı gösterir.

“There’s no copies being stored on your computer, but effectively it’s almost like there’s infinitely slightly different copies, because that’s what the model ends up sees.” — Howard, 17:38

```
bears = bears.new(  
    item_tfms=RandomResizedCrop(224, min_scale=0.5),  
    batch_tfms=aug_transforms()  
)  
dls = bears.dataloaders(path)
```

Aynı fotodan 6 farklı kopya — data augmentation



her epoch'ta bellekte üretilir, diske yazılmaz

Şekil 9.4: Veri artırımı (gerçek dönüşüm): RandomResizedCrop + çevirme + parlaklık ile aynı fotodan her epoch farklı bir kopya. Kopyalar diske yazılmaz, bellekte üretilir.

💡 Builder Notu — Bellekte Çoğalan Veri

- **Geriye (Stat 110):** Augmentation, modelin gördüğü dağılımı genişletip aşırı öğrenmeyi azaltır — örneklem çeşitliliği fikri.
- **İleriye:** Beş-on epoch'tan fazla eğiteceksen RandomResizedCrop + aug_transforms neredeyse her zaman işe yarar; production görüşü modellerinin standart parçasıdır.

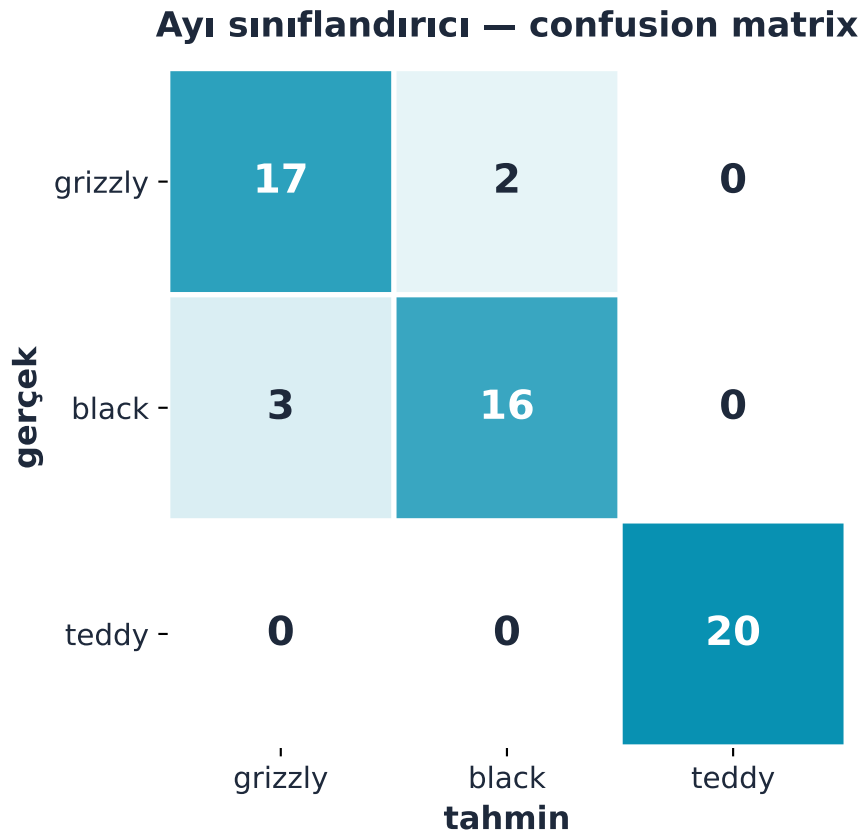
9.8 Confusion Matrix (Karışıklık Matrisi)

Model eğitildikten sonra ClassificationInterpretation ile hatalarını inceleriz. **Confusion matrix**, hangi kategorinin hangisiyle karıştırıldığını gösterir — yalnızca kategorik etiketlerde anlamlıdır.

```
interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()
```

Howard’ın ayı örneğinde model grizzly ile black bear’ı birbirine karıştırır ama teddy’yi hiç şaşırılmaz — mantıklı, çünkü teddy ikisinden de çok farklı görünür. Şekil 9.5 bu örüntüyü gösterir: çapraz hücreler dolu, teddy köşegeni temiz.

“What it says is, what category errors are you making? There were two times when it was actually a grizzly bear and it thought it was a black bear.” — Howard, 19:04



Şekil 9.5: Confusion matrix (illustratif sayılar — Howard ayı örneği): grizzly ile black bear karışır (çapraz hücreler 2 ve 3), teddy hiç şaşmaz (köşegen temiz). Satır=gerçek, sütun=tahmin.

💡 Builder Notu — Hata Analizinin İlk Adımı

- **Geriye (Stat 110):** Confusion matrix, sınıflandırma değerlendirmesinin temelidir; precision/recall buradan türer.
- **İleriye:** Hangi sınıfların karıştığını bilmek, veri toplama önceliğini belirler (production’da hata analizinin ilk adımı).

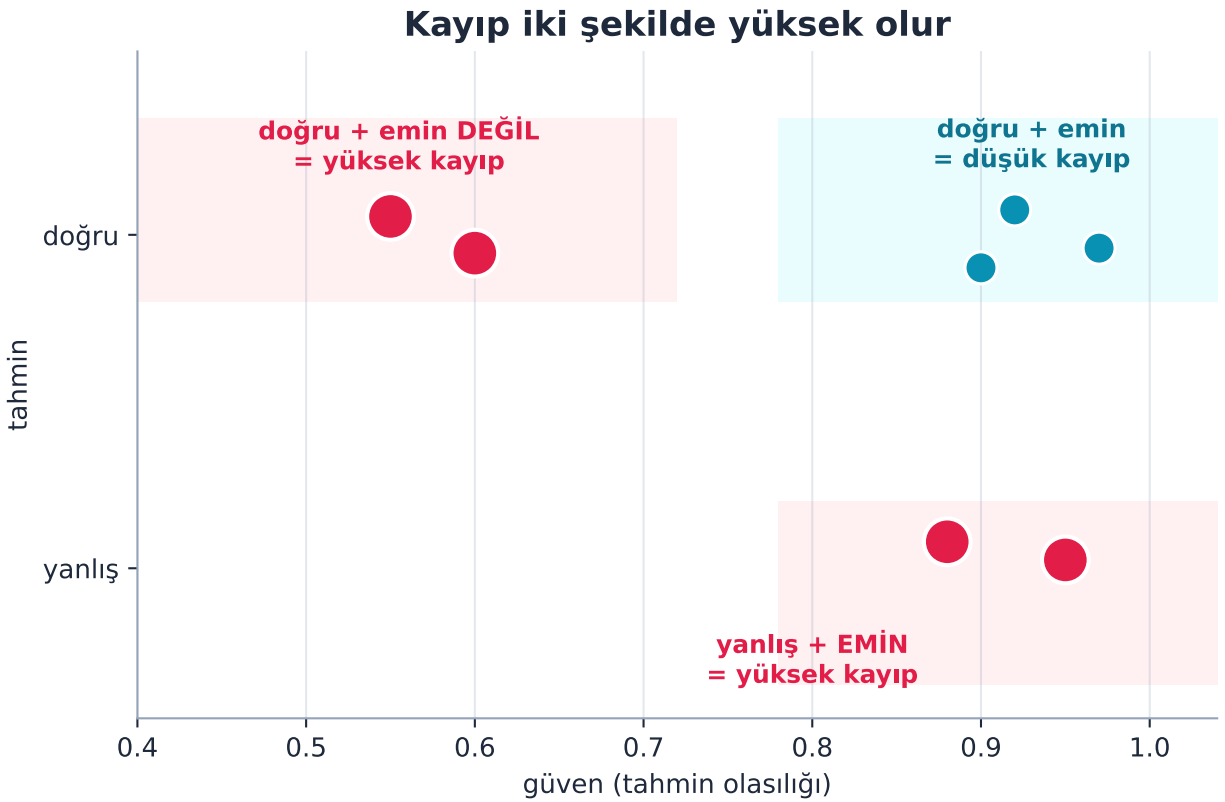
9.9 plot_top_losses

plot_top_losses, kaybın (loss) en yüksek olduğu örnekleri gösterir. Howard kritik bir ayrım yapar: kötü kayıp iki şekilde olur — ya **yanlış ve emin**, ya **doğru ama emin değil**.

“You can have a bad loss either by being wrong and confident, or being right and unconfident.”
— Howard, 22:02

```
interp.plot_top_losses(5, n_rows=1)
```

Bu, hangi örneklerin “zor” veya “şüpheli” olduğunu hızla ortaya çıkarır. Şekil 9.6 bu iki yüksek-kayıp durumunu güven ekseninde konumlandırır.



Şekil 9.6: plot_top_losses sezgisi (şematik): kayıp ya ‘yanlış + emin’ (sağ-alt) ya da ‘doğru + emin değil’ (düşük güven, sol-üst) olduğunda yüksektir. En şüpheli örnekler bunlardır — model nerede ve ne kadar emin yanlışlığını burada gösterir.

💡 Builder Notu — Emin ve Yanlış = Kalibrasyon Sorunu

- **Geriye (Ders 1):** Loss, Ders 1’deki “modelin ne kadar iyi olduğunun ölçüsü”dür; yüksek loss = ya hata ya düşük güven.
- **İleriye (calibration):** “Emin ve yanlış” örnekler, modelin kalibrasyon sorunlarını ve etiket hatalarını işaret eder.

9.10 ImageClassifierCleaner

Şimdi **önce eğit, sonra temizle** mantığı meyvesini verir. **ImageClassifierCleaner** modeli çalıştırır (bu yüzden ona `learn` veririz) ve her sınıfın örneklerini **kayba göre sıralı** (`plot_top_losses` mantığıyla) gösterir — en şüpheliler en üstte. Yanlış etiketlileri yeniden etiketler veya sileriz. Bu, Şekil 9.2 döngüsündeki “temizle” adımının somut aracıdır.

“Now we can use the fastai ImageClassifierCleaner to clean up the ones that are wrongly labeled. It actually runs our model, that’s why we pass it learn. And here’s what’s really important: they’re ordered by loss.” — Howard, 22:13

```
cleaner = ImageClassifierCleaner(learn)
cleaner
```

💡 Builder Notu — Veri Denetçisi Olarak Model

- **İleriye (veri-merkezli ML):** Modeli “etiket denetçisi” olarak kullanmak, büyük veri setlerini elle taramaya kıyasla devasa zaman kazandırır.
- **Pratik:** Temizledikten sonra modeli yeniden eğitirsin; döngü (eğit → temizle → eğit) doğruluğu artırır.

9.11 CPU RAM vs GPU RAM (VRAM)

Howard pratik bir uyarı yapar: GPU belleği (VRAM) CPU belleğinden farklı çalışır. CPU RAM dolduğunda işletim sistemi fazlasını diske yazabilir (swap); **GPU bunu yapamaz**. VRAM dolarsa “out of memory” hatası alırsın — çözüm genellikle Jupyter kernel’ini yeniden başlatmaktır.

“If you use up your GPU RAM, GPUs can’t swap that RAM onto the hard disk to use it later.” — Howard, 26:00

💡 Builder Notu — VRAM Gerçeği

- **Geriye (Ders 1):** Ders 1’de veriyi GPU’ya batch (mini-batch) hâlinde besliyorduk; işte o batch’ler VRAM’de tutulur — bu not, oradaki batch kavramının donanım tarafıdır.
- **İleriye (production):** Batch boyutu doğrudan VRAM tüketimini belirler; OOM hatası, batch boyutunu küçültmenin veya gradient accumulation’ın (Part 2 Ders 9 / Karpathy) gerekçesidir.

9.12 Production’a Alma: Hugging Face Spaces + Gradio

Modeli internette yayınlamak için Howard **Hugging Face Spaces** (ücretsiz barındırma) + **Gradio** (ücretsiz arayüz kütüphanesi) kullanır. Bir Space oluşturulur, Gradio uygulaması yazılır ve git ile yüklenir. Howard önce minimal bir “hello name” Gradio arayüzüyle mekaniği gösterir.

“Let’s go create a new space. We’re going to use Gradio, also free.” — Howard, 29:33

Süreç git tabanlıdır (GitHub Desktop veya terminal); Windows kullanıcıları için WSL/PowerShell notları da verilir.

💡 Builder Notu — En Hızlı Deploy Yolu

- **İleriye (deploy seçenekleri):** HF Spaces + Gradio, bir ML demosunu dünyaya açmanın en hızlı yoludur; alternatifler Streamlit, FastAPI, AWS/GCP container’larıdır.
- **Geriye:** Git, üretim akışının zorunlu aracı — coding-standards’taki branch/commit disiplini buraya da uygulanır.

9.13 learn.export — Modeli Dondurmak

Asıl köprü `learn.export`tır: eğitilmiş Learner’ı tek bir dosyaya (`model.pkl`) yazar. Bu, **GPU gerektiren tek adımdır**; geri kalan her şey (tahmin) CPU’da çalışır ([load_learner + predict bölümü](#)). Şekil 9.7 bu ayrımı gösterir — `model.pkl` GPU bölgesiyle CPU bölgesini köprüleyen tek artefaktır.

```
# Eğitim (Kaggle/Colab, GPU): deploy edilebilir bir model üret
path = untar_data(URLs.PETS)/'images'
def is_cat(x): return x[0].isupper()
dls = ImageDataLoaders.from_name_func('.',
    get_image_files(path), valid_pct=0.2, seed=42,
    label_func=is_cat, item_tfms=Resize(192))
learn = vision_learner(dls, resnet18, metrics=error_rate)
learn.fine_tune(3)
learn.export('model.pkl') # modeli tek dosyaya dondur
```

“I go `learn.export` and I give it a name. That’s going to create a file containing our trained model. And that’s the only thing you need a GPU for.” — Howard, 38:56

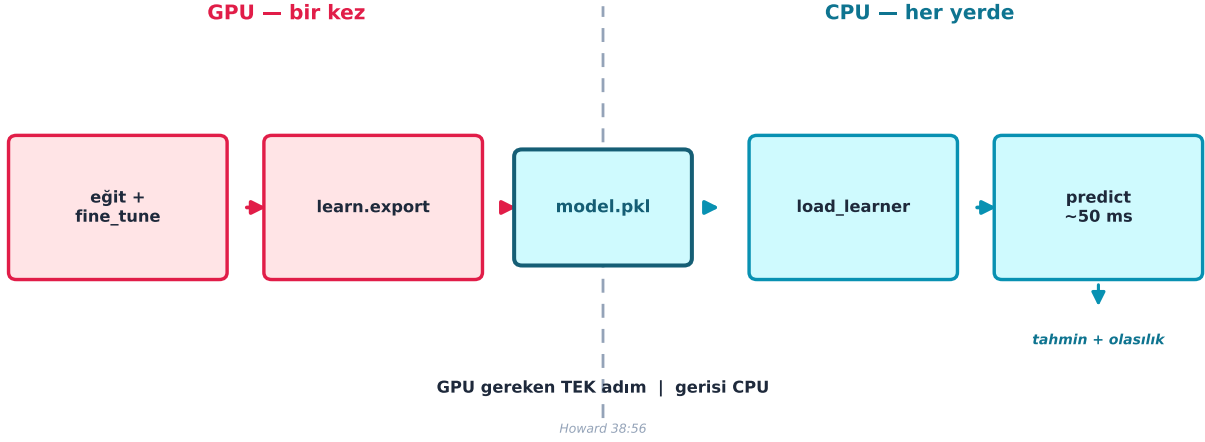
💡 Builder Notu — pickle = Taşınabilir Model Artefaktı

- **Geriye (Ders 1):** `ImageDataLoaders.from_name_func`, `DataBlock`’un kısa yoludur; etiket buradaki `is_cat` fonksiyonundan gelir.
- **İleriye:** `model.pkl` bir **pickle** dosyasıdır — eğitim ortamından bağımsız, taşınabilir bir model artefaktı; production’da versiyonlanır.

9.14 load_learner + predict

Dondurulmuş modeli kullanmak için onu eğitmeyiz, **yükleriz**: `load_learner` kaydedilen dosyayı açar ve eğitimin sonundakiyle birebir aynı Learner’ı geri verir.

“Instead of training a learner, we use `load_learner`, we pass in the file name that we saved, and that returns a learner. This learner is exactly the same as the learner you get when you finish training.” — Howard, 42:45



Şekil 9.7: Deploy ayrımı (şematik): `learn.export` GPU gereken TEK adımdır (`model.pkl` üretimi); `load_learner + predict` CPU’da ~50 ms çalışır. Pahalı GPU yalnızca eğitim için — `model.pkl` iki bölgeyi köprüler (Howard 38:56).

Howard bunu “zamanda dondurma” diye anlatır: `pk1` bir pickle (donmuş Python nesnesi) dosyasıdır. Tahmin, CPU’da bile anlıktır — Şekil 9.7 sağ tarafındaki CPU bölgesi (`load_learner → predict ~50 ms`) tam bu adımı betimler.

“We kind of froze it in time, something called a pickle file. It’s like a frozen object. We saved it to disk, transferred it to our computer, and we’ve now loaded it.” — Howard, 43:07

```
import gradio as gr
from fastai.vision.all import *

learn = load_learner('model.pkl')
# Etiketleyici fonksiyon (is_cat) tanımı burada da bulunmalı:
def is_cat(x): return x[0].isupper()

im = PILImage.create('dog.jpg')
pred, idx, probs = learn.predict(im) # ~54 ms, CPU'da bile anlık
# Cikti yapisi (ornek/illustratif degerler – is_cat vocab [False, True] sirali):
# pred = 'False' (string), idx = tensor(0), probs = tensor([P(dog), P(cat)])
# dog.jpg icin P(dog) yuksek cikar (or. ~0.9998 / ~0.0002)
```

“It took 54 milliseconds to figure out that this is not a cat. It’s returning: is it a cat as a string, is it a cat as a zero or one, and the probability that it’s a dog and a cat.” — Howard, 44:02



Builder Notu — Donmuş Model Tuzağı (`is_cat`)

- **Önemli tuzak:** `model.pkl` etiketleyici fonksiyona (`is_cat`) referans verir ama kaynak kodunu içermez — bu yüzden `is_cat` tanımı tahmin ortamında da bulunmalıdır; aynı kural [Gradio arayüzünün](#) `classify_image`’i için de geçerlidir.
- **İleriye:** ~50 ms çıkartım, gerçek zamanlı bir web servisi için fazlasıyla hızlı; CPU yeterli (deploy

maliyetini düşürür).

9.15 Gradio Arayüzü (app.py)

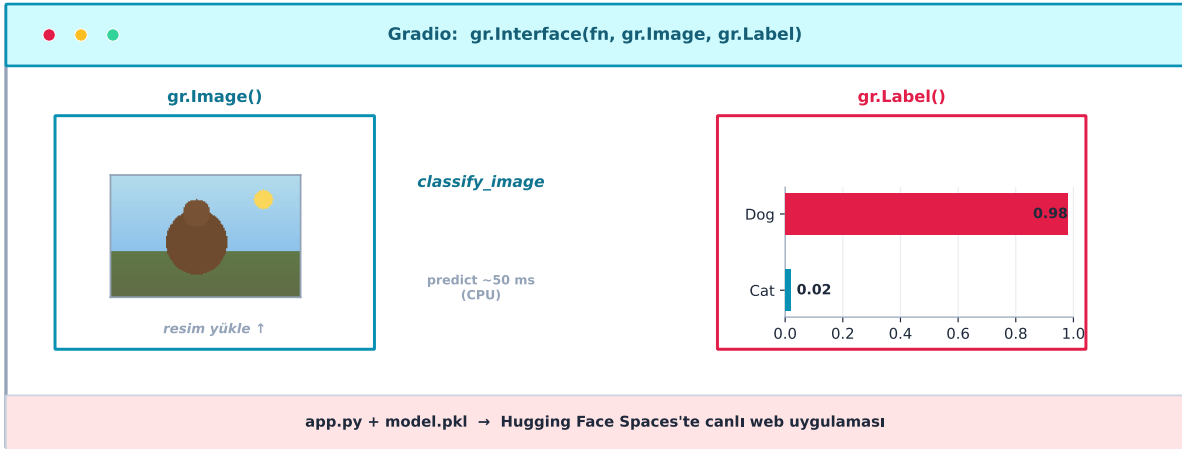
Gradio, tahmin fonksiyonunu bir web arayüzüne sarar. Sınıf adları tanımlanır, `predict` çıktısı bir olasılık sözlüğüne çevrilir ve `gr.Interface` görüntü girişi + etiket çıkışı kurar.

```
categories = ('Dog', 'Cat')

def classify_image(img):
    pred, idx, probs = learn.predict(img)
    return dict(zip(categories, map(float, probs)))

intf = gr.Interface(fn=classify_image, inputs=gr.Image(), outputs=gr.Label())
intf.launch()
```

`gr.Label`, olasılık sözlüğünü çubuk grafiği olarak gösterir. Şekil 9.8 bu arayüzü somutlaştırır: `gr.Image` girdisi → `classify_image` → `gr.Label` olasılık çubukları. Bu dosya Hugging Face Space'e yüklendiğinde model canlı bir web uygulaması olur.



Şekil 9.8: Gradio arayüzü (şematik — Gradio UI): `gr.Interface` tahmin fonksiyonunu web arayüzüne sarar — `gr.Image` girdisi, `classify_image`, `gr.Label` olasılık çubukları. `app.py + model.pkl` Hugging Face Spaces'e yüklenince canlı uygulama olur.

💡 Builder Notu — Gradio = REST API'nin İnce Katmanı

- **Geriye:** `classify_image`, `load_learner + predict` bölümündeki `learn.predict` çağrısını saran ince bir kabuktur — Gradio yeni bir tahmin mantığı eklemez, var olanı sarar.
- **İleriye:** Aynı `classify_image` fonksiyonu bir REST API'nin de çekirdeğidir; Gradio yalnızca onu görselleştiren ince bir katmandır.

9.16 #|export ile Notebook'tan Script

Howard pratik bir numara gösterir: deney yaptığı notebook'tan otomatik olarak `app.py` üretmek. `nbdev`'in `#|export` işareti, hangi hücrelerin script'e gireceğini belirler; bir `export` komutu bu hücreleri toplayıp `app.py` yazar.

“That creates a file for me called `app.py` containing that script. So this is a nice easy way.” — Howard, 48:25

```
#|export
categories = ('Dog', 'Cat')
def classify_image(img):
    pred, idx, probs = learn.predict(img)
    return dict(zip(categories, map(float, probs)))
# Son hücre: isaretili hücreleri app.py'ye yaz
from nbdev.export import nb_export
nb_export('app.ipynb', '.')
```

 Builder Notu — Notebook = Hem Deney Hem Kaynak

- **Geriye (Ders 1):** Ders 1'de notebook'u (Jupyter/Kaggle) deney ortamı olarak kullanmıştık; `#|export` aynı notebook'u `app.py` kaynağına dönüştürerek deney ile production'ı tek dosyada birleştirir.
- **İleriye (reproducibility):** “Notebook = hem deney hem kaynak” yaklaşımı, deneyle production kodunu senkron tutar; `nbdev` fast.ai ekosisteminin kendi geliştirme aracıdır.

9.17 Kaç Epoch Eğitmeli? + Kapanış

Topluluktan gelen yaygın bir soru: kaç epoch? Howard'ın pratik cevabı: ya zamanın/sabırın bitene kadar, ya da doğrulama hatası (error rate) artmaya başlayana kadar. Sabit bir sayı yoktur; küçük problemler için birkaç epoch, zor problemler için daha fazla.

“You train until the error rate starts getting worse, or until you run out of time.” — Howard, 53:14

Howard dersi, modeli Colab'da `export` edip indirme ve yerel ortam kurulumu (`mamba/conda`) notlarıyla kapatır. Asıl mesaj sabit: artık tam bir uçtan uca hattın var — `eğit`, `temizle`, `export`, `deploy`.

 Builder Notu — Erken Durdurma Sezgisi

- **Geriye (Ders 1):** “Doğrulama hatası artmaya başlayınca dur” kuralı, Ders 1'deki validation set + overfitting fikrinin doğrudan uygulamasıdır — `train` düşerken validation yükselmeye başladığı an, durulacak noktadır.
- **İleriye:** “Error rate kötüleşene kadar `eğit`” = erken durdurma (early stopping) sezgisi; Part 2'de (ve Stat 110 overfitting) bunu derinleştireceğiz.

9.18 Bu Dersin Özeti

1. Bir model ancak deploy edildiğinde değer üretir; **deploy üretimin asıl zor kısmıdır**.
2. fast.ai’de ters sezgi: **önce modeli eğit, sonra veriyi temizle** — model şüpheli örnekleri senin için bulur ([Ters Sezgi](#)).
3. **Veri artırımı** (RandomResizedCrop + aug_transforms) aynı görüntüden her epoch’ta farklı kopya üretip aşırı öğrenmeyi azaltır; kopyalar diske yazılmaz, bellekte üretilir ([Veri Artırımı](#)).
4. **Confusion matrix** hangi sınıfların karıştığını, **plot_top_losses** en şüpheli örnekleri (yanlış+emin veya doğru+emin değil) gösterir ([Confusion matrix](#), [plot_top_losses](#)).
5. **ImageClassifierCleaner** modeli kullanarak yanlış etiketli/bozuk örnekleri kayba göre sıralı temizler ([Cleaner](#)).
6. **GPU RAM (VRAM)** swap yapamaz; dolarsa kernel yeniden başlatılır.
7. **learn.export** modeli tek pickle dosyasına dondurur (GPU gereken tek adım); **load_learner** onu herhangi bir yerde geri yükler.
8. **predict** CPU’da bile ~50 ms’de tahmin döndürür (string + indeks + olasılıklar); **Gradio + Hugging Face Spaces** ile canlı web uygulamasına dönüştür.

! Tek Bir Cümle

Eğitmek işin yarısıdır; bir modeli dondurup (export), her yerde yükleyip (load_learner) milisaniyelerde tahmin ettiren ve Gradio + HF Spaces ile dünyaya açan **deploy zinciri**, derin öğrenmeyi gerçek bir ürüne dönüştürür.

9.19 Kontrol Soruları

i Soru 1: learn.export ne yapar ve neden “GPU gerektiren tek adım” odur?

Cevap:

learn.export('model.pkl') eğitilmiş Learner’ı (mimari + ağırlıklar + ön işleme) tek bir pickle dosyasına yazar. Eğitim (gradient hesabı, fine_tune) GPU ister; ama dosya oluşturma ve sonraki tahmin (predict) yalnızca ileri geçirir ve CPU’da hızlıca çalışır. Yani GPU’yu sadece modeli üretmek için kiralarsın; deploy edilen servis ucuz CPU’da koşar.

i Soru 2: “Önce eğit, sonra temizle” neden mantıklıdır? ImageClassifierCleaner bunu nasıl mümkün kılar?

Cevap:

Eğitilmiş model, hangi örneklerin “zor” olduğunu kayıpla ölçer. Yanlış etiketli veya bozuk görseller genellikle en yüksek kayba sahiptir. ImageClassifierCleaner(learn) modeli çalıştırıp her sınıfın örneklerini kayba göre sıralar — en şüpheliler en üstte. Böylece binlerce görseli elle taramak yerine, yalnızca üstteki birkaç şüpheliyi yeniden etiketler veya silersin. Model, veri kalitesinin denetçisi olur.

i Soru 3: `plot_top_losses`'ta bir örnek “doğru tahmin edilmiş” olduğu hâlde neden yüksek kayba sahip olabilir? Ayrıca `load_learner` sonrası `is_cat` tanımı neden gerekir?

Cevap:

Kayıp iki şekilde yüksek olur: (a) yanlış ve emin, (b) doğru ama emin değil. Model bir teddy'yi %66 güvenle doğru tahmin etse bile, düşük güven yüksek kayıp demektir — bu yüzden “doğru” örnek listede çıkar. `is_cat`'e gelince: `model.pkl` etiketleme fonksiyonuna referans tutar ama kaynak kodunu saklamaz; bu yüzden modeli yüklediğin ortamda `is_cat` tanımı yeniden bulunmalıdır, yoksa yükleme başarısız olur.

i Soru 4: Howard neden “deploy, eğitimden daha zor” der? CPU çıkarımın ~50 ms olması ne anlama gelir?

Cevap:

Eğitim bir kez olur ve net bir bitişi vardır; deploy ise sürekli çalışmalı, hataları kaldırmalı, güncellenmeli ve gerçek kullanıcılara hizmet vermelidir (servis, arayüz, izleme, sürüm). Asıl mühendislik buradadır. ~50 ms CPU çıkarımı ise pahalı GPU'ya gerek olmadığını gösterir: bir web servisi saniyede onlarca isteği ucuz donanımda karşılayabilir — deploy maliyetini ve karmaşıklığını ciddi biçimde düşürür.

9.20 Egzersizler

Egzersiz 1 (Direkt uygulama). Kendi iki sınıflı modelini (Ders 1'den) eğit ve `learn.export('model.pkl')` ile kaydet. Dosya boyutuna bak.

Egzersiz 2 (Temizleme döngüsü). `ImageClassifierCleaner` ile veri setini temizle, modeli yeniden eğit ve `error_rate`'i önce/sonra karşılaştır.

Egzersiz 3 (Resizing kıyası). Aynı veride `Resize(squish)`, `Resize(pad)` ve `RandomResizedCrop` ile üç model eğit; `plot_confusion_matrix` ile sonuçları kıyasla.

Egzersiz 4 (Gradio). `load_learner + classify_image + gr.Interface` ile yerel bir Gradio uygulaması yaz ve `launch()` ile çalıştır.

Egzersiz 5 (Deploy — dersin ana hedefi). Bir Hugging Face Space oluştur, `model.pkl + app.py` yükle ve modelini canlı web uygulaması olarak yayınla.

9.21 Sonraki Ders İçin Hazırlık

Ders 3: Sinir Ağı Temelleri (Neural net foundations)

Ders 1-2'de `vision_learner/fine_tune`'u kara kutu olarak kullandık. Ders 3 perdeyi aralar: bir sinir ağı **gerçekte nasıl çalışır**? Gradient descent, parametre uydurma, ReLU ve sıfırdan MNIST.

Ana konular:

- Bir fonksiyonu veriye uydurma (gradient descent)
- Türev ve gradient'in rolü

- ReLU ile esnek fonksiyonlar kurma
- Matrix multiplication ve MNIST

⚠ Ders 3 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 5 — bir model deploy et).
- Kendi modelini Hugging Face Spaces'te canlıya al.
- Ana cümleyi tekrar oku: “Önce eğit, sonra temizle; sonra dondur ve deploy et.”

9.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Deploy	Modeli kullanılabilir bir servise dönüştürme; asıl zor kısım	6:49
Önce eğit, sonra temizle	Eğitilmiş modelle şüpheli veriyi bulma (ters sezgi)	12:47
Data augmentation	Aynı görüntüden her epoch'ta farklı kopya üretme	15:53
RandomResizedCrop	Her seferinde görüntünün farklı parçasını kırpma	14:55
aug_transforms	Döndürme/eğme/rek dönüşümleri paketi	15:53
Confusion matrix	Hangi sınıfların birbirine karıştığını gösterir	19:04
plot_top_losses	En yüksek kayıplı (şüpheli) örnekler	22:02
ImageClassifierCleaner	Modelle veri temizleme aracı (kayba göre sıralı)	22:13
learn.export	Modeli tek.pkl dosyasına dondurma (GPU'lu tek adım)	38:56
load_learner	Dondurulmuş modeli geri yükleme	42:45
predict	CPU'da ~50 ms tahmin (string + indeks + olasılık)	43:30
Gradio + HF Spaces	Ücretsiz arayüz + barındırma ile canlı web uygulaması	27:20

9.23 ML Builder Bağlantıları

💡 Builder Notu — 6 ML Köprüsü

Bu ders, derin öğrenmeyi notebook'tan gerçek bir ürüne taşıyan deploy hattıdır — köprülerin özeti:

1. **Deploy = MLOps girişi** → model notebook'tan çıkıp servise dönüşür; production'da asıl iş budur.
2. **Veri-merkezli ML** → modeli etiket denetçisi olarak kullanmak (ImageClassifierCleaner), büyük veri setlerinde standart tekniktir.
3. **Data augmentation** → aşırı öğrenmeyi azaltır (Stat 110 örneklem çeşitliliği); tüm görü pipeline'larının parçası.
4. **pickle artefaktı** → taşınabilir, versiyonlanabilir model dosyası; üretimde model registry'nin temeli.
5. **CPU çıkarım** → ~50 ms; pahalı GPU olmadan ölçeklenebilir servis (deploy maliyeti düşer).
6. **Gradio/HF Spaces** → demo-to-production en hızlı yol; Part 2'de Stable Diffusion demoları da aynı altyapıyı kullanır.

! Bu dersten tek bir şey alıp gideceksen

Bu dersten tek bir şey alıp gideceksen: bir model ancak başkalarının kullanabildiği an gerçek olur. Önce eğit-sonra-temizle ile veriyi sağlamlaştırsın, export ile dondur, load_learner + predict ile her yerde çalıştır, Gradio + Hugging Face Spaces ile dünyaya aç. Eğitim laboratuvarıdır; **deploy üretilimdir**.

10 Sinir Ağı Temelleri — Derin Öğrenmeyi Türetmek (Neural net foundations)

Bir model = parametrelili bir fonksiyondur: önce quadratic $ax^2 + bx + c$ 'yi gürültülü veriye uydur, loss'u (MAE/MSE) ölç, türev/gradient ile yönü bul, gradient descent ($p \leftarrow p - \eta \nabla L$) ile parametreleri otomatik ayarla ve yeterince ReLU toplayarak derin öğrenmeyi sıfırdan türet

i Bölüm bilgisi

- **Howard'ın videosu:** course.fast.ai — Lesson 3: Neural net foundations (~90 dk)
- **Seri:** Practical Deep Learning for Coders — Part 1, Ders 3
- **Hoca:** Jeremy Howard
- **Playlist:** [Practical Deep Learning Part 1 \(2022\)](#)
- **Notebook:** [course22 — 04-how-does-a-neural-net-really-work](#)
- **Okuma süresi:** ~35 dk

10.1 Bu Derste Ne Var?

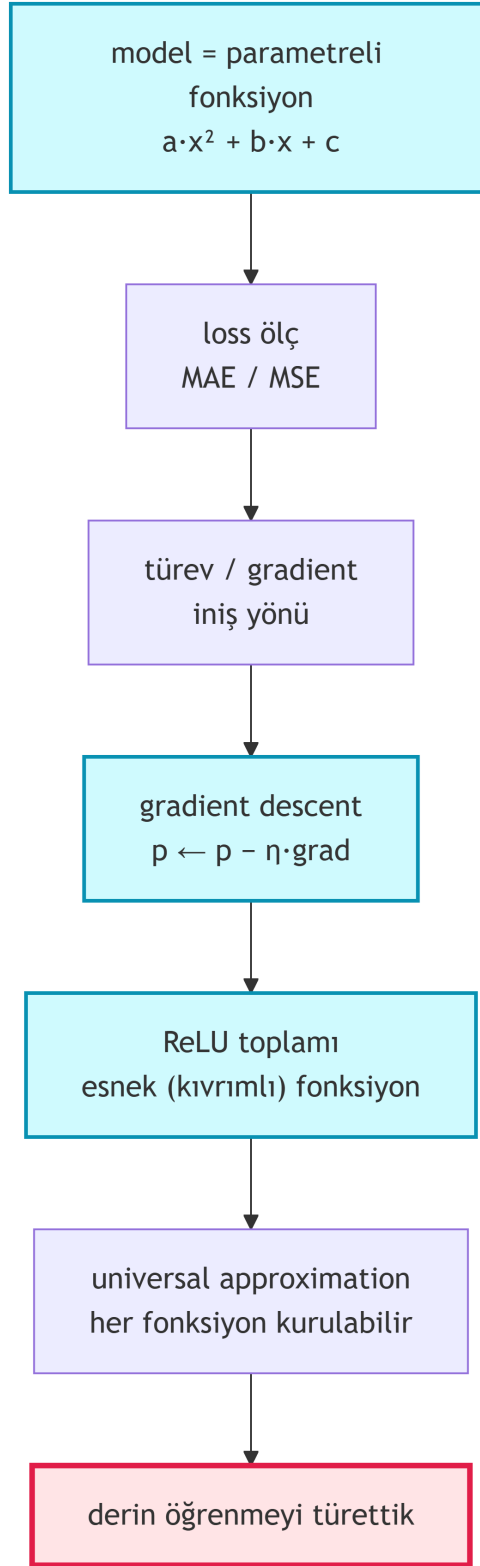
Ders 1-2'de vision_learner ve fine_tune'u bir kara kutu olarak kullandık. Ders 3 perdeyi aralar: bir sinir ağı **gerçekte nasıl çalışır?** Howard önce pratik tarafı (timmm ile daha iyi modeller, bulut ortamları) gösterir, sonra en aşağıya iner — bir modelin aslında **gradient descent ile parametreleri uydurulmuş bir fonksiyon** olduğunu sıfırdan kurar.

Üç temel fikir:

1. **Model = parametrelili fonksiyon** — basit bir quadratic $f(x) = ax^2 + bx + c$ bile bir modeldir; “öğrenmek” doğru a, b, c 'yi bulmaktır ([Model = Parametrelili Fonksiyon](#)).
2. **Gradient descent** — kayıp (loss) fonksiyonunun eğimini (gradient) hesaplayıp parametreleri eğimin tersine küçük adımlarla, $p \leftarrow p - \eta \nabla L$, güncellemek ([Gradient Descent](#)).
3. **ReLU toplamı = her şey** — yeterince rectified linear unit toplanırsa istenen kadar karmaşık herhangi bir fonksiyon kurulabilir; parametreleri gradient descent verir. “İşte derin öğrenmeyi türettik.” ([Sonsuz Karmaşık Fonksiyon → “Deep Learning”i Türettik](#)).

“We have just derived deep learning. Everything from now on is tweaks to make it faster and make it need less data.” — Howard, 47:59

Şekil 34.1 bu üç fikri, parametrelili fonksiyondan “derin öğrenmeyi türettik” sonucuna uzanan tek bir zincirde birleştirir.



Şekil 10.1: Ders 3’ün kavram haritası: model = parametrelili fonksiyon → loss (MAE/MSE) ile iyiliği ölç → türev/gradient yönü verir → gradient descent ($p \leftarrow p - \eta \cdot \text{grad}$) parametreleri uydurur → ReLU toplamı esnek fonksiyon kurar → universal approximation → ‘derin öğrenmeyi türettik’.

💡 Builder Notu — Bir Kara Kutuyu Türevden Türetmek

- **Geriye (Ders 1-2):** Ders 1-2’de `vision_learner + fine_tune` “büyü” gibiydi; bu ders o büyü’nün altındaki tek mekanizmayı — gradient descent ile parametre uydurma — elle kurar.
- **Geriye (Karpathy / Calculus):** Bu ders Karpathy’nin `micrograd`’ının kavramsal özüdür — `loss.backward()` + `param -= grad * lr`. Türev (Calculus) ve zincir kuralı tam burada işler.
- **İleriye:** “ReLU toplamı + gradient descent” universal approximation’dır; tüm modern mimariler bu çekirdeğin üstünde durur — “gerisi hız ve veri verimliliği ayarı”.
- **Tek cümle:** Bir model, gradient descent ile parametreleri uydurulan esnek bir fonksiyondur — gerisi hız ve verimlilik ayarıdır.

10.2 Bir fast.ai Dersi Nasıl Çalışılır?

Howard “Lesson 0” niteliğinde bir yöntem önerir: önce videoyu izle, sonra notebook’u aç, her hücreyi çalıştırmadan önce **çıkışının ne olacağını tahmin et**, sonra çalıştır ve doğrula. Kendi kendine çalışmanın (self-study) tuzaklarına da değinir; forum ve Discord topluluğunu önerir.

Howard ayrıca haftanın öğrenci işlerini gösterir (Marvel karakter tanıyıcı, müzik türü sınıflandırıcı, Power App uygulamaları) — Ders 1-2’nin birkaç satırı gerçek ürünlere dönüşüyor.

“Before you run each cell, try to think about what it’s going to show, to test your understanding.”
— Howard, 4:01

💡 Builder Notu — Çıktıyı Tahmin Et, Sonra Çalıştır

- **İleriye (aktif öğrenme):** “Çıktıyı tahmin et, sonra çalıştır” aktif öğrenmedir; pasif izlemeye kıyasla kavrayışı kat kat artırır — bu derste birazdan kuracağımız gradient descent döngüsünü adım adım kafanda canlandırmak tam bu disiplindir.
- **Geriye (Ders 1):** Ders 1’deki “her adımda veriye bak” alışkanlığının kardeşi; orada veriye, burada koda bakarsın.

10.3 Daha İyi Bir Pet Detektörü

Howard kedi/köpek değil, **cins** (breed) tanıyan daha zor bir model kurar. Doğruluğu artırmanın ilk yolu daha güçlü bir mimaridir: `resnet18` yerine `resnet34`. Kod Ders 1-2 ile birebir aynıdır; yalnızca mimari değişir.

“How do you make your neural network more accurate during the week?” — Howard, 8:15

💡 Builder Notu — Aynı Zincir, Tek Değişen Mimari

- **Geriye (Ders 1):** Aynı `DataBlock` → `vision_learner` → `fine_tune` zinciri; tek fark mimari adı (`resnet18` → `resnet34`).
- **İleriye (timm karşılaştırması):** Mimari seçimi doğruluk/hız dengesidir; bir sonraki bölüm bunu ölçen grafiği gösterir.

10.4 timm ile Model Karşılaştırması

fastai, Ross Wightman'ın **timm** (PyTorch Image Models) kütüphanesini entegre eder — dünyanın en büyük görü modeli koleksiyonu. Howard, modelleri **hız (saniye/örnek) vs doğruluk** ekseninde karşılaştıran bir grafik gösterir: sola doğru hızlı, yukarı doğru doğru. resnet'ler iyi bir başlangıç, convnext modelleri ise daha yeni ve güçlü.

```
import timm
timm.list_models('convnext*') # convnext ailesindeki modelleri listele
learn = vision_learner(dls, 'convnext_tiny', metrics=error_rate)
learn.fine_tune(3)
```

“On the x-axis we’ve got seconds per sample, so how fast is it; to the left is better, it’s faster.” — Howard, 15:08

💡 Builder Notu — Hız mı Doğruluk mu?

- **İleriye (production):** “Hız mı doğruluk mu?” kararı deploy bağlamına bağlıdır — mobil cihaz hız ister, batch işlem doğruluk; bu ayrımı Ders 2’deki [CPU çıkarım](#) tartışmasıyla birlikte düşün.
- **Pratik:** Önce küçük bir modelle (resnet18) hızlı denersin; mimari değiştirmek tek satırdır.

10.5 Bulut ve JupyterLab

Howard, kalıcı bir ortam isteyenler için Paperspace gibi bulut sağlayıcılarını ve **JupyterLab** arayüzünü tanıtır: git repo, terminal ve notebook’lar bir arada. Kaggle ücretsiz GPU verir ama kalıcı ortam sunmaz; Paperspace kalıcılık sağlar.

“Paperspace is another amazing platform; here you can see I’ve got my pets repo with a git repository.” — Howard, 9:19

💡 Builder Notu — Kalıcı Ortam = Reproducibility

- **İleriye (reproducibility):** Kalıcı ortam (kütüphaneler, veriler) gerçek projeler için önemlidir; tekrar üretilebilirlik ve hız kazandırır.
- **Geriye (Ders 1):** Ders 1’deki “notebook’un başına `!pip install -Uqq fastai`” disiplini, kalıcı ortamda bir kez kurulup unutulmuş bir adıma dönüşür.

10.6 model.pkl’in İçinde Ne Var?

Howard kritik soruyu sorar: bu “sihirli” `model.pkl` aslında nedir? İçine bakınca görülen şey, iç içe katmanlardan (Sequential, timm body) oluşan bir yapı — ve her katmanın **parametreleri**: büyük sayı yığınları.

“What did we just do, really? What is this magic model.pkl file? It’s an object.” — Howard, 20:49

Yani bir model = bir mimari (sayıların nasıl birleştirileceğinin tarifi) + parametreler (öğrenilen sayılar). Sıradaki bölümler bu “sayıları öğrenme” işinin en yalın hâlini kurar.

💡 Builder Notu — Mimari + Parametreler Ayrımı

- **Geriye (Ders 2):** Ders 2’de `model.pkl`’i deploy artefaktı olarak [dondurup taşımıştık](#); burada içini açıp “ne sakladığını” görüyoruz: mimari + öğrenilen sayılar.
- **İleriye (Model = Parametrelili Fonksiyon):** “Mimari sabit, parametreler eğitimle değişir” ayrımı tüm derin öğrenmenin temelidir; bir sonraki bölümde bu fikri 3 parametrelili bir quadratic’e indirgiyoruz.

10.7 Bir Model = Parametrelili Fonksiyon

Howard sinir ağı yerine çok daha basit bir örnek seçer: bir **quadratic** fonksiyon, $f(x) = ax^2 + bx + c$. Buradaki a, b, c parametrelerdir; onları değiştirerek fonksiyonun şeklini değiştiririz.

```
def quad(a, b, c, x): return a*x**2 + b*x + c

def mk_quad(a, b, c): return partial(quad, a, b, c)
f = mk_quad(3, 2, 1) # 3x^2 + 2x + 1 fonksiyonunu üretir
```

Gerçek veri gürültülüdür; Howard $f(x)$ ’e rastgele gürültü ekleyip “gerçek dünya verisi” taklit eder. Soru şudur: bu gürültülü veriye en iyi uyan a, b, c nedir? Şekil 10.2 üç farklı (a, b, c) seçiminin aynı gürültülü veriye nasıl uyduğunu gösterir — parametre değişince şekil değişir.

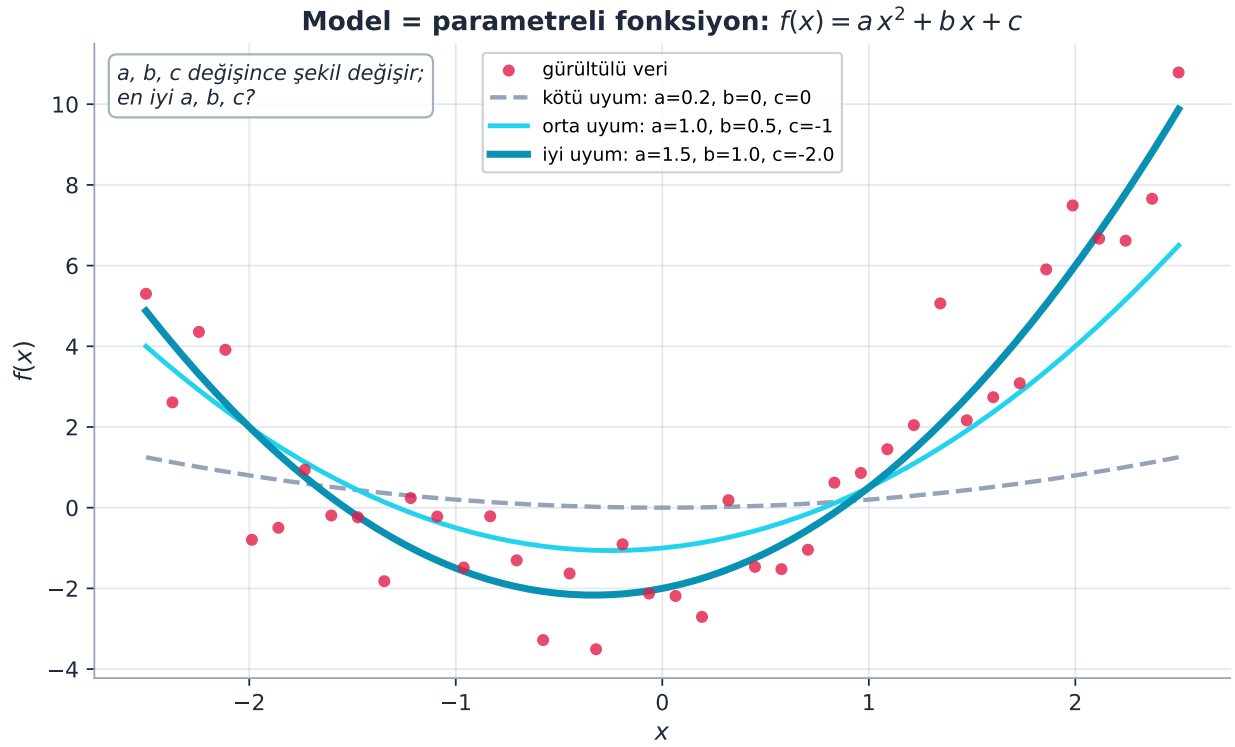
💡 Builder Notu — 3 Parametre, Sonra Milyonlar

- **Geriye (Stat 110):** Gürültü ekleme, normal dağılımdan örnekleme; veri = sinyal + gürültü — Howard’ın “gerçek dünya verisi” taklidi tam budur.
- **İleriye (Sonsuz Karmaşık Fonksiyon):** Quadratic’in 3 parametresi yerine bir sinir ağının milyonlarca parametresi olur; fikir aynıdır — yalnızca esneklik ve ölçek değişir.

10.8 Fonksiyonu Elle Uydurmak

Howard Jupyter’in `@interact` kaydırıcılarıyla a, b, c ’yi elle oynatıp veriye uydurmaya çalışır. Bir parametreyi artırıp azaltarak veriye yaklaşır — ama “iyi uyum”u gözle yargılamak öznel ve yavaştır.

```
@interact(a=1.1, b=1.1, c=1.1)
def plot_quad(a, b, c):
    plt.scatter(x, y)
    plot_function(mk_quad(a, b, c), ylim=(-3, 13))
```



Őekil 10.2: Model = parametrelili fonksiyon (gerçek hesaplama): $f(x) = a \cdot x^2 + b \cdot x + c$. a, b, c değıştikçe parabolün Őekli değışir; ‘öğrenmek’ gürültülü veriye en iyi uyan a, b, c’yi bulmaktır.

**Builder Notu — Gözle Ayarlama Ölçeklenemez**

- **İleriye (Loss → Gradient Descent):** “Gözle ayarlama” ölçeklenemez; bize iyiliği **sayıyla** ölçen bir araç (loss) ve onu otomatik iyileştiren bir yöntem (gradient descent) gerekir — sıradaki iki bölüm tam bunu kurar.
- **Geriye (Daha İyi Pet Detektörü):** Mimariyi tek satırla değiştirmek kolaydı; ama milyonlarca parametreyi elle oynatmak imkânsız — otomasyon şart.

10.9 Loss: Modelin İyiliğini Ölçmek

“İyi uyum”u nesnel kılmak için bir **loss fonksiyonu** gerekir: tahminlerle gerçek değerler arasındaki farkın bir özeti. Howard ortalama mutlak hatayı (MAE, mean absolute error) kullanır; karesel varyantı MSE’dir (mean squared error). Loss ne kadar küçükse model o kadar iyi.

```
def mae(preds, acts): return (torch.abs(preds - acts)).mean()
# MSE varyanti: ((preds - acts)**2).mean()
```

“That’s mean squared error.” — Howard, 32:07

Artık a, b, c ’yi öyle ayarlamak istiyoruz ki loss en küçük olsun. Bu loss değerinin tek bir parametreye göre nasıl bir “kâse” çizdiğini Şekil 10.3 gösterir.

**Builder Notu — Loss = Optimize Edilecek Hedef**

- **Geriye (Ders 1):** Loss, Ders 1’deki “modelin ne kadar iyi olduğunun ölçüsü”nün somut hâli; orada error_rate raporlanıyordu, burada optimize edilen sayı budur.
- **Geriye (Stat 110):** MSE/MAE, tahmin hatasının istatistiksel ölçütleri; MSE büyük hataları daha çok cezalandırır — [Egzersiz 5](#) tam bu farkı deneylerir.

10.10 Türev ve Gradient

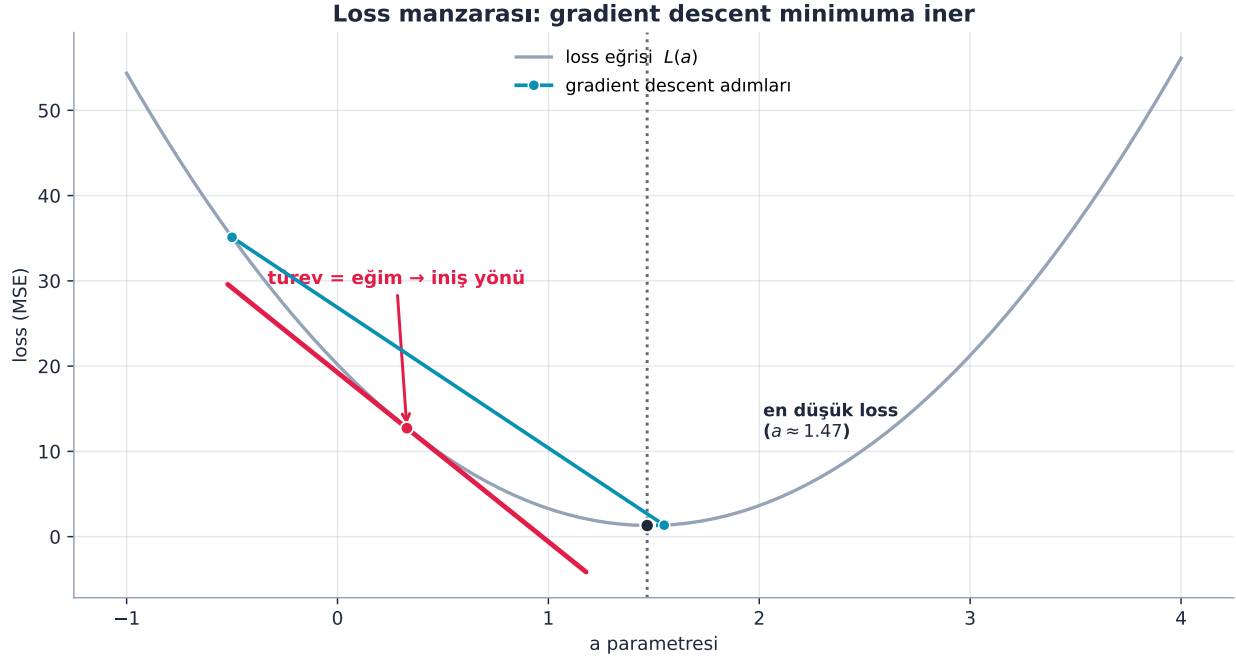
Loss’u küçültmek için her parametreyi hangi yönde değiştireceğimizi bilmeliyiz. Cevap **türev**dir: bir parametreyi azıcık artırınca loss artar mı azalır mı, ne kadar? Howard türevi hatırlatır — korkmaya gerek yok, fikir basittir: eğim.

“The derivative is, if I increase a , the loss will go down.” — Howard, 38:42

Tüm parametreler için türevlerin tamamına **gradient** (∇L) denir. Gradient bize en dik iniş yönünü verir. Şekil 10.3’teki rose teğet çizgisi tam bu eğimi — loss kâsesinin bir noktasındaki türevi — görselleştirir.

**Builder Notu — Türev = Eğim = İniş Yönü**

- **Geriye (Calculus):** Türev = anlık değişim oranı; gradient = çok değişkenli türev vektörü ∇L . Calculus’un derin öğrenmedeki tek ama temel rolü budur — eğimi bilmek, nereye adım atacağımızı söyler.



Şekil 10.3: Loss manzarası + gradient descent (gerçek hesaplama): bir parametreye (a) göre loss bir kâse çizer; türev (eğim) iniş yönünü verir ve gradient descent $p \leftarrow p - \eta \nabla L$ adım adım minimuma iner. En düşük loss noktası $a \approx 1.47$.

- **İleriye (Gradient Descent):** Şekil 10.3'teki cyan adımlar bu eğimi kullanarak minimuma iner; bir sonraki bölüm bu inişi PyTorch'la otomatikleştirir.

10.11 Gradient Descent

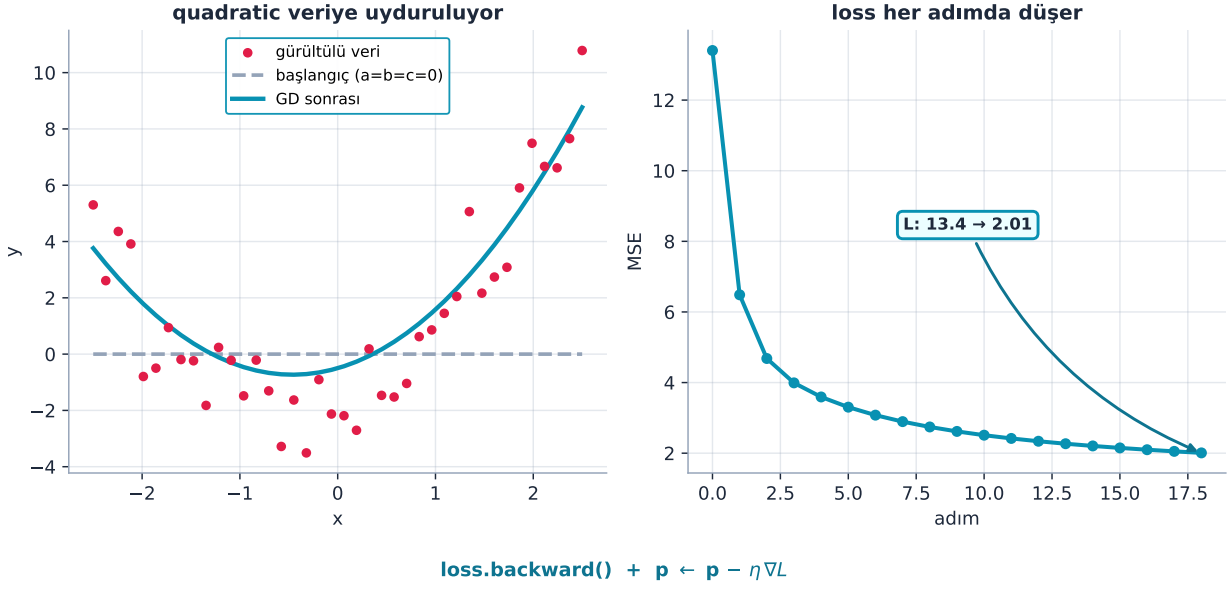
Şimdi sihir: PyTorch türevleri **otomatik** hesaplar. Parametrelere `requires_grad_()` deriz, loss'u hesaplayıp `loss.backward()` çağırırız — PyTorch her parametrenin gradyanını `.grad`'a yazar. Sonra parametreleri gradyanın **tersine** küçük bir adım kaydırırız, $p \leftarrow p - \eta \nabla L$. Bu döngü **gradient descent**tir.

```
abc = torch.tensor([1.1, 1.1, 1.1])
abc.requires_grad_()

for i in range(10):
    loss = quad_mae(abc)          # tahmin et, kaybi olc
    loss.backward()              # her parametrenin gradyanini hesapla
    with torch.no_grad():
        abc -= abc.grad * 0.01   # gradyanin tersine kucuk adim (lr=0.01)
    print(f'step={i}; loss={loss:.2f}')
```

“Call backward to calculate the gradients, and then with no grad, subtract the gradients times a small number. And there we go: the loss keeps improving.” — Howard, 41:42

Şekil 10.4 bu döngünün gerçek bir çalıştırmasını gösterir: solda başlangıçtaki düz çizgi adım adım verinin parabolüne yaklaşır, sağda loss her adımda düşer.



Şekil 10.4: Gradient descent (gerçek hesaplama — analitik MSE gradyanı): parametreler gradyanın tersine küçük adımlarla güncellenir; başlangıçta düz çizgi ($a=b=c=0$) verinin parabolüne yaklaşır, loss her adımda düşer (gürültü tabanına). $\text{loss.backward()} + p \leftarrow p - \eta \nabla L$.

💡 Builder Notu — micrograd'ın Tam Çekirdeği

- **Geriye (Karpthy):** `loss.backward() + param -= grad.lr` = Karpthy micrograd'ının tam çekirdeği; orada bu elle (her düğümün türevi zincir kuralıyla), burada PyTorch'la otomatik kurulur.
- **İleriye:** `with torch.no_grad()` bloğu, güncelleme adımının gradyan grafiğine girmemesini sağlar — production eğitim döngülerinin standart desendir; [learning rate](#) ise bu adımın çarpanıdır.

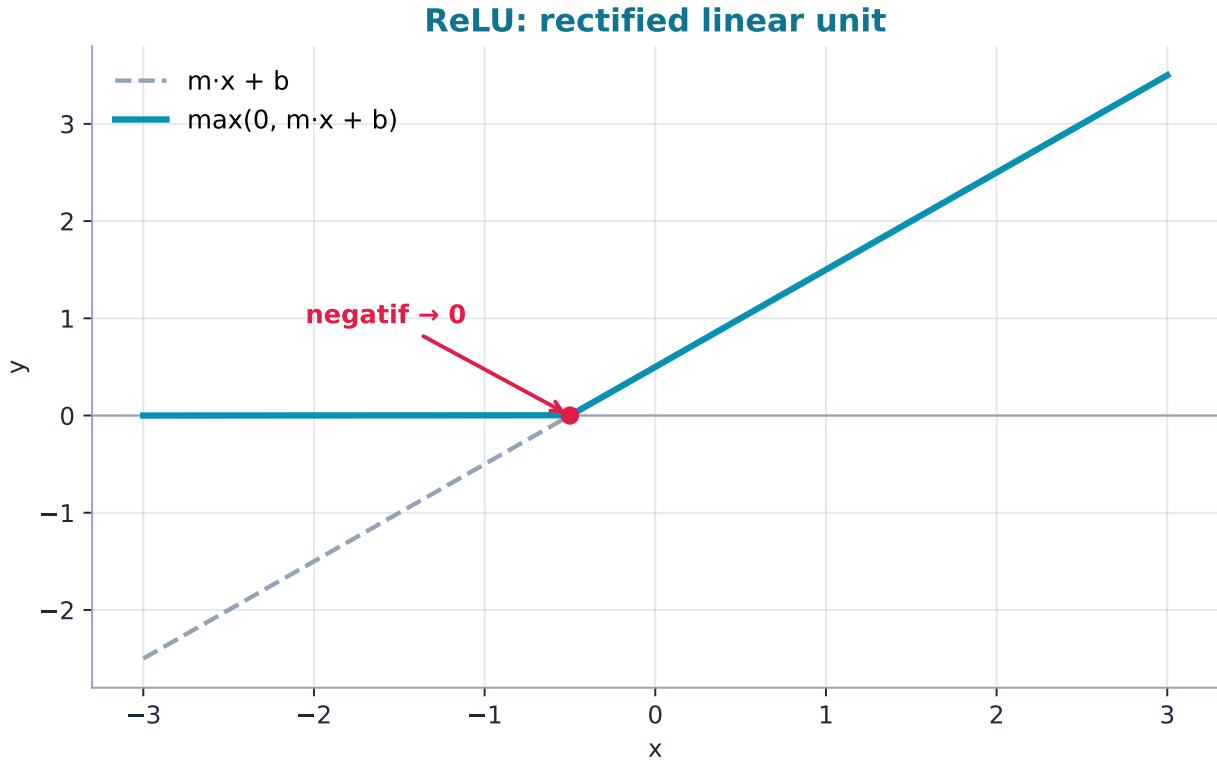
10.12 ReLU: Rectified Linear Unit

Quadratic basit kaldı; gerçek dünya çok daha karmaşık fonksiyonlar ister. Howard en küçük esnek yapı taşı tanıtır: bir doğru ($mx + b$) alıp negatif kısmını sıfıra kırmak. Buna **rectified linear unit (ReLU)** denir — kısaca $\max(0, mx + b)$.

```
def rectified_linear(m, b, x):
    y = m*x + b
    return torch.clip(y, 0.) # negatifleri sifira cek
# Esdegeri: F.relu(m*x + b)
```

“If it’s bigger than zero, leave it; if it’s smaller than zero, it’ll make it zero. So that’s rectified linear.” — Howard, 44:10

Şekil 10.5 bu tek kırılma noktasını (köşeyi) gösterir: ham doğru negatif bölgede sıfıra kırılır ve doğrusal-olmamayı verir.



Şekil 10.5: ReLU (gerçek hesaplama): bir doğru $m \cdot x + b$ alınır, negatif kısmı sıfıra kırılır $\rightarrow \max(0, m \cdot x + b)$. Tek kırılma noktası (köşe) doğrusal-olmamayı verir; en küçük esnek yapı taşı.

💡 Builder Notu — En Küçük Esnek Yapı Taşı

- **Geriye (6.S191):** ReLU, 6.S191 Ders 1'deki aktivasyon fonksiyonudur; doğrusal-olmamayı sağlayan en yaygın seçim — onsu katmanlar üst üste binse de tek bir doğrusal dönüşüme çökerdi.
- **İleriye (Sonsuz Karmaşık Fonksiyon):** Tek ReLU bir “kırık doğru”dur; ama Şekil 22.2’da göreceğin gibi yeterince toplanınca her şeyi kurar.

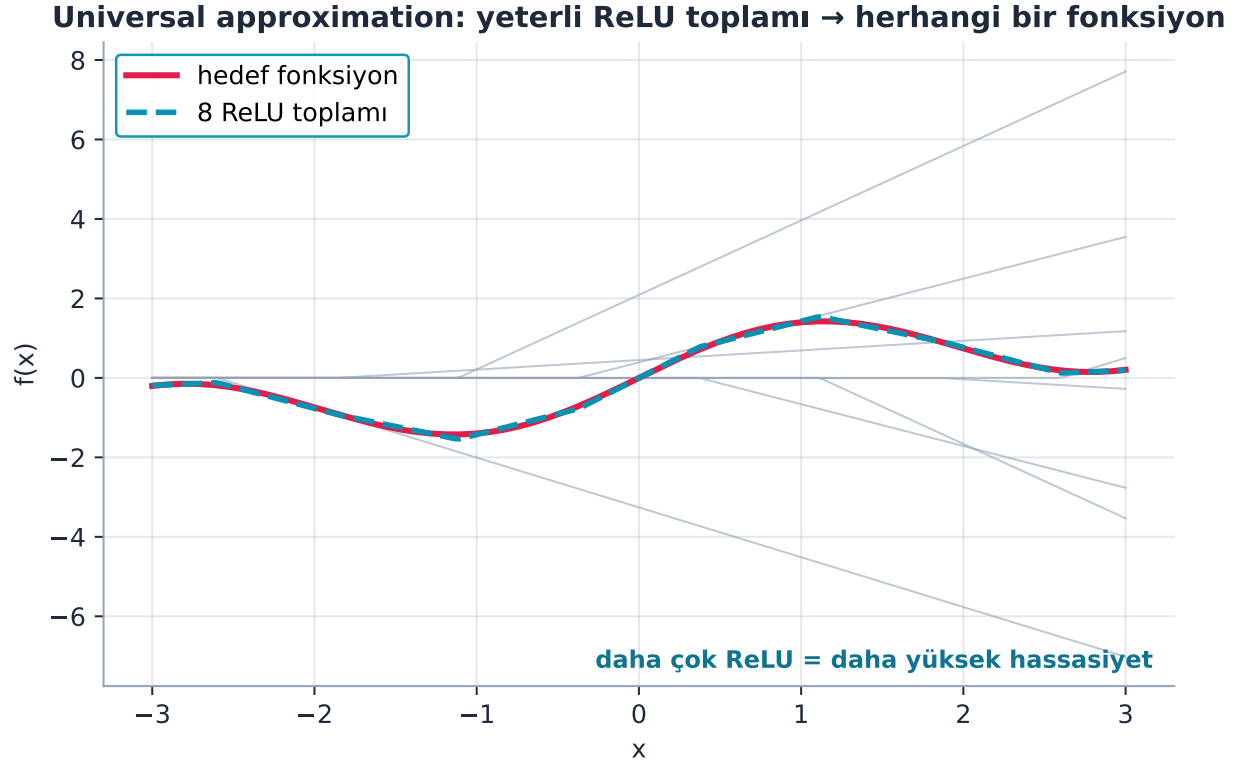
10.13 Sonsuz Karmaşık Fonksiyon

İşte anahtar fikir: iki ReLU’yu toplarsak inişli-çıkışlı bir şekil elde ederiz. İsteddiğimiz kadar ReLU toplarsak, istediğimiz kadar “kırımlı” herhangi bir fonksiyonu istediğimiz hassasiyette kurabiliriz.

```
def double_relu(m1, b1, m2, b2, x):
    return rectified_linear(m1, b1, x) + rectified_linear(m2, b2, x)
```

“We could add as many values together as we want, and with enough values we can match it as close as we want.” — Howard, 46:31

Howard ses dalgası örneği verir: 100 milyon ReLU verersen, konuşmanın dalga formunu neredeyse birebir eşleyebilirsin. Aynı fikir 2B, 3B, n-boyutlu girdiler için de geçerlidir. Şekil 22.2 bunu somutlaştırır: 8 ağırlıklı ReLU toplanarak kıvrımlı bir hedef fonksiyon yeniden kurulur.



Şekil 10.6: Universal approximation (gerçek hesaplama — 8 ReLU least-squares): yeterince ReLU toplanırsa istenen kadar kıvrımlı herhangi bir fonksiyon istenen hassasiyette kurulabilir (ince gri = bileşenler, rose = hedef, cyan kesik = toplam).

💡 Builder Notu — Universal Approximation’ın Resmi

- **Geriye (18.06):** Çok boyutlu girdi = matris-vektör çarpımı; her ReLU bir doğrusal dönüşüm + kırpmadır. Şekil 22.2’deki ince gri bileşenler tam bu tekil ReLU’lardır.
- **İleriye (“Deep Learning”i Türettik’):** Bu, **universal approximation** teoremidir; modern devasa ağlar bu basit fikrin ölçeklenmiş hâlidir — bir sonraki bölüm iki parçayı birleştirip sonucu ilan eder.

10.14 “Deep Learning”i Türettik”

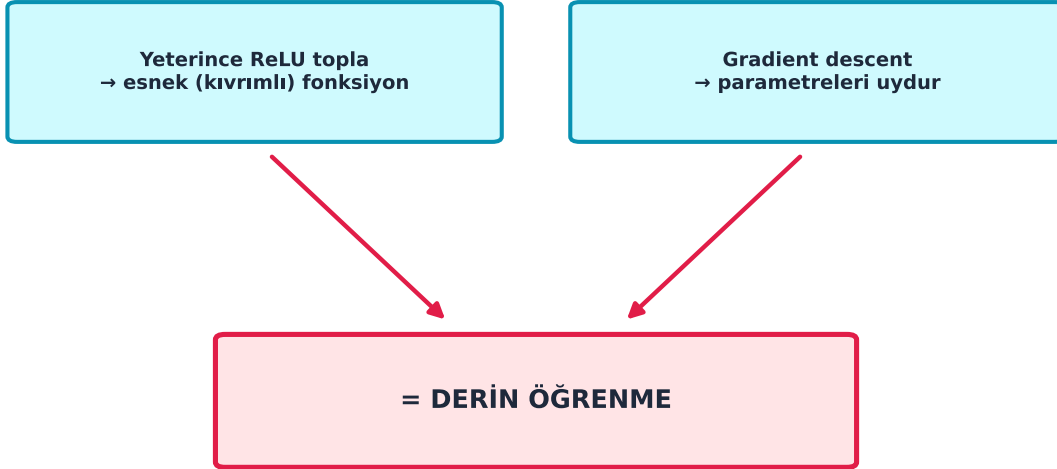
İki parça birleşince her şey tamamlanır: (1) yeterince ReLU toplayarak istenen kadar esnek bir fonksiyon kurarız (**ReLU toplamı**); (2) onun parametrelerini gradient descent ile uydururuz (**Gradient Descent**). Howard’ın ünlü cümlesi:

“We have just derived deep learning. Everything from now on is tweaks to make it faster and make it need less data. This is it.” — Howard, 47:59

Howard “draw the rest of the owl” şakasına gönderme yapar: aslında iki adım arasında gizli bir sihir yoktur — değerleri topla, gradient descent ile parametreleri optimize et, istediğin girdi-çıkı örneklerini ver; bilgisayar gerisini çizer. Şekil 10.7 bu sentezi tek bir şemada toplar.

“It’s using gradient descent to set some parameters to make a wiggly function, which is basically the addition of lots of rectified linear units, to match your data.” — Howard, 49:12

Howard 47:59: gerisi hız ve veri verimliliği ayarı



Şekil 10.7: Howard’ın sentezi (şematik): (1) yeterince ReLU toplayarak esnek (kırımlı) bir fonksiyon kur + (2) parametrelerini gradient descent ile uydur = derin öğrenme. ‘Gerisi hız ve veri verimliliği ayarı.’ (sematik — Howard’ın sentezi)



Builder Notu — Top-down ile Bottom-up Aynı Yerde Buluşur

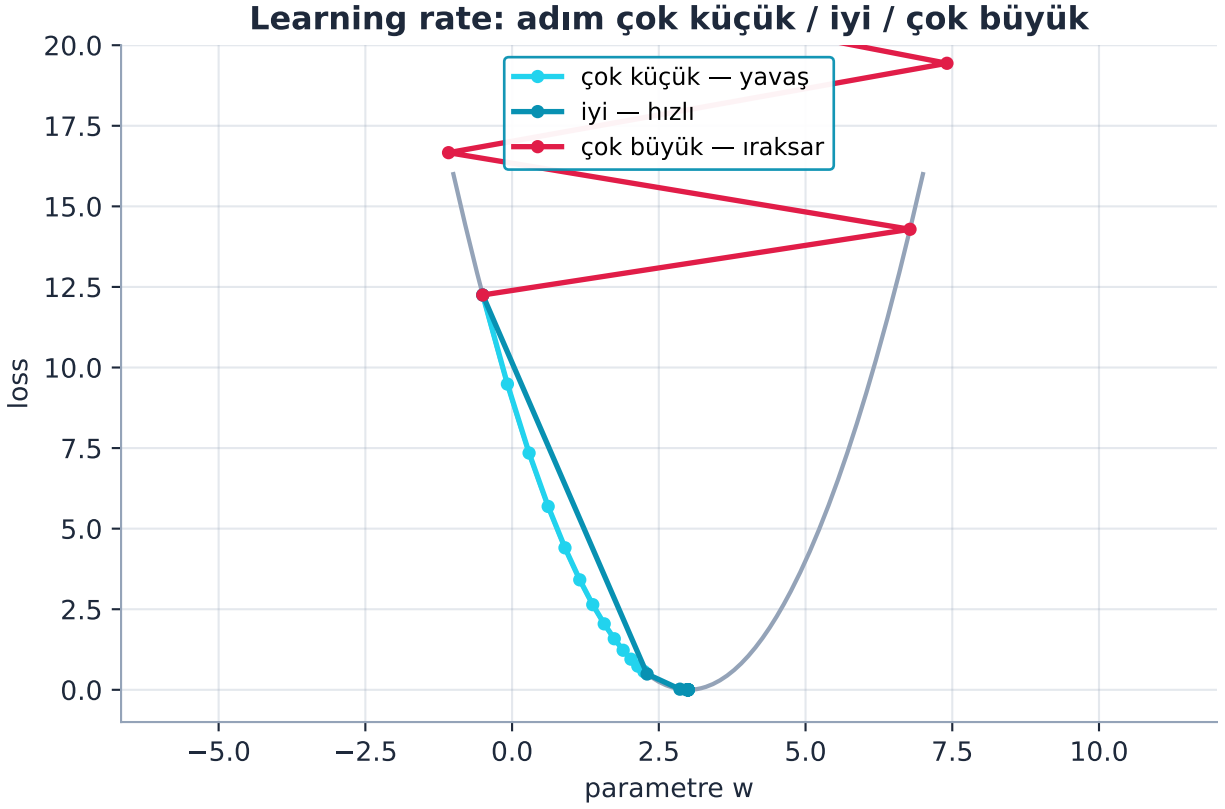
- **Geriye (tüm seri):** Bu cümle Karpathy serisinin de tezidir; fast.ai top-down bu sonuca varır, Karpathy bottom-up aynı yere ulaşır — iki yol, aynı çekirdek.
- **İleriye:** “Gerisi tweak” — Part 2’deki tüm teknikler (init, normalization, mimari) bu çekirdeği hızlandırma/iyileştirmedi; foundation burada kapanır.

10.15 Learning Rate

Gradient descent’teki “küçük adım” çarpanı (`abc.grad * 0.01`) kritik bir hiperparametredir: **learning rate** (η). Çok büyük seçilirse adımlar hedefi aşar, loss kötüleşir; çok küçük seçilirse ilerleme can sıkıcı derecede yavaş olur.

“If you pick a learning rate that’s too big, you’ll jump too far. Finding the right learning rate is a compromise.” — Howard, 57:24

Şekil 10.8 üç rejimi yan yana koyar: çok küçük adım minimuma sürünür, iyi adım hızlı iner, çok büyük adım hedefi aşırıraksar.



Şekil 10.8: Learning rate (gerçek hesaplama): çok küçük adım yavaş yakınsar, iyi adım hızlı iner, çok büyük adım hedefi aşırıraksar. Gradient descent adım çarpanı kritik hiperparametredir.

Builder Notu — η : Adım Uzunluğu Uzlaşması

- **İleriye (Ders 5-6, 18):** `lr_find` (öğrenme oranı bulucu) ve `OneCycle` politikası bu sorunu otomatikleştirir; Part 1 Ders 5-6 ve Part 2 Ders 18’de göreceğiz — Şekil 10.8’teki “ıraksar” eğrisi tam onların önlediği felakettir.
- **Geriye (Calculus):** Learning rate, gradyan yönündeki adım uzunluğu η ’dır; optimizasyonun yakınsamasını belirler — yön türevden, uzunluk η ’dan gelir.

10.16 Pratik Kararlar: Yeterli Veri ve Model Seçimi

Howard foundation’dan pratiğe döner. “Hangi modeli kullanmalıyım?” sorusuna pratik cevap: deney yapana kadar başlangıçta `resnet18` ile hızlı çalış, sonra daha güçlü mimariler dene. “Yeterli verim var mı?” sorusu ise **baştan** sorulmalıdır, sondan değil — yetersizse daha fazla veri toplamak ya da augmentation kullanmak gerekir.

“These are things you want to know at the start, not at the end.” — Howard, 53:39

💡 Builder Notu — Başarı Kriterini Baştan Tanımla

- **İleriye (goal-driven):** “Doğruluk yeterli mi?” sorusu probleme bağlıdır; başarı kriterini önceden tanımlamak projeyi yönlendirir — sonradan değil, baştan ölçülebilir bir hedef koymak işin kendisidir.
- **Geriye (timm karşılaştırması):** “Önce resnet18, sonra güçlü mimari” tavsiyesi, hız-doğruluk grafiğinin pratik özetidir; veri yetmiyorsa Ders 2’deki [data augmentation](#) devreye girer.

10.17 Kapanış

Ders 3, Ders 1-2’nin kara kutusunu açtı: bir model, gradient descent ile parametreleri uydurulan esnek bir fonksiyondur. Quadratic’ten ReLU toplamına, oradan otomatik gradyan inişine kadar her parça elle kuruldu. Bundan sonraki her şey — daha iyi mimariler, normalizasyon, hız — bu çekirdeğin üstüne eklenen ayarlardır.

💡 Builder Notu — Foundation Sağlamsa Gerisi Anlaşılır

- **İleriye:** Bu temel, Part 2’de (ve Karpathy serisinde) satır satır yeniden kurulacak; foundation sağlamsa gerisi anlaşılır.
- **Geriye (Bu Derste Ne Var?):** Dersin başında verilen üç fikir — [parametrelili fonksiyon](#), [gradient descent](#), [ReLU toplamı](#) — burada tek bir cümlede kapanır.

10.18 Bu Dersin Özeti

1. Bir model = bir mimari (sayıların nasıl birleşeceği) + parametreler (öğrenilen sayılar); `model.pkl` tam olarak budur ([model.pkl](#)).
2. En basit model bile parametrelili bir fonksiyondur (quadratic: $ax^2 + bx + c$) (**Model = Parametrelili Fonksiyon**).
3. **Loss** (MAE/MSE) modelin iyiliğini sayıyla ölçer; amaç onu en küçük yapmaktır (**Loss**).
4. **Türev/gradient** her parametreyi hangi yönde değiştireceğimizi söyler; PyTorch bunu `loss.backward()` ile otomatik hesaplar (**Türev ve Gradient**).
5. **Gradient descent:** parametreleri gradyanın tersine küçük adımlarla (learning rate) güncelleyen döngü, $p \leftarrow p - \eta \nabla L$ (**Gradient Descent**).
6. **ReLU** en küçük esnek yapı taşıdır; yeterince ReLU toplanırsa herhangi bir fonksiyon istenen hassasiyette kurulabilir (universal approximation) (**ReLU, Sonsuz Karmaşık Fonksiyon**).
7. ReLU toplamı + gradient descent = **derin öğrenmenin türetilmesi**; gerisi hız ve veri verimliliği ayarlarıdır (“**Deep Learning’i Türettik**”).
8. **Learning rate** kritik hiperparametredir: çok büyük aşar, çok küçük yavaşlatır (**Learning Rate**).

❗ Tek Bir Cümle

Bir sinir ağı, çok sayıda basit doğrusal-kırpma biriminin (ReLU) toplamıyla kurulan esnek bir fonksiyondur ve “öğrenmesi”, gradient descent’in kayıp fonksiyonunu küçültecek parametreleri bulmasından ibarettir.

10.19 Kontrol Soruları

i Soru 1: Gradient descent döngüsünün üç adımı nedir ve `loss.backward()` ne yapar?

Cevap:

Üç adım: (1) tahmin et ve kaybı hesapla (`loss = quad_mae(abc)`), (2) gradyanları hesapla (`loss.backward()`), (3) parametreleri gradyanın tersine küçük bir adım kaydır (`abc -= abc.grad * 1r`). `loss.backward()`, PyTorch'un otomatik türev (autograd) motorunu çalıştırır: her parametrenin kayba olan katkısının türevini hesaplayıp `.grad` özneliğine yazar. Güncelleme `with torch.no_grad()` içinde yapılır ki bu işlem gradyan grafiğine girmesin.

i Soru 2: Howard “deep learning’i türettik” derken hangi iki parçayı kastediyor? Quadratic ile sinir ağı arasındaki fark nedir?

Cevap:

İki parça: (1) yeterince **ReLU** (rectified linear unit) toplayarak istenen hassasiyette herhangi bir esnek fonksiyon kurmak (universal approximation), (2) bu fonksiyonun parametrelerini **gradient descent** ile uydurmak. Quadratic ile sinir ağı arasında kavramsal fark yoktur — ikisi de parametrelili fonksiyondur ve aynı gradient descent ile eğitilir. Fark ölçektir: quadratic 3 parametre, sinir ağı milyonlarca; ve sinir ağı ReLU toplamı sayesinde çok daha karmaşık şekilleri temsil edebilir.

i Soru 3: Learning rate çok büyük seçilirse ne olur, çok küçük seçilirse ne olur?

Cevap:

Çok büyük: her adım hedefi aşar (overshoot), parametreler minimumun etrafında zıplar veya tamamen ışınlanıp uzaklaşır — loss kötüleşir, eğitim ıraksar. Çok küçük: her adım minicik olur, minimuma ulaşmak çok uzun sürer (sıkıcı ve pahalı). Doğru learning rate bir uzlaşmadır; Part 1 Ders 5-6’daki `1r_find` ve Part 2 Ders 18’deki OneCycle bunu otomatikleştirir.

i Soru 4: Bir tek ReLU neden tek başına yetersizdir ama toplamaları neden her şeyi yapabilir? (builder bağlantısı)

Cevap:

Tek ReLU yalnızca bir “kırık doğru”dur (bir eğim + bir köşe); tek başına karmaşık bir eğriyi temsil edemez. Ama iki ReLU toplanınca inişli-çıkışlı bir şekil oluşur; n tane ReLU toplanınca istenen kadar kıvrımlı herhangi bir fonksiyon istenen hassasiyette kurulabilir. Bu, universal approximation teoremidir. Builder açısından: modern devasa ağlar (transformer, CNN) bu basit “ReLU toplamı + gradient descent” fikrinin ölçeklenmiş ve düzenlenmiş hâlidir — Karpathy serisi ve Part 2 bunu satır satır kurar.

10.20 Egzersizler

Egzersiz 1 (Direkt uygulama). `mk_quad`, `mae` ve gradient descent döngüsünü kendin kur; farklı başlangıç a , b , c ve learning rate’lerle loss’un nasıl düştüğünü gözle.

Egzersiz 2 (İki-aşamalı). `double_relu`’yu üç-dört ReLU toplamına genişlet; `@interact` ile parametreleri

oynatıp ne kadar karmaşık şekiller kurabildiğini gör.

Egzersiz 3 (Edge case). Learning rate’i kasıtlı olarak çok büyük (örn. 1.0) ve çok küçük (örn. 0.0001) seç; loss’un sırasıyla ıraksamasını ve aşırı yavaşlamasını gözlemle.

Egzersiz 4 (Python ile doğrulama). Bir tensöre `requires_grad_()` uygula, basit bir fonksiyonun loss’unu hesapla, `backward()` çağır ve `.grad`’ı yazdır; gradyanın işaretinin loss’u nasıl etkilediğini açıkla.

Egzersiz 5 (Sonraki dersin habercisi). MAE yerine MSE (karesel hata) kullanıp sonuçları kıyasla; büyük hataların neden daha çok cezalandırıldığını ve bunun gradyanı nasıl etkilediğini düşün.

10.21 Sonraki Ders İçin Hazırlık

Ders 4: Doğal Dil İşleme (NLP)

Şimdiye kadar görüntülerle çalıştık. Ders 4 metne geçer: bir cümlenin olumlu mu olumsuz mu olduğunu, iki sorunun aynı anlama gelip gelmediğini sınıflandırma. Hugging Face Transformers, tokenization ve ULMFiT.

Ana konular:

- Metni sayıya çevirme (tokenization, sayısallaştırma)
- Pretrained dil modelleri ve ince ayar (ULMFiT, Transformers)
- Validation vs test set ve overfitting tuzakları
- Pearson korelasyonu ile metrik



Ders 4 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 4 — gradient descent’i elle kur).
- Quadratic örneğini kendi notebook’unda baştan çalıştır.
- Ana cümleyi tekrar oku: “Model = gradient descent ile uydurulan esnek bir fonksiyon.”

10.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard’da
Model = fonksiyon	Mimari + parametreler; en basiti parametrelili bir denklem	20:49
Parametre	Modelin öğrendiği sayılar (quadratic’te a, b, c)	23:43
Loss (MAE/MSE)	Tahmin-gerçek farkının ölçüsü; küçük = iyi	31:02
Türev / gradient	Parametreyi hangi yönde değiştirince loss düşer	38:42
<code>requires_grad_</code>	PyTorch’a bu tensörün gradyanını izlemesini söyler	35:59
<code>loss.backward()</code>	Otomatik türev; gradyanları <code>.grad</code> ’a yazar	38:19

Kavram	Tanım	Howard'da
Gradient descent	Parametreleri gradyanın tersine küçük adımlarla güncelleme	39:23
ReLU	$\max(0, m \cdot x + b)$; en küçük esnek yapı taşı	43:23
Universal approximation	Yeterli ReLU toplamı her fonksiyonu kurar	46:31
Learning rate	Gradient descent adım çarpanı; büyük aşar, küçük yavaşlatır	56:28
timm	En büyük görü modeli koleksiyonu (resnet, convnext)	16:35
Mimari seçimi	Hız vs doğruluk dengesi (resnet18 -> convnext)	13:47

10.23 ML Builder Bağlantıları

💡 Builder Notu — 6 ML Köprüsü

Bu ders, derin öğrenmeyi en yalın çekirdeğinden — parametrelili fonksiyon + gradient descent — türetir; köprülerin özeti:

1. **Model = parametrelili fonksiyon** → tüm derin öğrenmenin tek cümlelik özü; mimari değişse de fikir sabit ([Model = Parametrelili Fonksiyon](#)).
2. **Gradient descent** → `loss.backward()` + `param -= grad * lr`; Karpathy micrograd'ının çekirdeği, Calculus türevinin uygulaması ([Gradient Descent](#)).
3. **ReLU + universal approximation** → modern devasa ağların matematiksel gerekçesi ([Sonsuz Karmaşık Fonksiyon](#)).
4. **Learning rate** → optimizasyonun en kritik hiperparametresi; `lr_find/OneCycle` ile otomatikleşir ([Learning Rate](#)).
5. **timm / mimari seçimi** → production'da hız-doğruluk dengesi; tek satırla model değiştirme ([timm](#)).
6. **PyTorch autograd** → otomatik türev; Karpathy bunu sıfırdan, fast.ai hazır kullanır ([Gradient Descent](#)).

❗ Bu dersten tek bir şey alıp gideceksen

Bu dersten tek bir şey alıp gideceksen: derin öğrenmede iki adım arasında gizli sihir yoktur. Çok sayıda ReLU'yu topla, gradient descent ile parametrelerini ayarla, istediğin girdi-çıkı örneklerini ver — bilgisayar gerisini çizer. “İşte derin öğrenmeyi türettik; gerisi hız ve veri verimliliği ayarıdır.”

11 Doğal Dil İşleme — Metin, Tokenization ve Validation Disiplini (Natural Language Processing)

Metni token'lara böl, sayıya çevir (`input_ids`), pretrained bir dil modelini (`deberta-v3-small`) Hugging Face ile fine-tune et — ama asıl ustalık overfitting'i yakalayan doğru validation/test set'i kurmaktır; rastgele ayırmak çoğu zaman yanıltır ve sessizce değer yok eder

Bölüm bilgisi

- **Howard'ın videosu:** [course.fast.ai — Lesson 4: Natural Language \(NLP\)](https://course.fast.ai/lesson4.html) (~95 dk)
- **Seri:** Practical Deep Learning for Coders — Part 1, Ders 4
- **Hoca:** Jeremy Howard
- **Playlist:** [Practical Deep Learning Part 1 \(2022\)](#)
- **Notebook:** [course22 — Getting started with NLP \(Hugging Face\)](#)
- **Okuma süresi:** ~35 dk

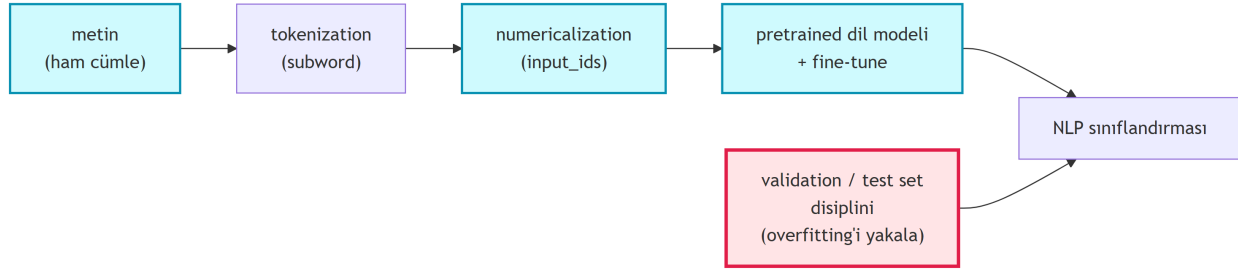
11.1 Bu Derste Ne Var?

Ders 1-3'te hep görüntülerle çalıştık; Ders 4 **metne** geçer. Howard bir Kaggle yarışması üzerinden (US Patent Phrase Matching — iki ifadenin anlamca ne kadar yakın olduğunu tahmin etme) NLP sınıflandırması kurar. Bu derste fastai değil, **Hugging Face Transformers** kullanılır. Ama dersin asıl kalbi mekanik değildir: ikinci yarı, tüm makine öğrenmesinin en kritik kavramına ayrılır — **overfitting** ve onu yakalayan **validation/test set** disiplini.

Üç temel fikir:

1. **Tokenization + numericalization** — metni token'lara böl, sonra her token'ı sözlükteki konumuna (`sayıya`, `input_ids`) çevir; modeller yalnızca sayıyla çalışır ([Tokenization](#) → [Numericalization](#)).
2. **Pretrained dil modeli + fine-tune** — görüntüdeki transfer learning aynen metne uygulanır; `deberta-v3-small`'dan başla, kendi verine uyarla ([Fine-tuning ve Transfer Learning](#)).
3. **İyi validation set = en kritik beceri** — rastgele ayırmak çoğu zaman yanıltır; modeli geleceği/yeni kişileri tahmin edecek şekilde test etmelisin, yoksa sessizce değer yok edersin ([İyi Bir Validation Set Kurmak](#)).

“In the real world outside of Kaggle you will often not even know that you overfit — you just destroy value for your organization silently.” — Howard, 54:19



Şekil 11.1: Ders 4’ün kavram haritası: metin → tokenization (subword) → numericalization (input_ids) → pretrained dil modeli (deberta-v3-small) + fine-tune; ve AYRI bir kol: validation/test set disiplini overfitting’i yakalar (rose vurgulu). İki kol birlikte ’NLP sınıflandırması’ nı verir.

Şekil 34.1 bu üç fikri, metinden başlayıp tokenization → numericalization → pretrained model + fine-tune zincirine ve ayrı bir kolda validation/test disiplinine uzanan tek bir haritada birleştirir.

💡 Builder Notu — Mekanik Kolay, Disiplin Zor

- **Geriye (6.S191):** NLP, 6.S191 Ders 2’deki dizi modelleme + transformer’ın pratik uygulamasıdır; “metin = token dizisi = sayı dizisi” — Howard burada o diziyi gerçek bir Kaggle problemine bağlar.
- **İleriye (Part 2 / Karpathy):** Tokenization Karpathy Ders 9’da (BPE) sıfırdan kurulur; Transformer Part 2 Ders 24’te. Bu derste ikisi de kara kutudur — [Hugging Face Trainer](#) hazır verir.
- **Tek cümle:** Metni sayıya çevirip pretrained bir modeli ince ayarlarsın; ama asıl ustalık doğru [validation set](#)’i kurmaktadır — gerisi birkaç satır.

11.2 Neden NLP için Farklı Kütüphane?

Howard bu derste fastai yerine **Hugging Face Transformers** kullanır. Sebebi: NLP’de transformer tabanlı modeller standart hâline geldi ve Transformers kütüphanesi bu ekosistemin merkezi. Howard, farklı bir kütüphane görmenin de öğretici olduğunu söyler — aynı kavramlar (tokenize, model, fine-tune) her yerde aynıdır.

“Why we’d be using a different library other than fast.ai.” — Howard, 1:13

💡 Builder Notu — Kavramlar Taşınır, API’ler Değişir

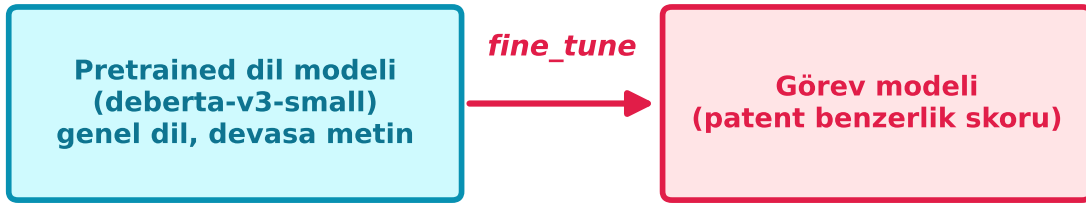
- **İleriye:** Tek bir kütüphaneye bağlı kalmamak builder için değerli; kavramlar taşınabilir, API’ler değişir. [Hugging Face Trainer](#)’ın fastai Learner’ına ne kadar benzediğini birazdan göreceğiz — aynı fikir, farklı isim.
- **Geriye (Ders 3):** Ders 3’te bir modelin “[gradient descent ile uydurulan esnek bir fonksiyon](#)” olduğunu kurmuştuk; o çekirdek kütüphaneden bağımsızdır — Transformers de aynı çekirdeği eğitir.

11.3 Fine-tuning ve Transfer Learning

NLP’de de fikir görüntüdekiyle aynıdır: sıfırdan eğitmeyiz, **pretrained** bir dil modelinden başlar, onu kendi görevimize ince ayarlarız. Howard, Zeiler-Fergus katman görselleştirmesini (Ders 3’teki gibi) hatırlatır: alt katmanlar genel, üst katmanlar göreve özel öznelilikler öğrenir.

“Finetuning a pretrained model.” — Howard, 3:25

Şekil 11.2 pretrained dil modelinin (genel dili öğrenmiş) `fine_tune` ile kendi görevimize (patent benzerliği) nasıl uyarlandığını gösterir — görüdeki `vision_learner` mantığının birebir aynısı.



Ders 1-3'teki `vision_learner` + `fine_tune` ile AYNI fikir; tek fark girdi metin

(transfer learning — görüntü → metin)

Şekil 11.2: Transfer learning, metne de uygulanır (şematik): pretrained bir dil modeli (deberta-v3-small) genel dili öğrenmiştir; `fine_tune` ile kendi görevimize (patent benzerliği) uyarlanır — görüdeki `vision_learner` mantığının aynısı.

💡 Builder Notu — Aynı Transfer, Yeni Modalite

- **Geriye (Ders 1-3):** `vision_learner` + `fine_tune` mantığı; tek fark girdinin metin olması. Şekil 11.2'deki cyan→rose akışı, görüde gördüğümüz “genel öğren, sonra uyarla” zincirinin metin karşılığıdır.
- **İleriye (ULMFIT):** NLP’ye transfer learning’i ilk getiren yöntem Howard’ın kendi icadı ULM-FiT’tir; pretrained + fine-tune fikrinin metindeki köküdür.

11.4 Problem: US Patent Phrase Matching

Howard somut bir Kaggle yarışması seçer: iki patent ifadesinin (anchor ve target, bir bağlam içinde) anlamca ne kadar yakın olduğunu 0-1 arası bir skorla tahmin etme. Bu aslında bir **regresyon** problemidir (sürekli skor), ama NLP sınıflandırma altyapısıyla çözülür.

“US Patent Phrase to Phrase Matching Kaggle competition.” — Howard, 14:49

💡 Builder Notu — Benzerlik Skoru Her Yerde

- **İleriye:** “Benzerlik skoru” problemi semantic search, öneri sistemleri ve embedding tabanlı eşleştirmenin temelidir; bir kez kurduğunuz NLP sınıflandırıcı pek çok yere taşınır.
- **İleriye (Metric vs Loss):** Skor sürekli (regresyon) olduğu için metrik de sürekli — bu yarışmada Pearson korelasyonu; sınıflandırma “doğruluk”undan farklı bir ölçü.

11.5 Veriyi Hazırlama

Model tek bir metin girdisi ister; Howard üç alanı (context, target, anchor) tek bir string’de birleştirir ve veriyi Hugging Face Dataset nesnesine çevirir.

```
df['input'] = 'TEXT1: ' + df.context + '; TEXT2: ' + df.target + '; ANC1: ' + df.anchor

from datasets import Dataset
ds = Dataset.from_pandas(df)
```

💡 Builder Notu — Yapıyı Modele İpucu Olarak Ver

- **Geriye (Python):** pandas DataFrame ile veri manipülasyonu; alanları işaretli (TEXT1/TEXT2/ANC1) birleştirmek modele yapıyı ipucu olarak verir — model, hangi parçanın ne olduğunu bu etiketlerden çıkarır.
- **İleriye (Numericalization):** Bu tek input string’i birazdan tokenize edilip input_ids’e dönüşecek; veri hazırlığı, tokenization’ın hammaddesidir.

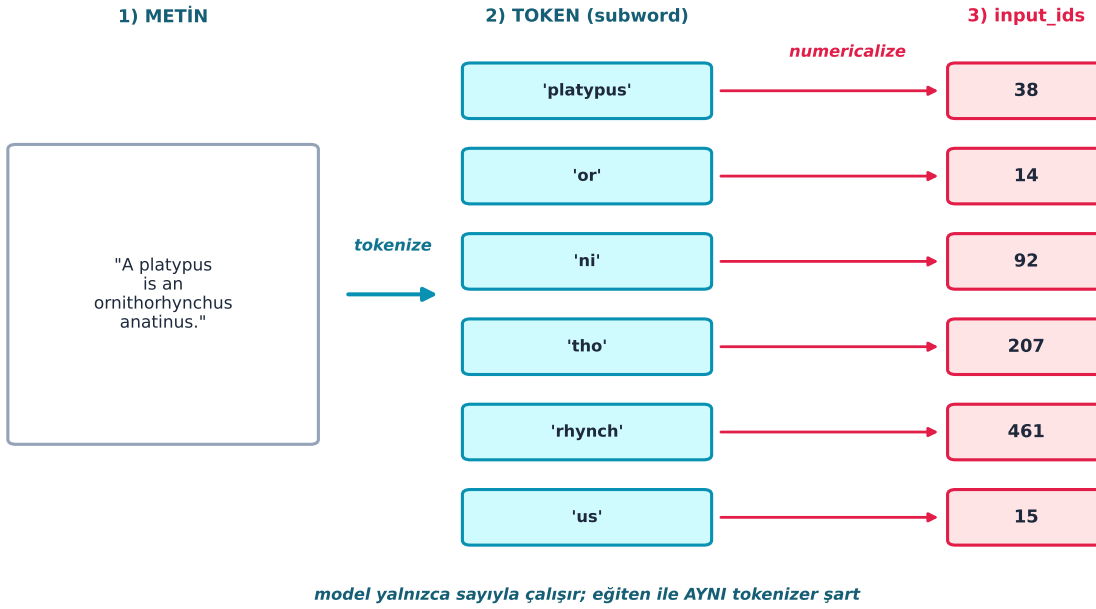
11.6 Tokenization

Metni modele vermeden önce **token**’lara bölmek gerekir. En basit yol kelimelere bölmektir ama o zaman sözlük çok büyük ve nadir kelimeler sorun olur. Modern yöntem **subword** tokenization: kelimeleri sık geçen parçalara böler.

“We don’t want a vocabulary to be too big. So instead, nowadays, people tend to tokenize into something called subwords.” — Howard, 31:02

Kritik kural: tokenize ederken modeli eğitenlerle **birebir aynı** yöntemi kullanmalısınız, yoksa farklı token’lar çıkar.

```
from transformers import AutoTokenizer
model_nm = 'microsoft/deberta-v3-small'
tokz = AutoTokenizer.from_pretrained(model_nm)
tokz.tokenize("A platypus is an ornithorhynchus anatinus.")
# platypus -> ['platypus'], ornithorhynchus -> ['or','ni','tho','rhynch','us']
```



Şekil 11.3: Tokenization + numericalization (şematik): metin subword token'lara bölünür (ornithorhynchus → or-ni-tho-rhynch-us), sonra her token sözlük konumuna (input_ids) çevrilir. Model yalnızca sayıyla çalışır; eğiten ile aynı tokenizer şart. (şematik — token örnekleri Howard'ın gerçek örneği; id'ler illüstratif)

Şekil 11.3 tüm boru hattını tek bir resimde toplar: ham cümle → subword token'lar (ornithorhynchus → or-ni-tho-rhynch-us) → her token'ın sözlük konumu (input_ids).

💡 Builder Notu — Neden Subword?

- **İleriye (Karpathy Ders 9):** Subword/BPE tokenization Karpathy'nin minbpe dersinde sıfırdan kurulur; "neden subword?" sorusu orada derinleşir — burada AutoTokenizer hazır verir, orada elle översin.
- **Pratik:** Şekil 11.3'daki "eğiten ile AYNI tokenizer şart" uyarısı kritiktir; yanlış tokenizer, modelin öğrendiği **sözlük konumlarıyla** uyuşmaz ve çıktı anlamsızlaşır.

11.7 Vocabulary ve Hugging Face Model Hub

Vocabulary (sözlük), tüm benzersiz token'ların listesidir. Model seçimi tokenizer'ı da belirler — bu yüzden Howard önce modeli (deberta-v3-small) seçer. Hugging Face Model Hub binlerce pretrained model barındırır; AutoTokenizer doğru sözlüğü ve tokenize kurallarını otomatik indirir.

"When we say AutoTokenizer, it will download the vocabulary and the details about how this particular model tokenized the dataset." — Howard, 35:57

💡 Builder Notu — Model + Tokenizer Ayrılmaz İkili

- **İleriye:** Model Hub, görüdeki timm’in NLP karşılığıdır; “doğru pretrained model + onun tokenizer’ı” ikilisi ayrılmaz. Ders 3’teki [timm karşılaştırması](#) görüde neyse, Model Hub metinde odur.
- **Geriye (Tokenization):** AutoTokenizer.from_pretrained(model_nm) tek satırla doğru sözlüğü çeker — bu yüzden modeli seçmek tokenizer’ı da seçmektir.

11.8 Numericalization

Tokenization metni token’lara böler; **numericalization** her token’ı sözlükteki konumuna (bir sayıya, input_ids) çevirir. Model yalnızca bu sayılarla çalışır. Howard bir fonksiyon yazıp tüm veri setini toplu (batched) tokenize eder.

```
def tok_func(x): return tokz(x["input"])
tok_ds = ds.map(tok_func, batched=True)
tok_ds = tok_ds.rename_columns({'score': 'labels'}) # HF 'labels' bekler
```

“These are the position in the vocabulary of each of the tokens in the string.” — Howard, 38:58

Şekil 11.3’in sağ kolonu (rose input_ids kutuları) tam bu adımı gösterir: her token kutusu bir sayıya eşlenir.

💡 Builder Notu — Her Şey Sayıdır (Metin Sürümü)

- **Geriye (Ders 1):** “Her şey sayıdır” ilkesi — görüntü pikselleri gibi metin de token id’lerine dönüşür; modelin kapısı her zaman bir sayı dizisidir.
- **Pratik:** Hugging Face hedef sütununu labels adıyla arar; yeniden adlandırmazsan eğitim sessizce çalışmaz — bu, [Trainer](#) kurarken en sık yapılan hatadır.

11.9 ULMFiT

Howard, NLP için transfer learning’i ilk getiren kendi yöntemine değinir: **ULMFiT**. Önce büyük bir metin külliyatında bir dil modeli eğitilir (sonraki kelimeyi tahmin), sonra bu model hedef göreve ince ayarlanır. ULMFiT, büyük belgeleri (transformer’ların aksine) kolayca işleyebilir.

“ULMFiT fits large documents easily.” — Howard, 43:27

💡 Builder Notu — Transfer Learning’in NLP Kökü

- **Geriye (Ders 1):** ULMFiT, modern NLP devriminin iki temelinden biri (Ders 1’de Howard’ın icadı olarak geçti); RNN tabanlı, uzun belgelerde hâlâ avantajlı. [Fine-tuning ve Transfer Learning](#) bölümündeki fikrin tarihsel köküdür.
- **İleriye:** Transformer’lar kısa bağlamda üstün; ULMFiT uzun belgede; ikisi tamamlayıcı. “Tek doğru mimari” yoktur — bağlam uzunluğu kararı belirler.

11.10 Overfitting ve Underfitting

Dersin kalbi: bir model veriyi ne **az** ne **çok** öğrenmelidir. Howard bir polinom örneğiyle gösterir. 1. derece polinom (doğru) veriye yetersiz uyar — **underfitting**; sistematik olarak yanlıştır. 10. derece polinom eğitim verisini ezberler ama gerçek deseni kaçıırır — **overfitting**. 2. derece tam isabettir.

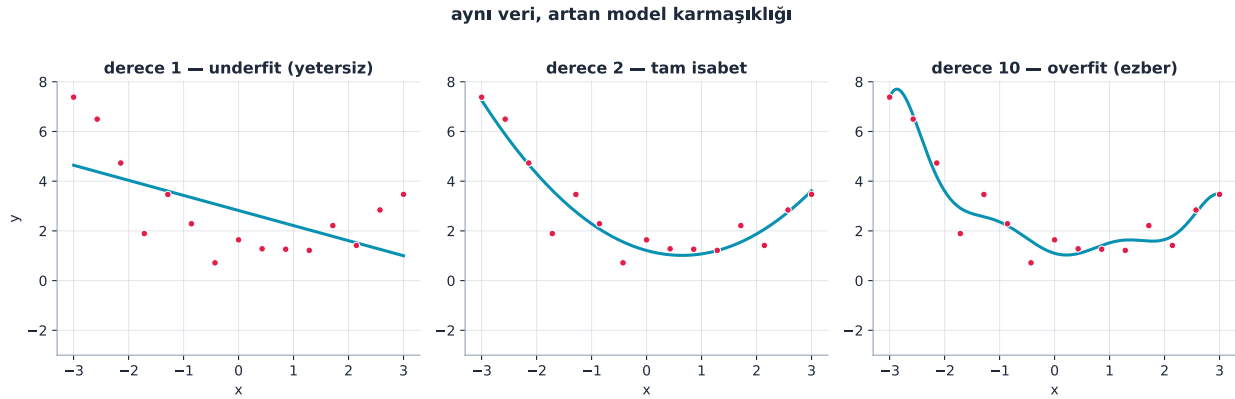
```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

def plot_poly(degree):
    model = make_pipeline(PolynomialFeatures(degree), LinearRegression())
    model.fit(x, y); plt.scatter(x, y); plot_function(model.predict)

plot_poly(1)    # underfit
plot_poly(10)   # overfit
plot_poly(2)    # tam isabet
```

“We don’t want underfitting or overfitting. Underfitting is pretty easy to recognize; overfitting is a bit harder.” — Howard, 45:55

Şekil 11.4 aynı veriye üç farklı karmaşıklıkta polinom uydurur: derece 1 yetersiz (underfit), derece 2 tam isabet, derece 10 gürültüyü ezberler (overfit).

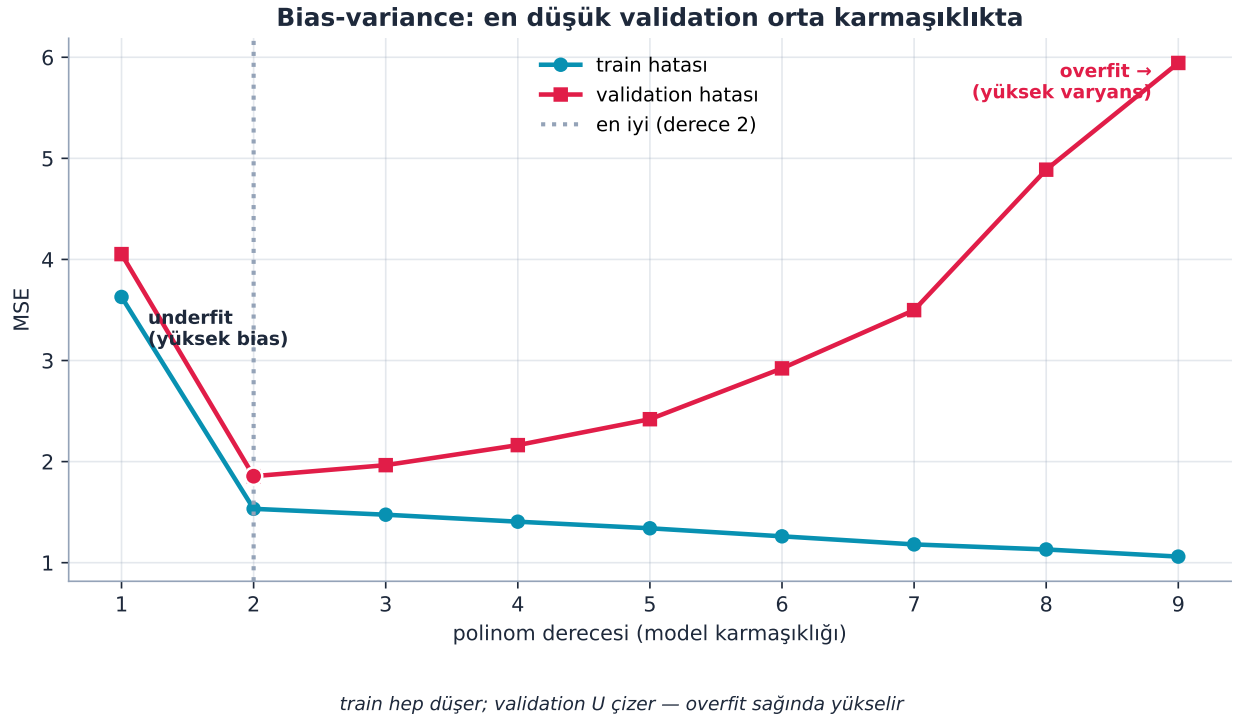


Şekil 11.4: Overfitting / underfitting (gerçek hesaplama — `np.polyfit`): 1. derece veriye yetersiz uyar (underfit, sistematik yanlıştır); 2. derece tam isabet eder; 10. derece gürültü noktalarını ezberler (overfit) ama gerçek deseni kaçıırır.

Aynı olguyu tek bir eğriyle özetlemenin yolu **bias-variance** dengesidir: model karmaşıklığı arttıkça train hatası hep düşer, ama validation hatası bir U çizer — en düşük noktası en iyi modeldir. Şekil 11.5 bunu gösterir.

💡 Builder Notu — Underfit Kolay, Overfit Sinsi

- **Geriye (Stat 110):** Bias-variance dengesi; underfit = yüksek bias, overfit = yüksek variance. Şekil 11.5’teki U eğrisi tam bu istatistiksel dengeyi çizer — solu bias, sağı varyans.



Şekil 11.5: Bias-variance dengesi (gerçek hesaplama — 60 gürültü ortalaması): model karmaşıklığı (polinom derecesi) arttıkça train hatası hep düşer, ama validation hatası U çizer — en düşük noktası (derece 2) en iyi modeldir; sağı overfit (yüksek varyans), solu underfit (yüksek bias).

- **İleriye (Validation Set):** Overfitting'i yakalamanın tek dürüst yolu validation set'tir — Şekil 11.4'teki derece-10 ezberini, ancak modelin görmediği veride ölçerek fark edersin; sıradaki bölüm bunu kurar.

11.11 Validation Set Neden Zorunlu?

Validation set, eğitimde kullanılmayan ama doğruluğu ölçmek için ayrılan veridir. Overfitting'i ancak modelin görmediği veride ölçerek yakalarsın. fastai bunu zorunlu kılar; çoğu kütüphane ise validation set'i atlamana izin vererek “ayağına sıkmanı” kolaylaştırır.

“Most libraries make it really easy to shoot yourself in the foot by not having a validation set. So fast.ai won't even let you do that.” — Howard, 51:58

💡 Builder Notu — fastai Zorlar, Diğerleri İzin Verir

- **Geriye (Ders 1):** `RandomSplitter(valid_pct=0.2)` tam bunu yapar; metrik yalnızca validation set'te ölçülür — Ders 1'de farkında olmadan kullandığın disiplin, burada açıkça gerekçelendirilir.
- **İleriye (İyi Bir Validation Set Kurmak):** Validation set'in var olması yetmez; **doğru** kurulması gerekir — bir sonraki bölüm “rastgele ayırmak neden çoğu zaman yanlış” sorusunu açar.

11.12 İyi Bir Validation Set Kurmak

En kritik nokta: validation set'i **rastgele** ayırmak çoğu zaman yanlıştır. Zaman serisinde geleceği tahmin edeceksen, validation set son haftalar olmalı (ortadan rastgele değil). Modeli “geçmiş değil, geleceği” tahmin edecek şekilde test etmelisin.

“In a time series, you're probably going to want to predict future dates. You should truncate and remove the last couple of weeks, testing whether you can predict the future, rather than the past.” — Howard, 53:17

Howard iki Kaggle örneği verir: dikkati dağılmış sürücü yarışmasında test set'te eğitimde **olmayan kişiler** vardı — iyi bir validation set de eğitimdekenden farklı kişiler içermeliydi. Balıkçılık yarışmasında ise model balığı değil tekneyi tanıyarak “kopya çekti.”

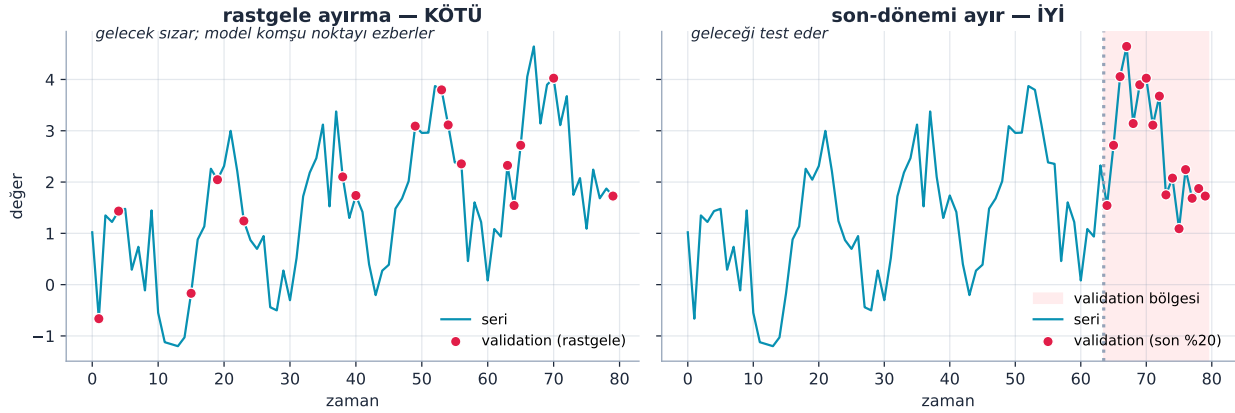
“Your validation set should contain photos of people that aren't in the data you're training your model on.” — Howard, 55:24

Şekil 11.6 aynı zaman serisini iki strateji ile ayırır: solda rastgele ayırma geleceği sızdırır (kötü — model komşu noktayı tanır), sağda son dönemi ayırmak geleceği tahmin etmeyi taklit eder (iyi).

💡 Builder Notu — Validation = Gerçek Kullanımın Provası

- **İleriye (production):** Veri sızıntısı (data leakage) ve kötü validation set, gerçek dünyada sessiz başarısızlığın bir numaralı sebebidir. `train_test_split/random_splitter` seni rastgele ayırmaya teşvik eder — Şekil 11.6'in sol paneli tam bu tuzağı gösterir; dikkatli ol.

Zaman serisinde validation set: rastgele DEĞİL, son-dönem



Şekil 11.6: İyi validation set (gerçek hesaplama — zaman serisi): rastgele ayırma geleceği sızdırır (kötü — model komşu noktayı tanıyıp aldattır); son dönemi ayırmak modeli gerçekte nasıl kullanacağını (geleceği tahmin) taklit eder (iyi).

- **Geriye (Overfitting):** İyi validation set, Şekil 11.5'teki U eğrisinin minimumunu **dürüst** ölçmeni sağlar; kötü validation set o minimumu olduğundan iyi gösterir ve seni overfit bir modele yönlendirir.

11.13 Test Set

Test set, üçüncü bir ayrımdır: model geliştirirken doğruluğunu izlemek için **bile** kullanmazsın. Neden? Üç ay boyunca her gün iki model denersen 180 model denemiş olursun; bazıları validation set'te salt **şans eseri** iyi çıkar. Test set, “validation set üzerinde overfit” etmediğinden emin olmanı sağlar.

“A test set is basically another validation set, but you don't even use it for tracking your accuracy while you build your model.” — Howard, 57:17

Howard acı gerçeği de söyler: üç ay çalışıp test set'te kötü sonuç alırsan, başa dönmekten başka çare yoktur — ama bilmek bilmemekten iyidir.

“If it's terrible, honestly you have to go back to square one. Better to know than to not know.” — Howard, 58:44

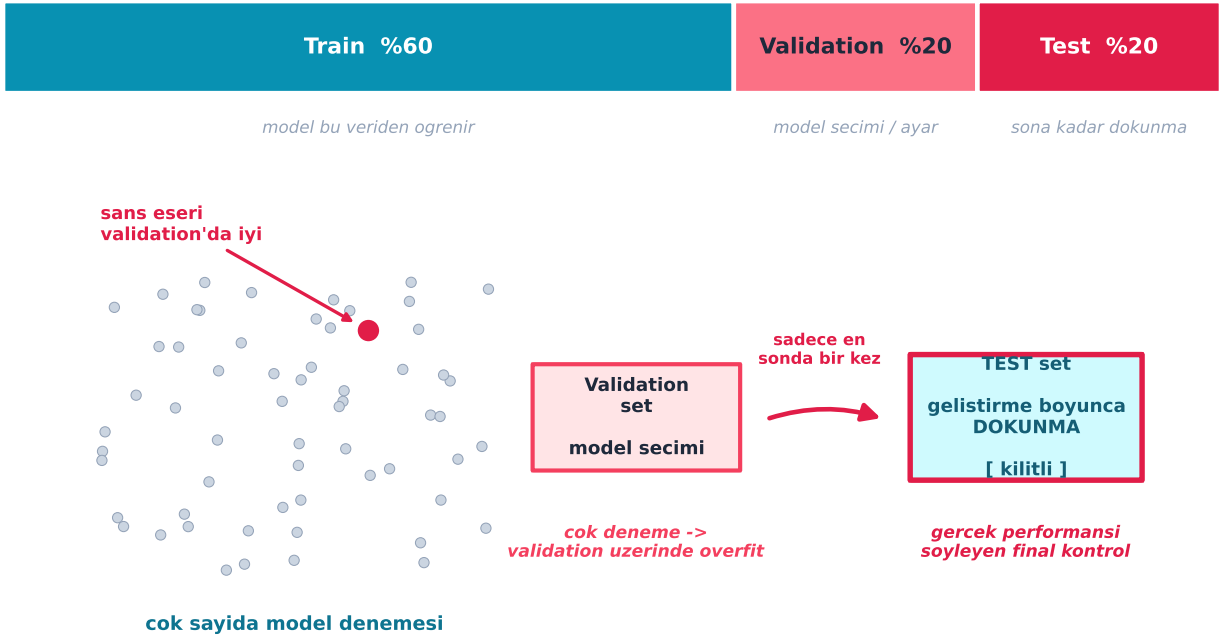
Şekil 11.7 üç-yönlü ayrımı ve “validation üzerinde overfit” kavramını gösterir: çok sayıda denemeden biri validation'da şans eseri parlar; test set, hiç dokunulmadığından gerçek performansı söyler.

💡 Builder Notu — Çok Deneme = Şans Eseri Galip

- **Geriye (Stat 110):** Çok sayıda hipotez deneninince şans eseri “anlamlı” sonuç çıkması = çoklu karşılaştırma problemi; test set bunun panzehiridir. Şekil 11.7'teki “şans eseri iyi” noktası tam bu istatistiksel tuzaktır.

Train / Validation / Test — ve 'validation üzerinde overfit'

Uç-yonlu ayrim



Şekil 11.7: Test set (şematik): train/validation/test üç-yönlü ayırım. Çok sayıda model denenince bazıları validation'da şans eseri iyi çıkar ('validation üzerinde overfit'); test set geliştirme boyunca hiç dokunulmadığından gerçek performansı söyler.

- **İleriye (Kaggle İş Akışı):** Kaggle’ın gizli test set’i tam bu disiplini dayatır — yalnızca en sonda görürsün; bu yüzden Howard onu validation becerisi için en iyi antrenman alanı sayar.

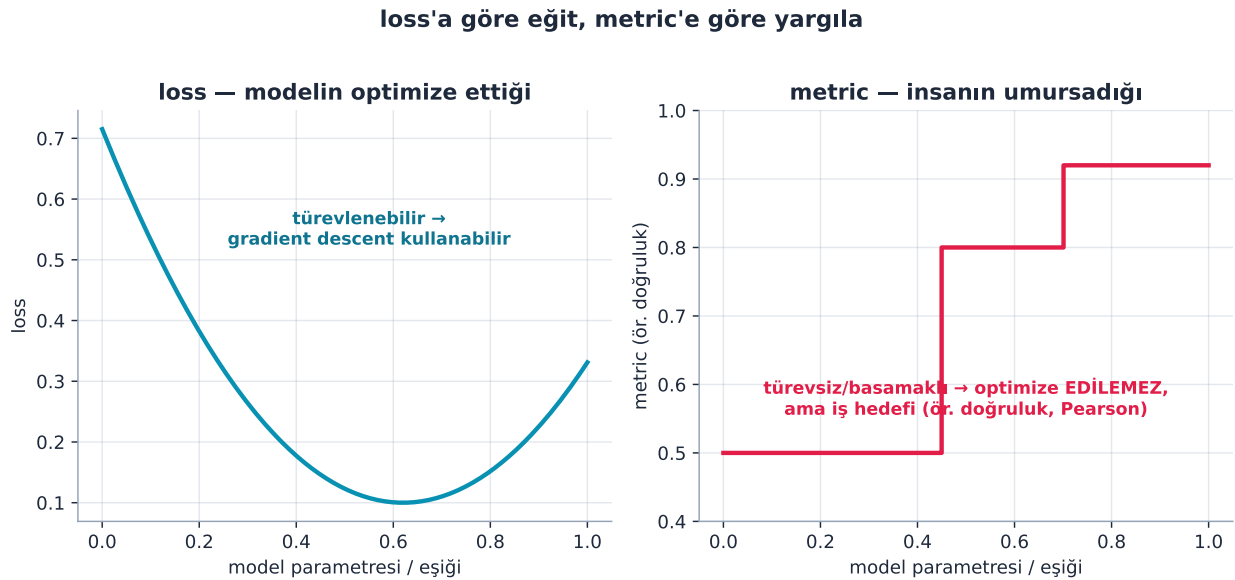
11.14 Metric vs Loss

Loss, modelin optimize ettiği şeydir (gradient descent için pürüzsüz olmalı). **Metric**, insanın umursadığı şeydir (örn. doğruluk, korelasyon). İkisi farklı olabilir. Bu yarışmada metrik **Pearson korelasyonudur** (tahmin ile gerçek arasındaki doğrusal ilişki).

```
def corr(x, y): return np.corrcoef(x, y)[0][1]
def corr_d(eval_pred): return {'pearson': corr(*eval_pred)}
```

“Metric versus loss.” — Howard, 59:05

Şekil 11.8 kontrastı görselleştirir: loss pürüzsüz ve türevlenebilirdir (gradient descent optimize eder), metric ise basamaklı/türevsizdir ama işin gerçek hedefidir.



Şekil 11.8: Metric vs loss (şematik kontrast): loss pürüzsüz ve türevlenebilirdir (gradient descent optimize eder); metric (doğruluk, Pearson) basamaklı/türevsizdir ama işin gerçek hedefidir. Modeli loss’a göre eğitir, başarıyı metric’e göre yargıların.

💡 Builder Notu — Eğit ve Yargıla Farklı Sayılarla

- **İleriye:** Loss optimize edilebilir olmalı; metric iş hedefini yansıtır. Pearson, aykırı değerlere duyarlıdır — Howard veri temizlemede bunu vurgular; Şekil 11.8’un basamaklı sağ paneli “neden doğrudan metric’i optimize edemeyiz” sorusunun cevabıdır.
- **Geriye (Ders 3):** Loss Ders 3’te gradient descent’in küçülttüğü pürüzsüz sayıydı; burada onun yanına “insanın umursadığı” metric’i koyup ikisini ayırıyoruz.

11.15 Hugging Face Trainer ile Eğitim

Her şey hazır olunca eğitim, Hugging Face Trainer ile birkaç satırdır: `TrainingArguments` hiperparametreleri (learning rate, batch size, epoch) tutar, `Trainer` model + veri + metrik'i birleştirir (fastai'nin `Learner`'ına çok benzer).

```
from transformers import TrainingArguments, Trainer
args = TrainingArguments('outputs', learning_rate=8e-5, warmup_ratio=0.1,
    lr_scheduler_type='cosine', fp16=True, evaluation_strategy="epoch",
    per_device_train_batch_size=128, num_train_epochs=4, weight_decay=0.01)
model = AutoModelForSequenceClassification.from_pretrained(model_nm, num_labels=1)
trainer = Trainer(model, args, train_dataset=dds['train'], eval_dataset=dds['test'],
    tokenizer=tokz, compute_metrics=corr_d)
trainer.train()
```

💡 Builder Notu — Trainer ≈ Learner

- **Geriye (Ders 1):** Trainer ≈ Learner; `TrainingArguments` ≈ fastai'nin eğitim ayarları. Kavram aynı, API farklı — [Neden Farklı Kütüphane?](#) bölümündeki “kavramlar taşınır” iddiasının somut kanıtı.
- **İleriye (Part 2 Ders 18):** `lr_scheduler_type='cosine'` + `warmup_ratio` = OneCycle benzeri çizelge; learning rate yönetimi Part 2’de derinleşir. Ders 3’teki [learning rate](#) tartışmasının üretimdeki hâlidir.

11.16 Kaggle İş Akışı ve Pratik Özet

Howard, Kaggle’ı validation set becerisini geliştirmenin en iyi yeri olarak önerir: günde sınırlı gönderim + gizli test set, overfitting’i “iliklerinde hissetmeni” sağlar. Gerçek dünyada overfit ettiğini fark bile etmezsin; Kaggle’da birkaç kez “battıktan” sonra öğrenirsin.

“Kaggle competitions are a fantastic way to test your ability to create a good validation set.” — Howard, 53:39

💡 Builder Notu — Önce Validation Stratejisi, Sonra Model

- **İleriye:** “Önce başarı kriterini ve validation stratejisini tanımla, sonra modele başla” — goal-driven disiplin; sona değil başa konur. Ders 3’teki “[başarı kriterini baştan tanımla](#)” tavsiyesinin NLP’deki yankısıdır.
- **Geriye (Test Set):** Kaggle’ın gizli test set’i, Şekil 11.7’teki “dokunulmayan final kontrol” kutusunun gerçek hayattaki örneğidir — bu yüzden disiplin antrenmanı için ideal.

11.17 Kapanış

Ders 4 iki şeyi öğretti: (1) metni modele sokmanın mekaniği (tokenization → numericalization → pretrained model + fine-tune, Hugging Face ile), ve (2) tüm ML'in en kritik becerisi olan doğru validation/test set kurma. Mimari ve kütüphane değişse de bu ikinci ders her yerde geçerlidir.

💡 Builder Notu — Mekanik Eskir, Disiplin Kalır

- **İleriye:** Tokenizer ve Transformer Part 2'de (ve Karpathy'de) sıfırdan kurulacak; validation disiplini ise her projede taşınır — model değişir, “[iyi validation set](#)” sorusu hiç değişmez.
- **Geriye (Bu Derste Ne Var?):** Dersin başındaki üç fikir — [tokenization](#), [transfer learning](#), [iyi validation set](#) — burada tek bir cümlede kapanır.

11.18 Bu Dersin Özeti

1. NLP'de fastai yerine **Hugging Face Transformers** kullanılır; kavramlar (tokenize, model, fine-tune) aynıdır ([Neden Farklı Kütüphane?](#)).
2. **Tokenization** metni subword token'lara böler; modeli eğitenlerle aynı tokenizer kullanılmalıdır ([Tokenization](#)).
3. **Numericalization** her token'ı sözlük konumuna (input_ids) çevirir; model yalnızca sayıyla çalışır ([Numericalization](#)).
4. **Transfer learning** metne de uygulanır: pretrained model (deberta-v3-small) + fine-tune; ULMFiT bunun RNN tabanlı öncüsüdür ([Fine-tuning](#), [ULMFiT](#)).
5. **Overfitting** (ezber) ve **underfitting** (yetersiz) arasında denge gerekir; polinom derecesi bunu somutlaştırır ([Overfitting ve Underfitting](#)).
6. **Validation set** overfitting'i yakalar; fastai onu zorunlu kılar ([Validation Set Neden Zorunlu?](#)).
7. **İyi validation set rastgele değildir** — geleceği/yeni kişileri tahmin edecek şekilde kurulmalı; aksi halde sessizce değer yok edilir ([İyi Bir Validation Set Kurmak](#)).
8. **Test set** üçüncü ayrımdır; validation üzerinde overfit etmediğini garantiler. **Metric** (insan hedefi) ile **loss** (model hedefi) farklıdır ([Test Set, Metric vs Loss](#)).

❗ Tek Bir Cümle

NLP'de metni token'lara bölüp sayıya çevirir ve pretrained bir modeli ince ayarlarsın; ama bir modelin gerçekten işe yarayıp yaramadığını yalnızca doğru kurulmuş bir validation/test set söyler — bu, tüm makine öğrenmesinin en kritik becerisidir.

11.19 Kontrol Soruları

i Soru 1: Tokenization ile numericalization arasındaki fark nedir? Neden modeli eğitenlerle aynı tokenizer kullanılmalı?

Cevap:

Tokenization metni parçalara (subword token'lara) böler; numericalization her token'ı sözlükteki (vocabulary) konumuna, yani bir sayıya (`input_ids`) çevirir. İkisi birlikte metni modelin anlayacağı sayı dizisine dönüştürür. Aynı tokenizer şart çünkü her model belirli bir sözlük ve bölme kuralıyla eğitildi; farklı bir tokenizer farklı token'lar üretir ve modelin öğrendiği sözlük konumlarıyla uyuşmaz — sonuç anlamsız olur. `AutoTokenizer.from_pretrained(model_nm)` doğru olanı otomatik indirir.

i Soru 2: Howard neden “rastgele bir validation set çoğu zaman yanlıştır” der? Bir örnek ver.

Cevap:

Çünkü validation set, modeli **gerçekte nasıl kullanacağını** taklit etmelidir. Zaman serisinde geleceği tahmin edeceksen, validation set son dönem olmalı (ortadan rastgele değil) — yoksa “geçmiş tahmin” etmeyi ölçersin, oysa geleceği tahmin edeceksin. Howard'ın dikkati dağılmış sürücü örneğinde test set eğitimde olmayan kişiler içeriyordu; rastgele ayırsaydın aynı kişinin bazı fotoğrafları hem eğitimde hem validation'da olur, model kişiyi tanıyıp aldatici yüksek skor verirdi. İyi validation set, eğitimden gerçekten farklı (gelecek/yeni kişi) örnekler içermelidir.

i Soru 3: Test set neden validation set'ten ayrı gerekir? “Validation set üzerinde overfit” ne demektir?

Cevap:

Model geliştirirken yüzlerce varyant denersin ve her birini validation set'te ölçersin. Bu kadar çok deneme arasında bazıları **salt şans eseri** validation set'te iyi çıkar — sen de o modeli seçersin. Bu, “validation set üzerinde overfit”tir: validation skorunu kendi karar verme sürecinle dolaylı olarak ezberlemişsindir. Test set, geliştirme boyunca **hiç dokunmadığın** üçüncü bir ayırmadır; yalnızca en sonda bakarsın ve böylece gerçek performansı öğrenirsin. Kötü çıkarsa başa dönmek gerekir — ama bilmek bilmemekten iyidir.

i Soru 4: Metric ile loss neden farklı olabilir? (builder bağlantısı)

Cevap:

Loss, gradient descent'in optimize ettiği şeydir; bu yüzden **pürüzsüz ve türevlenebilir** olmalı (örn. MSE, cross-entropy). Metric ise insanın/işin umursadığı şeydir (örn. doğruluk, Pearson korelasyonu, gelir) ve genellikle türevlenemez ya da optimize için uygun değildir. Bu yarışmada loss MSE benzeri, metric Pearson korelasyonudur. Builder açısından: modeli loss'a göre eğitirsin ama başarıyı metric'e göre yargıyorsun — ikisini karıştırmak yanlış model seçimine yol açar.

11.20 Egzersizler

Egzersiz 1 (Direkt uygulama). Bir Hugging Face tokenizer yükle (`AutoTokenizer.from_pretrained`) ve kendi cümlelerini tokenize et; subword bölünmelerini ve `input_ids`'i incele.

Egzersiz 2 (İki-aşamalı). Kendi metin verinde context/target alanlarını tek bir input string’inde birleştir, Dataset’e çevir ve map ile tokenize et.

Egzersiz 3 (Edge case). plot_poly ile 1, 2, 5, 16 derece polinomları çiz; underfit→tam→overfit geçişini ve eğitim verisine ezberlemeyi gözlemler.

Egzersiz 4 (Validation tasarımı). Bir zaman serisi veride hem rastgele hem “son dönemi ayır” validation set kur; iki yaklaşımın aynı modeli nasıl farklı değerlendirdiğini açıkla.

Egzersiz 5 (Sonraki dersin habercisi). Bir Trainer kurup küçük bir veride birkaç epoch eğit; compute_metrics ile Pearson’u izle, validation skoru eğitim skorundan ayrışınca (overfit) durdur.

11.21 Sonraki Ders İçin Hazırlık

Ders 5: Sıfırdan Model (From-scratch model)

Ders 4’te Hugging Face Trainer’ı kara kutu olarak kullandık. Ders 5 tekrar aşağı iner: bir tablo verisi (Titanic) üzerinde önce **lineer modeli ve sinir ağını sıfırdan** kurar, sonra neden framework kullanmamız gerektiğini gösterir.

Ana konular:

- Tablo verisini hazırlama (eksik değer, kategorik kodlama)
- Lineer modeli sıfırdan (matris çarpımı + gradient descent)
- Sinir ağına genişletme
- Framework’ün değeri



Ders 5 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 4 — iyi validation set tasarla).
- Bir Kaggle yarışmasına gir ve gizli test set deneyimini yaşa.
- Ana cümleyi tekrar oku: “İyi validation set, ML’in en kritik becerisidir.”

11.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard’da
Tokenization	Metni subword token’lara bölme	29:30
Vocabulary	Tüm benzersiz token’ların listesi	30:17
Numericalization	Token’ı sözlük konumuna (input_ids) çevirme	38:58
AutoTokenizer	Modelin sözlüğünü ve tokenize kuralını indirir	33:26
ULMFiT	RNN tabanlı NLP transfer learning; uzun belgelere uygun	43:27
Underfitting	Yetersiz kapasite; sistematik yanlış	45:55

Kavram	Tanım	Howard'da
Overfitting	Eğitim verisini ezberleme; yeni veride kötü	45:55
Validation set	Eğitimde görülmeyen, doğruluk ölçen ayırım	50:56
İyi validation set	Rastgele değil; geleceği/yeni kişileri test eder	52:32
Test set	Geliştirmede kullanılmayan üçüncü gizli ayırım	57:17
Metric vs loss	İnsan hedefi vs modelin optimize ettiği pürüzsüz şey	59:05
Trainer	HF'in Learner'ı; model + veri + metrik	59:05

11.23 ML Builder Bağlantıları

💡 Builder Notu — 6 ML Köprüsü

Bu ders, NLP mekaniğini (tokenization → numericalization → fine-tune) öğretir ama asıl dersi tüm ML'e taşınan validation/test disiplindir; köprülerin özeti:

1. **Tokenization/numericalization** → metni sayıya çeviren standart NLP ön işlemi; Karpathy Ders 9 (BPE) sıfırdan kurar ([Tokenization](#)).
2. **Transfer learning (metin)** → pretrained dil modeli + fine-tune; ULMFiT öncüsü, modern Transformers standardı ([Fine-tuning](#)).
3. **Overfitting/underfitting** → bias-variance dengesi (Stat 110); her modelde merkez sorun ([Overfitting ve Underfitting](#)).
4. **İyi validation set** → production'da sessiz başarısızlığı önleyen tek savunma; data leakage'ın panzehiri ([İyi Bir Validation Set Kurmak](#)).
5. **Test set** → çoklu deneme şansını eler; bilimsel değerlendirmenin temeli ([Test Set](#)).
6. **Metric vs loss** → loss optimize edilebilir, metric iş hedefini yansıtır; model seçimini doğru metric belirler ([Metric vs Loss](#)).

❗ Bu dersten tek bir şey alıp gideceksen

Bu dersten tek bir şey alıp gideceksen: en gelişmiş model bile kötü bir validation set'le değersizdir. Metni sayıya çevirip pretrained bir modeli ince ayarlamak artık birkaç satır; asıl ustalık, modeli gerçek dünyada nasıl kullanacağını taklit eden bir validation/test set kurmaktır — yoksa overfit ettiğini fark bile etmeden değer yok edersin.

12 Sıfırdan Model — Titanic’te Lineer Modelden Derin Ağa (From-scratch model)

Bir tablo veriyi (Titanic) sayıya çevir — eksik değer doldur, kategorikleri dummy variable’a çevir, normalize et — sonra önce bir lineer modeli (matris çarpımı + gradient descent), ardından sigmoid’i ve bir sinir ağını (matris + ReLU + matris) hiçbir model katmanı kütüphanesi olmadan, sadece tensör işlemleriyle sıfırdan kur; Howard’ın Excel’de gördüğü her şeyi Python’a taşıdığı, fast.ai’nin ’büyü’sünü çözen ders

i Bölüm bilgisi

- **Howard’ın videosu:** [course.fast.ai — Lesson 5: From-scratch model](https://course.fast.ai/lesson5.html) (~103 dk)
- **Seri:** Practical Deep Learning for Coders — Part 1, Ders 5
- **Hoca:** Jeremy Howard
- **Playlist:** [Practical Deep Learning Part 1 \(2022\)](#)
- **Notebook:** [course22 — 05-linear-model-and-neural-net-from-scratch](#)
- **Okuma süresi:** ~35 dk

12.1 Bu Derste Ne Var?

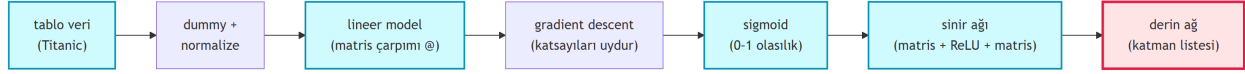
Ders 3’te bir modelin gradient descent ile uydurulan esnek bir fonksiyon olduğunu görmüştük; Ders 4’te ise Hugging Face Trainer’ı kara kutu olarak kullandık. Ders 5 tekrar aşağı iner ve bunu **gerçek bir tablo verisinde sıfırdan** kurar: Titanic veri setinde önce bir lineer modeli, sonra bir sinir ağını — hiçbir hazır model katmanı kullanmadan, sadece tensör işlemleriyle. Howard Excel’de gördüğü her şeyi Python’a taşır.

Üç temel fikir:

1. **Tablo verisini sayıya çevirme** — eksik değer doldurma, kategorik sütunları dummy variable’a (one-hot) çevirme, ölçekleri eşitleme (normalizasyon) (Titanic Verisi → Kategorik Veri → Normalizasyon).
2. **Lineer model = (girdi × katsayı) toplamı** — bu bir matris çarpımıdır; gradient descent’le katsayıları uydururuz, türevi PyTorch hesaplar (Lineer Modeli Kurmak → Matris Çarpımı).
3. **Sinir ağı = matris çarpımı + ReLU + tekrar** — lineer modeli iki katmana genişletip ReLU eklemek “sıfırdan derin öğrenme”yi tamamlar (Bir Sinir Ağı → Derin Ağa Genişletme).

“We’re going to build a couple of different types of tabular models from scratch.” — Howard, 0:13

12 Sıfırdan Model — Titanic’te Lineer Modelden Derin Ağa (From-scratch model)



Şekil 12.1: Ders 5’in kavram haritası: tablo veri → dummy + normalize → lineer model (matris çarpımı @) → gradient descent → sigmoid → sinir ağı (matris + ReLU + matris) → derin ağ (rose vurgulu). Tek bir zincir: veriyi sayıya çevir, ardışık matris çarpımlarından ReLU ile geçir, gradient descent ile katsayıları uydur — hepsi sıfırdan.

Şekil 34.1 bu üç fikri tek bir haritada birleştirir: tablo veriden başlayıp dummy + normalize ile sayıya geçer, lineer modele (matris çarpımı) ulaşır, gradient descent ile katsayıları uydurur, sigmoid’le olasılığa çevirir ve son olarak matris + ReLU + matris ile sinir ağına, oradan da derin ağa (rose vurgulu) genişler.

💡 Builder Notu — Karpathy’nin Yolu, Howard’ın Diliyle

- **Geriye (Ders 3 / Karpathy / 18.06):** Ders 3’ün gradient descent’i burada [gerçek veriye uygulanır](#); @ (matris çarpımı) 18.06’nın kalbidir; Karpathy makemore aynı “sıfırdan” ruhu — bu ders, fast.ai top-down’ın bir an için Karpathy bottom-up’a döndüğü yerdir.
- **İleriye (Ders 6):** “Sıfırdan yapmak öğretir ama yavaştır” — [bir sonraki ders](#) framework’ün (fastai) neden gerekli olduğunu gösterir.
- **Tek cümle:** Bir sinir ağı, tablo verisinde bile yalnızca matris çarpımları, ReLU ve gradient descent’ten ibarettir.

12.2 Sıfırdan Tabular Model

Howard tablo (spreadsheet/veritabanı) verisine döner ve modeli **sıfırdan** kurar. Excel’in neredeyse her şeyi yapabildiğini ama pratikte Python kullandığımızı belirtir — Excel küçük örnekleri sezdirmek için iyidir, gerçek iş Python’da yapılır.

“Excel can do nearly everything we need to build a neural network, but we don’t tend to use Excel in practice — instead we use a programming language like Python.” — Howard, 1:30

💡 Builder Notu — Önce Elle Sez, Sonra Kodda Ölçekle

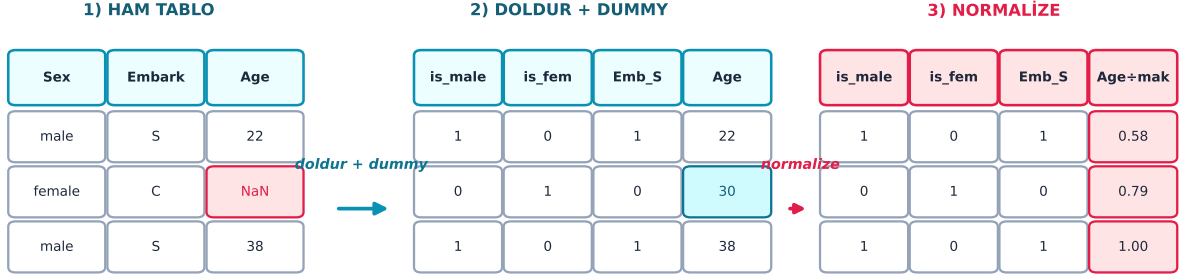
- **İleriye:** “Önce küçük ölçekte (Excel) sezgi, sonra kodda ölçek” — Howard’ın pedagojik deseni; karmaşık fikirleri elle görmek anlayışı kökleştirir. Aynı desen, [sinir ağını sıfırdan](#) kurarken zirvesine ulaşır.
- **Geriye (Ders 4):** Ders 4’te kütüphane (Hugging Face) merkezdeyken, burada bilinçli olarak kütüphaneyi bırakıp [Learner/Trainer’in içinde ne olduğunu](#) elle açıyoruz.

12.3 Titanic Verisi ve Eksik Değerler

İlk iş veriyi hazırlamaktır. Gerçek veri eksik (NaN) değerler içerir. Howard satır veya sütun atmaz; bunun yerine eksik değerleri o sütunun **moduyla** (en sık değer) doldurur. Böylece hiç veri kaybetmeyiz.

“I don’t throw out rows and I don’t throw out columns.” — Howard, 14:54

Şekil 12.2 bu hazırlığın ilk iki adımını (eksik değeri doldur, kategorikleri dummy’ye çevir) ve üçüncü adımı (normalize) tek bir akışta gösterir — soldaki kırmızı NaN hücresi, ortada modla doldurulur.



model yalnızca 0-1 aralığı sayı ister: kategorik metni 0/1'e, büyük sayıyı maks'a böl

Şekil 12.2: Tablo verisini sayıya çevirme (şematik): eksik değerleri modla (en sık değer) doldur, kategorikleri dummy variable’a (one-hot 0/1) çevir, sayısal sütunları normalize et (maks’a böl → 0-1). Model yalnızca 0-1 aralığı sayıyla çalışır. (şematik — kavramsal akış)

💡 Builder Notu — Atma, Doldur

- **İleriye (production):** Eksik değer stratejisi (doldur, işaretle, at) modeli ciddi etkiler; “at” çoğu zaman en kötüsüdür çünkü bilgi kaybeder. Şekil 12.2’in soldaki rose NaN hücresi, ortada modla doldurulup korunur — satır kaybetmeyiz.
- **İleriye (Kategorik Veri):** Doldurma sayısal sütun içindir; metin (kategorik) sütunları sayıya çevirmek için bambaşka bir araç (dummy variable) gerekir — sıradaki bölüm.

12.4 Kategorik Veri: Dummy Variable

Bir modeli “male” kelimesiyle veya “S” harfiyle çarpamayız — model sayı ister. Çözüm **dummy variable** (one-hot kodlama): her kategori için 0/1 değerli ayrı bir sütun. Cinsiyet, yolcu sınıfı, biniş limanı gibi kategorik alanlar bu şekilde sayısallaşır.

“We cannot multiply the letter ‘S’ by a coefficient. What we do is we create something called dummy variables.” — Howard, 19:09

Şekil 12.2’in ortadaki sütunu (is_male / is_fem / Emb_S) tam bu dönüşümü gösterir: tek Sex metni iki 0/1 sütununa açılır.

 Builder Notu — Her Kategori = Bir 0/1 Sütun

- **Geriye (Python/pandas):** `pd.get_dummies` kategorik sütunları otomatik açar; “her kategori = bir 0/1 sütunu” temsili tüm tablo ML’inin temelidir.
- **İleriye (Lineer Modeli Kurmak):** Her dummy sütununa model ayrı bir katsayı öğrenir; “is_male” sütununun katsayısı, cinsiyetin hayatta kalmaya etkisini doğrudan yakalar.

12.5 Ölçekleri Eşitlemek (Normalizasyon)

Sütunlar çok farklı büyüklüklerde olabilir — “Age” diğerlerinden çok daha büyüktür. Eğer normalize etmezsek, büyük sütun katsayıları domine eder. Howard her sütunu maksimumuna bölerek 0-1 aralığına çeker. Bu işlem broadcasting ile çok optimize biçimde yapılır.

“ ‘Age’ is much bigger than any of the other columns. So what we could do is divide by this vector, in a very optimized way.” — Howard, 35:22

Şekil 12.2’in en sağdaki rose sütunu ($\text{Age} \div \text{mak}$) bu adımı gösterir: $22/38 \rightarrow 0.58$, $38/38 \rightarrow 1.00$ gibi, tüm değerler 0–1 aralığına çekilir.

 Builder Notu — Ölçek Eşitlenmezse Büyük Sütun Domine Eder

- **Geriye (Stat 110):** Normalizasyon, özniteliklerin karşılaştırılabilir ölçekte olmasını sağlar; standardizasyonun basit hâli.
- **İleriye (Broadcasting):** “vektöre böl” ifadesi bir broadcasting işlemidir — 891 satırın hepsine aynı maks-vektörünü uygular; aynı mekanik birazdan lineer modelin kalbinde tekrar görünür.

12.6 Lineer Modeli Kurmak

Lineer model basittir: her sütunu bir **katsayı** ile çarp, satır boyunca topla. Howard katsayıları rastgele (-0.5 ile 0.5 arası) başlatır; tahmin = (girdi $\times$ katsayı) satır toplamı; kayıp = ortalama mutlak hata.

```
torch.manual_seed(442)
n_coeff = t_indep.shape[1]
coeffs = torch.rand(n_coeff) - 0.5          # rastgele baslangic katsayilari

preds = (t_indep * coeffs).sum(axis=1)      # her satir icin agirlikli toplam
loss = torch.abs(preds - t_dep).mean()      # ortalama mutlak hata
```

 Builder Notu — Quadratic’in Çok Değişkenli Hâli

- **Geriye (Ders 3):** Bu, Ders 3’teki **quadratic’in** çok değişkenli hâli; katsayılar = parametreler. Tek bir $a \cdot x^2 + b \cdot x + c$ yerine artık her sütunun bir katsayısı var.
- **Geriye (18.06):** (girdi $\times$ katsayı) toplamı bir **dot product**; tüm satırlar için yapılıncaya matris-vektör çarpımı — bu eşitlik **Matris Çarpımı** bölümünde açıkça kurulacak.

12.7 Broadcasting

Howard `t_indep * coeffs` ifadesinin nasıl çalıştığını açıklar: 891×12 'lik bir matrisi 12'lik bir vektörle çarpıyoruz. **Broadcasting**, vektörü her satıra otomatik uygular (sanki 891 kez kopyalanmış gibi) — hem kısa kod hem optimize C kodu, hatta GPU'da paralel.

“This is using the incredibly powerful technique of broadcasting. It's as if this was broadcast 891 times. And that all happened in optimized C code.” — Howard, 46:55

💡 Builder Notu — Açık Döngü Yerine Yayılma

- **Geriye (18.06 / NumPy):** Broadcasting, farklı şekilli tensörleri hizalama kuralıdır; “son eksenler uyuşursa yayılır.” Aynı kural [normalizasyondaki](#) “vektöre böl” işlemini de optimize eder.
- **İleriye:** Açık for döngüsü yerine broadcasting, modern sayısal hesabın hız sırrıdır; aynı satır toplamı birazdan tek bir @ ile [matris çarpımına](#) dönüşecek.

12.8 Gradient Descent Adımı

Ders 3'teki döngünün aynısı: katsayılara `requires_grad_()`, kaybı hesapla, `backward()` ile gradyanları al, katsayıları gradyanın tersine küçük adım kaydır, sonra gradyanı **sıfırla** (yoksa birikir).

```
loss = calc_loss(coeffs, t_indep, t_dep)
loss.backward()
with torch.no_grad():
    coeffs.sub_(coeffs.grad * 0.1) # gradyanın tersine adım
    coeffs.grad.zero_()           # gradyani sıfırla (birikmesin)
```

“The one bit we don't want to do from scratch is calculating derivatives — PyTorch does that for us.” — Howard, 39:44

💡 Builder Notu — `grad.zero_()` Karpathy'nin Meşhur Tuzağı

- **Geriye (Karpathy):** `grad.zero_()` unutulursa gradyanlar birikir — Karpathy'nin meşhur `zero_grad` hatasının ta kendisi; aynı satır birazdan [eğitim döngüsünde](#) bir fonksiyona sarılır.
- **İleriye:** “türevi PyTorch hesaplar” — sıfırdan yapmanın tek istisnası budur; geri kalan her şey (forward, loss, güncelleme) elle yazılır.

12.9 Eğitim Döngüsü

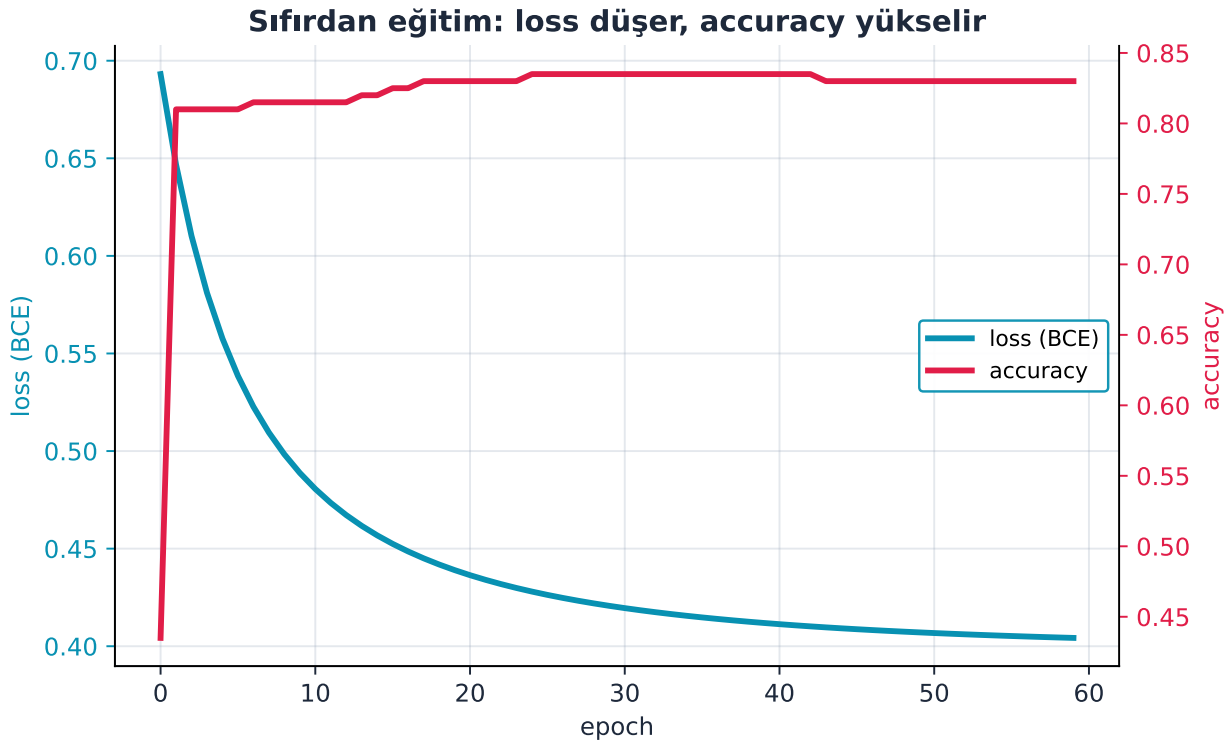
Howard adımları fonksiyonlara sarar: `update_coeffs` (adım + sıfırla), `one_epoch` (kayıp + backward + güncelle), `train_model` (katsayıları başlat + epoch'lar boyunca döndür). Bu, her ML eğitim döngüsünün iskeletidir.

```
def update_coeffs(coeffs, lr):
    coeffs.sub_(coeffs.grad * lr); coeffs.grad.zero_()

def one_epoch(coeffs, lr):
    loss = calc_loss(coeffs, trn_indep, trn_dep)
    loss.backward()
    with torch.no_grad(): update_coeffs(coeffs, lr)

def train_model(epochs=30, lr=0.01):
    torch.manual_seed(442)
    coeffs = init_coeffs()
    for i in range(epochs): one_epoch(coeffs, lr=lr)
    return coeffs
```

Şekil 12.3 bu döngünün davranışını gerçek bir sıfırdan eğitimde gösterir: her epoch'ta loss düşer, accuracy yükselir — `loss.backward() + coeffs -= grad·lr + grad.zero_()` üçlüsünün sonucu.



`loss.backward() + coeffs -= grad·lr + grad.zero_()`

Şekil 12.3: Sıfırdan gradient descent eğitimi (gerçek hesaplama — lojistik regresyon): her epoch'ta loss (BCE) düşer ve accuracy yükselir. `loss.backward()` ile gradyan, `coeffs -= grad·lr` ile güncelleme, `grad.zero_()` ile sıfırlama.

💡 Builder Notu — Learner ve Trainer’ın Çıplak Hâli

- **Geriye (Ders 1-4):** fastai Learner ve [Hugging Face Trainer](#) tam bu döngüyü senin için sarar; burada onu elle görüyoruz — Şekil 12.3’deki loss-aşağı / accuracy-yukarı eğrisi, o kara kutuların içinde olan biten şeydir.
- **İleriye (Sigmoid):** Şekil 12.3’deki accuracy, çıktıyı bir sigmoid’dan geçirince ek bir sıçrama yapar; sıradaki bölümler önce doğruluğu ölçmeyi, sonra sigmoid’i ekler.

12.10 Doğruluğu Ölçmek

Kayıp optimizasyon içindir; insan için **doğruluk** (accuracy) daha anlamlıdır. Howard tahmin 0.5’ten büyükse “hayatta kaldı” sayar ve gerçeğe karşılaştırır. Validation set’te ölçülür (eğitimde değil).

```
def acc(coeffs):
    return (val_dep.bool() == (calc_preds(coeffs, val_indep) > 0.5)).float().mean()
```

💡 Builder Notu — Loss Optimize İçin, Accuracy İnsan İçin

- **Geriye (Ders 4):** Metric (accuracy) ile loss (MAE) farkı; bu tam Ders 4’teki [metric vs loss](#) ayrımıdır — loss’a göre eğit, accuracy’ye göre yargıla.
- **Geriye (Ders 4):** Doğruluk yalnızca validation set’te ölçülür ([overfitting kontrolü](#)) — eğitim doğruluğu modelin gerçek başarısını söylemez.

12.11 Sigmoid

Lineer modelin çıktısı herhangi bir sayı olabilir, ama bir olasılık (0-1) istiyoruz. Çözüm **sigmoid** fonksiyonu: her sayıyı yumuşakça 0 ile 1 arasına sıkıştırır. Howard bunu ekleyince doğruluk 0.79’dan 0.82’ye çıkar.

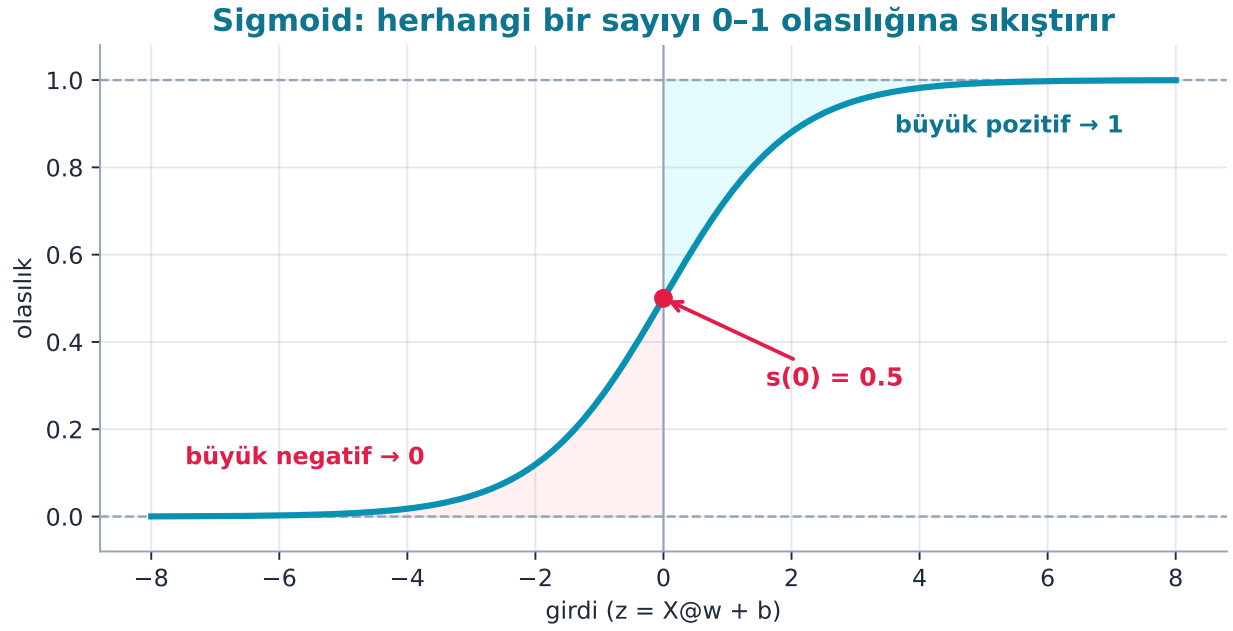
“If you’re wondering why is my model not very good, often the thing you need is chucking it through a sigmoid.” — Howard, 51:00

```
def calc_preds(coeffs, indeps):
    return torch.sigmoid((indeps * coeffs).sum(axis=1)) # ciktiyi 0-1'e sikistir
```

Şekil 12.4 bu fonksiyonun S şeklini gösterir: büyük negatif girdi 0’a, büyük pozitif girdi 1’e yaklaşır, merkez $s(0) = 0.5$. Matematiksel olarak $s(x) = \frac{1}{1 + e^{-x}}$.

💡 Builder Notu — Çıkış Aktivasyonu Her Modelde Var

- **Geriye (6.S191 / Calculus):** Sigmoid, 6.S191 Ders 1’in aktivasyon fonksiyonu (§4.D); S şeklinde, e^x tabanlı. Şekil 12.4’in merkez noktası $s(0) = 0.5$ tam karar eşiğidir.
- **İleriye:** Sınıflandırmada çıktıyı olasılığa çevirmek standarttır; ikili için sigmoid, çoklu için



büyük pozitif $\rightarrow 1$, büyük negatif $\rightarrow 0$; ikili sınıflandırmada çıkış aktivasyonu

Şekil 12.4: Sigmoid (gerçek hesaplama): her gerçek sayıyı yumuşakça 0 ile 1 arasına sıkıştırır (büyük pozitif $\rightarrow 1$, büyük negatif $\rightarrow 0$). Lineer çıktı olasılığa çevrilir; Howard'ın Titanic sıfırdan modelinde doğruluğu 0.79 $\rightarrow$ 0.82'ye taşıyan adım.

softmax. Bu sigmoid, [sinir ağının](#) ve [derin ağın](#) son adımında da yerini korur.

12.12 Kaggle’a Gönderme

Howard modeli test setine uygular ve tahminleri Kaggle formatında bir CSV’ye yazar. Sıfırdan kurulmuş bir lineer model bile makul bir sonuç verir — “from scratch” çalışıyor.

“Actually predicting who survived the Titanic. That’s cool — from scratch.” — Howard, 47:00

💡 Builder Notu — Deploy’un Kaggle Versiyonu

- **Geriye (Ders 2):** Bu, [deploy’un](#) Kaggle versiyonu — modeli yeni veride çalıştırıp çıktı üretme; eğittiğin model ancak yeni veriye uygulanınca değer üretir.
- **İleriye (Matris Çarpımı):** Bu noktaya kadar lineer model bir satır toplamıyla yazıldı; Howard birazdan onu tek bir @ operatörüne sadeleştirip sinir ağına zemin hazırlar.

12.13 Matris Çarpımı (@)

`(indeps * coeffs).sum(axis=1)` ifadesi aslında bir **matris çarpımı**dır. Python’ın @ operatörü tam bunu yapar (PyTorch tensörlerinde optimize biçimde). Howard kodu sadeleştirir ve sinir ağına hazırlanır.

```
val_indep @ coeffs          # (indep * coeffs).sum(axis=1) ile aynı
def calc_preds(coeffs, indeps): return torch.sigmoid(indeps @ coeffs)
```

“@ is an official Python operator. It means matrix-multiply. Because these are tensors, that will use PyTorch’s, and they’re exactly the same.” — Howard, 59:34

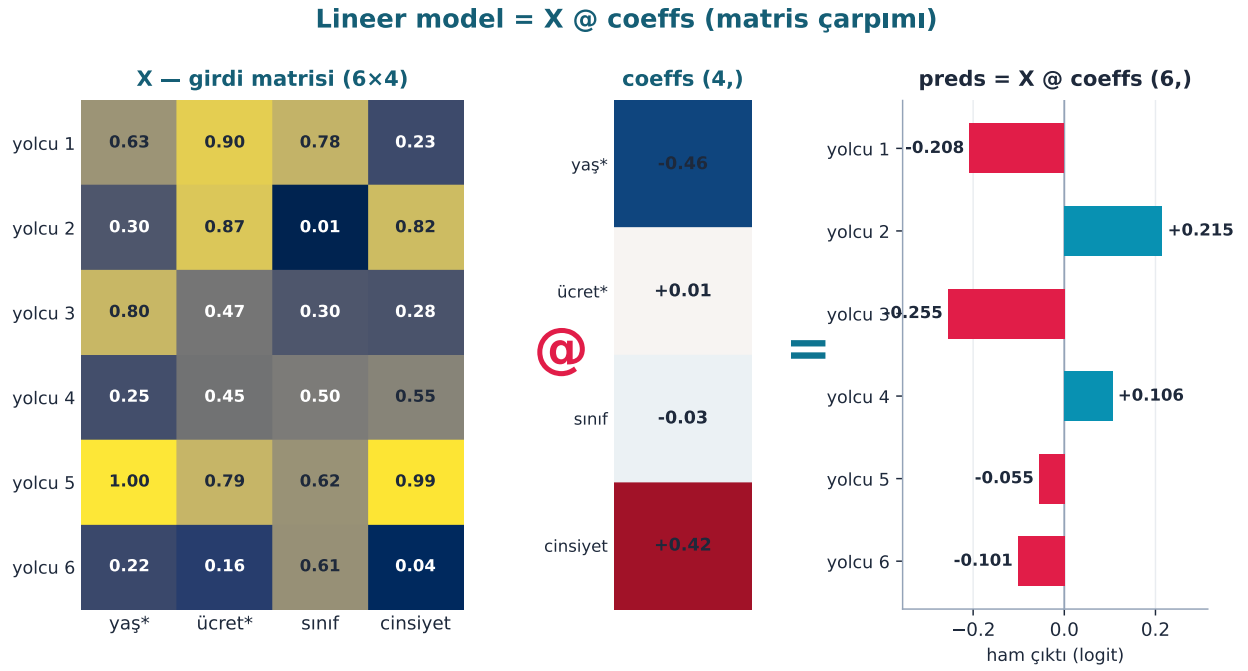
Şekil 12.5 bu eşitliği gerçek sayılarla gösterir: 6×4’lük bir X matrisi, 4’lük bir katsayı vektörüyle çarpılınca her satır için tek bir çıktı (`X @ coeffs`) üretir — `(X * coeffs).sum(axis=1)` ile birebir aynı.

💡 Builder Notu — @ Lineer Cebirin Kalbi

- **Geriye (18.06):** @ = matris çarpımı, lineer cebirin kalbi; her sinir ağı katmanı bir matris çarpımıdır. Şekil 12.5’deki tek katsayı sütunu (4×1), birazdan bir matrise (4×20) genişleyecek.
- **İleriye (Bir Sinir Ağı):** Vektörü “n×1 matris”e (sütun vektör) çevirmek, birden çok katsayı sütunu (yani gizli katman) eklemeye zemin hazırlar — sinir ağının tek satırlık adımı.

12.14 Bir Sinir Ağı

İşte zirve: matris çarpımına geçtikten sonra “çıldırıp” bir sinir ağı kurabiliriz. Tek katsayı sütunu yerine bir **katsayı matrisi** (`n_coeff × n_hidden`) kullanırız; her satır artık 20 değer (gizli aktivasyon) üretir. Bunları ReLU’dan geçirip ikinci bir katmanla tek çıktıya indirip sigmoid uygularız.



Şekil 12.5: Lineer model = matris çarpımı (gerçek hesaplama): her satır (yolcu) özellikleri katsayılarla çarpılıp toplanır — X @ coeffs. (girdi × katsayı).sum(axis=1) ile aynı sonucu verir; broadcasting/@ bu satır toplamını optimize eder.

```
def init_coeffs(n_hidden=20):
    layer1 = (torch.rand(n_coeff, n_hidden) - 0.5) / n_hidden
    layer2 = torch.rand(n_hidden, 1) - 0.3
    const = torch.rand(1)[0]
    return layer1.requires_grad_(), layer2.requires_grad_(), const.requires_grad_()

import torch.nn.functional as F
def calc_preds(coeffs, indeps):
    l1, l2, const = coeffs
    res = F.relu(indeps @ l1)  # 1. katman + ReLU (dogrusal-olmama)
    res = res @ l2 + const    # 2. katman -> tek cikti
    return torch.sigmoid(res)
```

“Now that we’ve done this using matrix-multiply, we can go crazy, and we can go ahead and create a neural network.” — Howard, 1:03:26

Şekil 12.6 lineer modelden sinir ağına ve derin ağa geçişi üç satırda gösterir: lineer = tek matris çarpımı; sinir ağı = matris @ + ReLU + matris @ + sigmoid (n_hidden=20 gizli birim); derin ağ = bu adımı bir katman listesinde tekrarlamak.

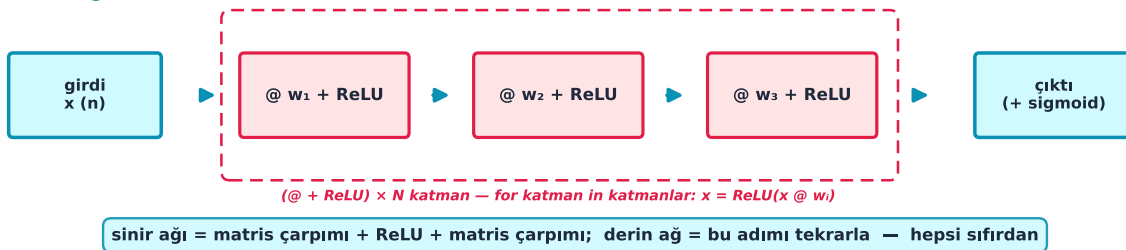
1 • Lineer model



2 • Sinir ağı



3 • Derin ağ



Şekil 12.6: Lineer modelden derin ağa (şematik): lineer model = tek matris çarpımı; sinir ağı = matris çarpımı + ReLU + ikinci matris çarpımı (n_hidden=20 gizli birim); derin ağ = bu adımı bir katman listesinde tekrarlamak — hepsi sıfırdan.

💡 Builder Notu — Bu Tam Ders 3’ün ReLU Toplamı

- **Geriye (Ders 3):** Bu tam Ders 3’ün “ReLU toplamı”dır — sadece matrislerle ifade edilmiş; $n\_hidden=20$ yani 20 ReLU birimi. Şekil 12.6’ın 2. satırı (rose ReLU kutusu) bu dönüşümün tam görüntüsüdür.
- **Geriye (Karpathy):** $layer1/layer2/ReLU = Karpathy \text{ makemore MLP’sinin elle hâli; aynı } W1 @ x \rightarrow ReLU \rightarrow W2 @ \cdot \text{ zinciri.}$

12.15 Derin Ağa Genişletme

Howard son adımı atar: iki katman yerine **istediğin kadar** katman. Bir katman listesi üzerinde döngü kurar, her birinde matris çarpımı + (son hariç) ReLU uygular. Artık bu gerçek bir **derin** sinir ağıdır — hepsi sıfırdan.

```
def calc_preds(coeffs, indeps):
    layers, consts = coeffs
    n = len(layers)
    res = indeps
    for i, l in enumerate(layers):
        res = res @ l + consts[i]
        if i != n-1: res = F.relu(res)    # son katman haric ReLU
    return torch.sigmoid(res)
```

Şekil 12.6’ın en alt satırı (rose noktalı çerçeveye sarılan $(@ + ReLU) \times N$ bloğu) tam bu döngüyü gösterir: bir katman listesi üzerinde matris çarpımı + ReLU.

💡 Builder Notu — nn.Sequential’in Özü

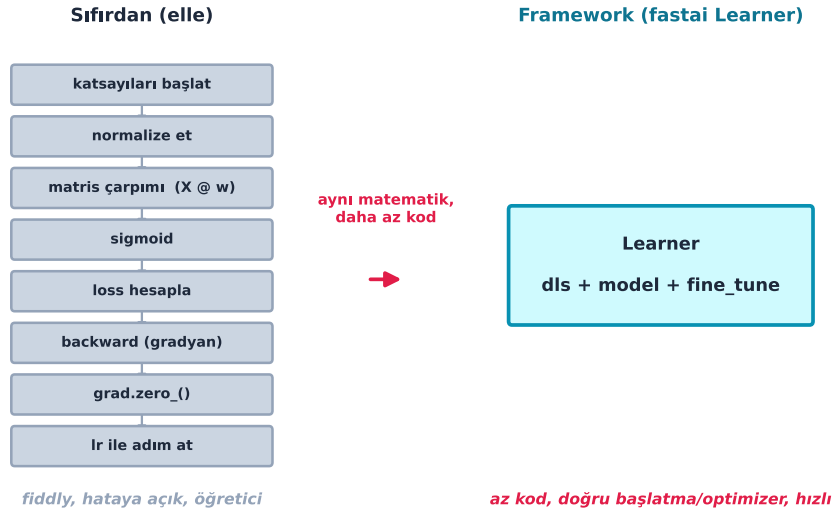
- **İleriye:** Bu döngü, PyTorch’un nn.Sequential’inin özüdür; “katman listesi üzerinde matris çarpımı + aktivasyon” tüm derin ağların iskeleti — Şekil 34.1’in en sağdaki rose “derin ağ” düğümü buraya işaret eder.
- **Geriye (Sinir Ağı):** İki katmanı genelleştirmenin tek farkı bir for döngüsüdür; mimari aynı (matris + ReLU), sadece tekrar sayısı değişken.

12.16 “Sıfırdan Derin Öğrenme” ve Framework

Howard büyük tabloyu kapatır: Titanic’te bir derin sinir ağını yalnızca tensör işlemleriyle, hiçbir model katmanı kütüphanesi olmadan kurduk. Ama elle yapmak zahmetlidir — katsayı başlatma, gradyan ölçeği, fiddly ayarlar. Bir sonraki ders neden framework (fastai) kullandığımızı gösterecek.

“When you do stuff by hand, everything does get more fiddly.” — Howard, 1:05:02

Şekil 12.7 bu kontrastı gösterir: solda elle yapılan 8 adımlık fiddly liste (başlat → normalize → matris çarpımı → sigmoid → loss → backward → grad.zero_ → adım), sağda aynı matematiği az ve sağlam kodla saran fastai Learner.



Şekil 12.7: Sıfırdan vs framework (şematik): elle yapmak her parçayı (başlatma, normalize, matris çarpımı, sigmoid, loss, backward, grad.zero_, lr adımı) gösterir ama fiddly’dir; framework (fastai Learner) aynı matematiği az ve sağlam kodla yapar. Sıfırdan bilmek neyin otomatikleştiğini anlatır.

💡 Builder Notu — Sıfırdan Bilmek = Neyin Otomatikleştiğini Bilmek

- **İleriye (Ders 6):** Framework, başlatma/optimizer/veri yükleme gibi “fiddly” parçaları doğru ve hızlı yapar; sıfırdan bilmek ise neyi otomatikleştirdiğini anlamayı sağlar. Şekil 12.7’ün sol panelindeki 8 adım, sağdaki tek Learner kutusunun içinde gizlidir.
- **Geriye (Eğitim Döngüsü):** Soldaki “backward → grad.zero_ → adım” üçlüsü, *elle yazdığımız one_epoch* fonksiyonunun ta kendisidir; framework onu görünmez kılar.

12.17 Kapanış

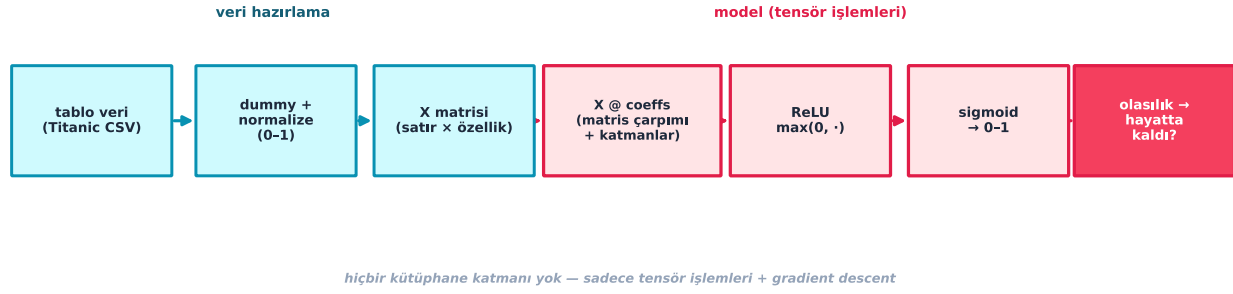
Ders 5, Ders 3’ün soyut çekirdeğini gerçek bir tablo verisinde somutlaştırdı: veri hazırlama (dummy + normalize) → lineer model (matris çarpımı + gradient descent) → sigmoid → sinir ağı (katman + ReLU) → derin ağ. Hiçbir sihir yok; sadece matris çarpımları, ReLU ve gradient descent.

Şekil 12.8 bu uçtan uca akışı tek bir şeritte özetler: Titanic CSV’den başlayıp dummy + normalize, X matrisi, matris çarpımı + katmanlar, ReLU, sigmoid ve sonunda hayatta-kalma olasılığına — hiçbir kütüphane katmanı olmadan.

💡 Builder Notu — Top-Down ile Bottom-Up’ın Buluştuğu Yer

- **Geriye (tüm seri):** Bu, fast.ai top-down ile Karpathy bottom-up’ın buluştuğu yer — Howard burada bir an için Karpathy moduna geçip her şeyi elle kurar. Şekil 12.8’un soldan sağa cyan→rose

12 Sıfırdan Model — Titanic’te Lineer Modelden Derin Ağa (From-scratch model)



Şekil 12.8: Titanic sıfırdan model — uçtan uca (şematik): tablo veri → dummy+normalize → X matrisi → matris çarpımı ($X @ coeffs$ + katmanlar) → ReLU → sigmoid → hayatta-kalma olasılığı. Hiçbir model katmanı kütüphanesi yok, sadece tensör işlemleri + gradient descent.

şeridi, [kavram haritasının](#) doğrusal hâlidir.

- **İleriye (Ders 6):** Bu uçtan uca elle kurulan zincir, framework’le birkaç satıra inecek; ama “ne otomatikleşti?” sorusunun cevabı tam burada, Şekil 12.8’de görünür kalır.

12.18 Bu Dersin Özeti

1. Tablo verisi önce sayıya çevrilir: eksik değerleri **modla doldur**, kategorikleri **dummy variable**’a çevir, sütunları **normalize** et ([Titanic Verisi](#), [Kategorik Veri](#), [Normalizasyon](#)).
2. **Lineer model** = (girdi × katsayı) satır toplamı; katsayılar gradient descent’le uydurulur ([Lineer Modeli Kurmak](#)).
3. **Broadcasting** bir vektörü tüm satırlara optimize biçimde uygular; açık döngüden çok daha hızlı ([Broadcasting](#)).
4. Eğitim döngüsü: kayıp → `backward()` → katsayıları güncelle → `grad.zero_()`; PyTorch yalnızca türevi hesaplar ([Gradient Descent Adımı](#), [Eğitim Döngüsü](#)).
5. **Sigmoid** çıktıyı 0-1 olasılığına sıkıştırır; sınıflandırmada doğruluğu artırır ([Sigmoid](#)).
6. **Matris çarpımı (@)** lineer katmanın özüdür; vektörü sütun matrise çevirmek gizli katman eklemeye zemin hazırlar ([Matris Çarpımı](#)).
7. **Sinir ağı** = matris çarpımı + ReLU + ikinci matris çarpımı; `n_hidden` gizli birim sayısıdır ([Bir Sinir Ağı](#)).
8. **Derin ağ** = bu adımı katman listesi üzerinde tekrarlamak — hepsi sıfırdan, sadece tensör işlemleriyle ([Derin Ağa Genişletme](#)).

! Tek Bir Cümle

Bir derin sinir ağı, tablo verisinde bile, normalize edilmiş sayıların ardışık matris çarpımları + ReLU’dan geçirilip gradient descent ile katsayılarının uydurulmasından ibarettir — fast.ai’nin gizlediği “büyü” budur.

12.19 Kontrol Soruları

i Soru 1: Kategorik bir sütunu (örn. cinsiyet) neden doğrudan modele veremeyiz? Çözüm nedir?

Cevap:

Model girdiyi bir katsayıyla çarpar; “male” veya “S” gibi metni sayıyla çarpamayız. Çözüm **dummy variable** (one-hot kodlama): her kategori için 0/1 değerli ayrı bir sütun açarız. Örneğin cinsiyet → “is_male” ve “is_female” sütunları (biri 1, diğeri 0). Böylece kategorik bilgi sayısal hâle gelir ve model her kategoriye ayrı bir katsayı öğrenebilir.

i Soru 2: Lineer modeli sinir ağına dönüştüren tam olarak nedir? Ders 3’ün ‘ReLU toplamı’yla bağlantısı nedir?

Cevap:

İki değişiklik: (1) tek katsayı vektörü yerine bir katsayı **matrisi** ($n_{\text{coeff}} \times n_{\text{hidden}}$) kullanmak — her satır artık tek değil n_{hidden} (örn. 20) değer üretir; (2) bu değerleri **ReLU**’dan geçirip ikinci bir katmanla (matris çarpımı) tek çıktıya indirmek. Bu, Ders 3’teki “yeterince ReLU topla” fikrinin matrislerle ifade edilmiş hâlidir: 20 gizli birim = 20 ReLU; `F.relu(indeps@11)` onları üretir, `res@12` toplar. Yani sinir ağı = matris çarpımı + ReLU + matris çarpımı.

i Soru 3: `grad.zero_()` neden gereklidir? Atlanırsa ne olur?

Cevap:

PyTorch’ta `backward()` gradyanları mevcut `.grad` değerinin **üstüne ekler** (biriktirir), üzerine yazmaz. Her gradient descent adımından sonra `grad.zero_()` ile sıfırlamazsan, bir sonraki adımın gradyanı öncekilerle toplanır ve güncelleme yanlış (çok büyük) olur — model düzgün eğitilmez. Bu, Karpathy’nin de vurguladığı meşhur `zero_grad` hatasıdır; production eğitim döngülerinde de sık görülen bir bug kaynağıdır.

i Soru 4: Sigmoid neden eklenir ve ne işe yarar? (builder bağlantısı)

Cevap:

Lineer modelin (veya son katmanın) çıktısı herhangi bir gerçek sayı olabilir (negatif, çok büyük). Ama ikili sınıflandırmada bir **olasılık** (0-1 arası) isteriz. **Sigmoid** her sayıyı yumuşakça 0 ile 1 arasına sıkıştırır (büyük pozitif → 1’e, büyük negatif → 0’a yakın). Howard’da bunu eklemek doğruluğu 0.79’dan 0.82’ye çıkardı. Builder açısından: çıktıyı görevin gerektirdiği aralığa eşleyen “çıkış aktivasyonu” her modelde standarttır — ikili için sigmoid, çok-sınıflı için softmax.

12.20 Egzersizler

Egzersiz 1 (Direkt uygulama). Titanic verisinde lineer modeli sıfırdan kur (`coeffs`, `calc_preds`, `calc_loss`) ve `train_model` ile eğit; doğruluğu validation set’te ölç.

Egzersiz 2 (İki-aşamalı). Sigmoid’i ekle/çıkar ve doğruluğun nasıl değiştiğini kıyasla; learning rate’i sigmoid’li/sigmoid’siz ayrı ayarla.

Egzersiz 3 (Edge case). `grad.zero_()` satırını kasıtlı olarak sil ve eğitimin nasıl bozulduğunu (loss'un patlaması) gözlemlle.

Egzersiz 4 (Python ile doğrulama). `(indeps * coeffs).sum(axis=1)` ile `indeps @ coeffs` sonuçlarının birebir aynı olduğunu doğrula; broadcasting ile matris çarpımı arasındaki ilişkiyi açıkla.

Egzersiz 5 (Sonraki dersin habercisi). Aynı modeli `fastai`'nin `tabular_learner`'ı ile kur ve sıfırdan yazdığın kodla sonuçları/kod miktarını kıyasla (Ders 6 framework).

12.21 Sonraki Ders İçin Hazırlık

Ders 6: Rastgele Ormanlar (Random forests)

Ders 5'te her şeyi elle yaptık. Ders 6 önce framework'ün (`fastai`) değerini gösterir, sonra tamamen farklı ama çok güçlü bir model ailesini tanıtır: **random forests** ve gradient boosting — tablo verisinde sinir ağına güçlü bir alternatif.

Ana konular:

- Framework neden gerekli (`fastai` ile az kod)
- Karar ağaçları ve random forest
- Feature importance
- Out-of-bag hatası



Ders 6 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 3 — sıfırdan eğit, `zero_` hatasını gör).
- Titanic modelini kendi notebook'unda baştan kur.
- Ana cümleyi tekrar oku: “Sinir ağı = matris çarpımı + ReLU + gradient descent.”

12.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Eksik değer doldurma	NaN'ları modla (en sık değer) doldur, satır/sütun atma	14:54
Dummy variable	Kategoriyi 0/1 sütunlara açma (one-hot)	19:09
Normalizasyon	Sütunları benzer ölçeğe çekme (maks'a bölme)	35:22
Lineer model	(girdi × katsayı) satır toplamı	26:53
Broadcasting	Vektörü tüm satırlara optimize uygulama	46:55
<code>grad.zero_()</code>	Her adımdan sonra gradyanı sıfırlama (birikmesin)	39:44
Sigmoid	Çıktıyı 0-1 olasılığına sıkıştırma	48:17

Kavram	Tanım	Howard'da
Matris çarpımı (@)	Lineer katmanın özü; dot product'ların matrisi	58:29
Gizli katman (n_hidden)	Ara aktivasyonlar; ReLU'dan geçen ara çıktılar	1:03:34
Sinir ağı	matris çarpımı + ReLU + matris çarpımı	1:03:34
Derin ağ	Katman listesi üzerinde matris çarpımı + ReLU	1:06:01
accuracy	Tahmin > 0.5 ile gerçeğin eşleşme oranı (validation)	46:11

12.23 ML Builder Bağlantıları

💡 Builder Notu — 6 ML Köprüsü

Bu ders, fast.ai top-down'ı bir an için Karpathy bottom-up'a çevirir: tablo veriyi sayıya çevirip sıfırdan bir derin ağ kurar. Köprülerin özeti:

1. **Sıfırdan model** → fast.ai'nin gizlediği her şeyin elle hâli; Karpathy makemore ile aynı ruh ([Sıfırdan Tabular Model](#)).
2. **Matris çarpımı (@)** → her sinir ağı katmanının özü (18.06); broadcasting ile birlikte hız sırrı ([Matris Çarpımı](#), [Broadcasting](#)).
3. **Sinir ağı = matris + ReLU + matris** → Ders 3'ün "ReLU toplamı"nın somut kodu ([Bir Sinir Ağı](#)).
4. **grad.zero_()** → gradyan birikimi tuzağı; Karpathy'nin meşhur hatası, production bug kaynağı ([Gradient Descent Adımı](#)).
5. **Sigmoid/softmax** → çıkış aktivasyonu; çıktıyı görevin aralığına eşler ([Sigmoid](#)).
6. **Dummy + normalizasyon** → tablo verisini sayısallaştırmanın standart ön işlemi ([Kategorik Veri](#), [Normalizasyon](#)).

❗ Bu dersten tek bir şey alıp gideceksen

Bu dersten tek bir şey alıp gideceksen: bir derin sinir ağında gizem yoktur. Veriyi sayıya çevir, ardışık matris çarpımlarından ReLU ile geçir, gradient descent ile katsayıları uydur. Howard bunu Titanic'te hiçbir kütüphane katmanı kullanmadan kurdu — fast.ai'nin "büyü"sü tam olarak budur, sadece daha az ve daha sağlam kodla.

13 Rastgele Ormanlar — Karar Ağacından Random Forest’a (Random forests)

Ders 5’te tablo verisinde bir sinir ağını sıfırdan kurduk; Ders 6 tamamen farklı ama tablo verisinde çoğu zaman daha güçlü bir aileyi tanıtır — karar ağaçlarını (ardışık ikili bölmeler) ve onların topluluğu random forest’ı: tek bir bölmeden Gini saflık ölçüsüne, bagging’e (Leo Breiman: çok sayıda bağımsız yansız modelin ortalaması), oradan random forest’a, feature importance’a, OOB hatasına ve gradient boosting’e — tablo verisinde ‘bozması neredeyse imkânsız’, hızlı, sağlam ve yorumlanabilir bir temel

Bölüm bilgisi

- **Howard’ın videosu:** [course.fast.ai — Lesson 6: Random forests](https://course.fast.ai/lesson6.html) (~103 dk)
- **Seri:** Practical Deep Learning for Coders — Part 1, Ders 6
- **Hoca:** Jeremy Howard
- **Playlist:** [Practical Deep Learning Part 1 \(2022\)](#)
- **Notebook:** [course22 — 07-how-random-forests-really-work](#)
- **Okuma süresi:** ~35 dk

13.1 Bu Derste Ne Var?

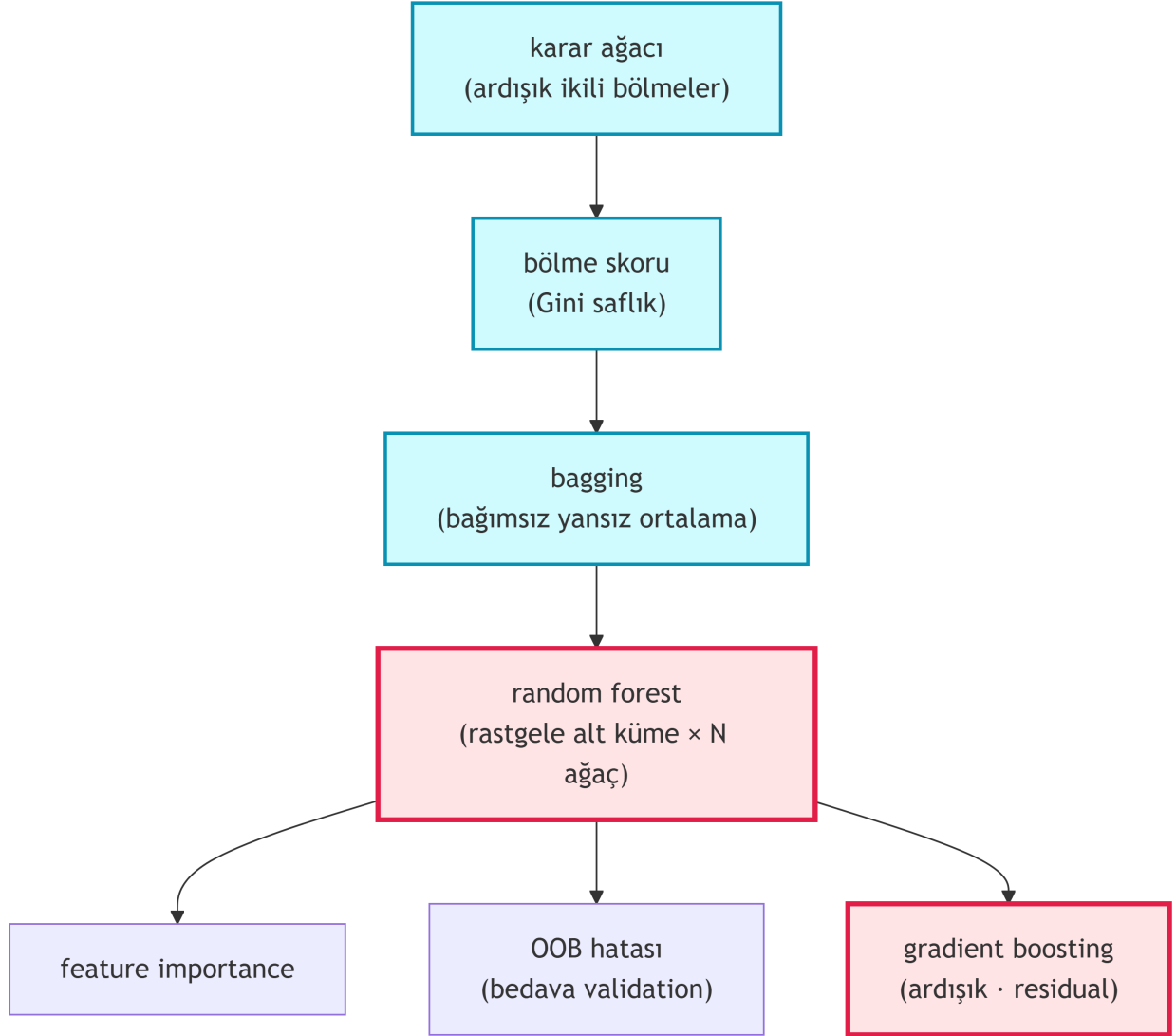
Ders 5’te tablo verisinde sinir ağını sıfırdan kurduk. Ders 6 tamamen farklı ama tablo verisinde çoğu zaman daha güçlü bir aileyi tanıtır: **karar ağaçları (decision trees)** ve onların topluluğu **random forests**. Howard adım adım, bir karar ağacından bagging’e, oradan random forest’a ve gradient boosting’e ilerler.

Üç temel fikir:

1. **Karar ağacı = ardışık ikili bölmeler** — veriyi en iyi ayıran sorulara göre dallara böler (örn. “kadın mı?”); derin öğrenme değildir, neredeyse hiç ön işleme istemez ([Karar Ağaçları](#) → [İlk Bölme](#) → [Bölme Skoru](#) → [Karar Ağacı Kurma](#)).
2. **Bagging** — çok sayıda **bağımsız, yansız** (ama kusurlu) modelin ortalaması, tek tek hepsinden iyidir; çünkü hatalar birbirini götürür (Leo Breiman) ([Bagging](#)).
3. **Random forest = bagging + karar ağaçları** — verinin rastgele alt kümeleriyle yüzlerce ağaç kur, ortalamasını al; “bozması neredeyse imkânsız” sağlam bir başlangıç ([Random Forest Fikri](#) → [Random Forest Kurma](#)).

“Leo Breiman came up with this idea called bagging.” — Howard, 16:52

Şekil 34.1 bu üç fikri tek bir haritada birleştirir: tek bir karar ağacından (bölme skoru = Gini) başlayıp bagging'e, oradan random forest'a (feature importance + OOB ile) ve son olarak güçlü akrabası gradient



Şekil 13.1: Ders 6’nın kavram haritası: karar ağacı (ardışık ikili bölmeler) → bölme skoru (Gini saflık) → bagging (bağımsız yansız modellerin ortalaması, Leo Breiman) → random forest (rastgele alt kümelerle çok ağaç) → feature importance + OOB hatası → gradient boosting (ardışık ağaçlar, rose vurgulu). Tek bir zincir: bir ağacı saflıkla böl, çok ağacı rastgelelikle çoğalt, ortalamasını al — ya da hatayı ardışık kovala.

💡 Builder Notu — Tablo Verisinde Ağaçlar, Sinir Ağına Karşı

- **Geriye (Stat 110 / Ders 5):** Bagging, “bağımsız yansız tahminlerin ortalaması varyansı düşürür” ilkesidir — istatistiğin örneklem ortalaması fikri; [Ders 5’in sıfırdan sinir ağı](#) tablo verisinde tek araçtı, bu ders ona güçlü bir alternatif koyar.

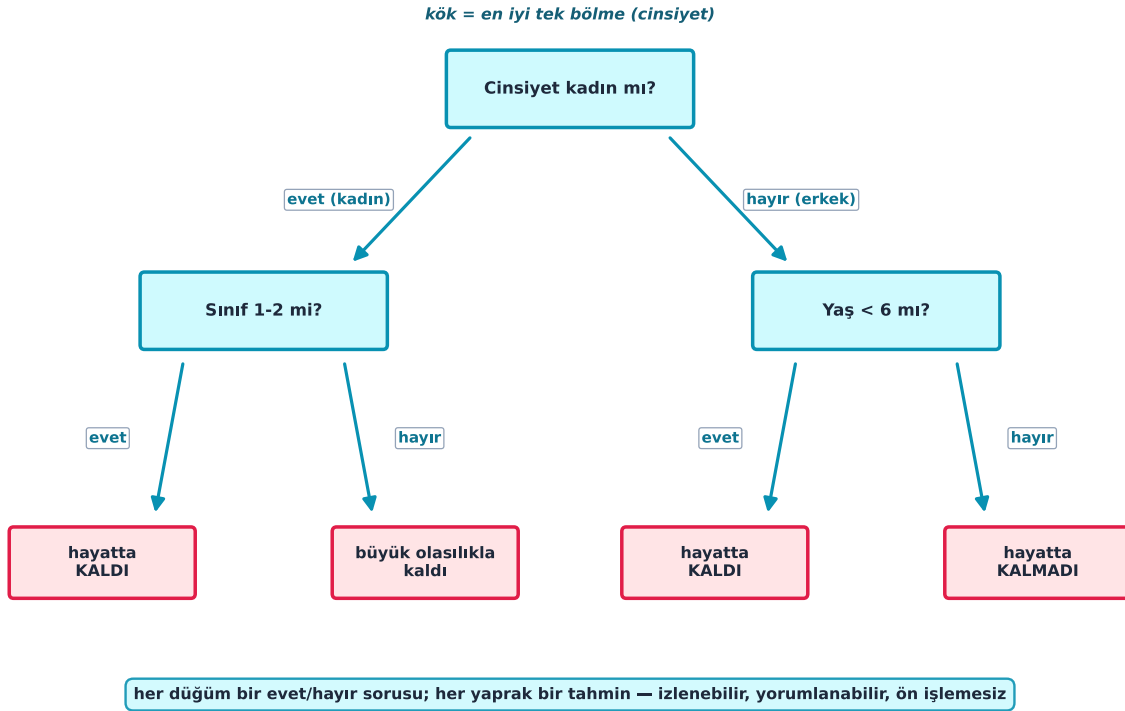
- **İleriye (production):** Random forest, tablo verisinde hızlı, sağlam ve yorumlanabilir bir temel; çoğu gerçek iş probleminde ilk denenecek model — sinir ağına koşmadan önce.
- **Tek cümle:** Random forest, çok sayıda kusurlu karar ağacının ortalamasını alarak güçlü, sağlam ve yorumlanabilir bir tablo modeli üretir.

13.2 Karar Ağaçları: Derin Öğrenme Olmayan ML

Howard derin öğrenmeden bir an için ayrılır ve **karar ağaçlarını** tanıtır: veriyi giderek ayıran bir dizi ikili bölme (örn. “cinsiyet kadın mı?”, “yaş < 6 mı?”). Bu, sinir ağı değil; klasik makine öğrenmesidir.

“Non-deep learning ish machine learning methods like decision trees.” — Howard, 5:09

Şekil 13.2 küçük bir Titanic karar ağacını gösterir: kökten başlayıp “kadın mı?” ve “yaş < 6 mı?” gibi ikili sorularla dallanır; her yaprak bir tahmindir (hayatta kaldı / kalmadı). Aynı ağaç [İlk Bölme](#), [Bölme Skoru](#) ve [Karar Ağacı Kurma](#) bölümlerinin somut görüntüsüdür.



Şekil 13.2: Karar ağacı (şematik — Titanic): kökten başlayan ardışık ikili bölmeler. ‘Cinsiyet kadın mı?’ en iyi ilk bölme; sonra her dal yeni bir eşik sorusuyla (yaş < 6?, sınıf < 2.5?) ikiye ayrılır. Yapraklar (rose) bir tahmin verir — hayatta kaldı / kalmadı. Derin öğrenme yok, sadece evet/hayır soruları.

💡 Builder Notu — Araç Çantasında İki Yöntem

- **İleriye:** Tablo verisinde karar ağacı tabanlı yöntemler çoğu zaman sinir ağından daha iyi ve çok daha hızlıdır; builder’ın araç çantasında ikisi de bulunmalı — Şekil 13.2’deki evet/hayır zinciri,

Ders 5'in matris çarpımı + ReLU yığınınına bambaşka bir alternatiftir.

- **Geriye (Ders 5):** Ders 5'te tablo için tek seçenek sıfırdan sinir ağıydı; bu ders aynı Titanic verisine ağaçla yaklaşır — [az ön işlemeyle](#) başlamak çok daha hızlıdır.

13.3 İlk Bölme: Bir Kuralla Başlamak

Howard en basit modelle başlar: tek bir bölme. Titanic'te en iyi tek ayırım cinsiyettir — kadınların çoğu hayatta kaldı, erkeklerin çoğu kalmadı. Soru: hangi sütun, hangi eşikte veriyi en iyi ikiye ayırır?

Şekil 13.2'nin kök düğümü ("Cinsiyet kadın mı?") tam bu tek bölmedir — ağacın geri kalanı bu ilk kuralın üstüne kurulur. Bu "en iyi"yi nesnel kılmak için bir skor gerekir, sıradaki [Bölme Skoru](#) bölümünün konusu.

💡 Builder Notu — Confusion Matrix Sezgisinin Kök Hâli

- **Geriye (Ders 2):** Bu, [Ders 2'deki confusion matrix sezgisinin](#) kök hâli — hangi özellik sınıfları en iyi ayırır; tek bölme, "en ayırıcı sütun" sorusunun en yalın cevabıdır.
- **İleriye (Karar Ağacı Kurma):** Tek bölmeyi tüm sütun/eşik kombinasyonları üzerinde aramak elle yorucudur; sklearn bunu otomatikleştirir — ama önce "iyi bölme" sayıya dökülmeli.

13.4 Bölme Skoru

"En iyi bölme"yi nesnel kılmak için bir skor gerekir. Howard her olası bölmeyi, iki tarafın **standart sapmasının** (ağırlıklı) toplamıyla puanlar: bir bölme iki tarafı ne kadar "saf" (homojen) yaparsa o kadar iyidir.

```
def _side_score(side, y):  
    tot = side.sum()  
    if tot <= 1: return 0  
    return y[side].std() * tot    # bir tarafın standart sapması x boyutu
```

Her olası "sütun + eşik" çifti için iki tarafın skorunu toplar, en düşük toplamı (en saf iki taraf) veren bölmeyi seçeriz. Şekil 13.2'deki her düğüm, bu skorla seçilmiş bir bölmedir; [Gini](#) bölümünde aynı saflık fikrinin sınıflandırmaya özel hâlini göreceğiz.

💡 Builder Notu — Standart Sapma = Homojenlik Ölçüsü

- **Geriye (Stat 110):** Standart sapma homojenlik ölçüsüdür; düşük std = saf grup. Bölme, toplam saflığı en çok artırmanı seçer — istatistiğin "varyansı azalt" sezgisinin ağaç hâli.
- **İleriye (Gini):** Regresyonda standart sapma; sınıflandırmada onun yerini Gini ($1 - \sum p^2$) alır — ikisi de aynı "iki tarafı saflaştır" amacına hizmet eder.

13.5 Karar Ağacı Kurma

Tüm bölmeleri elle aramak yerine scikit-learn'in `DecisionTreeClassifier`'ı bunu otomatik yapar. `max_leaf_nodes` ile ağacın büyüklüğünü sınırlarız; ağacı çizerek hangi soruları sorduğunu görebiliriz.

```
from sklearn.tree import DecisionTreeClassifier
m = DecisionTreeClassifier(max_leaf_nodes=4).fit(trn_xs, trn_y)
draw_tree(m, trn_xs, size=10)
```

“There’s no magic in here, just doing what we’ve just described.” — Howard, 5:09

`draw_tree`'nin çizdiği ağaç tam Şekil 13.2 gibidir: kökte en iyi bölme, dallarda ardışık eşik soruları, yapraklarda tahminler — hiçbir gizem yok, az önce [bölme skoruyla](#) anlattığımızın otomatik hâli.

💡 Builder Notu — Yorumlanabilir, İzlenebilir Model

- **İleriye (Model Yorumlama):** Karar ağacı yorumlanabilir bir modeldir — her tahmin, izlenebilir bir dizi evet/hayır sorusuyla açıklanır; bu yorumlanabilirlik random forest'a da taşınır.
- **Geriye (İlk Bölme):** `max_leaf_nodes=4`, ağacı [tek bölmenin](#) birkaç adım ötesine taşır; sınır koymazsak ağaç her satırı ezberler (overfit) — sınırlama saflık ile genelleme arasındaki dengeyi tutar.

13.6 Gini Saflık Ölçüsü

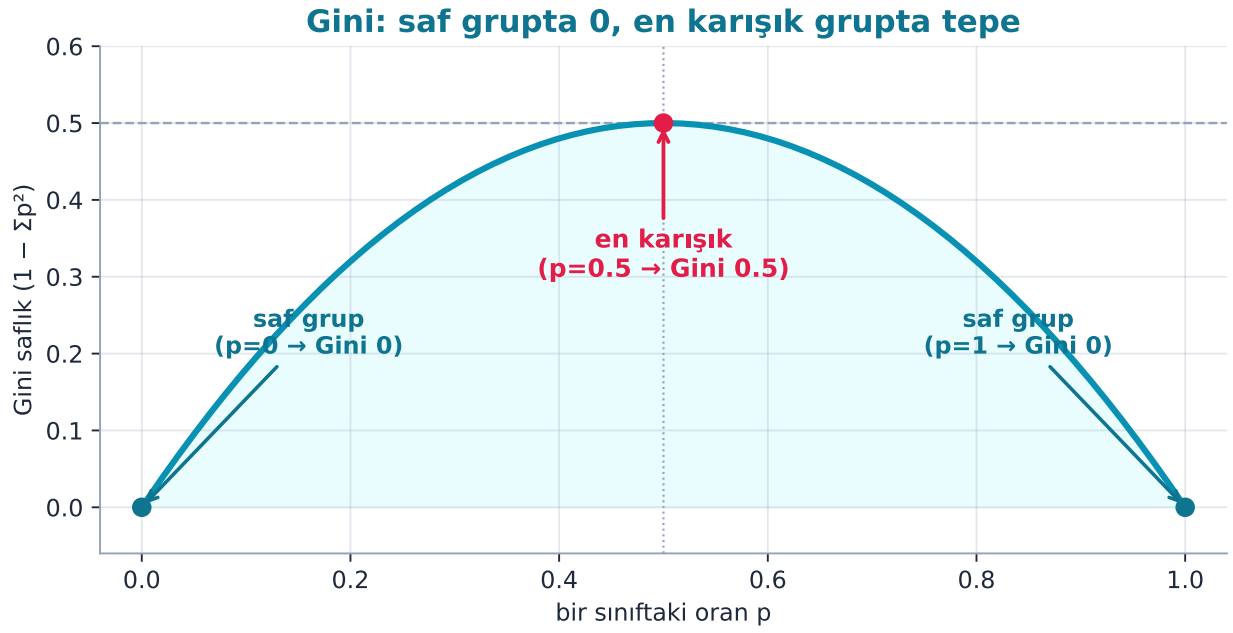
Sınıflandırmada saflık için sık kullanılan ölçü **Gini**'dir: bir gruptan rastgele iki örnek seçtiğinde farklı sınıfta olma olasılığı. Saf grupta (hepsi aynı) Gini düşük, karışık grupta yüksektir.

```
def gini(cond):
    act = df.loc[cond, dep]
    return 1 - act.mean()2 - (1 - act).mean()2
gini(df.Sex=='female'), gini(df.Sex=='male')
```

Şekil 13.3 bu ölçünün şeklini gerçek hesaplamayla gösterir: bir sınıftaki oran p değiştikçe $Gini = 1 - p^2 - (1 - p)^2 = 2p(1 - p)$ ters bir parabol çizer — saf grupta ($p = 0$ veya $p = 1$) sıfır, en karışık grupta ($p = 0.5$) tepe.

💡 Builder Notu — $Gini = 1 - \sum p^2$

- **Geriye (Stat 110):** $Gini = 1 - \sum p^2$; saf grupta olasılıkların karelerinin toplamı 1'e yaklaşır, Gini 0'a iner — Şekil 13.3'nin uç noktaları ($p = 0, 1$) tam bu sıfır saflığı gösterir.
- **İleriye (Feature Importance):** Bir sütunun “Gini'yi ne kadar düşürdüğü” birazdan feature importance'ın temel ölçütü olacak; Şekil 13.3'nin eğrisi o katkının nereden geldiğini anlatır.



bölme, toplam Gini'yi en çok düşüreni seçer

Şekil 13.3: Gini saflık ölçüsü (gerçek hesaplama): bir sınıftaki oran p için $Gini = 1 - p^2 - (1-p)^2 = 2p(1-p)$. Saf grupta ($p=0$ ya da $p=1$) Gini 0; en karışık grupta ($p=0.5$) tepe 0.5. Bir bölme, iki tarafın toplam Gini'sini en çok düşüreni seçer.

13.7 Az Ön İşleme: Karar Ağacının Avantajı

Howard kritik bir avantajı vurgular: karar ağaçları çok az ön işleme ister. Dummy variable oluşturabilirsin ama çoğu zaman gerekmez; normalizasyon gerekmez. Ağacın tek umursadığı, verinin **sıralamasıdır** (değerlerin büyüklüğü değil). Bu yüzden Howard tablo verisine her zaman bir karar ağacıyla başlar.

“All that a decision tree cares about is the ordering of the data. For tabular data I would always start by using a decision tree based approach.” — Howard, 17:53

💡 Builder Notu — Normalizasyon/Dummy Yükünü Kaldırır

- **Geriye (Ders 5):** Ders 5'te dummy + normalizasyon zorunluydu (lineer model için); karar ağacı bu yükü kaldırır — “Age < 24” eşiği, Age normalize edilse de edilmese de aynı bölmeyi verir, çünkü sadece sıralama önemlidir.
- **Geriye (Ders 5):** Lineer model/sinir ağı değerleri katsayılarla çarptığı için ölçek ve kodlamaya duyarlıdır; ağaç değeri eşiklerle böldüğü için duyarsızdır — başlamak çok hızlıdır.

13.8 Bagging

Tek bir karar ağacı ne kadar iyi olabilir ki? Howard, Leo Breiman'ın dâhiyane fikrini anlatır: **bagging**. Kusurlu ama **yansız** (sistemik olarak yüksek/düşük tahmin etmeyen) bir modelin hataları rastgeledir. Böyle çok sayıda **bağımsız** model üretilip ortalamaları alırsan, hatalar birbirini götürür ve ortalama, tek tek hepsinden iyi olur.

“The predictions of these multiple uncorrelated models — each of which is unbiased — will be better, because the average of the error will tend to zero.” — Howard, 16:52

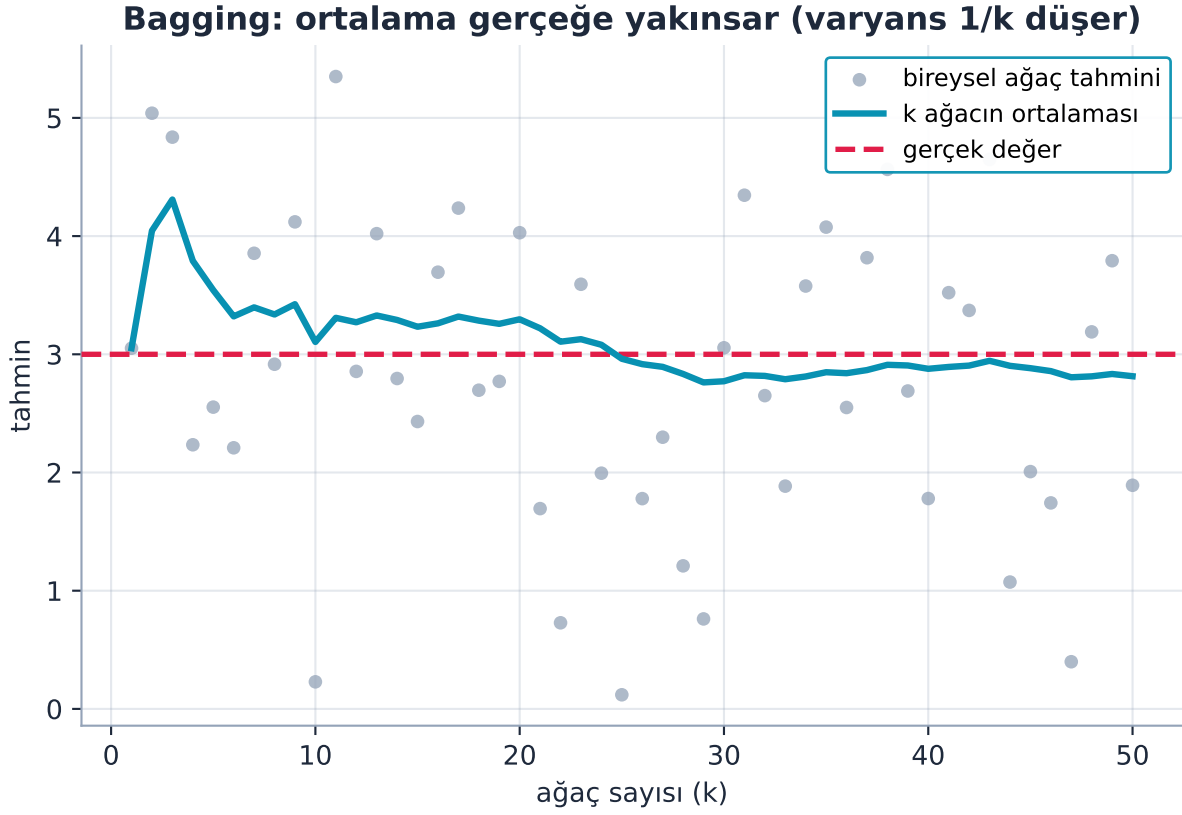
Şekil 13.4 bu yakınsamayı gerçek bir Monte Carlo simülasyonu ile gösterir: tek tek (gri) tahminler gerçek değer etrafında dağınıkken, k modelin ortalaması (cyan) gerçeğe yakınsar — varyans $1/k$ ile düşer.

💡 Builder Notu — Bagging = Büyük Sayılar Yasası

- **Geriye (Stat 110):** Bağımsız yansız tahminlerin ortalaması, varyansı örneklem sayısı ile düşürür (büyük sayılar yasası); bagging'in matematiksel temeli budur — Şekil 13.4'in $1/k$ daralması bu yasanın görsel kanıtıdır.
- **İleriye (Random Forest Fikri):** Bagging bir “meta yöntem”dir; onu karar ağaçlarına uygulayınca random forest çıkar — ama önce “bağımsız modeller nasıl üretilir?” sorusu yanıtlanmalı.

13.9 Random Forest Fikri

Çok sayıda **bağımsız** model nasıl üretilir? Basit: verinin her seferinde farklı bir rastgele alt kümesiyle (örn. %75) ayrı bir karar ağacı eğit. Her ağaç biraz farklı, hiçbiri mükemmel değil ama hepsi yansız. Bunların ortalaması bir **random forest**dir.

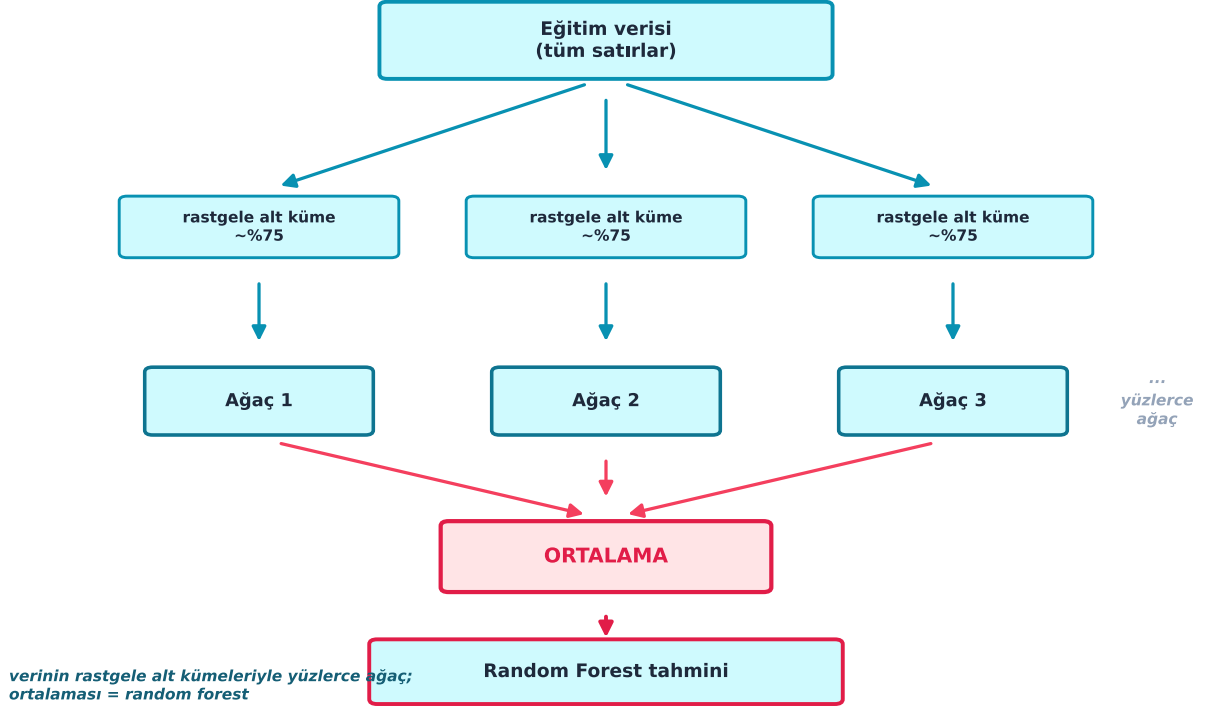


kusurlu ama yansız tahminler; hataları birbirini götürür (Leo Breiman)

Şekil 13.4: Bagging (gerçek hesaplama — Monte Carlo): bireysel tahminler (gri) gerçek değer etrafında dağınıktır (yansız ama gürültülü); k tahminin ortalaması (cyan) gerçeğe (rose kesik çizgi) yakınsar, varyans $1/k$ ile düşer. Leo Breiman'ın fikri: kusurlu ama yansız modellerin hataları birbirini götürür.

“Each of those decision trees is going to be not very good, but it will be unbiased. When you do this you create something called a Random Forest.” — Howard, 19:07

Şekil 13.5 bu mimariyi şematik olarak gösterir: tek bir eğitim verisinden rastgele alt kümeler çekilir, her birine bir ağaç eğitilir, sonra tüm ağaçların ortalaması (rose) alınır — bu ortalama random forest’ın tahminidir.



Şekil 13.5: Random forest (şematik): tek eğitim verisinden her seferinde farklı bir rastgele alt küme (~%75) çekilir, her birine ayrı bir karar ağacı eğitilir (cyan), sonra yüzlerce ağacın ortalaması (rose) alınır. Her ağaç kusurlu ama yansız; ortalamaları güçlü ve sağlamdır.

💡 Builder Notu — Bagging’in Karar Ağacına Uygulanmış Hâli

- **İleriye:** Bagging bir “meta yöntem”dir — modelleme değil, modelleri birleştirme yöntemi. Random forest, karar ağaçlarına uygulanmış bagging’dir; Şekil 13.5’in alt-küme → ağaç → ortalama akışı, Şekil 13.4’in yakınsamasının somut mimarisidir.
- **Geriye (Bagging):** Her ağacın farklı bir **rastgele alt kümeyle** eğitilmesi, ağaçları “bağımsız” kılan şeydir; bağımsızlık olmadan ortalamanın varyans-düşürme gücü çalışmaz.

13.10 Random Forest Kurma

Tüm bu fikir scikit-learn’de dört satırdır: kaç ağaç, yaprak başına kaç örnek, sonra fit.

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(100, min_samples_leaf=5)
rf.fit(trn_xs, trn_y)
mean_absolute_error(val_y, rf.predict(val_xs))
```

“So let’s create one in four lines of code.” — Howard, 20:18

Gerçek dünyada, makul boyuttaki veri setlerinde random forest neredeyse her zaman tek bir karar ağacından çok daha iyidir.

💡 Builder Notu — Classifier ve Regressor, Aynı Mantık

- **İleriye:** RandomForestClassifier (sınıflandırma) ve RandomForestRegressor (regresyon) — ikisi de aynı mantık; min_samples_leaf aşırı öğrenmeyi sınırlar (her yaprakta en az 5 örnek = ezber engeli).
- **Geriye (Ders 5):** [Ders 5’te sıfırdan yazdığımız onlarca satır](#) burada dört satıra iner; framework (sklearn) “fiddly” parçaları gizler — Howard’ın Ders 5 kapanışındaki “framework neden gerekli” sorusunun tablo-ağaç cevabı.

13.11 Feature Importance

Howard’ın en sevdiği özellik: **feature importance**. Her ağaçta, her sütunun Gini’yi ne kadar iyileştirdiğini (ve kaç kez seçildiğini) toplar, tüm ağaçlar üzerinde birleştirir. Sonuç: hangi özelliklerin önemli olduğunu gösteren bir grafik. Titanic’te cinsiyet ve sınıf açık ara önde çıkar.

```
import pandas as pd
pd.DataFrame(dict(cols=trn_xs.columns, imp=rf.feature_importances_)).plot('cols', 'imp', 'barh')
```

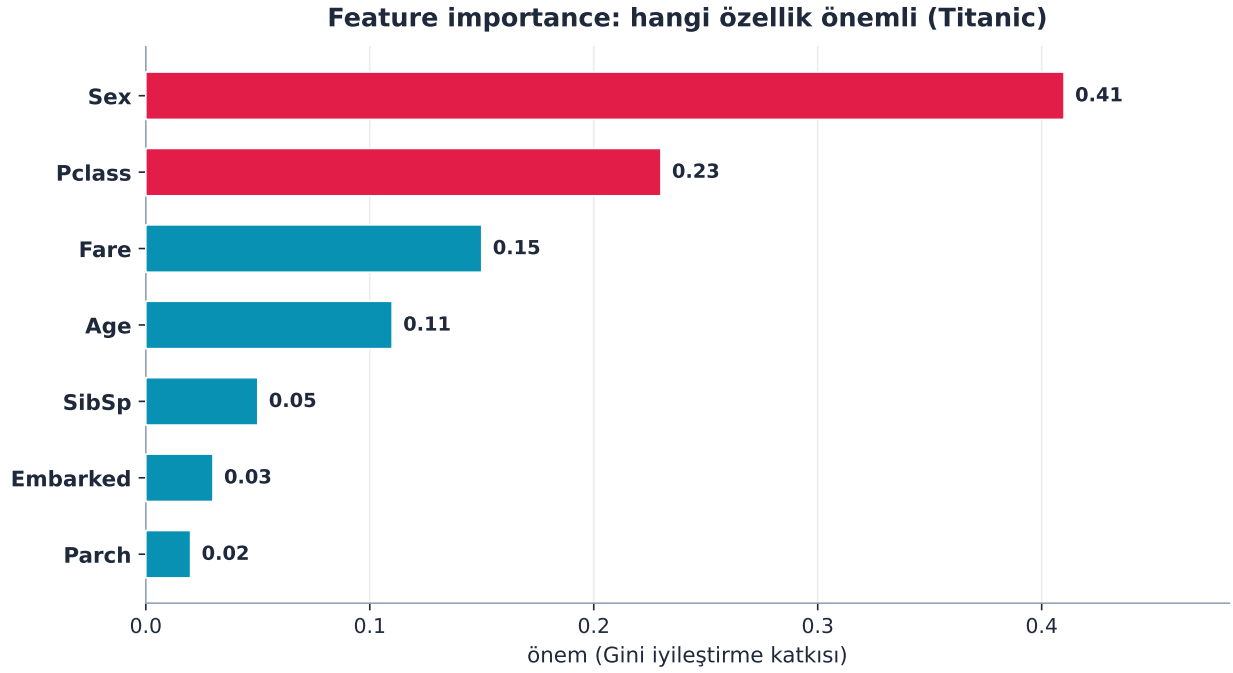
“A feature importance plot tells you how important is each feature, how often did the trees pick it and how much did it improve the Gini when it did.” — Howard, 24:16

Şekil 13.6 bu grafiği Titanic için (Howard’ın örnek değerleriyle) gösterir: cinsiyet (Sex) ve sınıf (Pclass) açık ara önde — yüzlerce sütunlu bir veride bile en önemli olanları saniyelerde bulur.

Howard bir hikâye anlatır: kredi skorlamada 7000 sütunluk bir veriyi random forest’a verip iki saatte ~30 önemli sütun bulmuş — bir danışmanlık firmasının iki yılda, milyonlarca dolara bulduğuyla aynı.

💡 Builder Notu — Önce Feature Importance Çiz

- **İleriye (production):** Yüzlerce sütunlu bir veride önce feature importance çiz, önemli ~30 sütunu bul — devasa zaman kazandıran ilk adım; Şekil 13.6’ın Howard’ın kredi-skorlama hikâyesiyle birleşimi (7000 sütun → 2 saat) bunun pratik gücünü gösterir.
- **Geriye (Gini):** Önem ölçüsü doğrudan [Gini iyileştirmesidir](#) — Şekil 13.3’deki eğrinin “bir sütun bölmeyi ne kadar saflaştırdı” katkısı, tüm ağaçlar üzerinde toplanır.



cinsiyet + sınıf açık ara önde; saniyelerde 100'lerce sütunda en önemliyi bul

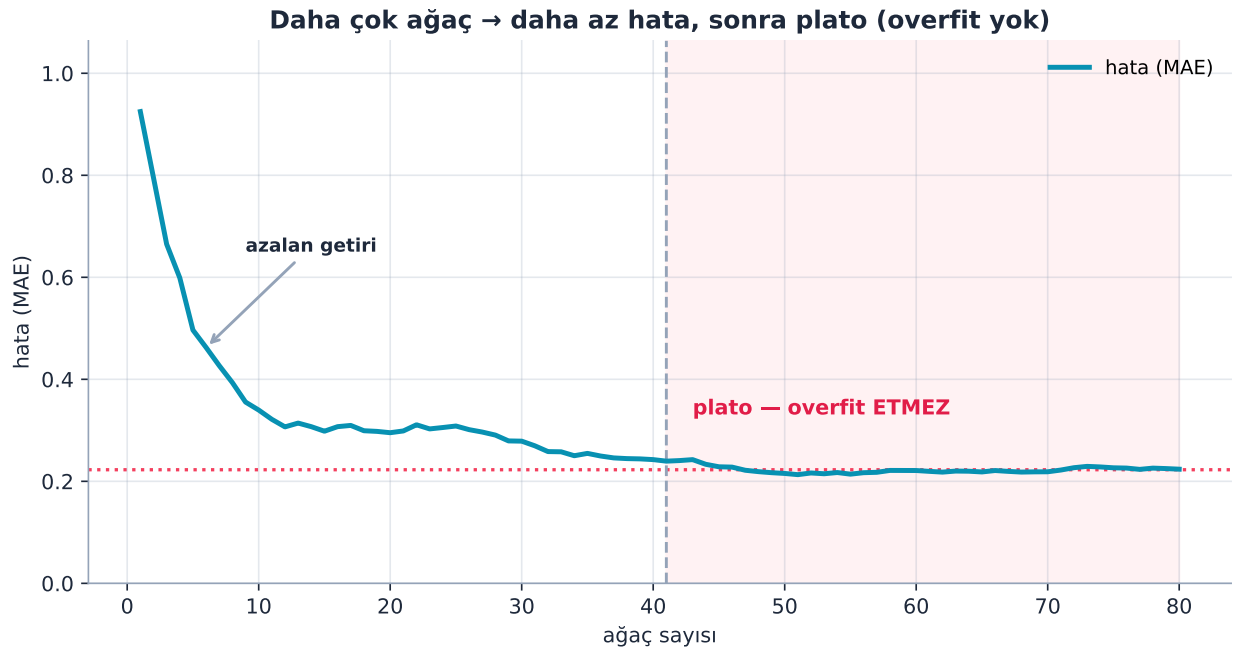
Şekil 13.6: Feature importance (illüstratif — Howard örneği): random forest, her sütunun tüm ağaçlar üzerinde Gini'yi ne kadar iyileştirdiğini toplar. Titanic'te cinsiyet (Sex) + sınıf (Pclass) açık ara önde (rose). Yüzlerce sütunlu veride en önemli ~30 sütunu saniyelerde bulur.

13.12 Kaç Ağaç?

Daha fazla ağaç her zaman hatayı düşürür (azalan getiriyle) — çünkü daha çok şey ortallanır. Ama her ağaç çıkarım (inference) zamanında maliyet getirir. Howard pratik kuralını verir.

“I don't often use more than 100 trees, this is a rule of thumb.” — Howard, 29:26

Şekil 13.7 bu davranışı gerçek bir bagging simülasyonu gösterir: ağaç sayısı arttıkça hata azalan getiriyle düşer ve bir platoya oturur — daha fazla ağaç eklemek artık pek bir şey kazandırmaz (ama **overfit de ettirmez**).



~100 ağaç iyi bir kural; sınır ağından FARKLI — ağaç eklemek overfit ettirmez

Şekil 13.7: Ağaç sayısı vs hata (gerçek hesaplama — bagging simülasyonu): daha çok ağaç hatayı azalan getiriyle düşürür ve bir platoya (rose bölge) oturur. ~100 ağaç iyi bir kural. Sınır ağından FARKLI: ağaç eklemek overfit ettirmez, yalnızca platoya yaklaştırır.

💡 Builder Notu — Ağaç Sayısı = Doğruluk/Hız Dengesi

- **İleriye:** Ağaç sayısı doğruluk/hız dengesidir; ~100 iyi bir varsayılan. Karar ağaçları çıkarımda çok hızlıdır (hatta C++ if'lerine derlenebilir) — Şekil 13.7'in platosu, “daha fazla ağacın maliyeti faydasını geçtiği” noktayı işaret eder.
- **İleriye (Random Forest Overfit Eder mi?):** Platonun yükselmemesi (sadece düzleşmesi) random forest'ın “güvenli” olmasının ta kendisi; aynı eğri **overfit bölümünde** sınır ağıyla zıtlık için tekrar okunur.

13.13 OOB: Out-of-Bag Hatası

Her ağaç verinin yalnızca bir alt kümesiyle (örn. %75) eğitildiğinden, her ağacın görmediği bir %25 vardır. Bu görülmeyen kısımda her ağacın doğruluğunu ölçüp ortalayabiliriz: **out-of-bag (OOB) hatası**. Bu, az veride ayrı bir validation set olmadan da genelleme tahmini verir.

“For each tree there was 25% of the data we didn’t train on... that is called the Out-of-Bag Error. You can get away with not having a validation set.” — Howard, 29:33

Random Forest Fikri bölümündeki “her ağaç ~%75 alt küme” yapısı (Şekil 13.5), OOB’yi bedava kılan şeydir: kalan %25, o ağaç için doğal bir test setidir.

Builder Notu — Validation Set’in Bedava Versiyonu

- **Geriye (Ders 4):** OOB, [validation set’in](#) random forest’a özgü “bedava” versiyonudur; yine de büyük/zaman serisi verilerde gerçek validation set tercih edilir.
- **Geriye (Random Forest Fikri):** OOB doğrudan [rastgele alt küme](#) yapısından doğar — her ağacın görmediği %25 zaten elimizde, ekstra veri ayırmaya gerek yok.

13.14 Model Yorumlama

Random forest yorumlamada çok güçlüdür. Feature importance’a ek olarak: **partial dependence** (bir özelliği değiştirince tahminin nasıl değiştiği), tek bir tahminin neden öyle yapıldığının açıklaması, ve gereksiz (redundant) özelliklerin tespiti. Bu yüzden Howard tablo projelerine random forest’la başlar.

“I would probably always start a tabular project with a Random Forest because they’re nearly impossible to mess up, and they give good insight, and they give a good base case.” — Howard, 31:31

Builder Notu — Üç Süper Güç: Bozulmaz + Yorumlanabilir + Hızlı

- **İleriye (production):** “Bozması imkânsız + yorumlanabilir + hızlı temel” = random forest’ın üç süper gücü; ML projesinin keşif aşamasının standart aracı.
- **Geriye (Feature Importance / Karar Ağacı Kurma):** Yorumlanabilirlik [tek ağacın izlenebilirliğinden](#) ve [feature importance’tan](#) gelir; orman bunları yüzlerce ağaç üzerinden istatistiksel olarak sağlamlaştırır.

13.15 Random Forest Overfit Eder mi?

Pratik bir güvence: ağaç eklemek random forest’ı overfit ettirmez — yalnızca platoya oturur. Çünkü her ağaç bağımsız ve yansızdır; ortalama almak varyansı düşürür, bias eklemesiz. Bu, random forest’ı bu kadar “güvenli” yapan şeydir.

Şekil 13.7’in platoya oturan (ama yükselmeyen) eğrisi tam bu güvenceyi gösterir: **Ders 4’teki overfitting eğrisinin** aksine, burada validation hatası bir noktadan sonra geri tırmanmaz.

💡 Builder Notu — Sinir Ağıyla Temel Fark

- **Geriye (Ders 4):** Sinir ağında daha fazla eğitim overfit edebilir; random forest'ta daha fazla ağaç etmez — temel bir fark. Şekil 13.7'in düz platosu, Ders 4'ün yükselen validation eğrisinin tam zıttıdır.
- **İleriye (Gradient Boosting):** “Ağaç eklemek overfit ettirmez” güvencesi yalnızca bagging (bağımsız ağaç) için geçerli; ardışık ağaç kuran gradient boosting bu güvenceyi kaybeder — daha güçlü ama daha dikkatli ayar ister.

13.16 Gradient Boosting

Howard güçlü bir akrobaya değinir: **gradient boosting**. Random forest bağımsız ağaçların ortalamasıyken, gradient boosting ağaçları **ardışık** kurar — her yeni ağaç, öncekilerin **hatalarını (residual)** düzeltmeye çalışır. Genellikle random forest'tan daha doğrudur ama overfit'e daha yatkındır, daha dikkatli ayar ister.

Şekil 13.8 bu iki paradigmayı yan yana koyar: üstte random forest (bağımsız ağaçlar paralel, ortalama), altta gradient boosting (ağaçlar ardışık, her biri öncekinin kalan hatasını düzeltir).

💡 Builder Notu — RF Başlangıç, Boosting Son Performans

- **İleriye (production):** XGBoost, LightGBM, CatBoost gibi gradient boosting kütüphaneleri, tablo verisi yarışmalarının (Kaggle) çoğunu kazanır; random forest sağlam başlangıç, boosting ise son performans — Şekil 13.8'in iki satırı tam bu iş bölümünü gösterir.
- **Geriye (Random Forest Overfit Eder mi?):** GBM'in ardışıklığı, RF'nin “overfit etmez” güvencesini bozar; her yeni ağaç öncekinin hatasına bakıp eğitim verisine daha çok yapışır — bu yüzden learning rate / ağaç derinliği / erken durdurma dikkatli ayarlanır.

13.17 Kapanış

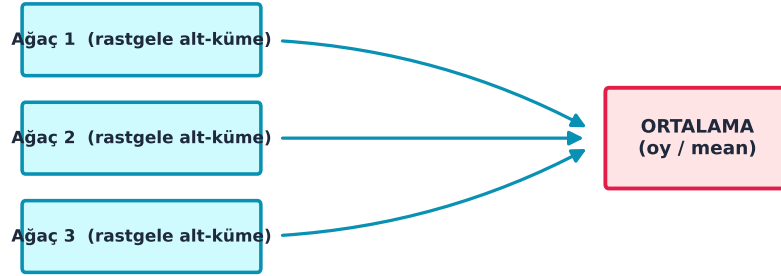
Ders 6, tablo verisi için sinir ağına güçlü bir alternatif gösterdi: tek karar ağacı → bagging → random forest → gradient boosting. Random forest hızlı, sağlam, yorumlanabilir bir temeldir ve “bozması neredeyse imkânsız”dır. Sinir ağıları görüntü/metin/ses gibi yapısız veride; ağaç tabanlı modeller tablo verisinde parlar.

💡 Builder Notu — İki Aile, İki Veri Türü

- **İleriye (Ders 7-8):** Sonraki dersler ortak filtreleme (collaborative filtering) ve evrişimlere (CNN) döner; tablo için ağaçlar, yapısız veri için sinir ağıları — builder ikisini de bilir.
- **Geriye (tüm seri):** Bu ders, Ders 5'in sıfırdan sinir ağına tablo verisinde bir rakip koydu; Şekil 34.1'in karar-ağacı → random-forest → gradient-boosting zinciri, bu rakibin tüm hikâyesidir.

1 · Random Forest — PARALEL

bağımsız, yansız; overfit etmez



ağaçlar birbirini GÖRMEZ — tam paralel kurulum

2 · Gradient Boosting — ARDIŞIK

her ağaç öncekinin hatasını düzeltir; daha güçlü ama overfit'e yatkın



ağaçlar SIRAYLA kurulur — her biri kalan hataya odaklanır

RF = bağımsız ağaçları ortalar (overfit etmez); GBM = ardışık ağaçlarla hatayı kovalar (XGBoost/LightGBM tablo yarışmalarını kazanır)

Şekil 13.8: Random forest vs gradient boosting (şematik): RF bağımsız ağaçları PARALEL kurup ortalar (overfit etmez); GBM ağaçları ARDIŞIK kurar — her biri öncekinin hatasını (residual) düzeltir (daha güçlü ama overfit'e yatkın, dikkatli ayar ister). XGBoost/LightGBM tablo yarışmalarını kazanır.

13.18 Bu Dersin Özeti

1. **Karar ağacı** veriyi ardışık ikili bölmelerle ayırır; derin öğrenme değildir, neredeyse hiç ön işleme istemez (yalnızca sıralama önemli) ([Karar Ağaçları](#), [Az Ön İşleme](#)).
2. Bir bölmenin iyiliği **saflıkla** ölçülür: standart sapma veya **Gini** (rastgele iki örneğin farklı sınıfta olma olasılığı) ([Bölme Skoru](#), [Gini](#)).
3. **Bagging** (Leo Breiman): çok sayıda bağımsız, yansız ama kusurlu modelin ortalaması, hataları götürdüğü için tek tek hepsinden iyidir ([Bagging](#)).
4. **Random forest** = verinin rastgele alt kümeleriyle kurulan yüzlerce karar ağacının ortalaması; dört satır kod, “bozması imkânsız” ([Random Forest Fikri](#), [Random Forest Kurma](#)).
5. **Feature importance** hangi özelliklerin önemli olduğunu saniyeler içinde gösterir — yüzlerce sütunlu veride paha biçilmez ([Feature Importance](#)).
6. Daha çok ağaç hatayı azaltır (azalan getiriyle) ve **overfit etmez**; ~100 ağaç iyi bir kural ([Kaç Ağaç?](#), [Overfit Eder mi?](#)).
7. **OOB hatası**, her ağacın görmediği %25 üzerinden ölçülür; az veride validation set yerine geçebilir ([OOB](#)).
8. **Gradient boosting** ağaçları ardışık kurar (her biri öncekinin hatasını düzeltir); daha güçlü ama overfit'e yatkın ([Gradient Boosting](#)).

! Tek Bir Cümle

Random forest, verinin rastgele parçalarıyla kurulan çok sayıda kusurlu karar ağacının ortalamasını alarak, tablo verisinde hızlı, sağlam, yorumlanabilir ve “bozması neredeyse imkânsız” bir model üretir.

13.19 Kontrol Soruları

i Soru 1: Bagging neden işe yarar? Random forest, bagging'i nasıl uygular?

Cevap:

Bagging, çok sayıda **bağımsız (uncorrelated)** ve **yansız (unbiased)** ama kusurlu modelin ortalamasını almaya dayanır. Yansız modellerin hataları rastgeledir (kimi yüksek, kimi düşük); ortalama alınca bu hatalar birbirini götürür ve ortalama tahmin tek tek hepsinden daha doğru olur (varyans düşer, bias eklenmez). Random forest bunu karar ağaçlarına uygular: verinin her seferinde farklı bir rastgele alt kümesiyle yüzlerce ağaç eğitir; her ağaç biraz farklı ve kusurludur ama yansızdır, ortalamaları güçlüdür. (Şekil 13.4 bu yakınsamayı, Şekil 13.5 mimariyi gösterir.)

i Soru 2: Karar ağaçları neden lineer modele/sinir ağına göre çok az ön işleme ister?

Cevap:

Çünkü karar ağacının tek umursadığı, verinin **sıralamasıdır**, değerlerin büyüklüğü değil. Bir bölme “Age < 24” gibi bir eşik sorusudur; Age'i normalize etsen de etmesen de aynı sıralama, aynı bölme çıkar. Bu yüzden **normalizasyon gerekmez**. Kategorik değişkenler de çoğu zaman sayısal kodla (dummy variable olmadan) çalışır, çünkü ağaç değeri eşiklerle böler. Lineer model/sinir ağı ise değerleri katsayılarla çarpıp topladığı için ölçek ve kodlamaya duyarlıdır.

i Soru 3: Feature importance nasıl hesaplanır ve neden bu kadar kullanışlıdır?

Cevap:

Her karar ağacında, her sütunun bir bölmede Gini'yi (örnek sayısıyla ağırlıklı) ne kadar iyileştirdiği toplanır; bu, tüm ağaçlar üzerinde her sütun için birleştirilir. Sonuç, her özelliğin “ne sıklıkla seçildiği ve seçildiğinde ne kadar fark yarattığı”nı gösteren bir önem sırasıdır. Kullanışlıdır çünkü random forest sanjiler içinde eğitilir ve veri dağılımına/ölçeğine duyarlıdır — yüzlerce (hatta binlerce) sütunlu bir veriyi doğrudan verip en önemli ~30 sütunu hemen bulabilirsin. Howard'ın kredi skorlama örneğinde bu, milyonlarca dolarlık iki yıllık danışmanlığa eşdeğer içgörüyü iki saatte verdi.

i Soru 4: Random forest ile sinir ağı arasında ne zaman hangisini seçersin? (builder bağlantısı)

Cevap:

Tablo verisinde (spreadsheet, veritabanı) önce **random forest** (veya gradient boosting): hızlı, sağlam, yorumlanabilir, bozması zor, az ön işleme. **Yapısız veride** (görüntü, metin, ses) **sinir ağı**: çünkü öznitelikleri kendisi öğrenir ve mekânsal/dizisel yapıyı yakalar. Pratikte builder, bir tablo projesine random forest ile başlar (temel + feature importance), gerekirse gradient boosting ile performansı zorlar; görüntü/metin için doğrudan derin öğrenmeye gider. İkisi rakip değil, farklı işler için araçtır.

13.20 Egzersizler

Egzersiz 1 (Direkt uygulama). Titanic verisinde bir `DecisionTreeClassifier(max_leaf_nodes=4)` eğit ve `draw_tree` ile hangi soruları sorduğunu incele.

Egzersiz 2 (İki-aşamalı). Bir `RandomForestClassifier(100, min_samples_leaf=5)` kur, doğruluğunu tek ağaçla kıyasla; ağaç sayısını 10, 40, 100 yapıp hatayı izle.

Egzersiz 3 (Edge case). Ağaç sayısını 500'e çıkar; hatanın overfit etmeyip platoya oturduğunu gözlemler (sinir ağıyla zıtlık).

Egzersiz 4 (Python ile doğrulama). `feature_importances_`'ı çiz; en önemli iki-üç sütunu çıkarıp modeli yeniden eğit ve doğruluk kaybını ölç.

Egzersiz 5 (Sonraki dersin habercisi). Aynı veride bir gradient boosting modeli (örn. `GradientBoostingClassifier` veya `XGBoost`) dene; random forest'la doğruluk ve overfit eğilimini kıyasla.

13.21 Sonraki Ders İçin Hazırlık

Ders 7: Ortak Filtreleme (Collaborative filtering)

Ders 6 tablo verisi için ağaçları gösterdi. Ders 7 öneri sistemlerinin kalbine iner: bir kullanıcının bir ürünü/filmi ne kadar seveceğini, **embedding**'ler ve latent faktörlerle tahmin etme.

Ana konular:

- Collaborative filtering ve latent faktörler
- Embedding'ler (gömme tabloları)

- Bias terimleri
- Derin öğrenme ile öneri

⚠ Ders 7 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 2 ve 4 — random forest + feature importance).
- Bir tablo veri setinde feature importance çiz ve sonuçları yorumla.
- Ana cümleyi tekrar oku: “Random forest = çok kusurlu ağacın ortalaması.”

13.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Karar ağacı	Ardışık ikili bölmelerle veriyi ayırma	4:43
Bölme skoru	İki tarafı en saf yapan bölme (std/Gini)	7:05
Gini	$1 - \sum p^2$; rastgele iki örneğin farklı sınıfta olma olasılığı	7:05
Az ön işleme	Ağaç yalnızca sıralamayı umursar; normalize gerekmez	17:53
Bagging	Bağımsız yansız modellerin ortalaması (Leo Breiman)	16:52
Random forest	Rastgele alt kümelerle çok ağacın ortalaması	19:07
Feature importance	Her sütunun Gini'yi iyileştirme katkısı	24:16
Ağaç sayısı	Daha çok ağaç = daha az hata (azalan getiri); ~100	29:26
OOB hatası	Her ağacın görmediği %25 üzerinden genelleme tahmini	29:33
Yorumlanabilirlik	Partial dependence, tahmin açıklama, redundant tespit	32:08
Gradient boosting	Ardışık ağaçlar; her biri öncekinin hatasını düzeltir	49:08
RF vs sinir ağı	Tablo → ağaç; yapısız veri → sinir ağı	31:31

13.23 ML Builder Bağlantıları

💡 Builder Notu — 6 ML Köprüsü

Bu ders, derin öğrenmeden bir an için ayrılıp tablo verisinin en güçlü klasik ailesini — karar ağaçları ve random forest’ı — kurar. Köprülerin özeti:

1. **Bagging** → bağımsız yansız tahminlerin ortalaması varyansı düşürür (Stat 110 büyük sayılar yasası) ([Bagging](#)).
2. **Random forest** → tablo verisinde hızlı, sağlam, yorumlanabilir temel; production keşif aşamasının standardı ([Random Forest Fikri](#), [Model Yorumlama](#)).
3. **Feature importance** → yüzlerce sütunda önemli olanları saniyelerde bulma; veri keşfinin ilk adımı ([Feature Importance](#)).
4. **OOB hatası** → random forest’a özgü “bedava” validation; az veride pratik ([OOB](#)).
5. **Gradient boosting** → XGBoost/LightGBM; tablo yarışmalarının çoğunu kazanır ([Gradient Boosting](#)).
6. **Ağaç vs sinir ağı** → tablo için ağaç, görüntü/metin/ses için sinir ağı; builder ikisini de kullanır ([Karar Ağaçları](#), [Kapanış](#)).

! Bu dersten tek bir şey alıp gideceksen

Bu dersten tek bir şey alıp gideceksen: tablo verisinde sinir ağına koşma — önce bir random forest kur. Bozması neredeyse imkânsızdır, saniyelerde eğitilir, feature importance ile sana veriyi anlatır ve sağlam bir temel verir. “Çok sayıda kusurlu ağacın ortalaması” basit ama şaşırtıcı derecede güçlü bir fikirdir.

14 Ortak Filtreleme — Embedding, Latent Faktör ve Dot Product (Collaborative filtering)

Serinin sondan bir önceki dersi iki büyük yarıyı birleştirir: önce yarışma araçları — gradient accumulation (küçük GPU’da büyük model), ensembling, multi-target ve çok-sınıflı kaybın altındaki softmax + cross-entropy — sonra dersin asıl başlığı collaborative filtering: kullanıcı $\times$ ürün matrisindeki boşlukları, her ikisi için öğrenilen latent faktör vektörlerinin dot product’ıyla doldurma; ve perdenin arkası — ‘embedding’ gizemli bir kavram değil, yalnızca bir diziden satır aramaktır (one-hot çarpımının hızlı kısıyolu)

Bölüm bilgisi

- **Howard’ın videosu:** [course.fast.ai — Lesson 7: Collaborative filtering](https://course.fast.ai/lesson7.html) (~106 dk)
- **Seri:** Practical Deep Learning for Coders — Part 1, Ders 7
- **Hoca:** Jeremy Howard
- **Playlist:** [Practical Deep Learning Part 1 \(2022\)](#)
- **Notebook:** [fastbook — 08_collab](#)
- **Okuma süresi:** ~38 dk

14.1 Bu Derste Ne Var?

Serinin sondan bir önceki dersi iki büyük konuyu kapsar. İlk yarı, yarışma performansını zorlamanın araçlarını verir: büyük modelleri sığdırmak için **gradient accumulation**, doğruluğu artırmak için **ensembling**, ve **multi-target** modeller ile **cross-entropy/softmax** derinlemesine. İkinci yarı dersin başlığıdır: **collaborative filtering** (öneri sistemleri) — **latent faktörler** ve **embedding**’lerle.

Üç temel fikir:

1. **Gradient accumulation + ensembling** — küçük GPU’da büyük model eğitmenin ve birden çok modeli birleştirip doğruluğu artırmanın pratik yolları ([Gradient Accumulation](#) → [Ensembling](#)).
2. **Cross-entropy = softmax + log likelihood** — çok-sınıflı sınıflandırmanın standart kaybı; her modelin altında bu yatar ([Softmax](#) → [Cross-Entropy](#)).
3. **Collaborative filtering = matris tamamlama** — kullanıcı ve ürün için **latent faktörler** öğren, **dot product**’larını al; embedding ise “bir diziden arama yapmak”tan başka bir şey değildir ([Collaborative Filtering](#) → [Latent Faktörler](#) → [Dot Product](#) → [Embedding = Arama](#)).

“An embedding actually means: look something up in an array.” — Howard, 1:19:29

Şekil 34.1 bu üç fikri tek bir haritada birleştirir: yarışma araçlarından (gradient accumulation / ensembling) çok-sınıflı kaybın çekirdeğine (softmax → cross-entropy), oradan dersin kalbine — collaborative filtering’e (latent faktör + dot product) ve hepsinin altındaki “embedding = arama” fikrine (rose vurgulu) ilerler.

💡 Builder Notu — İki Yarı, Tek Zincir: Yarışma Araçları ve Öneri Çekirdeği

- **Geriye (Ders 4-6):** Cross-entropy [Ders 4’ün loss’unu](#), embedding [Ders 6’nın dummy variable’ını](#) genelleştirir; [dot product](#) ise 18.06’nın kalbidir.
- **İleriye (Part 2 / Karpathy):** [Embedding](#) ve [cross-entropy](#), Karpathy makemore ve GPT’nin temel taşları; collaborative filtering modern öneri sistemlerinin çekirdeği.
- **Tek cümle:** Embedding bir arama tablosudur; collaborative filtering bu tablolardaki latent faktörlerin dot product’ıyla boş hücreleri tahmin eder.

14.2 Daha Büyük Modeller ve GPU Belleği

Howard daha büyük (daha doğru) modeller kullanmak ister ama bunlar GPU belleğini (VRAM) aşar — “out of memory” hatası. Sorun: bellek, batch’teki görüntü sayısı ve model boyutuyla artar.

“Out of memory error. What do we do? We can use a cool little trick called gradient accumulation.”
— Howard, 5:58

Çözüm hemen [Gradient Accumulation](#) bölümünde geliyor: batch’i belleğe sığacak küçük parçalara böl, gradyanları biriktir.

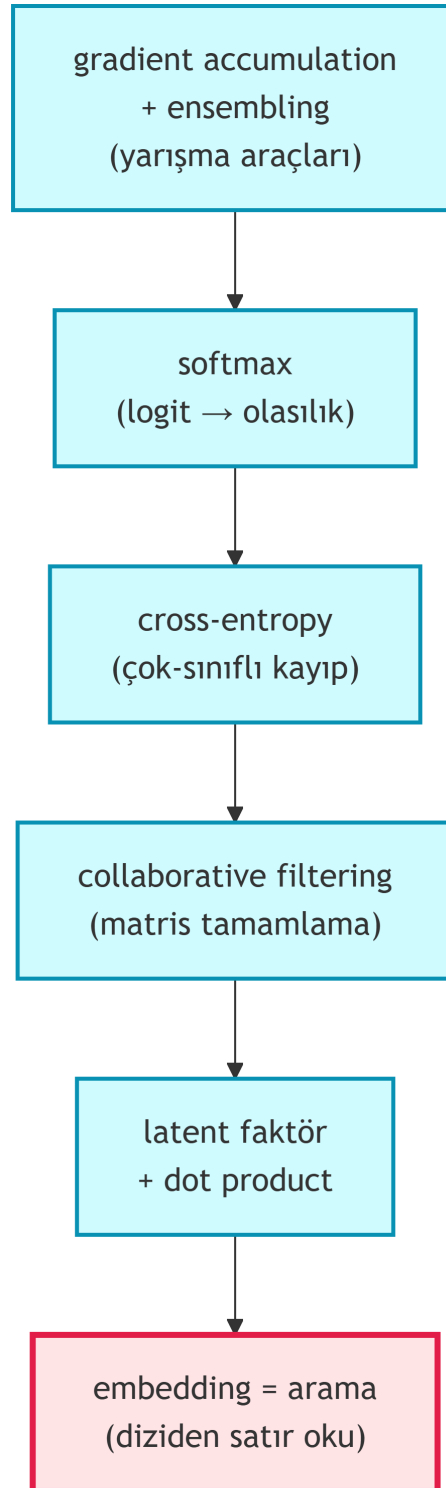
💡 Builder Notu — ‘GPU Swap Yapamaz’ Notunun Çözümü

- **Geriye (Ders 2):** [Ders 2’deki](#) “GPU swap yapamaz” notunun çözümü tam burada geliyor — CPU RAM’ine taşamayan VRAM, batch’i bölmekle idare edilir.
- **İleriye (Gradient Accumulation):** “Out of memory” hatası modelin yanlış olduğunu değil, batch’in büyük olduğunu söyler; çözüm modeli küçültmek değil, batch’i bölmektir.

14.3 Gradient Accumulation

Gradient accumulation: büyük bir batch yerine birkaç küçük batch’in gradyanlarını **biriktir**, sonra tek adım at. Matematiksel olarak büyük batch’le neredeyse aynıdır ama belleğe sığar.

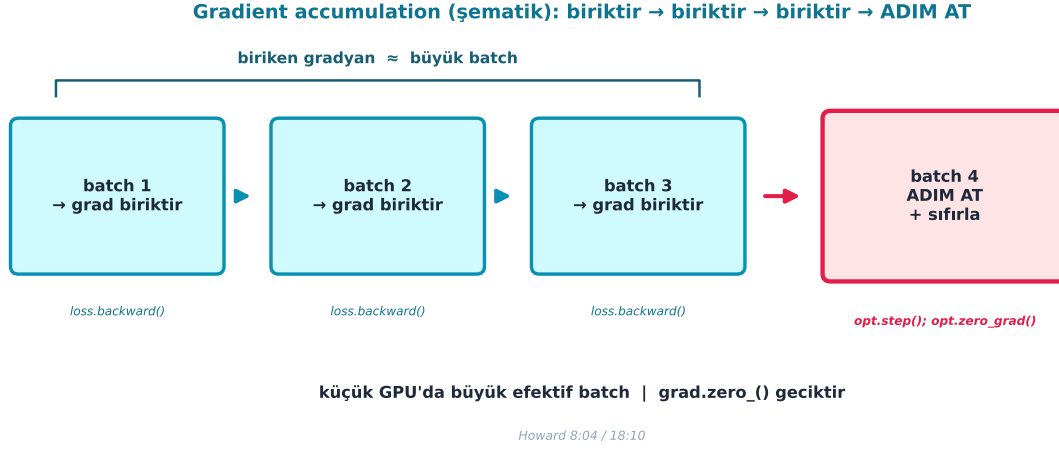
```
# Kavram: küçük batch'lerin gradyanlarını biriktir, n_acc batch'te bir adım at
for i, (xb, yb) in enumerate(dl):
    loss = loss_func(model(xb), yb)
    loss.backward() # gradyanlar birikir (henüz sıfırlama)
    if (i+1) % n_acc == 0:
        opt.step(); opt.zero_grad() # adım at, sonra sıfırla
# fastai'de: cbs=GradientAccumulation(n_acc) callback'i bunu otomatik yapar
```



řekil 14.1: Ders 7'nin kavram haritası: gradient accumulation + ensembling (yariřma araları) → softmax (logit→olasılık) → cross-entropy (ok-sınıflı kayıp) → collaborative filtering (latent faktör + dot product) → embedding = arama (rose vurgulu). Tek bir zincir: nce ok-sınıflı kaybı kur, sonra kullanıcı/rn iin latent faktör ğren, dot product'larını al — ki o faktrleri okumak yalnızca bir diziden satır aramaktır.

“Gradient accumulation in fastai is very straightforward, you just [add a parameter] and it updates the weights once it’s got 64 images.” — Howard, 18:10

Şekil 14.2 bu mekanizmayı şematik olarak gösterir: ilk birkaç küçük batch yalnızca gradyanı biriktirir (`loss.backward()`, sıfırlama yok), son batch’te tek adım atılır ve gradyan sıfırlanır — biriken gradyan büyük bir batch’in gradyanına yaklaşıp.



Şekil 14.2: Gradient accumulation (şematik): ilk üç küçük batch yalnızca gradyanı biriktirir (cyan, `loss.backward()` — henüz `opt.step` yok); dördüncü batch’te ADIM AT + sıfırla (rose, `opt.step()`; `opt.zero_grad()`). Biriken gradyan $\approx$ büyük batch’in gradyanı; böylece küçük GPU’da büyük efektif batch elde edilir. `grad.zero_()`’ı geciktirmek = gradyan biriktirmek.

💡 Builder Notu — `grad.zero_()`’ı Geciktirmek = Biriktirmek

- **Geriye (Ders 5):** [Ders 5’teki](#) `grad.zero_()`’ı geciktirmek = gradyan biriktirmek; Şekil 14.2’in “henüz sıfırlama yok” adımı, aynı mekanizmanın bilinçli kullanımıdır.
- **İleriye (Karpathy Ders 10):** Gradient accumulation, GPT-2 eğitiminde büyük efektif batch için standart tekniktir — sınırlı VRAM’de devasa modelleri eğitmenin yolu.

14.4 Ensembling

Doğruluğu artırmanın bir başka yolu: birden çok modeli eğitip tahminlerini **birleştirmek** (ensembling). Howard farklı mimarileri (convnext, vit) ve farklı validation set’leri (k-folds) birleştirir; her ekleme skoru biraz daha iyileştirir.

“These are my four submissions, and you can see each one got better.” — Howard, 23:01

💡 Builder Notu — Bagging’in Mimariye Uygulanmış Hâli

- **Geriye (Ders 6):** Ensembling, [random forest’in bagging fikrinin](#) (bağımsız modellerin ortalaması) farklı mimarilere uygulanmış hâli — orada rastgele alt kümeler, burada farklı mimari/validation.

- **İleriye (production):** Yarışma kazanan çözümlerin neredeyse hepsi ensemble'dır; production'da maliyet/doğruluk dengesi gerektirir — her ek model çıkarım maliyetini de artırır.

14.5 Multi-Target Modeller

Howard tek değil **iki şey** tahmin eden bir model kurar (paddy yarışmasında: hem hastalık hem pirinç çeşidi). Model son katmanında iki çıktı grubu üretir ve kayıp ikisinin toplamıdır. Tek bir modelin birden çok hedefi öğrenmesi çoğu zaman her birini ayrı öğrenmekten iyidir.

“So now we need a model that predicts two things.” — Howard, 37:51

Bu “iki çıktı grubu” fikri doğrudan **Softmax** bölümüne bağlanır: her grup kendi içinde softmax + cross-entropy ile puanlanır, kayıplar toplanır.

💡 Builder Notu — Paylaşılan Temsil, İki Hedef

- **İleriye:** Multi-task learning, paylaşılan temsillerden yararlanır; bir görevdeki bilgi diğerine yardım eder (transfer'in içsel hâli) — model gövdesi ortak, yalnızca son katman ikiye ayrılır.
- **İleriye (Cross-Entropy):** İki hedefin her biri kendi cross-entropy'sini üretir; toplam kayıp ikisinin toplamıdır — aynı kayıp tek modelde iki kez kullanılır.

14.6 Softmax

Çok-sınıflı sınıflandırmada modelin ham çıktıları (logit'ler) bir olasılık dağılımına çevrilmelidir. **Softmax** bunu yapar: her çıktının üstelini alıp toplama böler — sonuçlar pozitif ve toplamı 1 olur, en büyük logit baskınlaşır.

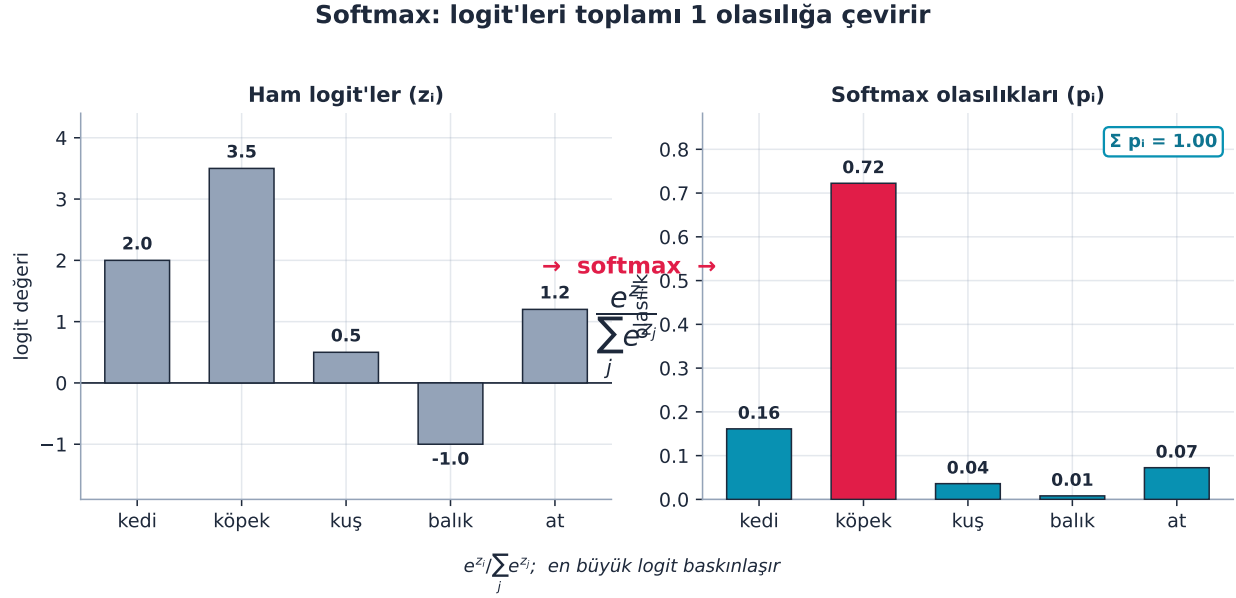
$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

“We're trying to predict which one is it, and so this is called softmax.” — Howard, 45:45

Şekil 14.3 bu dönüşümü gerçek hesaplamayla gösterir: solda ham logit'ler (negatif olabilir), sağda softmax'tan geçince toplamı 1 olan olasılıklar — en büyük logit (rose) baskın olasılığı alır.

💡 Builder Notu — Sigmoid'in Çok-Sınıflı Genellemesi

- **Geriye (Ders 5 / Calculus):** Softmax, **Ders 5'teki sigmoid'in** çok-sınıflı genellemesidir; e^x tabanlı (Calculus), “tek doğru sınıf” varsayımı için — Şekil 14.3'in toplamı-1 kısıtı tam bu “tek doğru” varsayımının matematik karşılığıdır.
- **Pratik:** Birden çok şey aynı anda doğru olabiliyorsa softmax DEĞİL, her çıktıya ayrı sigmoid kullanılır; softmax sınıfları birbiriyle yarışır (biri kazanırsa diğerleri kaybeder).



Şekil 14.3: Softmax (gerçek hesaplama): ham logit'ler z_i (sol, negatif olabilir) $\rightarrow$ softmax(z_i) = $e^{z_i} / \sum e^{z_j}$ $\rightarrow$ olasılıklar p_i (sağ, hepsi pozitif, toplamı 1). En büyük logit (köpek) baskın olasılığı alır (rose). Üstel, büyük logiti orantısız büyütür; bu yüzden 'soft' bir argmax gibi davranır.

14.7 Cross-Entropy Loss

Cross-entropy, softmax + negatif log likelihood'dur. Modelin doğru sınıfa verdiği olasılığın logaritmasını cezalandırır: doğru sınıfa yüksek olasılık $\rightarrow$ düşük kayıp; düşük olasılık $\rightarrow$ yüksek kayıp.

$$\text{cross-entropy} = - \sum_i y_i \cdot \log(p_i)$$

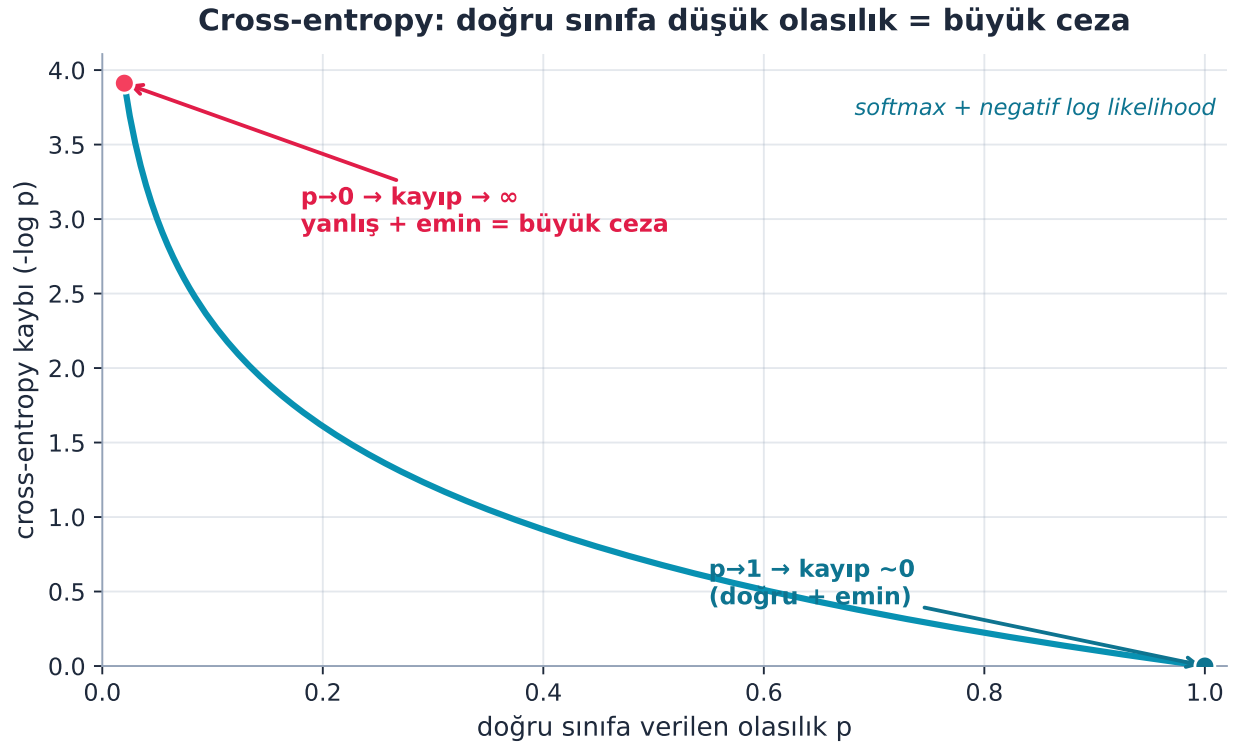
PyTorch `F.cross_entropy` softmax'ı (aslında log-softmax'ı) ve negatif log likelihood'u tek adımda, sayısal olarak kararlı biçimde hesaplar.

"The first part of what cross-entropy loss in PyTorch does is to calculate the softmax. It's actually the log of the softmax." — Howard, 46:24

Şekil 14.4 bu kaybı gerçek hesaplamayla gösterir: doğru sınıfa verilen olasılık p 1'e yaklaşıncaya kayıp ~ 0 ; p 0'a yaklaşıncaya kayıp asimptotik olarak sonsuza gider — "yanlış ve emin" en sert cezayı alır.

💡 Builder Notu — Maximum Likelihood'dan Türeyen Kayıp

- **Geriye (Stat 110 / 6.S191):** Cross-entropy, iki dağılım arasındaki farkın ölçüsü; maximum likelihood'dan türer. 6.S191 Ders 1'in ana kaybı — Şekil 14.4'nin — $-\log p$ eğrisi, "doğru sınıfa atanan olasılığı en büyük yap" hedefinin matematik karşılığıdır.
- **İleriye (Karpathy):** Dil modellerinin (makemore, GPT) kaybı tam budur — sonraki token'ın cross-entropy'si; **softmax** + bu eğri, her dil modelinin çıkış katmanında çalışır.



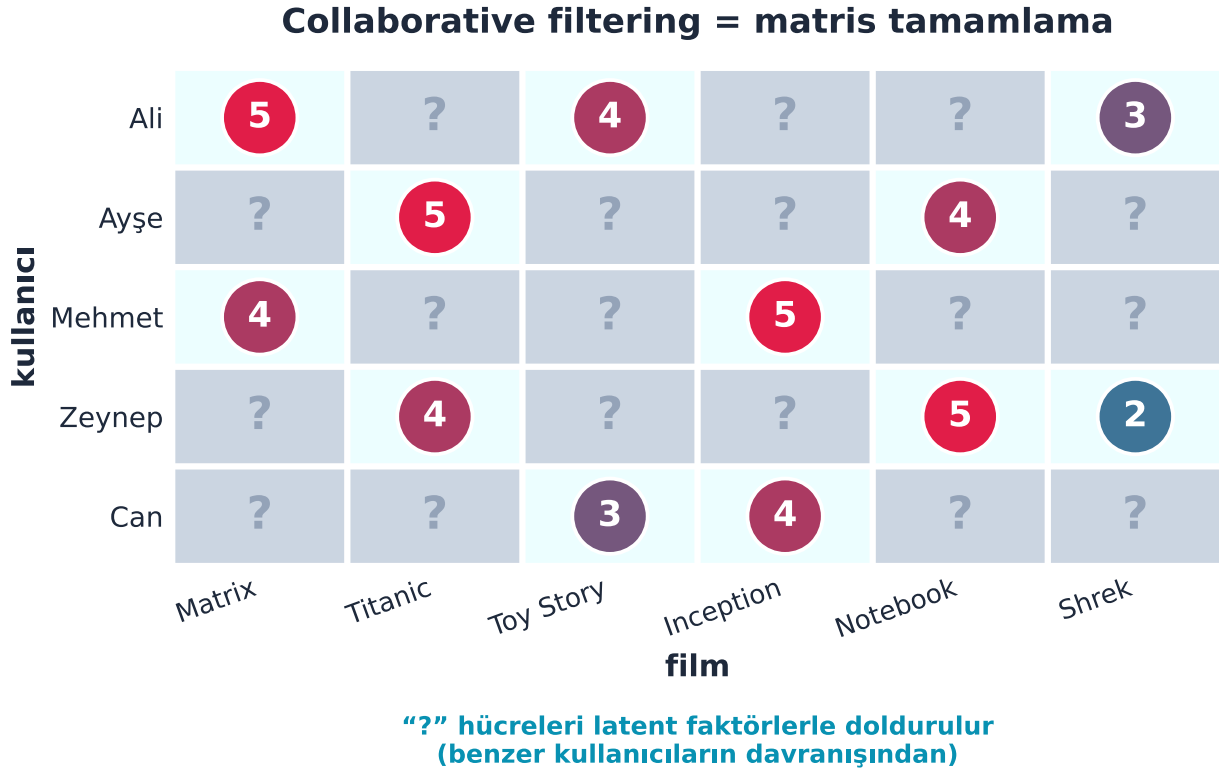
Şekil 14.4: Cross-entropy (gerçek hesaplama): doğru sınıfa verilen olasılığın $-\log$ 'u. Yüksek olasılık ($p \rightarrow 1$) $\rightarrow \sim 0$ kayıp; düşük olasılık $\rightarrow$ büyük ceza (asimptotik ∞). softmax + negatif log likelihood.

14.8 Collaborative Filtering: Öneri Problemi

İkinci yarı dersin başlığıdır. **Collaborative filtering**: bir kullanıcının daha önce puanlamadığı bir filmi/ürünü ne kadar seveceğini, benzer kullanıcıların davranışından tahmin etme. Veri, kullanıcı $\times$ ürün matrisidir; çoğu hücre boştur. Amaç bu boşlukları doldurmaktır.

“The core of collaborative filtering is a matrix completion exercise.” — Howard, 1:14:00

Şekil 14.5 bu problemi şematik bir kullanıcı $\times$ film matrisiyle gösterir: birkaç hücrede puan (1-5) var, çoğu hücre boş (“?”); collaborative filtering’in işi bu boşlukları, benzer kullanıcıların davranışından tahmin ederek doldurmaktır.



Şekil 14.5: Collaborative filtering = matris tamamlama (şematik): 5 kullanıcı $\times$ 6 film puan matrisi. Birkaç hücrede gerçek puan (1-5, düşük→cyan, yüksek→rose daire); çoğu hücre BOŞ (“?”). Amaç: “?” hücrelerini benzer kullanıcıların davranışından (latent faktörlerle) doldurmak. Puanlar kavramsal — sahte istatistik değil, illüstratif örnek.

💡 Builder Notu — Netflix/Spotify’ın Ölçeklenmiş Hâli

- **İleriye (production)**: Netflix, Spotify, Amazon önerileri bu fikrin ölçeklenmiş hâli; “bunu sevenler şunu da sevdi.” Şekil 14.5’in 5×6 oyuncak matrisi, gerçekte milyonlarca kullanıcı $\times$ milyon ürün boyutunda çalışır.
- **İleriye (Latent Faktörler)**: Boşlukları doldurmanın yolu, sıradaki bölümde gelen latent faktörlerdir — her kullanıcı ve ürün için öğrenilen gizli sayılar.

14.9 Latent Faktörler

Boş hücreleri nasıl doldururuz? **Latent faktörler** ile: her kullanıcı için (örn. 5) gizli sayı, her film için (örn. 5) gizli sayı icat ederiz. Bu faktörlerin ne anlama geldiğini önceden bilmeyiz — model onları öğrenir. Belki biri “aksiyon sevmeye derecesi”, biri “eskilik” olur; ama biz tanımlamayız.

“We can create things called latent factors. There are five latent factors — I don’t know what they’re for — they’re just five latent factors.” — Howard, 1:21:00

Bu faktörler nasıl puana dönüşür? Sıradaki **Dot Product** bölümünün konusu: kullanıcı ve film faktör vektörlerinin eleman-eleman çarpımının toplamı.

💡 Builder Notu — Matris Faktörizasyonu / SVD Ruhı

- **Geriye (18.06):** Latent faktörler, kullanıcı-film matrisinin düşük-ranklı çarpanlara ayrılmasıdır (matris faktörizasyonu / SVD ruhu); **collaborative filtering matrisinin** boşlukları, bu iki küçük faktör matrisinin çarpımıyla doldurulur.
- **İleriye (Latent Uzak):** Anlamı önceden bilinmeyen bu faktörler, eğitimden sonra yorumlanabilir bir **latent uzak** oluşturur — benzer filmler yakın düşer.

14.10 Dot Product

Bir kullanıcının bir filmi ne kadar seveceği = kullanıcının latent faktörleri ile filmin latent faktörlerinin **dot product**’ı (eleman eleman çarp, topla). Kullanıcının “aksiyon sevgisi” yüksek + filmin “aksiyonluğu” yüksek → yüksek skor.

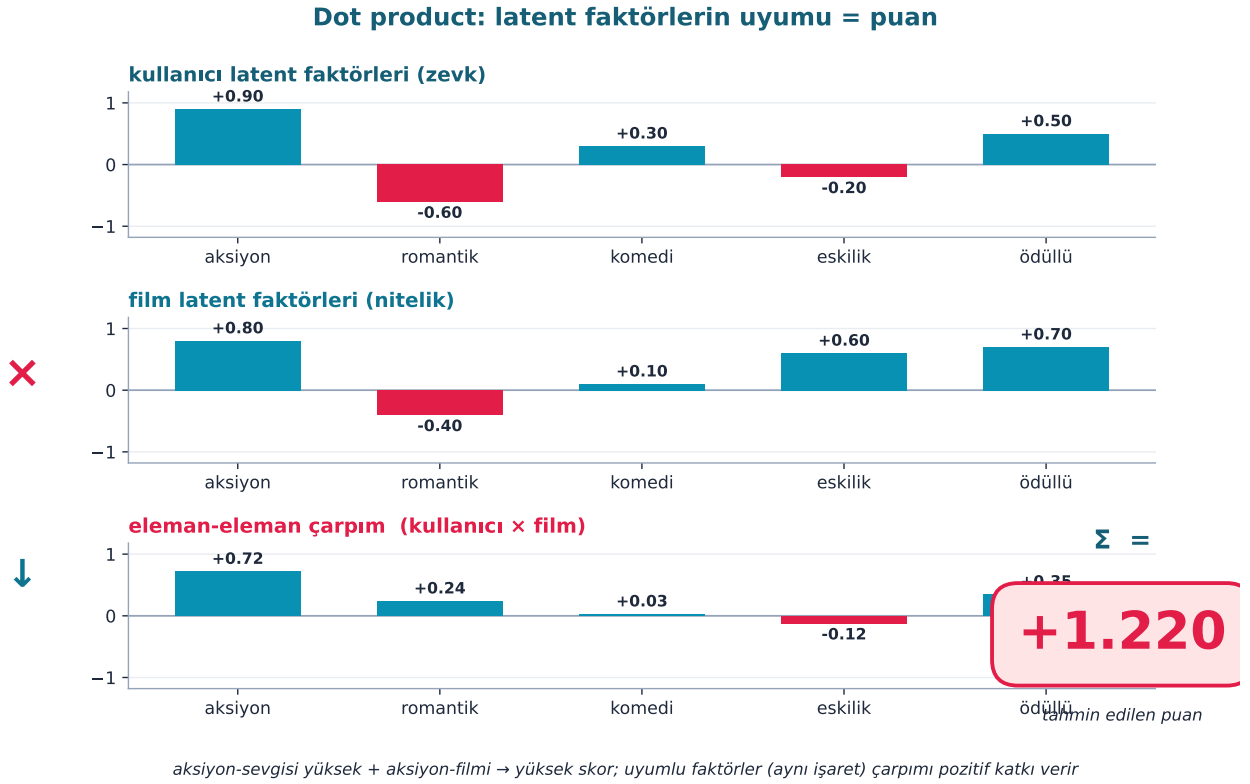
```
n_factors = 5
user_factors = torch.randn(n_users, n_factors) # her kullanıcı için 5 gizli sayı
movie_factors = torch.randn(n_movies, n_factors) # her film için 5 gizli sayı
# tahmin = ilgili kullanıcı ve film faktörlerinin dot product'ı
```

“This is the dot product of the user’s preferences and the movie’s... the matrix product of a row with a column is the same thing as a dot product.” — Howard, 1:08:00

Şekil 14.6 bu işlemi gerçek hesaplamayla gösterir: kullanıcı faktör vektörü (zevk) ile film faktör vektörü (nitelik) eleman-eleman çarpılır; çarpımların toplamı (Σ , rose) tahmin edilen puandır — uyumlu faktörler (aynı işaret) pozitif katkı verir.

💡 Builder Notu — Dot Product = SUMPRODUCT, 18.06’nın Kalbi

- **Geriye (18.06 / Ders 5):** Dot product = SUMPRODUCT (Excel); 18.06’nın temel işlemi, **Ders 5’teki lineer modelin** çekirdeği — Şekil 14.6’in eleman-eleman çarp-topla akışı, “bir satır $\times$ bir sütun = skaler” matris çarpımının ta kendisidir.
- **İleriye (Sıfırdan Dot Product Modeli):** Bu skaler, sıfırdan kurulacak modelin forward’ında satır başına bir kez hesaplanır; tüm model bu tek işlemin etrafında döner.



Şekil 14.6: Dot product (gerçek hesaplama): kullanıcı latent faktörleri (zavk) × film latent faktörleri (nitelik) → eleman-eleman çarpım → Σ = tahmin edilen puan (rose). Negatif değer rose, pozitif değer cyan. Uyumlu faktörler (aynî işaret, örn. aksiyon-sevgisi yüksek + aksiyon-filmi) çarpımı pozitif katkı verir; ters faktörler skoru düşürür. user/movie değerleri engine'den gelir.

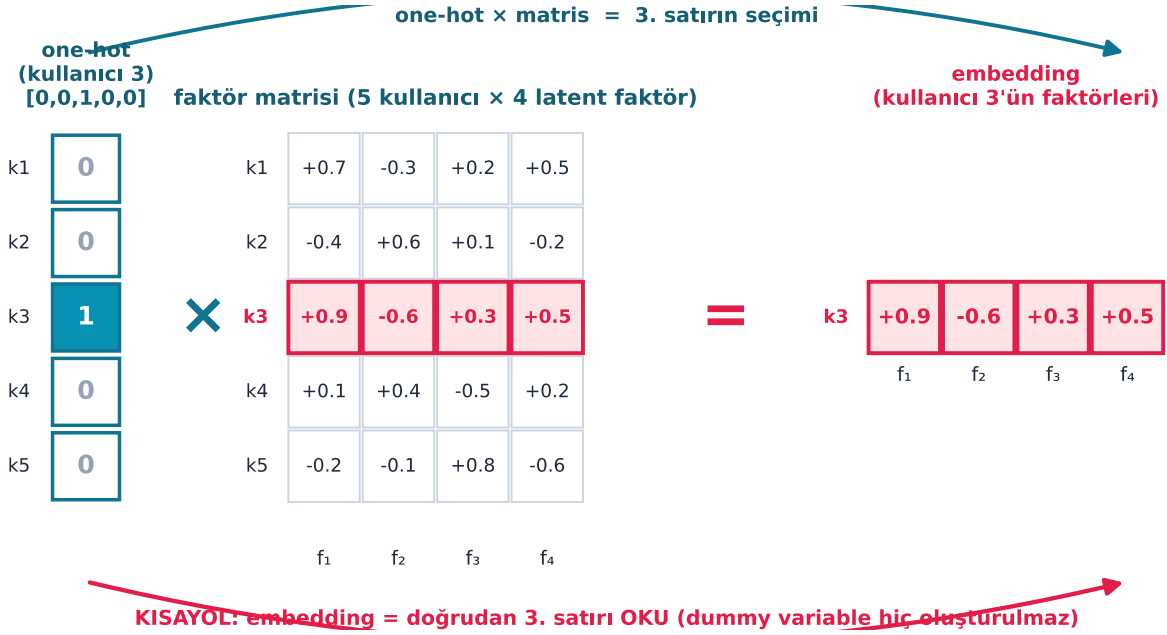
14.11 Embedding = Arama Tablosu

“Embedding” kulağa gizemli gelir ama Howard perdesini kaldırır: **embedding, bir diziden (matristen) satır aramaktan ibarettir**. Kullanıcı 14’ün faktörlerini istiyorsak, faktör matrisinin 14. satırını alırız. Bu, bir one-hot vektörle matris çarpımıyla matematiksel olarak aynıdır — embedding sadece bunun hızlı kısayoludur (dummy variable hiç oluşturulmaz).

“An embedding actually means: look something up in an array. You can think of an embedding as a computational shortcut for multiplying something by a one-hot-encoded vector — we never even have to create the dummy variables.” — Howard, 1:19:29

Şekil 14.7 bu eşitliği şematik olarak gösterir: bir one-hot vektörle faktör matrisini çarpmak = o satırı seçmek; embedding bu çarpımı atlayıp doğrudan ilgili satırı okur (alt rose ok = kısayol).

Embedding = diziden satır arama (one-hot çarpımının kısayolu)



Şekil 14.7: Embedding = arama tablosu (semantik): kullanıcı 3’ün faktörlerini almak = faktör matrisinin 3. satırını okumak. Bu, [0,0,1,0,0] one-hot vektörüyle matris çarpımına BİREBİR eşittir; embedding bunun hızlı kısayolu (dummy variable hiç oluşturulmaz). Latent faktör değerleri illüstratif — gerçek modelde öğrenilir.

💡 Builder Notu — Dummy Variable’ın Hızlı Hâli

- **Geriye (Ders 6):** Embedding = **dummy variable** + matris çarpımının hızlı hâli; “her kategori bir 0/1 sütunu” fikrinin verimli versiyonu — Şekil 14.7’in alt rose oku, dummy variable’ları hiç oluşturmadan satırı okur.
- **İleriye (NLP/Karpathy):** Token embedding’leri (Ders 4, Karpathy makemore) tam budur — kelime id’sini bir vektöre çeviren arama tablosu; aynı one-hot kısayolu dil modellerinin girişinde

çalışır.

14.12 Sıfırdan Dot Product Modeli

Howard collaborative filtering modelini PyTorch'ta bir sınıf olarak kurar. Embedding katmanları faktörleri tutar; forward metodu (PyTorch modeli çağırınca otomatik çalışır) kullanıcı ve film faktörlerini arar, dot product'larını alır. sigmoid_range çıktıyı puan aralığına (0-5) sıkıştırır.

```
class DotProduct(Module):
    def __init__(self, n_users, n_movies, n_factors, y_range=(0, 5.5)):
        self.user_factors = Embedding(n_users, n_factors)
        self.movie_factors = Embedding(n_movies, n_factors)
        self.y_range = y_range
    def forward(self, x):
        users = self.user_factors(x[:, 0])
        movies = self.movie_factors(x[:, 1])
        return sigmoid_range((users * movies).sum(dim=1), *self.y_range)

model = DotProduct(n_users, n_movies, 50)
learn = Learner(dls, model, loss_func=MSELossFlat())
learn.fit_one_cycle(5, 5e-3)
```

“PyTorch is going to call a method called forward in your class. This is where you put your calculation of your model.” — Howard, 1:29:58

forward içindeki `(users * movies).sum(dim=1)` tam **dot product**'tır (Şekil 14.6); `user_factors(x[:, 0])` ise **embedding araması**'dır (Şekil 14.7) — model bu iki işlemten ibarettir.

 **Builder Notu** — Module + forward = calc_preds'in Nesne Hâli

- **Geriye (Ders 5):** Module + forward = **Ders 5'teki calc_preds'in** nesne hâli; PyTorch'un model API'si — sıfırdan yazdığımız tahmin fonksiyonu, artık bir sınıfın forward'ı.
- **İleriye (Ders 5):** sigmoid_range, **Ders 5'teki sigmoid'in** aralık genelleştirilmiş hâli; çıktıyı 0-5.5 puan aralığına sıkıştırır, böylece model “aralık dışı” tahmin etmez.

14.13 Bias Terimleri

Sadece dot product yetmez: bazı kullanıcılar her şeye yüksek puan verir, bazı filmler herkesçe sevilir. Bunu yakalamak için her kullanıcıya ve her filme bir **bias** (sabit terim) eklenir. Model artık “bu kullanıcı genelde cömert + bu film genelde sevilir + ikisinin uyumu” der.

```
class DotProductBias(Module):
    def __init__(self, n_users, n_movies, n_factors, y_range=(0, 5.5)):
        self.user_factors = Embedding(n_users, n_factors)
        self.user_bias = Embedding(n_users, 1)
        self.movie_factors = Embedding(n_movies, n_factors)
        self.movie_bias = Embedding(n_movies, 1)
        self.y_range = y_range
    def forward(self, x):
        users = self.user_factors(x[:, 0]); movies = self.movie_factors(x[:, 1])
        res = (users * movies).sum(dim=1, keepdim=True)
        res += self.user_bias(x[:, 0]) + self.movie_bias(x[:, 1])
        return sigmoid_range(res, *self.y_range)
```

Tahmin artık “uyum (dot product) + kullanıcı cömertliği + film popülerliği” toplamıdır; bias terimleri de birer **embedding**’dir (yalnızca tek sütunlu).

💡 Builder Notu — Bias = $Wx+b$ ’deki b

- **Geriye (Ders 3):** Bias, [Ders 3’teki](#) $Wx + b$ ’deki b ile aynı fikir — sabit kaydırma terimi (§4.D “bias tuzağı”: istatistiksel yanlılık değil, modelin sabit ofseti).
- **İleriye (Weight Decay):** Bias eklemek modeli daha esnek (ve overfit’e yatkın) kılar; sıradaki [weight decay](#) tam bu esnekliği frenler.

14.14 Weight Decay (Regularization)

Bias eklenince model overfit etmeye başlayabilir. Çözüm **weight decay** (L2 regularization): kayba, katsayıların karelerinin toplamını ekleyerek modeli küçük ağırlıklar tutmaya zorlar. fastai’de `wd` parametresiyle verilir.

```
learn.fit_one_cycle(5, 5e-3, wd=0.1) # weight decay ile overfit'i azalt
```

“It’s a mathematical function that fits my intuition about what works well.” — Howard, 1:22:24

💡 Builder Notu — Occam’ın Usturası, Sayıya Dökülmüş

- **Geriye (Ders 4 / Stat 110):** Weight decay, [Ders 4’teki overfitting’in](#) panzehirlerinden biri; büyük ağırlıkları cezalandırarak modeli basit tutar (Occam’ın usturası) — [bias terimlerini](#) ekleyince doğan esneklik, weight decay ile dengelenir.
- **İleriye (collab_learner):** `wd=0.1` tek bir parametredir ama overfit/underfit dengesini kurar; [collab_learner](#) çağrısında doğrudan verilir.

14.15 collab_learner ve Embedding Boyutu

Tüm bu sıfırdan kodu fastai tek satırda sarar: `collab_learner`. Latent faktör sayısını ve puan aralığını verir. Howard, embedding boyutu için kendi “rule of thumb” formülünü kullandığını söyler — kâğıt üstünde

sezgisini bir tabloya yazıp ona fonksiyon uydurmuş; şimdi birçok makalede kullanılıyor.

```
learn = collab_learner(dls, n_factors=50, y_range=(0, 5.5))
learn.fit_one_cycle(5, 5e-3, wd=0.1)
```

“It’s a mathematical function that fits my intuition about what works well. Lots of papers now say: using fastai’s rule of thumb for embedding sizes.” — Howard, 1:22:24

Bu tek satır, [sıfırdan kurduğumuz DotProductBias](#)’ı ve [weight decay](#)’i birlikte sarar — framework, “fiddly” parçaları gizler.

💡 Builder Notu — vision_learner Ailesinin Collab Üyesi

- **Geriye (Ders 1):** `collab_learner`, [vision_learner/text_learner](#) ailesinin collaborative filtering üyesi (§4.I) — aynı `learn.fit_one_cycle` API’si, farklı veri türü.
- **İleriye (Latent Uzay):** `n_factors=50` embedding boyutudur; bu 50 boyutlu uzay, eğitimden sonra [yorumlanabilir bir latent uzaya](#) dönüşür.

14.16 Derin Öğrenme ile Collab ve Latent Uzay

Dot product yerine embedding’leri birleştirip bir sinir ağına besleyebiliriz (`use_nn=True`). Daha da önemlisi: embedding fikri her yere taşınır — kategorik tablo sütunları, NLP token’ları, hatta öneri dışı problemler. Öğrenilen embedding’ler yorumlanabilir bir **latent uzay** oluşturur (benzer filmler yakın olur).

```
learn = collab_learner(dls, use_nn=True, y_range=(0, 5.5), layers=[100, 50])
learn.fit_one_cycle(5, 5e-3, wd=0.1)
```

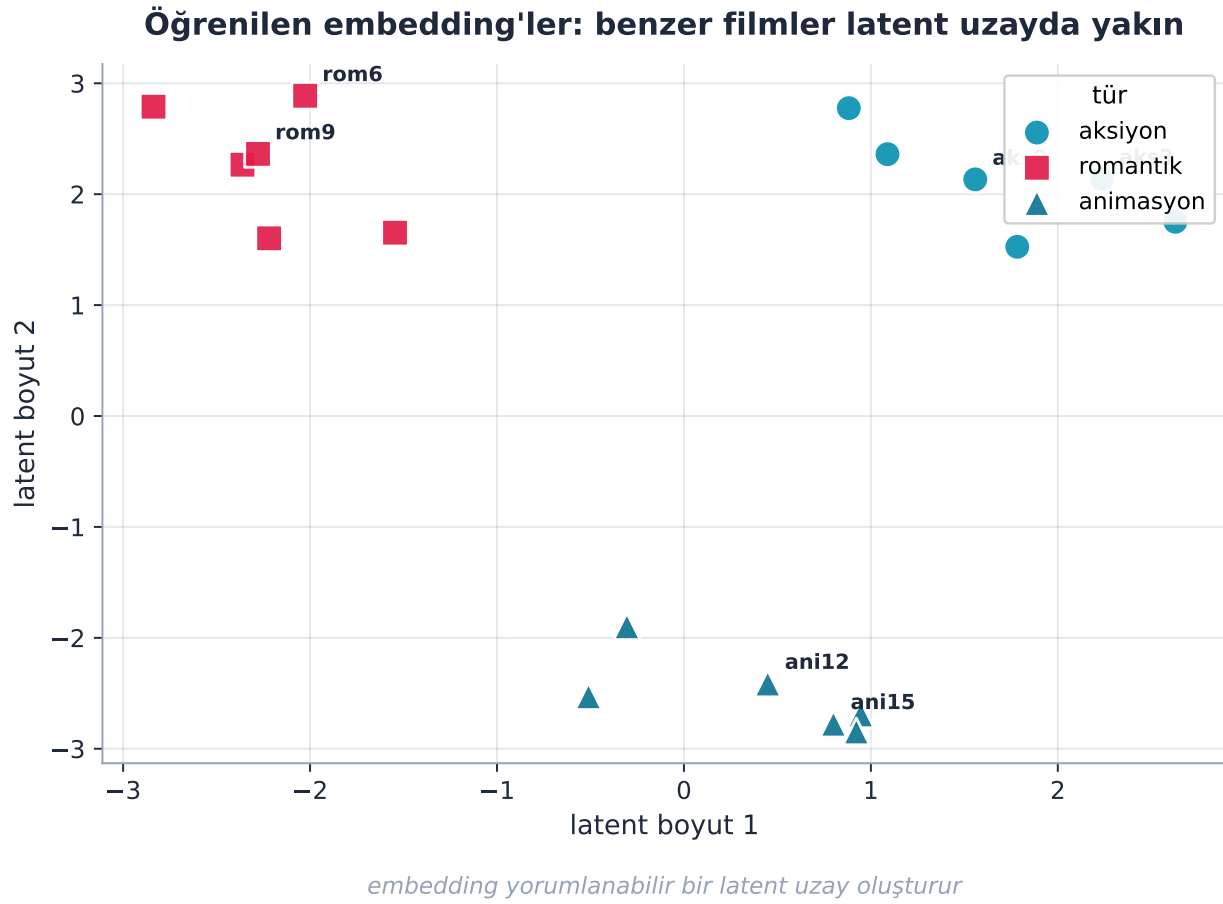
Şekil 14.8 bu yorumlanabilirliği gerçek (kümeli sentetik) 2B noktalarla gösterir: öğrenilen film embedding’leri latent uzayda türlerine göre kümelenir — aksiyon, romantik ve animasyon filmleri ayrı bölgelere düşer; benzer filmler yakın olur.

💡 Builder Notu — Embedding Her Yerde: Tablo, NLP, Öneri

- **İleriye:** Tablo modellerinde kategorik değişkenler için embedding (EmbeddingNN), NLP’de kelime embedding’leri, öneri sistemlerinde kullanıcı/ürün embedding’leri — hepsi aynı “[arama tablosu](#)” fikri; Şekil 14.8’in kümeleri, modelin yapıyı kendiliğinden bulduğunu gösterir.
- **Geriye (Latent Faktörler):** Anlamı önceden bilinmeyen [latent faktörler](#), eğitimden sonra bu kümeleri oluşturur — “aksiyon-sevgisi” eksenini biz tanımlamadan ortaya çıkar.

14.17 Kapanış

Ders 7 iki güçlü konuyu birleştirdi. İlk yarı yarışma araçlarını verdi: gradient accumulation (bellek), ensembling (doğruluk), multi-target + cross-entropy/softmax (çok-sınıflı kayıp). İkinci yarı collaborative filtering’i kurdu:



Şekil 14.8: Öğrenilen embedding'lerin latent uzayı (gerçek hesaplama — kümeli sentetik 2B noktalar): film embedding'leri türlerine göre kümelenir (aksiyon/romantik/animasyon ayrı bölgelerde). Benzer filmler yakın düşer; embedding yorumlanabilir bir latent uzay oluşturur. Eksenler latent boyut — model öğrenir, biz tanımlamayız.

latent faktörler + dot product + bias + weight decay, hepsi embedding üstünde — ki embedding sadece “bir diziden arama yapmak”tır.

💡 Builder Notu — Son Derse ve Part 2’ye Köprü

- **İleriye (Ders 8):** Son ders evrişimlere (CNN) döner; **embedding + cross-entropy + dot product** burada öğrenilen temeller Part 2 ve Karpathy boyunca tekrar tekrar karşımıza çıkacak.
- **Geriye (tüm seri):** Şekil 34.1’in yarışma-araçları → softmax/cross-entropy → collaborative-filtering → embedding zinciri, bu dersin iki yarısını tek bir hikâyede toplar — kayıp fonksiyonundan öneri çekirdeğine.

14.18 Bu Dersin Özeti

1. **Gradient accumulation:** küçük batch’lerin gradyanlarını biriktirip tek adım atarak küçük GPU’da büyük model eğitme (**Gradient Accumulation**).
2. **Ensembling:** birden çok modelin tahminlerini birleştirip doğruluğu artırma (bagging’in mimari versiyonu) (**Ensembling**).
3. **Multi-target** modeller birden çok şeyi aynı anda tahmin eder; kayıp hedeflerin toplamıdır (**Multi-Target**).
4. **Softmax** logit’leri toplamı 1 olan olasılıklara çevirir; **cross-entropy** = softmax + negatif log likelihood (**Softmax**, **Cross-Entropy**).
5. **Collaborative filtering** bir matris tamamlama problemidir: kullanıcı × ürün matrisindeki boşlukları doldurma (**Collaborative Filtering**).
6. **Latent faktörler** her kullanıcı/ürün için öğrenilen gizli sayılardır; anlamlarını model belirler (**Latent Faktörler**, **Latent Uzak**).
7. Tahmin = kullanıcı ve ürün latent faktörlerinin **dot product**’ı (+ **bias** terimleri); **weight decay** overfit’i azaltır (**Dot Product**, **Bias**, **Weight Decay**).
8. **Embedding**, bir diziden satır aramaktır — one-hot çarpımının hızlı kısayolu; her yerde (öneri, NLP, tablo) kullanılır (**Embedding = Arama**).

❗ Tek Bir Cümle

Collaborative filtering, kullanıcılar ve ürünler için öğrenilen latent faktör vektörlerinin (embedding’lerin) dot product’ıyla puanları tahmin eden bir matris tamamlama yöntemidir; ve embedding, gizemli görünse de yalnızca bir diziden hızlıca satır aramaktan ibarettir.

14.19 Kontrol Soruları

i Soru 1: Embedding nedir ve neden one-hot çarpımıyla aynı şeydir?

Cevap:

Embedding, bir matristen (faktör tablosundan) belirli bir satırı **aramaktır** — örneğin kullanıcı 14’ün faktör vektörünü almak için matrisin 14. satırını çekmek. Bu, “14. eleman 1, gerisi 0” olan bir one-hot vektörle matrisi çarpmakla matematiksel olarak birebir aynıdır: one-hot çarpımı yalnızca o satırı seçer.

Embedding bu işlemin hızlı kısayoludur — one-hot vektörü ve dummy variable’ları hiç oluşturmadan doğrudan satırı okur. Yani gizemli bir kavram değil, verimli bir arama işlemidir. (Şekil 14.7 bu eşitliği gösterir.)

i Soru 2: Collaborative filtering’de latent faktörler ve dot product nasıl çalışır? Bias neden eklenir?

Cevap:

Her kullanıcı ve her ürün için (örn. 5) **latent faktör** (gizli sayı) öğrenilir; anlamlarını model kendisi belirler (belki “aksiyon sevgisi”, “eskilik”). Bir kullanıcının bir ürünü ne kadar seveceği, ikisinin faktör vektörlerinin **dot product**’ıdır (eleman eleman çarp, topla) — uyumlu faktörler yüksek skor verir. **Bias** eklenir çünkü dot product yalnızca “uyum”u yakalar; ama bazı kullanıcılar her şeye yüksek puan verir, bazı ürünler herkesçe sevilir. Kullanıcı ve ürün bias’ları bu genel eğilimleri ayrıca modeller, böylece tahmin “uyum + kullanıcı cömertliği + ürün popülerliği” olur. (Şekil 14.6 dot product’ı, Şekil 14.5 doldurulacak boşlukları gösterir.)

i Soru 3: Cross-entropy loss ne yapar ve softmax ile ilişkisi nedir?

Cevap:

Cross-entropy iki adımdır: önce **softmax** modelin ham çıktılarını (logit) toplamı 1 olan bir olasılık dağılımına çevirir ($e^{z_i} / \sum e^z$); sonra **negatif log likelihood** doğru sınıfa verilen olasılığın logaritmasını cezalandırır ($-\log p$). Doğru sınıfa yüksek olasılık → küçük kayıp; düşük olasılık → büyük kayıp. PyTorch `F.cross_entropy` bunu tek adımda ve log-softmax kullanarak sayısal olarak kararlı biçimde yapar. Birden çok şey aynı anda doğru olabiliyorsa softmax yerine her çıktıya ayrı sigmoid + binary cross-entropy kullanılır. (Şekil 14.3 softmax’ı, Şekil 14.4 kaybı gösterir.)

i Soru 4: Gradient accumulation neden daha büyük batch ile (neredeyse) aynıdır? (builder bağlantısı)

Cevap:

Bir batch’in gradyanı, o batch’teki örneklerin gradyanlarının ortalamasıdır. Gradient accumulation, büyük bir batch’i belleğe sığmadığı için birkaç küçük parçaya böler; her parçanın gradyanını hesaplar ama optimizer adımını **atmadan** biriktirir (`backward()` gradyanları toplar). `n_acc` parça birikince tek adım atılır. Toplanan gradyan, büyük batch’in gradyanıyla matematiksel olarak (neredeyse) aynıdır — fark yalnızca BatchNorm gibi batch-istatistiği kullanan katmanlarda küçük sapmalar olur. Builder açısından: bu, sınırlı VRAM’de büyük efektif batch elde etmenin standart yoludur (Karpathy GPT-2 eğitiminde de kullanır). (Şekil 14.2 akışı gösterir.)

14.20 Egzersizler

Egzersiz 1 (Direkt uygulama). MovieLens verisinde `CollabDataLoaders.from_df` ile veri yükle ve `collab_learner(n_factors=50, y_range=(0,5.5))` ile bir model eğit.

Egzersiz 2 (İki-aşamalı). Sıfırdan `DotProduct` ve `DotProductBias` sınıflarını kur; bias eklemenin doğruluğu nasıl değiştirdiğini kıyasla.

Egzersiz 3 (Edge case). `wd` (weight decay) değerini 0, 0.1, 1.0 yapıp validation kaybını izle; overfit ile underfit

arasındaki dengeyi gözlemle.

Egzersiz 4 (Python ile doğrulama). Bir one-hot vektör oluştur ve `onehot @ user_factors` ile `user_factors[index]` sonuçlarının aynı olduğunu doğrula (embedding = arama).

Egzersiz 5 (Sonraki dersin habercisi). Öğrenilen film embedding'lerini PCA ile 2 boyuta indir ve benzer filmlerin latent uzayda yakın olduğunu gözlemle.

14.21 Sonraki Ders İçin Hazırlık

Ders 8: Evrişimler / CNN (Convolutions)

Son ders görüntülerin nasıl verimli işlendiğine iner: **convolution**. Bir filtreyi görüntü üzerinde kaydırarak yerel desenleri (kenar, köşe) yakalama — CNN'lerin temel işlemi. Ayrıca collaborative filtering ve embedding'i tablo modellerine bağlar.

Ana konular:

- Convolution (evrişim) ve filtreler
- CNN mimarisi
- Embedding'in tabloda kullanımı
- Serinin kapanışı



Ders 8 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 4 — `collab_learner + embedding=arama`).
- Bir öneri veri setinde latent faktörleri eğit ve yorumla.
- Ana cümleyi tekrar oku: “Embedding = bir diziden arama; collab = latent faktörlerin dot product'ı.”

14.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Gradient accumulation	Küçük batch gradyanlarını biriktirip tek adım at	8:04
Ensembling	Birden çok modelin tahminlerini birleştirme	23:01
Multi-target	Tek modelle birden çok şeyi tahmin etme	37:51
Softmax	Logit'leri toplamı 1 olan olasılıklara çevirme	45:45
Cross-entropy	Softmax + negatif log likelihood; çok-sınıflı kayıp	46:24
Collaborative filtering	Kullanıcı × ürün matrisi tamamlama (öneri)	1:14:00
Latent faktör	Kullanıcı/ürün için öğrenilen gizli sayılar	1:21:00

Kavram	Tanım	Howard'da
Dot product	İki faktör vektörünün eleman çarpımı toplamı	1:08:00
Embedding	Bir diziden satır arama (one-hot çarpımının kısayolu)	1:19:29
Bias	Kullanıcı/ürün genel eğilimini yakalayan sabit terim	1:29:58
Weight decay (wd)	L2 regularization; büyük ağırlıkları cezalandırır	1:22:24
collab_learner	fastai'nin collaborative filtering Learner'ı	1:22:24

14.23 ML Builder Bağlantıları

💡 Builder Notu — 6 ML Köprüsü

Bu ders, yarışma araçlarından öneri sistemlerinin çekirdeğine uzanır; köprülerin özeti:

1. **Embedding** → kategorik veri, NLP token'ları ve öneri için ortak “arama tablosu” fikri; her yerde ([Embedding = Arama, Latent Uzay](#)).
2. **Cross-entropy/softmax** → çok-sınıflı sınıflandırmanın standart kaybı; Karpathy make-more/GPT'nin temeli ([Softmax, Cross-Entropy](#)).
3. **Collaborative filtering** → matris tamamlama; Netflix/Spotify önerilerinin çekirdeği ([Collaborative Filtering](#)).
4. **Gradient accumulation** → sınırlı VRAM'de büyük efektif batch; Karpathy GPT-2 eğitiminde standart ([Gradient Accumulation](#)).
5. **Ensembling** → bagging'in mimari versiyonu; yarışma kazanan çözümlerin standardı ([Ensembling](#)).
6. **Weight decay** → overfit panzehiri (Ders 4); büyük ağırlıkları cezalandırır ([Weight Decay](#)).

❗ Bu dersten tek bir şey alıp gideceksen

Bu dersten tek bir şey alıp gideceksen: derin öğrenme jargonu seni korkutmak için tasarlanmış gibidir ama altı basittir. “Embedding” sadece bir diziden satır aramaktır; “collaborative filtering” kullanıcı ve ürün için öğrenilen gizli sayıların dot product'ıyla boş hücreleri doldurmaktır. Basitliği keşfettiğinde, kendi başına yapabileceğini fark edersin — ve yapabilirsin.

15 Evrişimler / CNN — Convolution ve Embedding’in Her Yerdeği (Convolutions)

Part 1’in son dersi iki şeyi tamamlar: önce Ders 7’nin embedding hikâyesini bitirir — embedding’i sıfırdan kurma, yorumlama, mesafeyle benzerlik ve embedding’in collaborative filtering’den NLP’ye, oradan tablo verisine (entity embeddings) taşınması: hepsi aynı ‘arama tablosu’; sonra dersin asıl başlığı convolution — küçük bir kerneli (filtre) görüntü üzerinde kaydırıp her konumda dot product alan, kenar gibi yerel desenleri yakalayan, tam bağlı ağa kıyasla çok az parametre kullanan işlem; ve CNN — conv + ReLU + pooling + BatchNorm yığını: her katman görüntüyü küçültürken kanalı artırır, sonunda tek bir tahmin kalır

Bölüm bilgisi

- **Howard’ın videosu:** [course.fast.ai — Lesson 8: Convolutions \(CNNs\)](https://course.fast.ai/lesson8.html) (~97 dk)
- **Seri:** Practical Deep Learning for Coders — Part 1, Ders 8 (FİNAL)
- **Hoca:** Jeremy Howard
- **Playlist:** [Practical Deep Learning Part 1 \(2022\)](#)
- **Notebook:** [fastbook — 13_convolutions](#)
- **Okuma süresi:** ~36 dk

15.1 Bu Derste Ne Var?

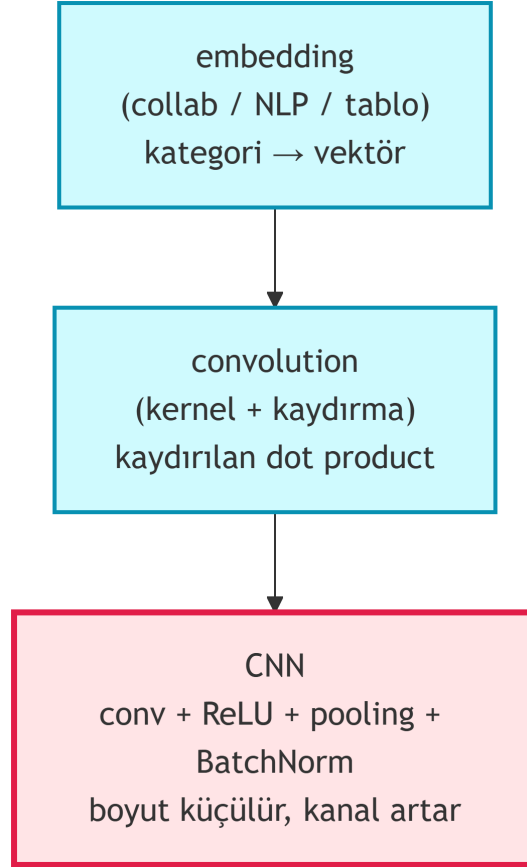
Part 1’in son dersi iki şeyi tamamlar. İlk yarı, Ders 7’nin **embedding** hikâyesini bitirir: embedding’i sıfırdan kurma, yorumlama, mesafeyle benzerlik, ve embedding’in NLP ile tablo verisine (entity embeddings) taşınması. İkinci yarı dersin başlığıdır: **convolution** — CNN’lerin, yani görüntü modellerinin temel işlemi.

Üç temel fikir:

1. **Embedding her yerde** — collaborative filtering, NLP ve tablo verisi; hepsi aynı “arama tablosu”. Tablo verisinde kategorik sütunlar için embedding (entity embeddings) güçlü bir tekniktir ([Embedding Her Yerde](#) → [Tablo İçin Embedding](#)).
2. **Convolution = kaydırılan mini dot product** — küçük bir filtreyi (kernel) görüntü üzerinde kaydırıp yerel desenleri (kenar) yakalama; tam bağlı ağa kıyasla çok az parametre ([Convolution: Kenar Bulma](#) → [Filtreyi Kaydırma](#)).
3. **CNN = conv + ReLU + pooling yığını** — her katman görüntüyü küçültürken kanal sayısını artırır; sonunda tek bir tahmin kalır ([simple_cnn](#) → [Pooling](#) → [BatchNorm](#)).

“You can think of a convolution as being a sliding window of little mini dot products.” — Howard, 53:00

Şekil 34.1 bu üç fikri tek bir haritada birleştirir: her veri tipinde kategoriye vektöre çeviren **embedding**'den, görüntü üzerinde kaydırılan **convolution**'a, oradan conv + ReLU + pooling + BatchNorm'dan oluşan **CNN**'e ilerler — rose vurgulu “CNN” düğümü dersin doruğudur.



Şekil 15.1: Ders 8'in kavram haritası: embedding (collaborative filtering / NLP / tablo — kategoriye vektöre çeviren ortak arama tablosu) → convolution (kerneli görüntüde kaydır, her konumda dot product al, kenar gibi yerel desenleri yakala) → CNN (conv + ReLU + pooling + BatchNorm yığını; boyut küçülür, kanal artar — rose vurgulu). Tek bir zincir: önce her veri tipindeki arama tablosunu (embedding) tamamla, sonra görüntü için kaydırmalı dot product'a (convolution) geç, en sonunda bu işlemleri yığarak CNN'i kur.



Builder Notu — İki Yarı, Tek Hikâye: Embedding'in Her Yerdeliği ve Convolution

- **Geriye (Ders 1/3/7):** Bu ders [Ders 1'deki](#) “görüntü = sayı”yı, [Ders 3'teki](#) Zeiler-Fergus öznetelik dedektörlerini ve [Ders 7'deki](#) embedding'i birleştirip somutlaştırır.
- **İleriye (Part 2 / Karpathy):** [Convolution](#) Part 2'de (Ders 15, 19) U-Net ve diffusion'ın temeli; [entity embeddings](#) modern tablo modellerinin standardı; Şekil 34.1'in rose “CNN” düğümü buradan büyür.
- **Tek cümle:** Convolution, az parametreyle yerel desen yakalayan kaydırmalı bir dot product'tır; embedding ise her veri tipinde kategoriye vektöre çeviren arama tablosudur.

15.2 Part 1 Finali ve Embedding'i Sıfırdan Kurma

Howard son derse, [Ders 7'nin embedding'ini sıfırdan](#) kurarak başlar. PyTorch'un Embedding katmanı aslında basittir: eğitilecek bir parametre matrisi oluşturur ve onu küçük rastgele sayılarla doldurur.

```
def create_params(size):
    # eğitilecek bir parametre matrisi: küçük normal rastgele sayılarla doldur
    return nn.Parameter(torch.zeros(*size).normal_(0, 0.01))
```

“This is the last lesson of Part One.” — Howard, 0:00

Bu, Ders 7'deki Embedding'in içini açar: gizem yok, sadece eğitilebilir bir tablo — [embedding'in arama tablosu](#) olduğu fikrinin kod hâli.

 Builder Notu — nn.Parameter: Gradient Descent'in Güncellediği Tablo

- **Geriye (Ders 5/7):** nn.Parameter, [gradient descent'in](#) güncelleyeceği ağırlıklardır; embedding bunların bir arama tablosu hâli — [Ders 7'de elle kurduğumuz](#) Embedding tam bu create_params ile aynı işi yapar.
- **İleriye (Part 2):** nn.Parameter ile saklanan bu ağırlıklar Part 2'de sıfırdan optimize edilir — matmul'dan backprop'a kadar her parametrenin gradyanı elle hesaplanıp güncellenecek; requires_grad mekanizması orada açılır.

15.3 Embedding Yorumlama

Eğitilmiş embedding'ler yorumlanabilir. Howard film faktörlerinin en düşük/en yüksek değerli olanlarına bakar ve hangi filmlerin uçlarda olduğunu görür — model, hiç söylenmeden, filmleri anlamlı eksenlerde (örn. klasik vs gişe) düzenlemiştir.

“What we could do is we could find which Movie-IDs [have the lowest factors], the names of those movies.” — Howard, 12:32

Bu, [Ders 7'deki latent uzayın](#) yorumlanabilirliğinin devamıdır: model, anlamı biz tanımlamadan eksenleri kendi keşfeder.

 Builder Notu — Latent Uzayı İncelemek = Yorumlanabilirlik

- **Geriye (Ders 7):** Bu, [Ders 7'deki latent faktörlerin](#) yorumlanabilirliğinin devamı; orada PCA ile gördüğümüz “klasik vs gişe” eksenini, burada en uç film faktörlerine bakarak doğrudan okunur.
- **İleriye (yorumlanabilirlik):** Öğrenilen latent uzayı incelemek, modelin “ne öğrendiğini” anlamın güçlü bir yolu; “bu eksen ne demek?” sorusu modelin iç temsiliyi açar.

15.4 fastai'de Collab: EmbeddingDotBias

Ders 7'de sıfırdan kurduğumuz dot-product-bias modelini fastai hazır sunar: EmbeddingDotBias. collab_learner bunu otomatik kullanır — elle yazdığımız sınıfın kütüphane hâli.

“EmbeddingDotBias, so we can take a look at that.” — Howard, 18:11

💡 Builder Notu — Sıfırdan Bilmek, Hazırı Güvenle Kullanmak

- **Geriye (Ders 7):** Elle kurduğün DotProductBias = fastai'nin EmbeddingDotBias'ı; sıfırdan bilmek, hazırı güvenle kullanmanı sağlar — “framework sihir değil, bizim yazdığımızın temiz hâli.”
- **İleriye (Part 2):** Bu “önce sıfırdan, sonra hazırı” disiplini Part 2'nin omurgasıdır — Learner ve optimizier dâhil her soyutlama “from the foundations” yeniden kurulup ancak sonra kütüphane hâliyle kullanılır.

15.5 Embedding Mesafesi

Embedding'ler bir uzayda nokta olduğundan, iki film arasındaki **mesafeyi** ölçebiliriz (Öklid mesafesine benzer, normalize edilmiş). Yakın embedding'ler benzer filmler demektir. Howard bir filme en yakını sorar ve “Dial M for Murder”a en yakın filmi bulur — mantıklı bir sonuç.

“Based on this embedding distance, the closest is ‘Dial M for Murder’, which makes a lot of sense.” — Howard, 23:16

Bu mesafe fikri **embedding'in her yerdeliğine** köprüdür: kategoriye vektöre çeviren her tabloda “yakınlık = benzerlik” geçerlidir.

💡 Builder Notu — Semantic Search ve Vektör Veritabanlarının Çekirdeği

- **Geriye (Ders 7 dot product):** Mesafe, Ders 7'deki dot product'ın ikiz fikridir — dot product benzerliği büyütür, mesafe ise farkı ölçer; ikisi de embedding'leri aynı vektör uzayında karşılaştırır.
- **İleriye (production):** Embedding mesafesi, semantic search ve öneri sistemlerinin (“benzer ürünler”) çekirdeğidir; vektör veritabanlarının (Qdrant vb.) ve RAG sistemlerinin temel işlemi — içerikleri embedding'e çevir, en yakın komşuları ara.

15.6 Derin Öğrenme ile Collaborative Filtering

Dot product yerine, kullanıcı ve film embedding'lerini birleştirip bir sinir ağına besleyebiliriz. fastai bunu PyTorch'un hazır işlevleriyle kolaylaştırır; Ders 7'de sıfırdan yazdığımız yapının daha esnek hâli.

“A neural net exactly the same as the style that we created, but we're using PyTorch's functionality to do it more easily.” — Howard, 24:26

💡 Builder Notu — Sinir Ağı Tabanlı Collab: Daha Esnek

- **Geriye (Ders 7):** Bu, [Ders 7’de sıfırdan yazdığımız](#) collab modelinin sinir ağı varyantı; orada embedding’leri dot product’la birleştirmiştik, burada aynı embedding’ler concat edilip katmanlara beslenir.
- **İleriye:** Sinir ağı tabanlı collab, ek özellikleri (kullanıcı yaşı, tür) kolayca dâhil eder; saf dot product’tan daha esnektir — embedding’leri girdi olarak alıp katmanlarla işler.

15.7 NLP için Embedding

Embedding fikri metne de taşınır: her kelime/token bir embedding vektörüne çevrilir ([Ders 4’teki](#) tokenization + numericalization’ın devamı). Model, kelimeleri anlamlı bir vektör uzayında konumlandırır.

“Look up something in an array. And then, inside the model, we’re basically [working with those vectors].” — Howard, 30:28

Bu, “kelime id → vektör” — yani [embedding’in arama tablosu](#) fikrinin metne uygulanmış hâli; sonraki bölümdeki [embedding’in her yerdeliğinin](#) ikinci kolu.

💡 Builder Notu — Token Embedding’leri = Karpathy’nin İlk Katmanı

- **Geriye (Ders 7 arama tablosu):** “Kelime id → vektör”, [Ders 7’deki embedding arama tablosunun](#) metne uygulanmış hâli; kullanıcı/film id’si yerine token id’si aynı diziden satır çeker.
- **İleriye (Karpathy):** Token embedding’leri Karpathy makemore ve GPT’nin ilk katmanı; “kelime id → vektör” tam bu arama tablosu — dil modeli, bu embedding’leri attention ve MLP ile işleyerek sonraki token’ı tahmin eder.

15.8 Tablo İçin Embedding (Entity Embeddings)

En güçlü genelleme: tablo verisindeki **kategorik sütunlar** için embedding (entity embeddings). Mağaza, ürün, gün gibi kategoriler, dummy variable yerine öğrenilen vektörlerle temsil edilir. `tabular_learner` bunu otomatik yapar. Howard, Rossman mağaza satış tahmini yarışmasında bu yöntemin (entity embeddings) yayımlanmış bir makaleye dönüştüğünü anlatır.

“It’s not just Collaborative Filtering and NLP but also Tabular Analysis.” — Howard, 30:28

Bu üçüncü kol — collaborative filtering, NLP ve tablo — sıradaki [Embedding Her Yerde](#) figürünü tamamlar: aynı mekanizma üç farklı veri tipinde.

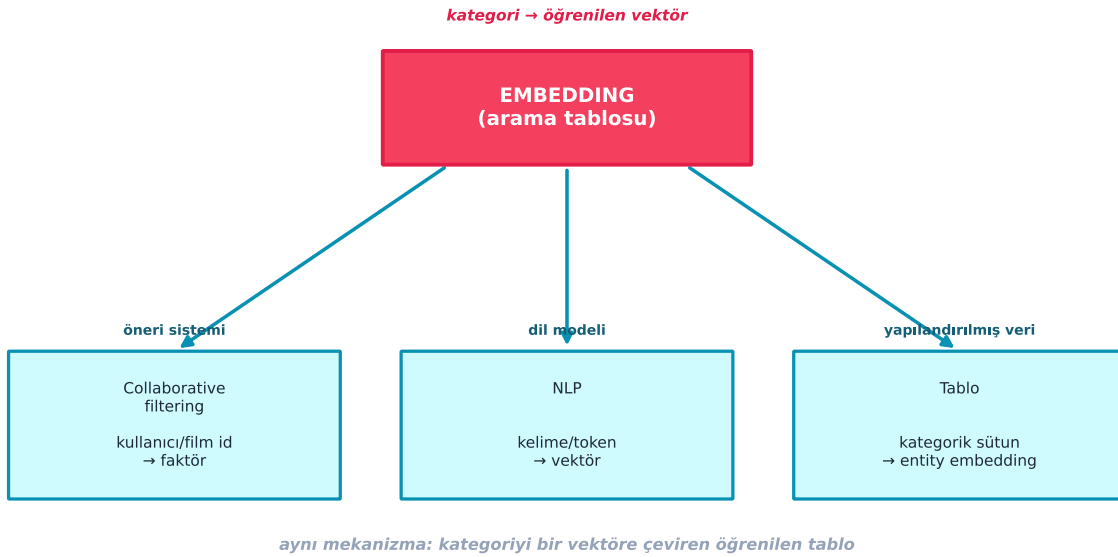
💡 Builder Notu — Dummy Variable’dan Öğrenilen Vektöre

- **Geriye (Ders 5/6):** [Ders 5’te kategorikler dummy variable](#), [Ders 6’da ağaç için ham koddu](#); entity embeddings ise onları öğrenilen vektörlere çevirir — sinir ağının tablo gücü.
- **İleriye (production):** Entity embeddings, tablo derin öğrenmesinin standardı; kategorinin “anlamını” öğrenir (benzer mağazalar latent uzayda yakın olur).

15.9 Embedding Her Yerde

Dersin ilk yarısının özeti tek bir resimde toplanır: **embedding her yerdedir**. Collaborative filtering (kullanıcı/film id), NLP (kelime/token) ve tablo verisi (kategorik sütun) — üçü de aynı “arama tablosu”. Kategoriyi öğrenilen bir vektöre çeviren tek fikir, üç farklı alanda yeniden ortaya çıkar.

Şekil 15.2 bu birliği gösterir: merkezde **embedding** (rose vurgulu, “arama tablosu”), üç kola ayrılır — **collaborative filtering**, **NLP** ve **tablo (entity embeddings)** — hepsi aynı mekanizma.



Şekil 15.2: Embedding her yerde (sematik): collaborative filtering (kullanıcı/ürün id), NLP (kelime token) ve tablo verisi (entity embeddings) — üçü de aynı ‘arama tablosu’. Kategoriyi öğrenilen bir vektöre çeviren tek fikir.

💡 Builder Notu — Tek Mekanizma, Üç Alan

- **Geriye (Ders 4/6/7):** Üç kol da daha önce ayrı ayrı gördüğümüz şeyler: **NLP tokenization**, **tablo kategorikleri**, **collab latent faktörleri** — Şekil 15.2 bunları tek fikirde birleştirir.
- **İleriye (Karpathy / production):** “Kategori → öğrenilen vektör” modern derin öğrenmenin her yerinde; öneri, dil modeli ve yapılandırılmış veri aynı arama tablosunu paylaşır.

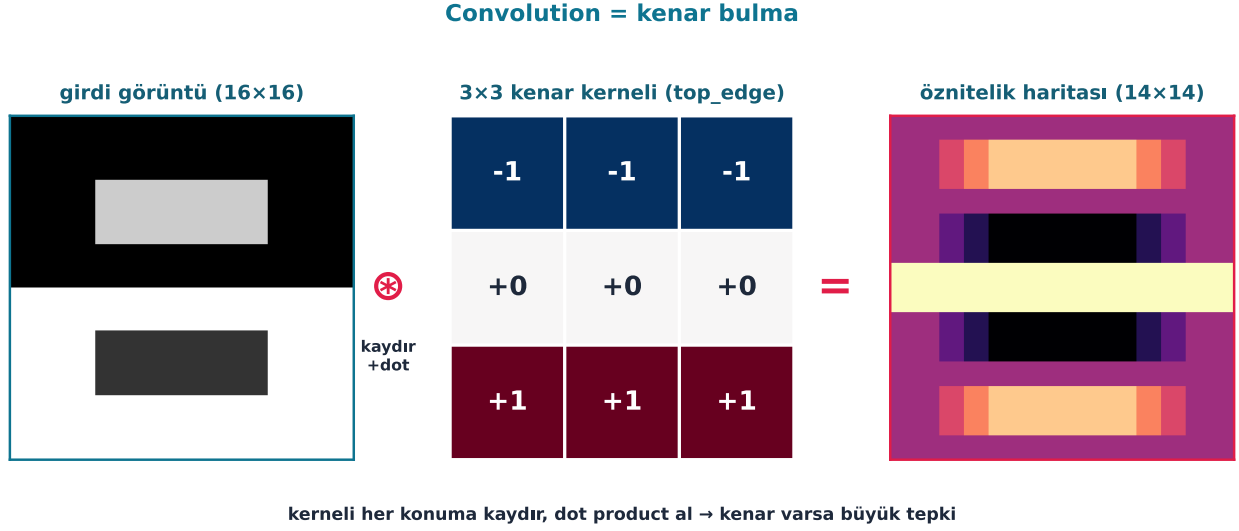
15.10 Convolution: Kenar Bulma

İkinci yarı dersin başlığıdır. Howard convolution’ı bir **kenar bulucu** ile tanıtır. Küçük bir 3×3 matris (kernel) tasarlar: üst satır -1, alt satır +1. Bu kernel bir görüntü yamasıyla eleman eleman çarpılıp toplanınca, yatay kenar varsa büyük, yoksa küçük bir sayı verir.

```
top_edge = tensor([[ -1, -1, -1],
                   [  0,  0,  0],
                   [  1,  1,  1]]).float() # yatay kenar bulucu kernel
```

“How did I do this? I did this using something called a convolution.” — Howard, 44:33

Şekil 15.3 bu işlemi gerçek hesaplamayla gösterir: solda girdi görüntü, ortada 3×3 top_edge kerneli (üst -1, alt +1), sağda öznelik haritası — kerneli her konuma kaydırıp dot product alınca yatay kenarlar büyük tepki verir. Bu, [Ders 3'teki](#) Zeiler-Fergus “öğrenilmiş kenar dedektörleri”nin elle kurulmuş hâlidir.



Şekil 15.3: Convolution = kenar bulma (gerçek hesaplama): 3×3 kenar kerneli (top_edge: üst -1, alt +1) görüntünün her konumuna kaydırılıp dot product alınır; sonuç öznelik haritası yatay kenarları vurgular (büyük tepki). Zeiler-Fergus dedektörlerinin elle hali.

💡 Builder Notu — Elle Kernel vs Öğrenilen Kernel

- **Geriye (Ders 3):** Bu, [Ders 3'teki](#) Zeiler-Fergus “öğrenilmiş kenar dedektörleri”nin elle kurulmuş hâli; Şekil 15.3’in top_edge’i bir insanın tasarladığı kernel — CNN bunları kendi öğrenir.
- **İleriye (Filtreyi Kaydırma):** Tek bir konumda hesaplanan bu dot product, sıradaki bölümde tüm görüntüye kaydırılarak öznelik haritasına dönüşür.

15.11 apply_kernel: Filtreyi Kaydırma

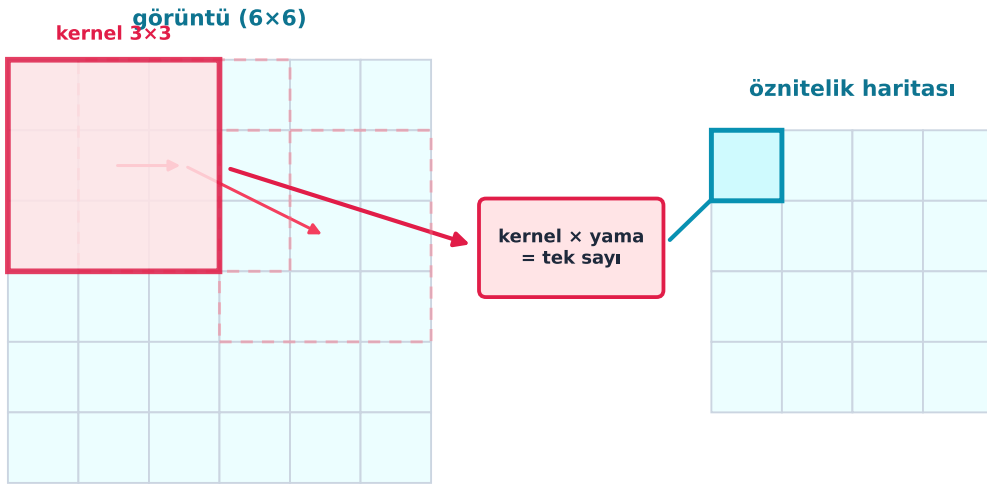
Bir convolution, bu kerneli görüntünün **her konumuna kaydırarak** uygular. Her konumda kernel ile o yamanın dot product’ı alınır; sonuç yeni bir “öznelik haritası”dır (feature map). Howard bunu açıkça “kaydırılan mini dot product’lar” diye tanımlar.

```
def apply_kernel(row, col, kernel):
    # (row,col) merkezli 3x3 yamayı kernel ile carpip topla
    return (im3_t[row-1:row+2, col-1:col+2] * kernel).sum()
```

```
# kerneli tum konumlara kaydir -> oznitelik haritasi
rng = range(1, 27)
top_edge3 = tensor([[apply_kernel(i, j, top_edge) for j in rng] for i in rng])
```

“You can think of a convolution as being a sliding window of little mini dot products. The size of this is called its kernel size — this is a 3 by 3 kernel.” — Howard, 53:00

Şekil 15.4 bu kaydırmayı şematik olarak gösterir: kernel (rose pencere) görüntü üzerinde her konuma kayar; her konumda kernel $\times$ yama bir tek sayı üretir, bu sayılar öz nitelik haritasını oluşturur. Her konumdaki işlem bir dot product'tır — convolution, mekânsal olarak tekrarlanan dot product.



her konum bir dot product; pencere kayar → öz nitelik haritası

Şekil 15.4: Kaydırılan mini dot product (şematik): kernel (rose pencere) görüntü üzerinde her konuma kayar; her konumda kernel ile o yamanın dot product'ı bir sayı üretir → bunlar öz nitelik haritasını (sağda) oluşturur. Kernel boyutu = pencere boyutu (3 $\times$ 3).



Builder Notu — Convolution = Mekânsal Olarak Tekrarlanan Dot Product

- **Geriye (18.06 / Ders 5):** Her konumdaki işlem bir dot product; convolution = mekânsal olarak tekrarlanan dot product — Şekil 15.4'in her penceresi, 18.06'nın “satır $\times$ yama = skaler” işlemini bir kez daha çalıştırır.
- **İleriye (F.conv2d):** Bu açık döngü öğreticidir ama yavaştır; sıradaki bölüm aynı işlemi optimize biçimde yapar.

15.12 F.conv2d: Verimli Convolution

Elle döngü yavaştır; PyTorch `F.conv2d` aynı işlemi optimize biçimde, hatta birden çok kerneli (kanalı) ve tüm batch'i aynı anda yapar.

```
batch_features = F.conv2d(xb, edge_kernels) # tum batch + tum kerneller, optimize
```

Bu, `apply_kernel`'in açık döngüsünü optimize çekirdek koda çevirir — sonuç aynı, hız çok daha yüksek (GPU'da paralel).

💡 Builder Notu — Açık Döngüyü Optimize Çekirdeğe Çevirmek

- **Geriye (Ders 5):** Broadcasting/matris çarpımı gibi, conv2d de [açık döngüyü](#) optimize çekirdek koda çevirir (GPU'da paralel) — [Ders 5'teki](#) “döngü yerine matris işlemi” dersinin görüntü hâli.
- **İleriye:** `F.conv2d` birden çok kerneli (kanalı) aynı anda uygular; her kanal farklı bir desen (yatay kenar, dikey kenar, köşe) öğrenir.

15.13 Neden CNN? Parametre Paylaşımı

Kritik avantaj: 28×28 bir görüntüyü tam bağlı bir katmana versek 784 ağırlık gerekirdi (her piksel için biri). Convolution ise aynı 3×3 kerneli (yalnızca **9 parametre**) tüm görüntüde paylaşır. Bu hem parametre sayısını düşürür hem de “kenar her yerde kenardır” (öteleme-değişmezlik) bilgisini gömer.

“We don't have 784 parameters, we only have nine parameters.” — Howard, 44:33

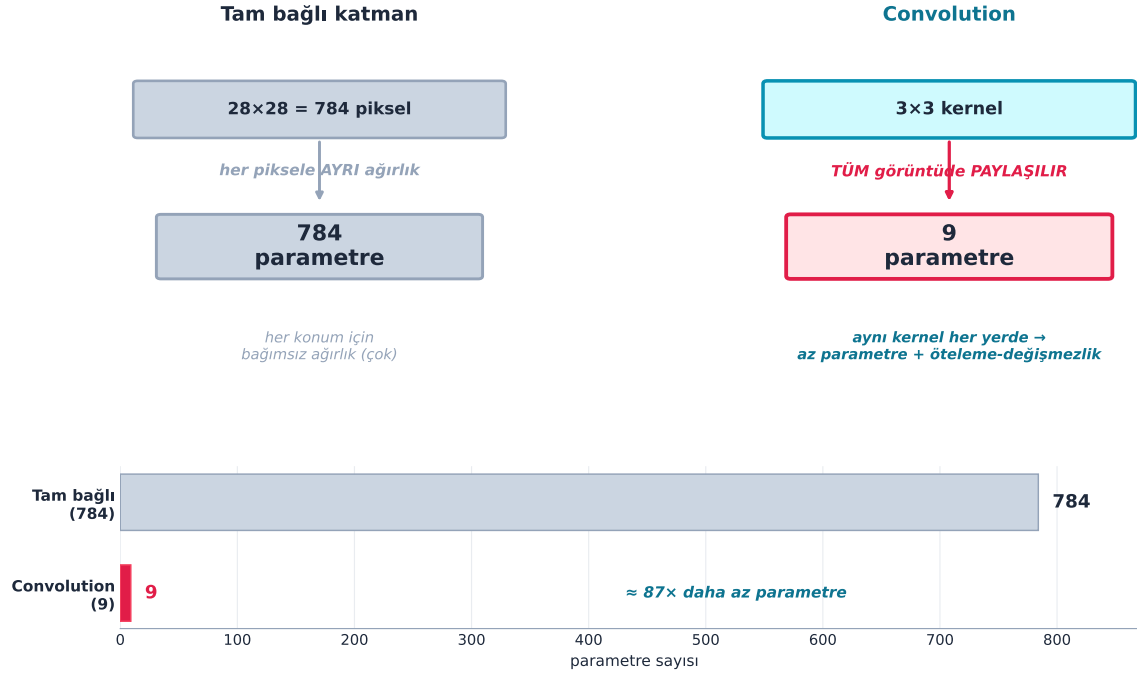
Şekil 15.5 bu karşıtlığı gösterir: solda tam bağlı katman (her piksele ayrı ağırlık $\rightarrow 784$), sağda convolution (tek 3×3 kernel tüm görüntüde paylaşılır $\rightarrow 9$). Aynı kernel her yerde $\rightarrow$ az parametre + öteleme-değişmezlik = görüntü için doğru inductive bias.

💡 Builder Notu — Paylaşım + Yerellik = Doğru Inductive Bias

- **Geriye (Ders 5):** Tam bağlı katman her piksele ayrı ağırlık verir; conv aynı ağırlığı paylaşır — daha az parametre, daha iyi genelleme. Şekil 15.5'in $784 \rightarrow 9$ karşıtlığı bu farkın somut hâli.
- **İleriye (18.06):** Parametre paylaşımı + yerel bağlantı = convolution'ı görüntü için doğru “inductive bias” yapan şey; modelin yapıya dair önbilgisini mimariye gömer.

15.14 simple_cnn Mimarisi

Bir CNN, **conv + ReLU** katmanlarının yığındır. Howard `stride=2` kullanır: her katman görüntüyü yarıya indirirken ($28 \rightarrow 14 \rightarrow 7 \rightarrow 4 \rightarrow 2 \rightarrow 1$) kanal sayısını artırır. Sonunda tek bir aktivasyon kalır ve Flatten ile tahmine dönüşür.



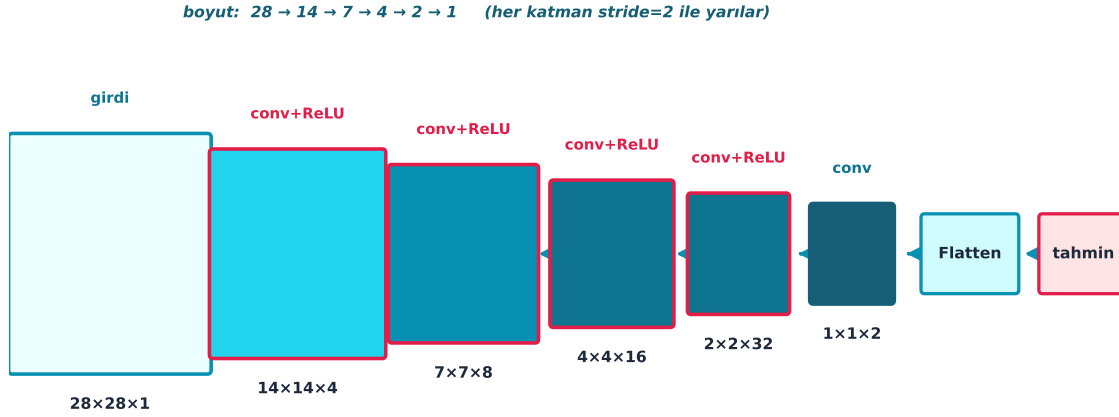
Şekil 15.5: Parametre paylaşımı (sematik): tam bağlı katman 28×28 görüntü için 784 ağırlık ister; convolution aynı 3×3 kerneli (yalnızca 9 parametre) tüm görüntüde paylaşır. Az parametre + ‘kenar her yerde kenardır’ (öteleme-değişmezlik) = görüntü için doğru inductive bias.

```
def conv(ni, nf, ks=3, act=True):
    res = nn.Conv2d(ni, nf, stride=2, kernel_size=ks, padding=ks//2)
    if act: res = nn.Sequential(res, nn.ReLU()) # conv + ReLU
    return res

simple_cnn = sequential(
    conv(1, 4), # 14x14
    conv(4, 8), # 7x7
    conv(8, 16), # 4x4
    conv(16, 32), # 2x2
    conv(32, 2, act=False), # 1x1
    Flatten(),
)
learn = Learner(dls, simple_cnn, loss_func=F.cross_entropy, metrics=accuracy)
```

“A convolutional neural network is very, very, very simple.” — Howard, 45:00

Şekil 15.6 bu yığını şematik olarak gösterir: her conv + ReLU katmanı stride=2 ile boyutu yarılar (28→14→7→4→2→1) ve kanalı artırır (1→4→8→16→32→2); sonunda Flatten ile tek tahmine iner. Bu, Ders 3’ün “ReLU toplamı”nın mekânsal hâli; kayıp ise Ders 7’deki F.cross_entropy.



her katman: stride=2 boyutu yarılar, kanal sayısı artar; sonunda Flatten ile tek tahmine iner — Ders 3'ün 'ReLU toplamı'nın mekânsal hâli

Şekil 15.6: simple_cnn mimarisi (şematik): conv + ReLU yığını; her katman stride=2 ile görüntüyü küçültür ($28 \rightarrow 14 \rightarrow 7 \rightarrow 4 \rightarrow 2 \rightarrow 1$) ve kanal sayısını artırır ($1 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32$). Sonunda Flatten ile tek tahmine iner. Ders 3'ün 'ReLU toplamı'nın mekânsal hâli.

💡 Builder Notu — conv + ReLU = 'ReLU Toplamı'nın Mekânsal Hâli

- **Geriye (Ders 3/5):** conv + ReLU yığını = Ders 3'ün "ReLU toplamı"nın mekânsal hâli; [F.cross_entropy](#) (Ders 7) sınıflandırma kaybı — Şekil 15.6'in son kutusu bu kayba bağlanı.
- **İleriye (Part 2):** Bu basit yığın, Part 2'de ResNet ve U-Net gibi derin mimarilerin iskeletidir; stride'lı conv boyut indirgemenin temel aracı.

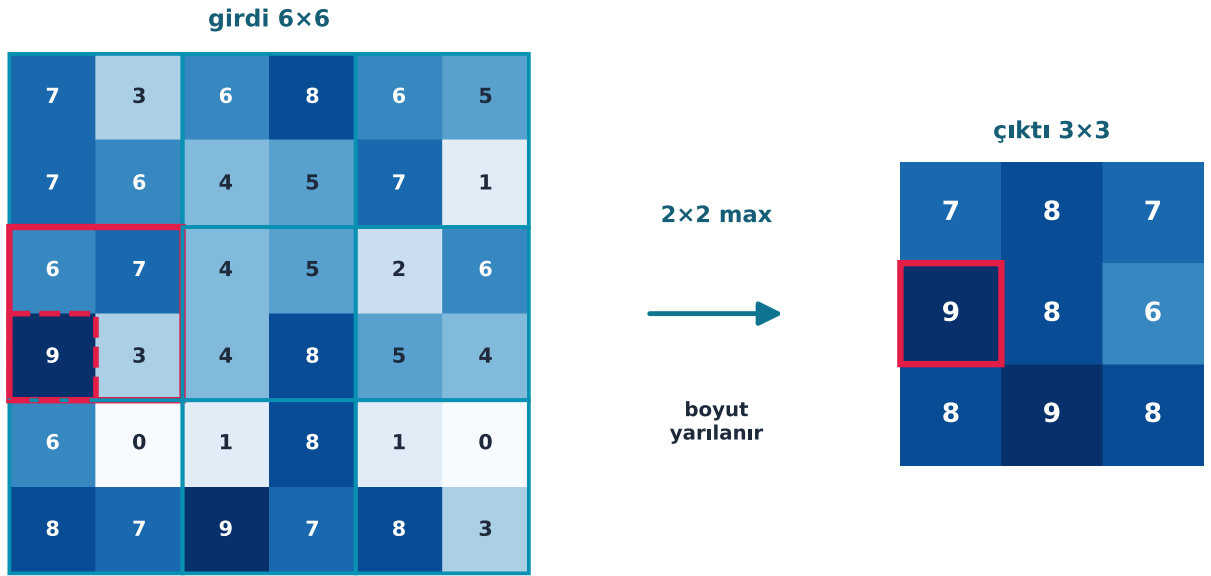
15.15 Pooling

Howard **max pooling**'e değinir: convolution gibi kaydırılan bir pencere, ama bu kez her penceredeki **en büyük** değeri alır. Bu, görüntüyü küçültür (çözünürlük düşer) ve özeti çıkarır. Modern ağlar genelde stride'lı conv veya pooling ile boyutu giderek azaltır.

"Max Pooling is kind of like a convolution — a sliding window — but this time [it takes the max]. With 2 by 2 max pooling we end up losing [resolution]." — Howard, 58:00

Şekil 15.7 bu işlemi gerçek hesaplamayla gösterir: 2x2 kaydırılan pencere her bölgenin en büyük değerini alır $\rightarrow 6 \times 6$ girdi 3×3 'e iner. Çözünürlüğü düşürür, hesabı azaltır ve küçük kaymalara dayanıklılık (translation invariance) katar — [simple_cnn](#)'in stride'lı conv'una alternatif bir boyut indirgeme yolu.

Max pooling (gerçek hesaplama) — 2×2 pencere her bloğun EN BÜYÜK değerini alır



Kaydırılan pencerede EN BÜYÜK değer seçilir (kesik kırmızı: vurgulanan bloğun max'ı); çözünürlüğü düşürür, özeti çıkarır.

Şekil 15.7: Max pooling (gerçek hesaplama): 2×2 kaydırılan pencere her bölgenin EN BÜYÜK değerini alır → 6×6 görüntü 3×3'e iner. Çözünürlüğü düşürür, hesabı azaltır, küçük kaymalara dayanıklılık (translation invariance) katar.

💡 Builder Notu — Max = En Güçlü Tepki, Ortalama = Beklenen Değer

- **Geriye (Stat 110):** Max pooling, bir bölgenin “en güçlü tepkisini” alır; ortalama pooling ise beklenen değeri. Şekil 15.7’in $6 \times 6 \rightarrow 3 \times 3$ indirgemesi, boyut indirgeme + öteleme dayanıklılığı sağlar.
- **İleriye (simple_cnn):** Modern ağılar boyutu ya max pooling ya stride’lı conv ile azaltır; ikisi de “kaydırılan pencere” fikrinin farklı kullanımı.

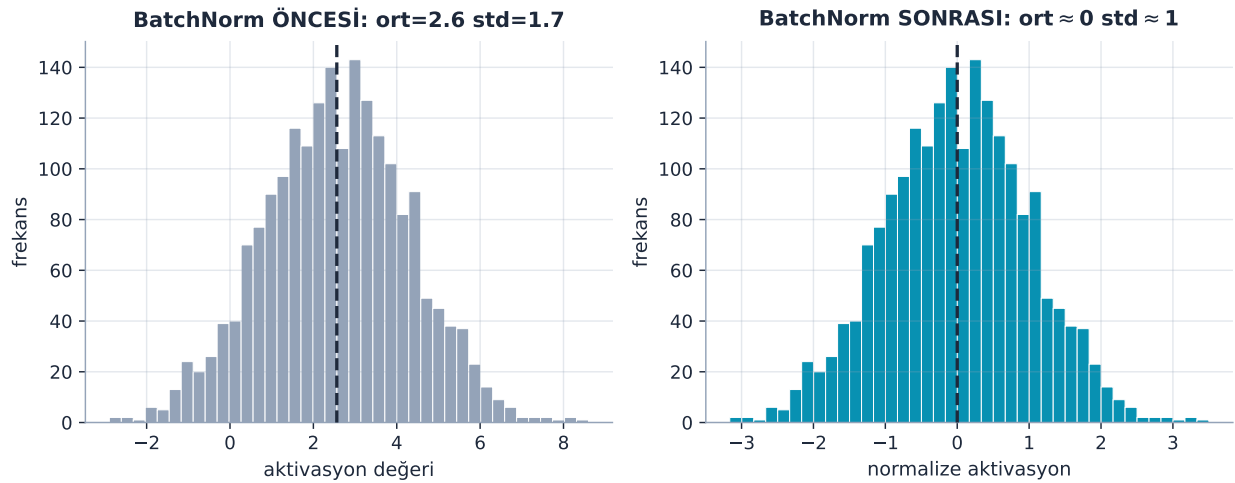
15.16 BatchNorm

Howard daha derin ve hızlı eğitim için conv katmanına **BatchNorm** (toplu normalizasyon) ekler: aktivasyonları batch üzerinden normalize ederek eğitimi kararlı ve hızlı kılar.

```
def conv(ni, nf, ks=3, act=True):
    layers = [nn.Conv2d(ni, nf, stride=2, kernel_size=ks, padding=ks//2)]
    if act: layers.append(nn.ReLU())
    layers.append(nn.BatchNorm2d(nf)) # aktivasyonlari normalize et
    return nn.Sequential(*layers)
```

Şekil 15.8 bu normalizasyonu gerçek hesaplamayla gösterir: kaymış/geniş bir aktivasyon dağılımı (sol) $\text{ort} \approx 0$ $\text{std} \approx 1$ ’e (sağ) çekilir. Aktivasyon istatistiğini koruyarak eğitimi kararlı ve hızlı kılar; daha derin ağıları eğitilebilir yapar.

BatchNorm: aktivasyonları $\text{ort}=0$ $\text{std}=1$ ’e normalize eder



aktivasyon istatistiğini koruyup eğitimi kararlı/hızlı kılar; Karpathy makemore'da sıfırdan

Şekil 15.8: BatchNorm (gerçek hesaplama): aktivasyonları batch üzerinden normalize eder — kaymış/geniş dağılım (sol, $\text{ort}=2.6$ $\text{std}=1.7$) $\text{ort} \approx 0$ $\text{std} \approx 1$ ’e (sağ) çekilir. Aktivasyon istatistiğini koruyarak eğitimi kararlı ve hızlı kılar; daha derin ağıları eğitilebilir yapar.

💡 Builder Notu — BatchNorm: Karpathy makemore'da Sıfırdan

- **Geriye (§4.F / Karpathy):** BatchNorm, Karpathy makemore Ders 4'te sıfırdan kurulur (§4.F); aktivasyon istatistiğini koruyup derin ağları eğitilebilir kılar — Şekil 15.8'un $\text{ort} \approx 0$ $\text{std} \approx 1$ çıktısı tam bu normalizasyon.
- **İleriye (Part 2 Ders 17):** Başlatma + normalizasyon, Part 2'de derinlemesine işlenecek; “neden bu kadar işe yarıyor?” sorusu hâlâ aktif bir araştırma konusu.

15.17 Serinin Kapanışı: Part 1 → Part 2

Howard Part 1'i kapatır. Ana mesajı **tenacity** (azim): çoğu insan erken vazgeçer; bir şeyi gerçekten “güzelce bitirmek” seni küçük bir azınlığa sokar. Howard kendi araçlarını (nbdev) bunu kolaylaştırmak için kurduğunu söyler.

“If you just keep going until something's actually finished, that's going to put you in a small minority. Most people don't do that.” — Howard, 1:35:13

💡 Builder Notu — Top-Down'dan 'From the Foundations'a

- **Geriye (Ders 1 / Part 1):** Part 1 Ders 1'den beri “önce çalıştır, sonra derinleş” (top-down) izledi — çalışan modelden başlayıp parçalarını açtı; bu ders o yaklaşımın doruğu ve kapanışı.
- **İleriye (Part 2):** Part 1 pratik DL'i (top-down) verdi; Part 2 her şeyi sıfırdan kurup Stable Diffusion'a varacak — Karpathy ruhuna en yakın bölüm. Bu dersin [convolution'u](#) ve [BatchNorm'u](#) orada sıfırdan yeniden kurulacak.

15.18 Kapanış

Part 1 burada bitti. Ders 8 embedding'i (collab, NLP, tablo) ve convolution'u (CNN'lerin temeli) birleştirdi. Sekiz derste sıfırdan çalışan modeller kurduk, deploy ettik, hem ağaçları hem sinir ağlarını öğrendik. Howard'ın kapanış sözü bir davet:

“I hope you'll join us in Part 2. Thanks very much, everybody. I really enjoyed this process.” — Howard, 1:36:04

💡 Builder Notu — Part 2 / Karpathy: Builder'ın Derinlik Katmanı

- **Geriye (tüm Part 1 zinciri):** Sekiz ders boyunca Ders 1'in çalışan modelinden bu dersin [convolution](#) ve [embedding'in her yerdeliğine](#) kadar kurduğumuz her parça, Şekil 34.1'in tek zincirinde toplanır — Part 2 bu zincirin üzerine inşa edilir.
- **İleriye (Part 2 / Karpathy):** Part 2 “from the foundations” omurgasıyla matmul'dan diffusion'a her şeyi sıfırdan kurar; Karpathy serisiyle birlikte builder'ın derinlik katmanı — Şekil 34.1'in [embedding](#) → [convolution](#) → CNN zinciri, Part 2'de tek tek parçalarına ayrılıp yeniden kurulacak.

15.19 Bu Dersin Özeti

1. **Embedding**, PyTorch'ta sıfırdan kurulabilir: küçük rastgele sayılarla başlatılan, eğitilebilir bir parametre tablosu ([Embedding'i Sıfırdan Kurma](#)).
2. Öğrenilen embedding'ler **yorumlanabilir**; faktörler ve aralarındaki **mesafe** benzerliği (örn. benzer filmler) ortaya çıkarır ([Embedding Yorumlama](#), [Embedding Mesafesi](#)).
3. Embedding fikri **her yerde**: collaborative filtering, NLP (kelime embedding) ve tablo verisi (entity embeddings) ([Embedding Her Yerde](#), [Tablo İçin Embedding](#)).
4. **Convolution**, küçük bir kerneli (filtre) görüntü üzerinde kaydırıp her konumda dot product alır — “kaydırılan mini dot product’lar” ([Convolution: Kenar Bulma](#), [Filtreyi Kaydırma](#)).
5. Aynı kernel tüm görüntüde **paylaşılır**: 784 yerine 9 parametre; az parametre + öteleme-değişmezlik ([Parametre Paylaşımı](#)).
6. **CNN** = conv + ReLU (+ pooling/BatchNorm) yığını; her katman boyutu küçültür, kanalı artırır ([simple_cnn](#)).
7. **Pooling** (örn. max) görüntüyü küçültüp özeti çıkarır; **BatchNorm** eğitimi kararlı ve hızlı kılar ([Pooling](#), [BatchNorm](#)).
8. **Part 1**, top-down yaklaşımla pratik derin öğrenmeyi verdi; **Part 2** her şeyi sıfırdan kuracak ([Serinin Kapanışı](#)).

! Tek Bir Cümle

Convolution, az parametreyle yerel desenleri yakalayan, görüntü üzerinde kaydırılan bir dot product’tır; embedding ise collaborative filtering’den NLP’ye ve tablo verisine kadar her yerde kategoriye öğrenilen bir vektöre çeviren arama tablosudur.

15.20 Kontrol Soruları

i Soru 1: Convolution nedir ve neden tam bağlı bir katmandan çok daha az parametre kullanır?

Cevap:

Convolution, küçük bir kerneli (örn. 3×3 matris) görüntünün her konumuna **kaydırıp** her konumda kernel ile o yamanın dot product’ını alır; sonuç bir öznitelik haritasıdır (kenar, köşe gibi yerel desenleri yakalar). Az parametre kullanır çünkü **aynı kernel tüm görüntüde paylaşılır**: 28×28 bir görüntü için tam bağlı katman 784 ağırlık isterken, 3×3 convolution yalnızca 9 ağırlık kullanır. Bu paylaşım hem parametreyi düşürür hem de “bir kenar görüntünün her yerinde kenardır” (öteleme-değişmezlik) bilgisini gömer — görüntü için doğru inductive bias. (Şekil 15.3 kenar bulmayı, Şekil 15.5 $784 \rightarrow 9$ karşıtlığını gösterir.)

i Soru 2: ‘Embedding her yeredir’ ne demek? Üç farklı kullanım alanı ver.

Cevap:

Embedding, bir kategoriye (id’yi) öğrenilen bir vektöre çeviren arama tablosudur ve aynı fikir üç alanda kullanılır: (1) **Collaborative filtering** — kullanıcı ve ürün id’leri için latent faktör vektörleri (Ders 7); (2) **NLP** — her kelime/token için kelime embedding’i (Ders 4, Karpathy); (3) **Tablo verisi** — kategorik

sütunlar (mağaza, ürün) için entity embeddings, dummy variable yerine öğrenilen vektörler (Rossmann örneği). Üçünde de mekanizma aynıdır: kategoriye vektöre çeviren, gradient descent'le öğrenilen bir tablo. (Şekil 15.2 üç kolu tek mekanizmada gösterir.)

i Soru 3: Pooling ve BatchNorm bir CNN'de ne işe yarar?

Cevap:

Pooling (örn. max pooling), convolution gibi kaydırılan bir penceredir ama her pencerede en büyük (max) değeri alır; bu, görüntüyü küçültür (çözünürlük düşer), hesabı azaltır ve küçük kaymalara dayanıklılık (translation invariance) katar. Modern ağlar bunu stride'lı conv ile de yapar. **BatchNorm** (toplu normalizasyon) ise her katmanın aktivasyonlarını batch üzerinden normalize eder; bu, aktivasyon istatistiğini koruyarak eğitimi kararlı ve çok daha hızlı kılar, daha derin ağları eğitilebilir yapar. (Şekil 15.7 pooling'i, Şekil 15.8 normalizasyonu gösterir.)

i Soru 4: Embedding mesafesi neden kullanışlıdır? Üretimde nerede karşına çıkar? (builder bağlantısı)

Cevap:

Embedding'ler bir vektör uzayında nokta olduğundan, iki öge arasındaki **mesafe** (Öklid benzeri) onların benzerliğini ölçer — yakın embedding'ler benzer öğeler demektir (Howard'ın örneğinde “Dial M for Murder”a en yakın filmler mantıklı çıktı). Builder açısından bu, modern **semantic search** ve **öneri sistemlerinin** (“benzer ürünler/filmler”) çekirdeğidir; içerikleri embedding'e çevirip en yakın komşuları aramak, vektör veritabanlarının (Qdrant, FAISS) ve RAG sistemlerinin temel işlemidir. (Bkz. [Embedding Mesafesi](#).)

15.21 Egzersizler

Egzersiz 1 (Direkt uygulama). `top_edge` ve `left_edge` kernellerini kur, `apply_kernel` ile bir görüntüye uygula ve ürettikleri öznetelik haritalarını karşılaştır.

Egzersiz 2 (İki-aşamalı). `simple_cnn`'i (conv + ReLU yığını) kur ve MNIST benzeri bir veride `F.cross_entropy` ile eğit; her katmanın çıktı boyutunu (`simple_cnn(xb).shape`) izle.

Egzersiz 3 (Edge case). conv'a BatchNorm2d ekle/çıkar ve eğitim hızı/kararlılığı farkını gözlemle.

Egzersiz 4 (Python ile doğrulama). Bir tam bağlı katman (784 girdi) ile 3×3 conv'un parametre sayısını hesaplayıp karşılaştır; parametre paylaşımının etkisini açıkla.

Egzersiz 5 (Sonraki dersin habercisi — Part 2). Convolution'ın diffusion modellerinde (U-Net) nasıl kullanıldığını araştır; Part 2'de bunu sıfırdan kuracağız.

15.22 Sonraki: PART 2 İçin Hazırlık

Part 2: Deep Learning Foundations to Stable Diffusion

Part 1 bitti — pratik derin öğrenmeyi top-down öğrendin. Part 2 (Ders 9-25) yön değiştirir: önce Stable Diffusion’ı kullanır, sonra **her şeyi sıfırdan** (from the foundations) yeniden kurar — matris çarpımından backprop’a, Learner’dan diffusion’a. Karpathy ruhuna en yakın bölüm.

Ana konular (Part 2):

- Stable Diffusion ve diffusion modelleri
- Sıfırdan: matmul, backprop, Learner, init/norm
- DDPM, DDIM, U-Net, attention, latent diffusion

⚠ Part 2 Öncesi Yapılacak

- Part 1’in tüm egzersizlerini tamamla ve bir model deploy et.
- Bir şey üret ve forum’da paylaş (Howard’ın tavsiyesi: azim + bitirmek).
- Ana cümleyi tekrar oku: “Convolution = kaydırılan dot product; embedding = arama tablosu.”

15.23 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard’da
Embedding (sıfırdan)	Eğitilebilir, küçük rastgele başlatılan parametre tablosu	7:43
Embedding yorumlama	Faktörlere bakıp modelin öğrendiği eksenleri görme	12:32
Embedding mesafesi	Vektör uzayında yakınlık = benzerlik (öneri/arama)	22:13
Entity embeddings	Tablo kategorik sütunları için öğrenilen vektörler	35:01
Convolution	Kerneli kaydırarak her konumda dot product	44:33
Kernel	Küçük filtre matrisi (örn. 3×3 kenar bulucu)	44:33
Parametre paylaşımı	Aynı kernel her yerde; 784 yerine 9 parametre	44:33
F.conv2d	Optimize convolution (tüm batch + kanal)	53:00
simple_cnn	conv + ReLU yığını; boyut küçülür, kanal artar	45:00
Pooling	Kaydırmalı max/ortalama; boyut indirgeme	58:00
BatchNorm	Aktivasyonları normalize; kararlı/hızlı eğitim	57:07
stride	Kernelin adım büyüklüğü; görüntüyü küçültür	45:00

15.24 ML Builder Baęlantıları

💡 Builder Notu — 6 ML Köprüsü

Bu ders, embedding'in her yerdelięinden convolution'a ve CNN'e uzanır; köprülerin özeti:

1. **Convolution** → görüntü için doęru inductive bias (paylařım + yerellik); CNN/U-Net/diffusion'ın temeli (Part 2) ([Convolution: Kenar Bulma](#), [Parametre Paylařımı](#)).
2. **Embedding her yerde** → collab + NLP + tablo (entity embeddings); kategoriye vektöre çeviren ortak fikir ([Embedding Her Yerde](#), [Tablo İin Embedding](#)).
3. **Embedding mesafesi** → semantic search, öneri, vektör veritabanları (Qdrant), RAG ([Embedding Mesafesi](#)).
4. **BatchNorm** → Karpathy makemore'da sıfırdan (§4.F); derin aę eęitimini kararlı kılar ([Batch-Norm](#)).
5. **Parametre paylařımı** → az parametre + genelleme; modern mimarilerin verimlilik sırrı ([Parametre Paylařımı](#)).
6. **Pooling/stride** → boyut indirgeme; hesabı azaltıp soyutlama seviyesini artırma ([Pooling, simple_cnn](#)).

! Bu dersten ve Part 1'den tek bir řey alıp gideceksen

Bu dersten ve Part 1'den tek bir řey alıp gideceksen: derin öęrenmenin paraları basittir ve hepsi birbirine baęlanır. Embedding bir arama tablosu, convolution kaydırılan bir dot product, model gradient descent'le uydurulan bir fonksiyondur. Howard'ın son tavsiyesi en deęerlisi: bir řeyi gerekten bitir ve paylař — çoęu insan vazgeer; sen vazgeme. Part 2'de görüřmek üzere.

16 Ders 9 — Stable Diffusion (Part 2 Başlangıcı)

Part 2'nin ilk dersi yönü değiştirir: Part 1 pratik derin öğrenmeyi top-down verdi, Part 2 'deep learning foundations to Stable Diffusion' — her şeyi sıfırdan kurup Stable Diffusion'a varacak. Ama bu ilk ders yine top-down başlar: önce Stable Diffusion'la oynarız (prompt → görüntü, guidance scale, negative prompt, img2img, textual inversion), sonra nasıl çalıştığını sezgisel açarız. Çekirdek fikir: 'bir şeyin ne kadar geçerli bir görüntü olduğunu' söyleyen sihirli bir fonksiyonun pixel gradyanı yönünde pikselleri itirmek — gürültüden görüntü doğar. Üç bileşen: U-Net (gürültü tahmincisi), VAE (latent sıkıştırma 786K → 16K), CLIP (metin koşullama)

Bölüm bilgisi

- **Howard'ın videosu:** [course.fast.ai — Lesson 9: Stable Diffusion](https://course.fast.ai/lesson-9-stable-diffusion) (~135 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 9 (Part 2 BAŞLANGICI)
- **Hoca:** Jeremy Howard
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [diffusion-nbs](#) — [stable_diffusion](#)
- **Okuma süresi:** ~40 dk

16.1 Bu Derste Ne Var?

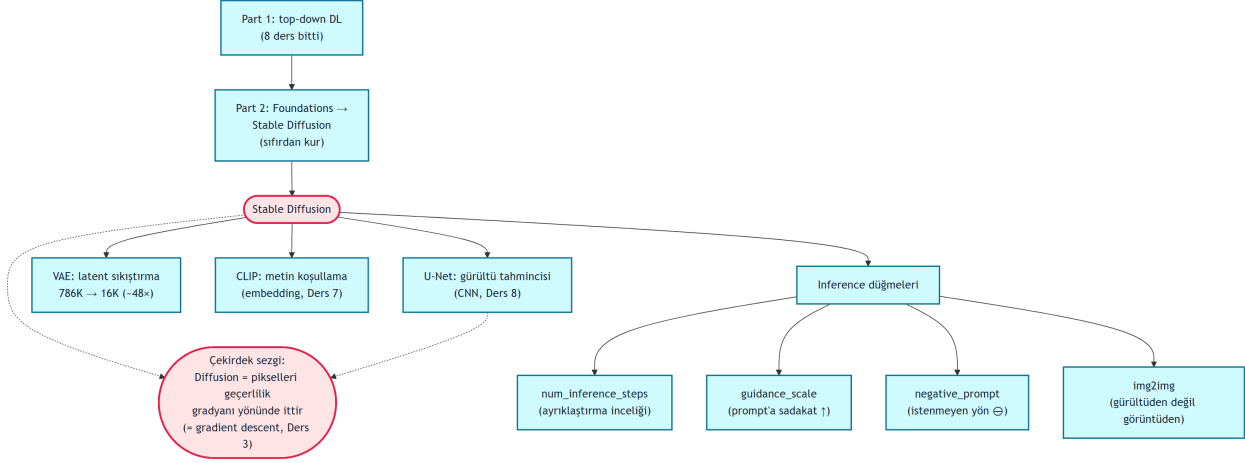
Part 2'nin ilk dersine geldik (Howard'ın numaralandırmasıyla Lesson 9, çünkü Part 1 sekiz dersti). Yön değişiyor: Part 1 pratik derin öğrenmeyi **top-down** verdi; Part 2'nin tam adı **“deep learning foundations to Stable Diffusion”** — her şeyi sıfırdan kurup Stable Diffusion'a varacak. Ama bu ilk ders yine top-down başlar: önce Stable Diffusion'la **oynarız**, sonra nasıl çalıştığını sezgisel olarak açarız.

Üç temel fikir bu dersin omurgasını kurar:

1. **Stable Diffusion'ı kullanmak** — StableDiffusionPipeline ile prompt → görüntü; guidance_scale, negative_prompt, img2img, textual inversion ve DreamBooth gibi araçlarla üretimi yönlendirme ([Oynayalım](#) → [Pipeline](#)).
2. **Diffusion'ın sezgisi** — “bir şeyin ne kadar geçerli bir görüntü olduğunu” söyleyen sihirli bir fonksiyon hayal et; pikselleri o fonksiyonun gradyanı yönünde ittir → gürültüden görüntü doğar ([Sihirli API](#) → [Pixel Gradyanı](#)).
3. **Üç bileşen** — **U-Net** (gürültü tahmincisi), **VAE/autoencoder** (latent sıkıştırma: 786K → 16K), **CLIP** (metin koşullama) ([U-Net](#) → [VAE](#) → [CLIP](#)).

“We’re certainly not going to be spending all of our time seeing how to do important things; we’ll be doing generative modely fun things, and understanding lots of details.” — Howard, 0:45

Şekil 34.1 bu üç fikri tek bir yol haritasında birleştirir: Part 1’in (top-down, 8 ders) bittiği yerden Part 2’nin “önce kullan, sonra sıfırdan kur” stratejisine, oradan Stable Diffusion’un üç bileşenine (U-Net + VAE + CLIP) ve çıkarım düğmelerine (num_inference_steps, guidance_scale, negative_prompt, img2img) uzanır — rose vurgulu “Stable Diffusion” ve “Çekirdek sezgi” düğümleri dersin doruğudur.



Şekil 16.1: Part 2 yol haritası: Stable Diffusion’ı önce kullan, sonra üç bileşeniyle (U-Net + VAE + CLIP) sıfırdan kur. Çekirdek sezgi — diffusion, pikselleri geçerlilik gradyanı yönünde iter (= Ders 3’teki gradient descent, ağırlık yerine piksel). (şematik)

💡 Builder Notu — Üç Tanıdık Parça: Diffusion’ın Gradient Descent Kökü

- **Geriye (Ders 3/8 + Karpathy):** Diffusion’ın çekirdeği gradient descent’tir (Ders 3) — ama bu kez ağırlıkları değil **pikselleri** günceller; `f.backward()` aynı autograd. U-Net bir CNN’dir (Ders 8); üretim, embedding’leri (Ders 7-8) kullanan CLIP ile koşullanır.
- **İleriye (Part 2 / NYU §4.J):** Part 2’nin geri kalanı (L11-25) U-Net, VAE ve scheduler’ı sıfırdan kurar. Diffusion, NYU’nun enerji-tabanlı modelleri (EBM) ve self-supervised learning fikriyle akrabadır — “geçerlilik skoru” bir enerji fonksiyonu sezgisidir.
- **Tek cümle:** Stable Diffusion, metinle koşullanmış bir gürültü-giderme döngüsüdür; bugün kullanırız, Part 2 boyunca sıfırdan kurarız.

16.2 Stable Diffusion’la Oynayalım

Howard dersi top-down açar: önce modeli **kullanırız**. Hugging Face `diffusers` kütüphanesiyle, bir prompt’tan görüntü üreten hazır bir pipeline yükleriz. “Hepimiz oynamak için sabırsızlanıyoruz” der; mekanik sonra gelir. Bu, Part 2’nin “önce kullan, sonra sıfırdan kur” stratejisidir — [konsept haritasının](#) sol kolunda gördüğümüz “önce kullanım” adımı.

“Because we’re all dying to play with it, right?” — Howard, 1:42

💡 Builder Notu — Önce Çalışan Sonuç: Part 1 Top-Down Ruhunun Devamı

- **Geriye (Ders 1):** Top-down ruhu [Ders 1](#) ile aynı; önce çalışan sonuç, sonra derinlik — modelin ne yaptığını görmeden iç mekaniğine inmeyiz.
- **İleriye (bu ders):** Bugün önce `pipe()`'ı çağırıp görüntü üreteceğiz, dersin ikinci yarısında ([Sihirli API](#)) o “sihrin” aslında tanıdık gradient descent olduğunu açacağız.

16.3 Kaynaklar ve Compute

Howard diffusion-nbs repo'sunu (oynamak için notebook'lar), Lexica gibi prompt galerilerini ve compute seçeneklerini (Colab, Paperspace, kendi makinen) tanıtır. Part 2, Part 1'den **daha fazla GPU** gerektirir; diffusion modelleri ağırdır.

“Part 2 requires quite a bit more compute than Part 1.” — Howard, 12:15

💡 Builder Notu — Compute Maliyeti: Latent Uzayın Varlık Sebebi

- **İleriye (production / VAE):** Diffusion compute-yoğundur; latent uzayda çalışmak (sonraki bölümlerde [VAE](#) ile) bu maliyeti ~48× düşürür. “Pahalı işi küçük uzayda yap” disiplini, Stable Diffusion'ı pratik kılan anahtardır.
- **İleriye (Ders 14):** fp16 (yarı hassasiyet) bellek tasarrufu da aynı maliyet mücadelesinin parçasıdır; Part 2 Ders 14'te derinleşir.

16.4 StableDiffusionPipeline

İlk kod: hazır bir pipeline yükle, bir prompt ver, görüntü al. `from_pretrained` modeli (CompVis/stable-diffusion-v1-4) indirir; fp16 bellek için yarı hassasiyet kullanır. Bu, dersin geri kalanında açacağımız “yüksek seviye” arayüzdür — [düşük seviye pipeline](#) bunun içini gösterir.

```
from diffusers import StableDiffusionPipeline
pipe = StableDiffusionPipeline.from_pretrained(
    "CompVis/stable-diffusion-v1-4", variant="fp16",
    torch_dtype=torch.float16).to("cuda")

prompt = "a photograph of an astronaut riding a horse"
pipe(prompt).images[0]
```

💡 Builder Notu — `from_pretrained` = Transfer Learning'in Üretken Hâli

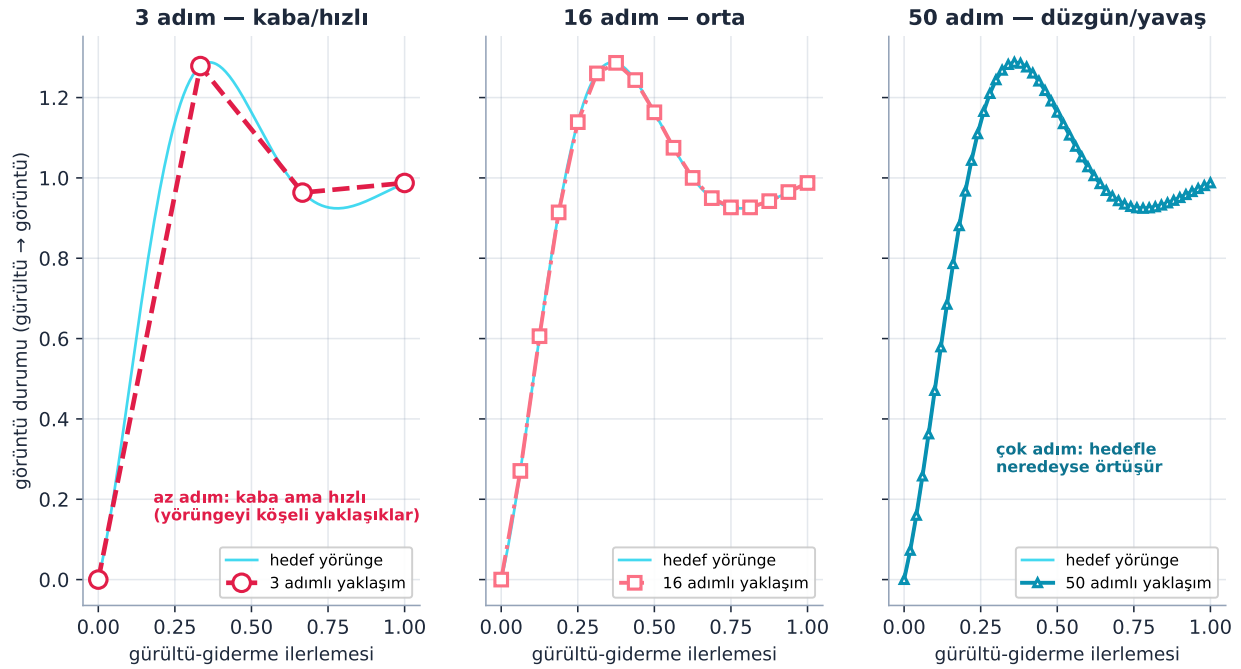
- **Geriye (Ders 2):** `from_pretrained` = transfer learning; başkasının (CompVis) devasa GPU'larla eğittiği modeli indirip kullanırız — [Ders 2'deki](#) “önceden eğitilmiş modeli al, kendi işine uydur” fikrinin üretken model hâli.
- **İleriye (Ders 14):** fp16 (mixed precision) bellek ve hız için yarı hassasiyet kullanır; Part 2 Ders 14'te neden işe yaradığı ve nasıl kurulduğu derinleşir.

16.5 num_inference_steps

Diffusion, gürültüden görüntüyü **adım adım** çıkarır. num_inference_steps kaç adımda çıkaracağını belirler: az adım hızlı ama kaba, çok adım yavaş ama kaliteli. torch.manual_seed aynı başlangıç gürültüsünü sabitler, böylece adım sayısının etkisini tek başına gözleyebiliriz.

```
torch.manual_seed(1024)
pipe(prompt, num_inference_steps=3).images[0]    # hızlı, kaba
pipe(prompt, num_inference_steps=16).images[0]   # yavaş, kaliteli
```

Şekil 16.2 bu dengeyi gerçek bir yörünge üstünde gösterir: gürültü-giderme süreci sürekli bir hedef yörüngeye yaklaşır, num_inference_steps ise bu yörüngeyi kaç parçaya böldüğümüzdür. 3 adım yörüngeyi köşeli ve kaba yaklaşıklar (hızlı), 16 adım daha düzgün, 50 adım hedefle neredeyse örtüşür (yavaş). Yani parametre, gürültü-giderme yörüngesinin **ayrıklaştırma inceliğidir** — her adım pratikte bir U-Net çağrısıdır.



Şekil 16.2: (şematik: num_inference_steps = gürültü-giderme yörüngesinin ayrıklaştırma inceliği; gerçekte her adım bir U-Net çağrısıdır)

💡 Builder Notu — Adım Sayısı = Yörünge Ayrıklaştırması (DDIM'in Hedefi)

- **Geriye (Ders 3):** Az adımla kaba, çok adımla düzgün yaklaşma, [Ders 3'teki](#) gradient descent'in learning-rate/adım sayısı dengesinin ikizidir — büyük adım hızlı ama kaba, küçük adım yavaş ama hassas.
- **İleriye (Ders 21 DDIM):** Adım sayısını azaltıp kaliteyi korumak Part 2'nin (DDIM, Ders 21) ana hedeflerindendir; Şekil 16.2'in “50 adım $\approx$ 3 adımın kalitesi” hayali tam bu araştırma yönüdür.

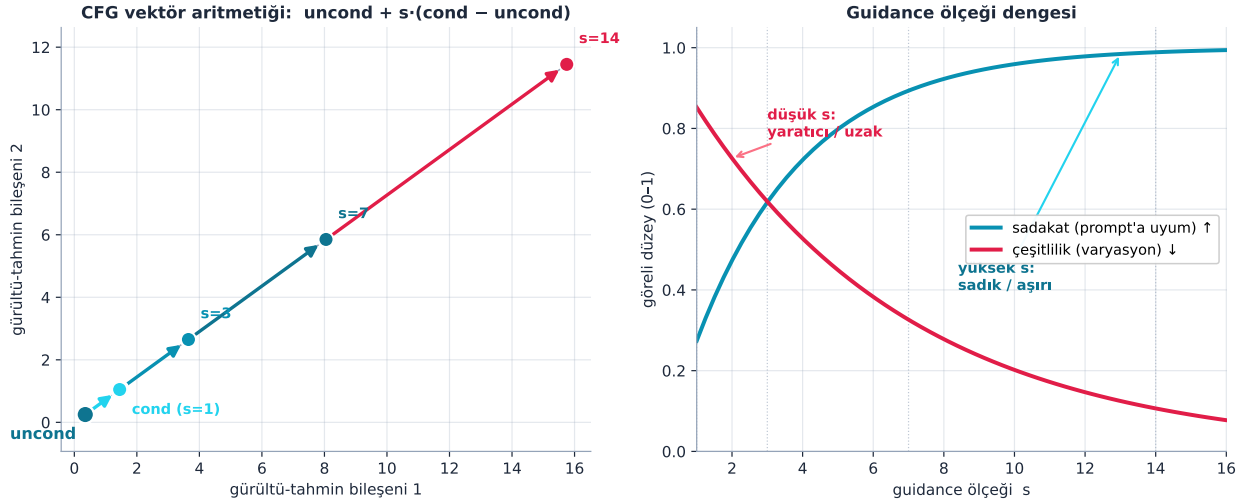
16.6 guidance_scale (Classifier-Free Guidance)

guidance_scale, modelin prompt’a ne kadar “sıkı” uyacağını ayarlar. Düşük → yaratıcı ama prompt’tan uzak; yüksek → prompt’a sadık ama bazen aşırı. Bu, **classifier-free guidance** (CFG)’in parametresidir: koşullu (prompt’lu) ve koşulsuz (boş prompt’lu) tahmini harmanlar.

```
images = concat(pipe(prompts, guidance_scale=g).images for g in [1.1, 3, 7, 14])
```

“That’s what guidance scale does — you can think of the guidance scale as how strongly should we apply the prompt.” — Howard, 40:00

Şekil 16.3 CFG’nin vektör aritmetiğini somutlaştırır. Sol panel formülü gösterir: $\text{tahmin} = \text{uncond} + s \cdot (\text{cond} - \text{uncond})$ — ölçek s büyüdükçe tahmin vektörü koşulsuz noktadan koşullu yöne, hatta ötesine taşar. Sağ panel dengeyi gösterir: yüksek s daha **sadık** (prompt’a uyum ↑) ama daha az **çeşitli** (varyasyon ↓). İkisi birlikte, neden çok yüksek guidance’in aşırı doymun/yapay görüntüler verdiğini açıklar.



Şekil 16.3: Classifier-free guidance (CFG): $\text{tahmin} = \text{uncond} + s \cdot (\text{cond} - \text{uncond})$. SOL: ölçek s büyüdükçe tahmin vektörü koşulsuzdan koşullu yöne, hatta ötesine taşar. SAĞ: yüksek s daha sadık ama daha az çeşitli. (CFG vektör aritmetiği gerçek; sadakat/çeşitlilik denge eğrileri illüstratif)



Builder Notu — CFG = Koşullu ile Koşulsuz Arası Vektör Aritmetiği

- **Geriye (Ders 7 dot product):** CFG’nin “koşullu – koşulsuz” yönü, Ders 7’deki vektör aritmetiğinin üretken hâlidir — iki tahmin arasındaki farkı bir yön olarak alıp ölçekleriz.
- **İleriye (Ders 10/9A):** Classifier-free guidance’ın tam matematiği Ders 9A (Whitaker) ve Ders 10’da açılır (§4.I); Şekil 16.3’in sadakat/çeşitlilik eğrileri o derslerde formel hâle gelir.

16.7 negative_prompt

negative_prompt ile modele “şunu İSTEME” deriz: çıktıyı belirtilen kavramdan **uzaklaştırır**. Howard “blue” negatif prompt’uyla mavi tonları çıkarır. CFG açısından bu, **guidance** yönünü tersine çevirmektir — istenen

yöne itmek yerine istenmeyen yönden çekmek.

```
torch.manual_seed(1000)
pipe(prompt, negative_prompt="blue").images[0] # mavi tonlardan uzaklas
```

💡 Builder Notu — Negatif Prompt = Guidance Yönünün Tersisi

- **Geriye (guidance):** Negatif prompt, [classifier-free guidance](#)'ın yönünü tersine çeviren bir kavramdır — “cond – uncond” yerine “uncond – unwanted” yönüne iter; aynı vektör aritmetiği, ters işaret.
- **İleriye (9A):** Ders 9A'da Whitaker bunu derinleştirir: negatif prompt'u ve embedding interpolasyonunu latent uzayda elle kontrol eder.

16.8 img2img: Görüntüden Görüntü

Sıfır gürültü yerine bir **başlangıç görüntüsünden** başlayabiliriz: model onu kısmen gürültüleyip prompt'a göre yeniden çizer. strength ne kadar değiştireceğini belirler (0 = aynı, 1 = tamamen yeniden). Bu, [num_inference_steps](#) yörüngesine baştan değil ortadan girmek gibidir — gürültü-giderme döngüsünü saf gürültüyle değil, kısmen gürültülenmiş bir görüntüyle başlatırız.

```
from diffusers import StableDiffusionImg2ImgPipeline
images = pipe(prompt="Wolf howling at the moon, photorealistic 4K",
               image=init_image, strength=0.8, num_inference_steps=50).images
```

💡 Builder Notu — img2img: Stil Aktarımı ve Düzenlemenin Temeli

- **İleriye (production):** img2img, stil aktarımı ve düzenlemenin (inpainting, DiffEdit) temelidir; başlangıç görüntüsünü kısmen gürültüleyip yeniden çizme fikri Ders 10'da derinleşir.
- **Geriye (num_inference_steps):** strength, gürültü-giderme yörüngesine hangi noktadan gireceğimizi belirler — düşük strength orijinale yakın kalır, yüksek strength neredeyse sıfırdan üretir.

16.9 Textual Inversion ve DreamBooth

İki güçlü özelleştirme yöntemi: **Textual inversion** tek bir yeni embedding eğitir (yeni bir kavrama/stile bir “kelime” öğretir); **DreamBooth** modeli birkaç fotoğrafla ince ayarlar (örn. belirli bir kişiyi/nesneyi). İkisi de modele yeni bir şey öğretir — biri tek bir embedding satırını, diğeri model ağırlıklarını günceller.

“Textual Inversion is where we actually fine-tune just a single embedding... we add that token to the text model and then train the embedding.” — Howard, 40:00

💡 Builder Notu — Textual Inversion = Tek Bir Embedding Satırını Eğitmek

- **Geriye (Ders 7-8):** Textual inversion = tek bir embedding’i (arama tablosu satırını) eğitmek; [embedding’in arama tablosu](#) olduğu bilgisi (Ders 7-8) doğrudan işler — yeni “kelime”, [CLIP](#) sözlüğüne eklenen yeni bir satırdır.
- **İleriye (production):** DreamBooth modelin tamamını birkaç görselle ince ayarlar; ikisi de transfer learning’in üretken modellere taşınmış hâlidir.

16.10 Stable Diffusion Nasıl Çalışır: Sihirli API

Şimdi sezgiye geçiyoruz. Howard “el yazısı rakam üreten” basit bir örneklerle açıklar. Bir **sihirli API** hayal et: ona bir görüntü verirsin, sana “bunun el yazısı bir rakam olma olasılığı”nı döner (örn. 0.98 veya 0.02). Bu fonksiyona sahipsek, onu görüntü **üretmek** için kullanabiliriz. Bu, dersin ikinci yarısının çekirdek kurgusudur — sonraki bölümde ([Pixel Gradyanı](#)) bu fonksiyonun gradyanını alıp üretime çevireceğiz.

“You pass an image into this magic API, and it returns the probability that this thing is a handwritten digit.” — Howard, 47:00

💡 Builder Notu — Görüntü = Sayı + Bir Skor Fonksiyonu

- **Geriye (Ders 1):** “Görüntü = sayı” (Ders 1) artı bir skor fonksiyonu — tanıdık parçalar; sihirli API yalnızca “bu görüntü ne kadar geçerli?” sorusuna bir sayı döndüren bir fonksiyondur.
- **İleriye (Pixel Gradyanı):** Bu fonksiyona sahipsek, onun gradyanını alıp pikselleri “daha geçerli” yöne itebiliriz — sıradaki bölümün konusu.

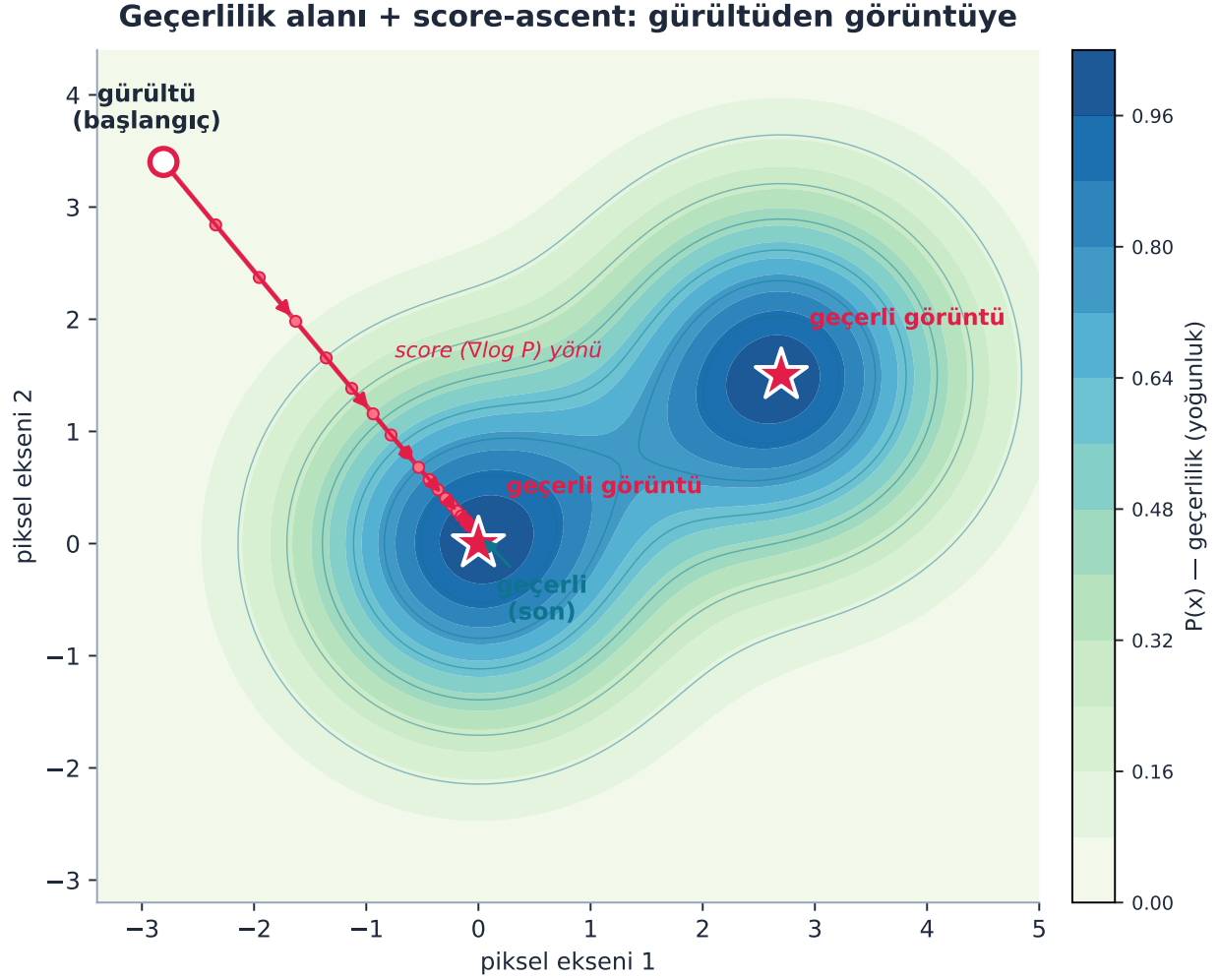
16.11 Pixel Gradyanı → Gürültü Giderme

Anahtar fikir: bu sihirli fonksiyonun, **her pikselle ilgili gradyanını** alırız — “bu pikseli biraz değiştirsem, rakam olma olasılığı ne kadar artar?”. Sonra her pikseli bu gradyan yönünde azıcık ittirirsek, görüntü biraz daha “rakam gibi” olur. Bunu tekrarlırsak, gürültüden bir rakam doğar. İşte diffusion budur — ama ağırlıkları değil **pikselleri** günceller.

```
# Kavram (gerçek SD bunu U-Net ile yapar):
# pikselleri "daha gecerli goruntu" yonunde ittir
# x = x - c * gradient(P(x gecerli), x) # her adimda biraz gurultu gider
```

“We’ve calculated the gradient of the probability that X3 is a handwritten digit with respect to the pixels of X3. We take every pixel and subtract a little bit times its gradient.” — Howard, 49:41

Şekil 16.4 bu çekirdeği gerçek hesaplamayla gösterir: arka plandaki **geçerlilik alanı** $P(x)$ (yüksek = geçerli görüntü bölgesi), iki rose yıldız iki geçerli görüntü (mod), ve gürültülü bir başlangıçtan yıldızlardan birine inen **score-ascent yörüngesi**. Her küçük ok, score $\nabla \log P$ yönüdür; ardışık adımlarla en yakın geçerli moda yaklaşıyoruz. Diffusion’ın özü tam budur — ağırlık değil, pikselin kendisi güncellenir.



Şekil 16.4: Geçerlilik alanı $P(x)$ ve score-ascent yörüngesi: gürültülü başlangıçtan, $\nabla \log P$ (score) yönünde ardışık adımlarla en yakın geçerli moda iniş. Diffusion’ın çekirdeği — ağırlık değil, pikselin kendisi güncellenir. (gerçek hesaplama: Gaussian-karışım score’u $\nabla \log P$ ile gradient ascent — diffusion’ın çekirdeği)

Matematiksel olarak bu yön bir **score**'dur: olasılık yoğunluğunun logaritmasının gradyanı, $\nabla_x \log p(x)$. Score yüksek olasılıklı bölgelere doğru işaret eder; pikselleri o yönde adım adım iterek gürültüden geçerli bir görüntüye varırız.

💡 Builder Notu — Bu, Loss Yerine Skor Üstünde Gradient Descent

- **Geriye (Ders 3/5):** Bu, gradient descent'in ta kendisidir ([Ders 3](#)) — ama loss yerine “geçerlilik skoru”, ağırlık yerine piksel; `f.backward()` ([Ders 5](#)) aynı autograd mekanizmasıdır.
- **İleriye (Part 2):** Bu “sihirli fonksiyon”u sıfırdan kuracağız (L15-L25); o, gürültü tahmin eden bir sinir ağı ([U-Net](#))dir. Score $\nabla \log p$ fikri, NYU'nun enerji-tabanlı modelleriyle ([sentez figürü](#)) akrabadır.

16.12 U-Net: Gürültü Tahmincisi

Pratikte “sihirli fonksiyon” bir sinir ağıdır ve adı **U-Net**'tir — Stable Diffusion'ın ilk bileşeni. Girdisi gürültülü bir görüntü; çıktısı o görüntüdeki **gürültünün tahmini** (veya gürültüsüz hâli). Gürültüyü çıkarıp tekrarlayarak görüntü oluşur. [Pixel gradyanı](#) bölümündeki “score” işte bu ağı çıktısıdır.

“This is the first component of Stable Diffusion, it's the U-Net. The input to the U-Net is a somewhat noisy image, and from the input we end up with the un-noisy image.” — Howard, 1:00:00

💡 Builder Notu — U-Net = CNN'in Encoder-Decoder Hâli

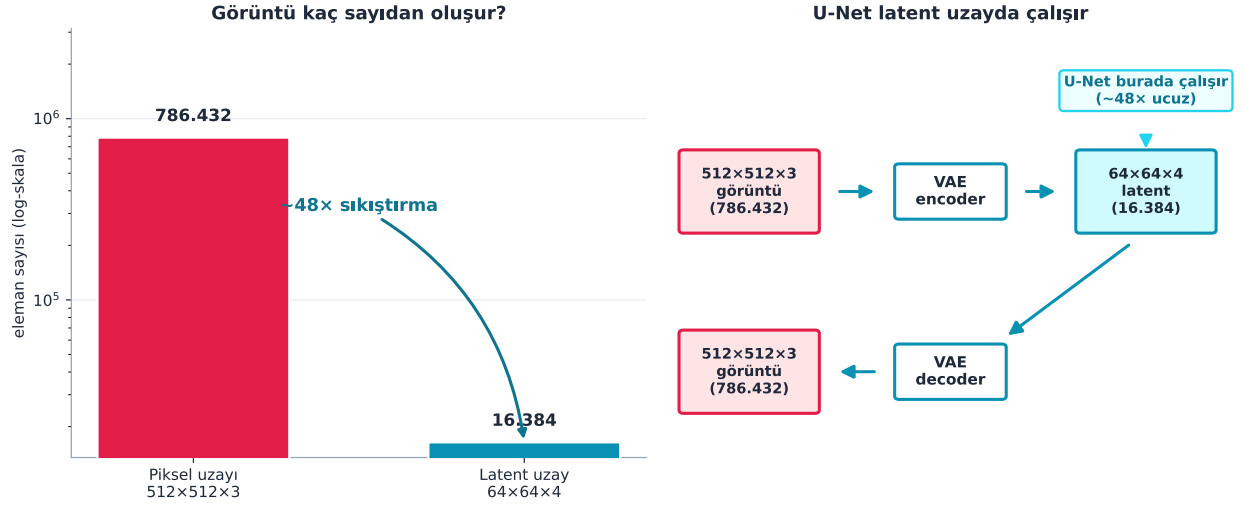
- **Geriye (Ders 8):** U-Net bir CNN mimarisidir ([Ders 8 convolution](#)); encoder-decoder + skip bağlantıları. Conv katmanlarıyla görüntüyü küçültüp büyütür — [simple_cnn](#) yığınının simetrik, üretken hâli.
- **İleriye (Ders 19/23):** U-Net, Part 2 Ders 19 ve 23'te sıfırdan kurulur (§4.I); attention ile koşullama orada eklenir.

16.13 Autoencoder / VAE: Latent Sıkıştırma

Sorun: $512 \times 512 \times 3$ bir görüntü = 786.432 sayı; U-Net'i bu kadar pikselle eğitmek çok pahalı. Çözüm bir **autoencoder** (VAE): görüntüyü $64 \times 64 \times 4 = 16.384$ sayıya **sıkıştırır** (encoder), sonra geri açar (decoder). U-Net'i bu küçük **latent** uzayda çalıştırırız — $\sim 48\times$ daha az veri. Stable Diffusion'ı *latent* diffusion yapan şey budur.

“We've compressed it from 786,432 to 16,384. We've created a very powerful compression algorithm. Why on earth would we train our U-Net with 786,432 pixels?” — Howard, 1:10:00

Şekil 16.5 bu sıkıştırmayı gerçek sayılarla gösterir: sol panel piksel uzayı (786.432) ile latent uzayı (16.384) log-skalada karşılaştırır — $\sim 48\times$ fark; sağ panel VAE encode $\rightarrow$ latent diffusion $\rightarrow$ decode akışını şematize eder. U-Net pahalı işi pikselde değil, bu $\sim 48\times$ küçük latent uzayda yapar.



Şekil 16.5: VAE latent sıkıştırma: U-Net pikselleri değil, ~48× küçük latent uzayı işler (gerçek sayılar: Howard 786.432→16.384, ~48× — ‘latent diffusion’).

💡 Builder Notu — Latent Uzayda Çalış: Pahalı İş Küçük Temsilde Yap

- **Geriye (6.S191 / Ders 8):** VAE, 6.S191 Ders 4’te kavramsaldır; autoencoder [Ders 8’de](#) geçti. “Latent uzayda çalışmak” = pahalı işlemi düşük boyutlu, anlamlı bir temsilde yapmak — aynı verimlilik mantığı embedding ve quantization’da da geçerli.
- **İleriye (Ders 25/29):** VAE, Part 2 Ders 25/29’da sıfırdan kurulur (§4.I “latent diffusion”); encoder/decoder ve reconstruction loss orada açılır.

16.14 CLIP: Metin Koşullama

Üçüncü bileşen: **CLIP**. Prompt’u (metni) U-Net’in anlayacağı bir embedding’e çevirir. CLIP, metin ve görüntüyü **ortak bir uzaya** gömecek şekilde eğitilmiştir; böylece “astronaut riding a horse” metni, üretimi o yöne koşullar. **U-Net** her gürültü-giderme adımında bu embedding’i girdi olarak alır.

```
from transformers import CLIPTokenizer, CLIPTextModel
tokenizer = CLIPTokenizer.from_pretrained("openai/clip-vit-large-patch14")
text_encoder = CLIPTextModel.from_pretrained("openai/clip-vit-large-patch14").to("cuda")
```

💡 Builder Notu — CLIP = Embedding’in Çok-Modal Hâli

- **Geriye (Ders 4/7):** CLIP metni tokenize edip ([Ders 4](#)) embedding’e çevirir ([Ders 7](#)); metin-görüntü ortak uzayı, [Ders 8’deki](#) embedding mesafesinin çok-modal hâlidir — “yakın embedding = benzer anlam” burada metin ile görüntü arasında geçerlidir.
- **İleriye (Part 2):** CLIP’in attention ile U-Net’i nasıl koşullandırdığı (cross-attention) Part 2’de açılır; metin embedding’i, gürültü-giderme döngüsünün her adımına yön verir.

16.15 Latent + Scheduler + Noise Schedule

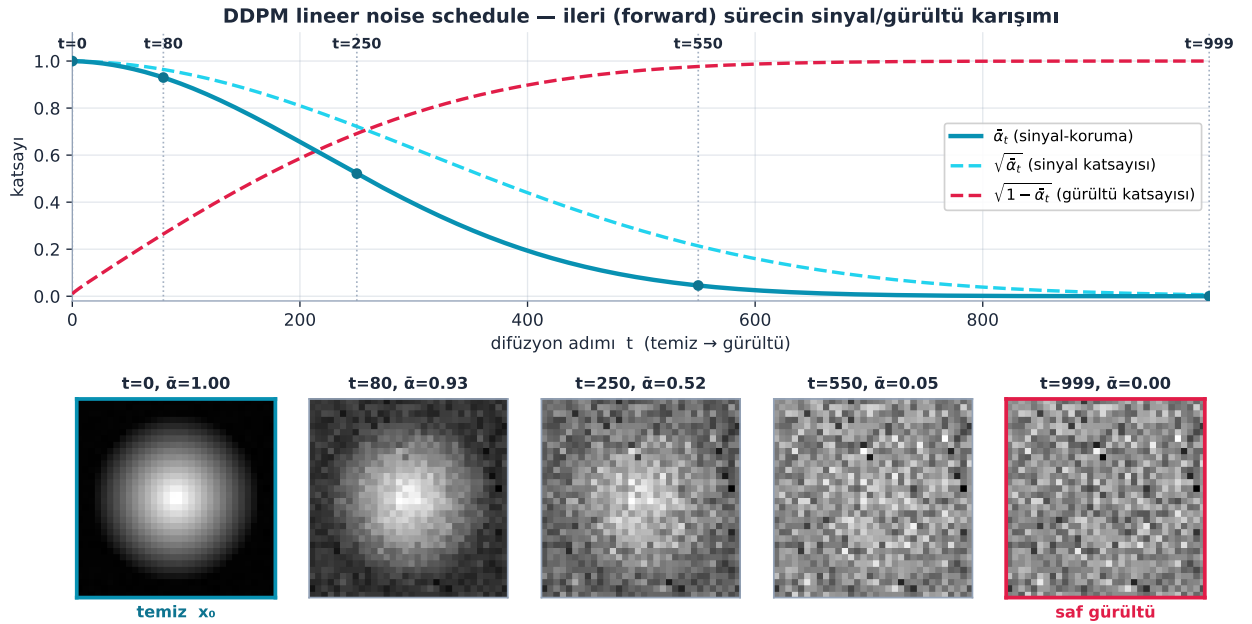
Üretim, latent uzayda **rastgele gürültüyle** başlar (64×64×4). **Scheduler** her adımda ne kadar gürültü kaldırılacağını yöneten **noise schedule**'ı (β , $\bar{\alpha}$ dizileri) belirler. U-Net her adımda gürültüyü tahmin eder, scheduler onu kısmen çıkarır; bu döngü görüntüyü oluşturur.

```
torch.manual_seed(100)
latents = torch.randn((batch_size, unet.in_channels, height//8, width//8)).half().to("cuda")
scheduler.set_timesteps(num_inference_steps) # noise schedule adimlari
```

Üretim bu noise schedule'in **tersine** gider. İleri (forward) süreç temiz bir görüntüye adım adım gürültü ekler:

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \varepsilon$$

burada x_0 temiz görüntü, ε saf gürültü, $\bar{\alpha}_t$ ise t adımında ne kadar sinyal korunduğunu belirler. Şekil 16.6 bu ileri süreci gösterir: üst panel DDPM linear noise schedule'ında $\bar{\alpha}_t$ ve sinyal/gürültü katsayılarını ($\sqrt{\bar{\alpha}_t}$, $\sqrt{1 - \bar{\alpha}_t}$), alt satır da temiz x_0 'dan saf gürültüye giden gerçek x_t karelerini çizer. Stable Diffusion üretimi bu okun **tersidir** — saf gürültüden başlayıp $\bar{\alpha}_t$ 'yi geri tırmanır.



Şekil 16.6: (gerçek DDPM forward: $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \varepsilon$; SD üretimi bunun tersidir)



Builder Notu — Noise Schedule: DDPM/DDIM/Karras'ın Çekirdeği

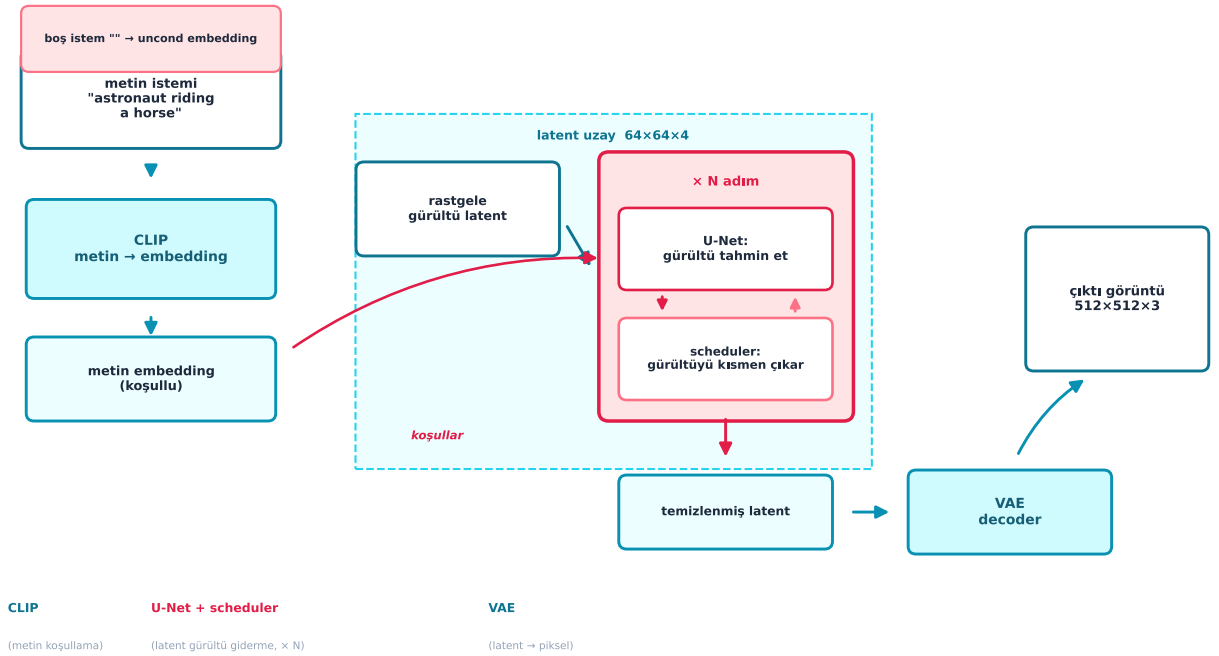
- **İleriye (Ders 19-22):** Noise schedule, DDPM (Ders 19) ve DDIM (Ders 21) ile Karras (Ders 22) Part 2'de sıfırdan kurulur (§4.I); Şekil 16.6'un linear schedule'ı orada gelişmiş schedule'larla karşılaştırılır.
- **Geriye (num_inference_steps):** scheduler.set_timesteps tam olarak num_inference_steps bölümündeki ayırıklaştırmayı kurar — kaç adımda gürültü kaldırılacağı bu noise schedule'dan okunur.

16.16 Düşük Seviye Pipeline

Howard hazır `pipe()`'in içini açar — beş adım: (1) **tokenizer + CLIP** prompt'u embedding'e çevirir; (2) classifier-free guidance için boş prompt embedding'i de hazırlanır; (3) latent rastgele gürültüyle başlar; (4) döngüde U-Net gürültüyü tahmin eder, scheduler çıkarır; (5) VAE decoder latent'i görüntüye açar.

```
# Classifier-free guidance: kosullu + kosulsuz embedding
text_emb = text_encoder(text_input.input_ids.to("cuda"))[0].half()
uncond_emb = text_encoder(uncond_input.input_ids.to("cuda"))[0].half()
emb = torch.cat([uncond_emb, text_emb])
# latent gürültü -> dongude U-Net + scheduler -> VAE decode
latents = torch.randn((len(prompts), unet.in_channels, height//8, width//8)).half().to("cuda")
latents = latents * scheduler.init_noise_sigma
```

Şekil 16.7 bu beş adımı tek bir çıkarım hattında birleştirir — bu derslerin **flagship** diyagramıdır. Solda **CLIP** metni embedding'e çevirir (ve CFG için boş prompt'tan uncond embedding hazırlanır); merkezde latent uzayda rastgele gürültü, rose vurgulu “× N adım” döngüsünde **U-Net** + **scheduler** ile temizlenir (CLIP embedding'i döngüyü koşullar); sağda **VAE** decoder temizlenmiş latent'i 512×512×3 görüntüye açar. Üç bileşen tek bir akışta: CLIP yön verir, U-Net+scheduler latent'te gürültü giderir, VAE açar.



Şekil 16.7: (şematik: Stable Diffusion çıkarım hattı — CLIP koşullar, U-Net+scheduler latent'te gürültü giderir, VAE açar)

💡 Builder Notu — Beş Adım = Part 2'nin Yol Haritası

- **İleriye (Part 2):** Bu beş adım — **CLIP**, latent, **U-Net** döngüsü, **scheduler**, **VAE** — Part 2'nin geri kalanının yol haritasıdır; her biri sıfırdan kurulacak.

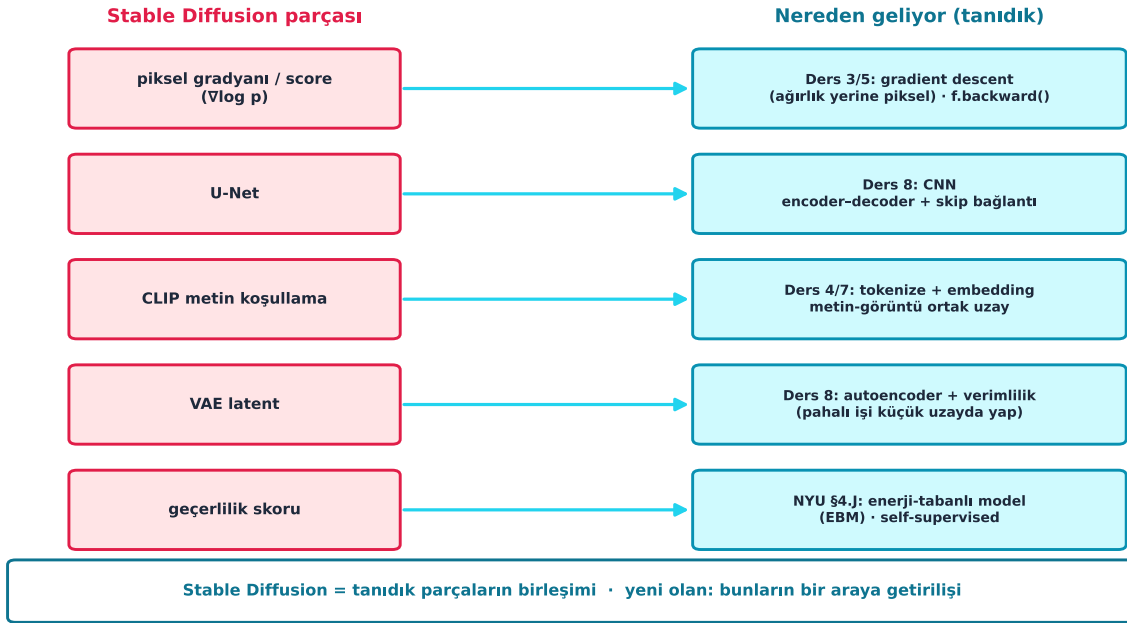
- **Geriye (pipeline):** Yüksek seviye `pipe()` bu beş adımı tek satırda gizler; düşük seviye açılım, “framework sihir değil, bizim kuracağımızın temiz hâli” disiplindir (Ders 8’deki “sıfırdan bil, hazırı güvenle kullan” ile aynı).

16.17 Kapanış

Ders 9 Part 2’yi top-down açtı: Stable Diffusion’ı kullandık (prompt, guidance, negative prompt, img2img, textual inversion) ve nasıl çalıştığını sezgisel olarak gördük — “bir şeyi daha geçerli yapan gradyan yönünde pikselleri ittir”. Üç bileşeni tanıdık: **U-Net** (gürültü tahmincisi), **VAE** (latent sıkıştırma), **CLIP** (metin koşullama). Part 2’nin geri kalanı bunları sıfırdan kuracak.

“In Part 2 we’re going to build our own everything from scratch, including our own calculus things.” — Howard, 0:45

Şekil 16.8 dersin sentezidir: Stable Diffusion sihir değil — her parçası Ders 3-8 ve NYU’dan tanıdık. Pixel gradyanı/score (Ders 3/5 gradient descent), U-Net (Ders 8 CNN), CLIP (Ders 4/7 tokenize + embedding), VAE (Ders 8 autoencoder + verimlilik), geçerlilik skoru (NYU §4.J enerji-tabanlı model). Yeni olan parçalar değil, **bir araya getirilişleridir**.



Şekil 16.8: (sentez: Stable Diffusion sihir değil — Ders 3-8 + NYU parçalarının birleşimi)

💡 Builder Notu — 9A/9B/10: Derinliğe İniş

- **İleriye (Ders 9A/9B/10):** Ders 9A (Whitaker) latent uzayı ve guidance’ı derinleştirir; Ders 9B (Waseem + Tanishq) diffusion’ın matematiğini; Ders 10 “from the foundations”a başlar — matmul’dan backprop’a.

- **Geriye (tüm ders):** Şekil 16.8’in beş satırı bu dersin her bölümünü (pixel gradyanı, U-Net, CLIP, VAE) tek bir “tanıdık parçalar” mesajında toplar.

16.18 Bu Dersin Özeti

1. **Part 2 top-down başlar:** önce Stable Diffusion’ı **kullan**, sonra sıfırdan kur (Oynayalım, Pipeline).
2. StableDiffusionPipeline ile prompt → görüntü; num_inference_steps kalite/hız, guidance_scale prompt sadakati, negative_prompt kaçınma (num_inference_steps, guidance, negative_prompt).
3. **img2img** bir başlangıç görüntüsünü prompt’a göre yeniden çizer; **textual inversion/DreamBooth** modele yeni kavram öğretir (img2img, textual inversion).
4. Diffusion sezgisi: “bir görüntünün ne kadar geçerli olduğunu” söyleyen bir fonksiyonun **pixel gradyanı** yönünde itirmek (sihirli API, pixel gradyanı).
5. Bu, gradient descent’tir — ama ağırlıkları değil **pikselleri** günceller; gürültüden görüntü doğar (pixel gradyanı).
6. **U-Net** gürültü tahmincisidir (SD’nin 1. bileşeni); girdi gürültülü görüntü, çıktı gürültüsüz tahmin (U-Net).
7. **VAE/autoencoder** görüntüyü latent uzaya sıkıştırır (786K → 16K); U-Net orada çalışır (latent diffusion) (VAE).
8. **CLIP** metni embedding’e çevirip üretimi koşullar; **scheduler/noise schedule** adım adım gürültü kaldırmayı yönetir (CLIP, scheduler).

! Tek Bir Cümle

Stable Diffusion, CLIP ile metne koşullanmış bir U-Net’in, VAE’nin sıkıştırdığı latent uzaya, rastgele gürültüyü adım adım giderek görüntü oluşturmaktır — ve özünde bu, pikselleri “daha geçerli görüntü” yönünde ittiren bir gradient descent’tir.

16.19 Kontrol Soruları

i Soru 1: Howard’ın ‘sihirli API’ örneği diffusion’ı nasıl açıklıyor? Bu, neyin sezgisidir?

Cevap:

Sihirli API, bir görüntü alıp “bunun geçerli bir görüntü (örn. el yazısı rakam) olma olasılığını” döndüren bir fonksiyondur. Eğer bu fonksiyonumuz varsa, onun **her pikselle ilgili gradyanını** alabiliriz — “bu pikseli değiştirsem olasılık ne kadar artar?”. Sonra her pikseli bu gradyan yönünde azıcık ittirip görüntüyü biraz daha “geçerli” yaparız; tekrarlırsak gürültüden görüntü doğar. Bu, **gürültü giderme (denoising)** sürecinin sezgisidir ve pratikte o “sihirli fonksiyon” bir sinir ağıdır (U-Net). Mekanizma gradient descent’tir — ama ağırlıklar yerine pikseller güncellenir. (Şekil 16.4 geçerlilik alanı + score-ascent yörüngesini gösterir.)

i Soru 2: Stable Diffusion’ın üç bileşeni (U-Net, VAE, CLIP) ne işe yarar?

Cevap:

U-Net gürültü tahmincisidir: girdisi gürültülü bir (latent) görüntü, çıktısı o gürültünün tahmini; çıkarılınca görüntü netleşir. **VAE (autoencoder)** görüntüyü latent uzaya sıkıştırır ($512 \times 512 \times 3 \approx 786K$ sayı $\rightarrow 64 \times 64 \times 4 \approx 16K$) ve geri açar; U-Net bu küçük uzayda çalışır, $\sim 48\times$ daha ucuz (bu yüzden “latent diffusion”). **CLIP** prompt’u (metni) U-Net’in anlayacağı bir embedding’e çevirir; metin ve görüntüyü ortak uzaya gömdüğü için üretimi prompt yönünde koşullar. Üçü birlikte: CLIP yön verir, U-Net latent’te gürültü giderir, VAE sonucu görüntüye açar. (Şekil 16.7 üçünü tek hatta, Şekil 16.5 VAE sıkıştırmasını gösterir.)

i Soru 3: guidance_scale ve num_inference_steps üretimi nasıl etkiler?

Cevap:

num_inference_steps gürültüden görüntüye kaç adımda gidileceğidir: az adım (örn. 3) hızlı ama kaba, çok adım (örn. 50) yavaş ama kaliteli — kalite/hız dengesi. guidance_scale (classifier-free guidance) modelin prompt’a ne kadar sıkı uyacağıdır: düşük değer yaratıcı ama prompt’tan uzak, yüksek değer prompt’a sadık ama bazen aşırı doymun/yapay. İkisi de üretim sırasında ayarlanır (modeli yeniden eğitmeden); birlikte kalite, hız ve prompt sadakati arasındaki dengeyi belirler. (Şekil 16.2 adım sayısını, Şekil 16.3 CFG dengesini gösterir.)

i Soru 4: Neden U-Net’i piksel uzayında değil latent uzayında çalıştırırız? (builder bağlantısı)

Cevap:

$512 \times 512 \times 3$ bir görüntü ~ 786.000 sayıdır; U-Net’i milyonlarca böyle görüntüyle, her adımda bu kadar pikselle eğitmek/çalıştırmak devasa compute ister. VAE görüntüyü ~ 16.000 sayılıklı bir latent’e sıkıştırdığından ($\approx 48\times$ az), U-Net’i bu küçük uzayda çalıştırmak çok daha ucuz ve hızlıdır — kalite neredeyse korunur çünkü VAE algısal olarak önemli bilgiyi tutar. Builder açısından bu, Stable Diffusion’ı pratik kılan anahtar fikirdir (“latent diffusion”): pahalı işlemi düşük boyutlu, anlamlı bir temsilde yapmak — aynı verimlilik mantığı embedding ve quantization’da da geçerli. (Bkz. VAE, Şekil 16.5.)

16.20 Egzersizler

Egzersiz 1 (Direkt uygulama). diffusion-nbs repo’sunu aç, StableDiffusionPipeline ile kendi prompt’larını üret; torch.manual_seed’i sabitleyip tekrarlanabilirliği gözle.

Egzersiz 2 (İki-aşamalı). Aynı prompt + seed ile num_inference_steps’i 3, 16, 50 yap; guidance_scale’i 1, 7, 14 yap; kalite/sadakat farkını karşılaştır.

Egzersiz 3 (Edge case). negative_prompt ile bir kavramı (örn. renk, nesne) çıkarmayı dene; img2img’de strength’i 0.2 ve 0.9 yapıp farkı gözle.

Egzersiz 4 (Kavramsal). El yazısı rakam örneğini kâğıt üstünde izle: bir “geçerlilik fonksiyonu” + pixel gradyanı ile gürültüden rakam nasıl doğar, adım adım açıkla.

Egzersiz 5 (Sonraki dersin habercisi — 9A). Latent uzayda iki prompt embedding’i arasında interpolasyon yapıp ara görüntüler üretmeyi araştır (Whitaker Ders 9A).

16.21 Sonraki: Ders 9A İçin Hazırlık

Ders 9A: Stable Diffusion Derinlemesine (Jonathan Whitaker)

Ders 9 Stable Diffusion'ı kullanıp sezgisini verdi. Ders 9A (misafir: Jonathan Whitaker) latent uzaya ve guidance'a derinlemesine iner: latent'lerle oynama, classifier-free guidance'ın içi, embedding interpolasyonu.

Ana konular (Ders 9A):

- Latent uzay ve VAE ile kodlama/çözme
- Classifier-free guidance'ın mekaniği
- Embedding interpolasyonu ve negative prompts
- Pipeline'ı parça parça kurma



Ders 9A Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 4 — pipeline'la oyna, sezgiyi izle).
- diffusion-nbs notebook'unu kendi compute'unda çalıştır.
- Ana cümleyi tekrar oku: “Diffusion = pikselleri geçerlilik gradyanı yönünde itirmek.”

16.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
StableDiffusionPipeline	Prompt'tan görüntü üreten hazır diffusers pipeline'ı	20:48
num_inference_steps	Gürültüden görüntüye adım sayısı (kalite/hız)	21:38
guidance_scale	Prompt'a sadakat (classifier-free guidance)	40:00
negative_prompt	Çıktıyı belirtilen kavramdan uzaklaştırma	40:10
img2img	Başlangıç görüntüsünü prompt'a göre yeniden çizme	41:16
Textual inversion	Tek bir embedding eğitip yeni kavram öğretme	40:00
Sihirli API sezgisi	Geçerlilik skorunun pixel gradyanı ile gürültü giderme	47:00
U-Net	Gürültü tahmincisi; SD'nin 1. bileşeni	1:00:00
VAE / autoencoder	Görüntüyü latent'e sıkıştırma (786K → 16K)	1:10:00
CLIP	Metni embedding'e çevirip üretimi koşullama	1:00:00
Latent	Sıkıştırılmış görüntü temsili (64×64×4)	1:10:00

Kavram	Tanım	Howard'da
Scheduler / noise schedule	Her adımda kaldırılacak gürültüyü yöneten dizi	1:00:00

16.23 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: Stable Diffusion'ın Kökleri

Bu ders, Stable Diffusion'ı kullanmaktan üç bileşenine ve gradient descent köküne uzanır; köprülerin özeti:

1. **Diffusion = pixel gradient descent** → [Ders 3/5'in](#) gradient descent'i, ağırlık yerine piksel üstünde; `f.backward()` aynı autograd ([pixel gradyanı](#)).
2. **U-Net** → CNN ([Ders 8](#)); encoder-decoder + skip; Part 2'de sıfırdan (Ders 19/23) ([U-Net](#)).
3. **VAE / latent diffusion** → 6.S191 Ders 4 + [Ders 8](#) autoencoder; pahalı işlemi küçük uzayda yapma verimliliği ([VAE](#)).
4. **CLIP** → metin tokenize ([Ders 4](#)) + embedding ([Ders 7](#)); metin-görüntü ortak uzayı (çok-modal) ([CLIP](#)).
5. **guidance / negative prompt** → üretimi yönlendirme; Ders 9A/10'da matematik ([guidance](#), [negative_prompt](#)).
6. **NYU (§4.J)** → diffusion, enerji-tabanlı modeller (EBM) ve self-supervised learning ile akraba; “geçerlilik skoru” = enerji fonksiyonu sezgisi ([sentez](#)).

❗ Bu dersten tek bir şey alıp gideceksen

Stable Diffusion sihir değil, üç tanıdık parçanın birleşimidir — bir CNN (U-Net) gürültüyü tahmin eder, bir autoencoder (VAE) işi küçük bir latent uzaya taşır, CLIP metinle yön verir. Özü ise [Ders 3'ten](#) beri bildiğin gradient descent'tir: bir şeyi “daha geçerli” yapan yönde adım at. Bugün kullandık; Part 2 boyunca her parçasını sıfırdan kuracağız.

17 Ders 9A — Stable Diffusion Derinlemesine (Whitaker)

Misafir ders: Jonathan Whitaker (fastai), Ders 9’da kavramsal gördüğümüz hazır `pipe()`’ı parça parça açar ve sampling loop’u sıfırdan replike eder. VAE ile latent uzaya inip çıkarız (0.18215 ölçek), CLIP ile metni embedding’e çeviririz, noise schedule (σ) ile gürültü ekleriz, classifier-free guidance’lı bir U-Net her adımda iki tahmin yapar, scheduler gürültüyü kısmen çıkarır. `img2img` bu mekanizmanın bir uygulamasıdır; en güçlüsü ise özel guidance: istediğin herhangi bir hedefin gradyanını sampling’e enjekte etmek — tam da Ders 9’un ‘pixel gradyanı’ sezgisi.

i Bölüm bilgisi

- **Whitaker’ın videosu:** [course.fast.ai — Lesson 9A: Stable Diffusion Deep Dive](https://course.fast.ai/lesson-9a-stable-diffusion-deep-dive) (~41 dk, misafir: Jonathan Whitaker)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 9A (Ders 9’un derinlemesine eki)
- **Misafir hoca:** Jonathan Whitaker (fastai)
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [diffusion-nbs — Stable Diffusion Deep Dive](#)
- **Okuma süresi:** ~28 dk

17.1 Bu Derste Ne Var?

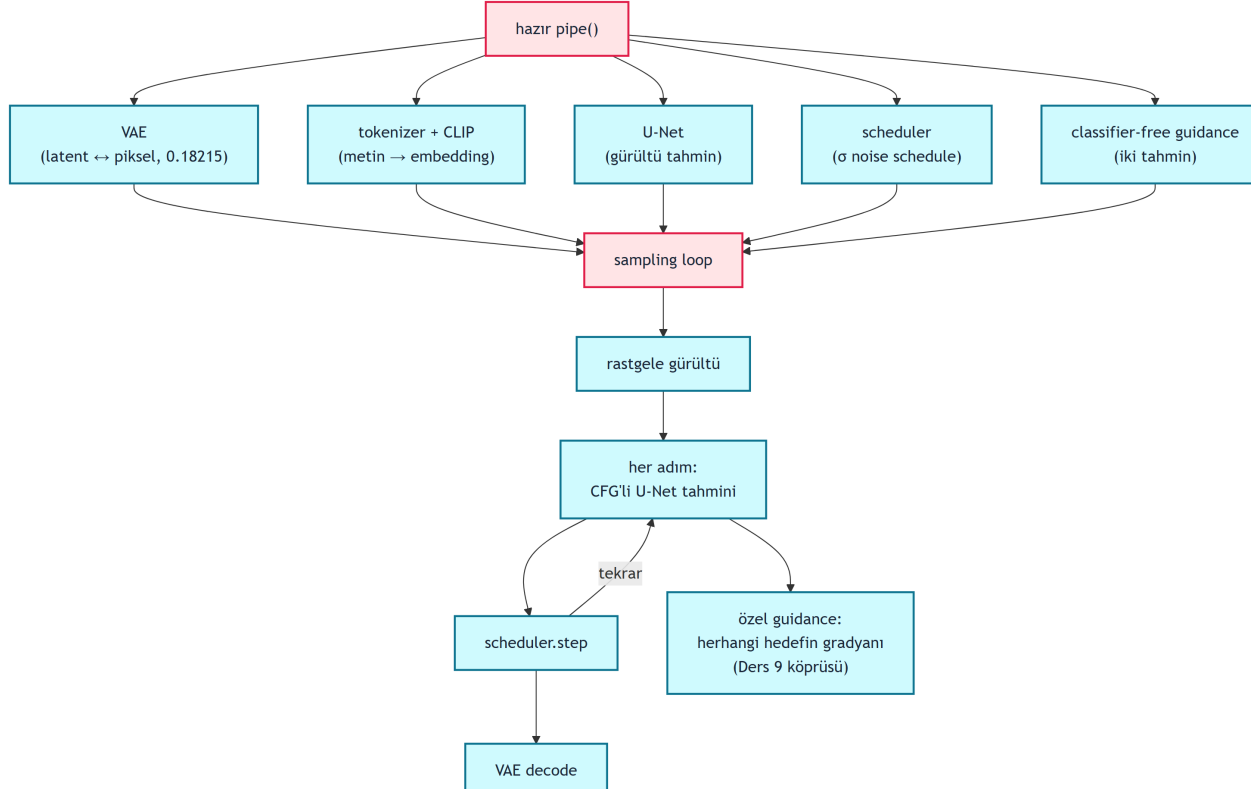
Misafir ders. Jonathan Whitaker (fastai), [Ders 9’da](#) kavramsal gördüğümüz hazır `StableDiffusionPipeline`’ı **parça parça açar** ve sampling loop’u **sıfırdan replike eder**. Ders 9 “üç bileşen + pixel gradyanı” sezgisini verdi; Whitaker o sezgiyi çalışan koda çevirir: VAE, tokenizer + CLIP, U-Net, scheduler ve classifier-free guidance birer ayrı bileşen olur, gürültü-giderme döngüsü elle kurulur.

Üç temel fikir bu dersin omurgasını kurar:

1. **Latent uzayda çalışma** — VAE görüntüyü $4 \times 64 \times 64$ ’e sıkıştırır; tüm diffusion bu küçük uzayda olur (`pil_to_latent / latents_to_pil`, 0.18215 ölçekleme) (VAE/latent → [encode/decode](#)).
2. **Gürültü ekleme + sampling loop** — eğitimde gürültü eklenir, çıkarımda σ çizelgesine göre adım adım çıkarılır; `img2img` bu mekanizmanın bir uygulamasıdır ([noise schedule](#) → [sampling döngüsü](#)).
3. **Classifier-free guidance** — her adımda **iki tahmin** (prompt’lu + prompt’suz) yapılır; fark `guidance_scale` ile çarpılıp prompt yönüne itirilir ([U-Net + CFG](#)).

“We’re going to be basically replicating this code, but now we’ll be doing it ourselves.” — Whitaker, 0:43

Şekil 34.1 bu yapıyı tek bir yol haritasında birleştirir: hazır `pipe()` beş bileşene (VAE, CLIP, U-Net, scheduler, CFG) ayrılır; sampling loop bunları her adımda birleştirir (rastgele gürültü → CFG’li U-Net tahmini → scheduler.step → VAE decode); ve özel guidance, istediğin herhangi bir hedefin gradyanını döngüye enjekte eder — rose vurgulu “hazır `pipe()`” ve “sampling loop” düğümleri dersin doruğudur.



Şekil 17.1: Whitaker’ın hazır `pipe()` çağrısını parça parça açışı: VAE, tokenizer+CLIP, U-Net, scheduler ve classifier-free guidance ayrı bileşenler; sampling loop bunları her adımda birleştirir, özel guidance ise herhangi bir hedefin gradyanını enjekte eder (Ders 9 köprüsü). (şematik)

💡 Builder Notu — `pipe()`’ın İç: Sezgiden Çalışan Koda

- **Geriye (Ders 9):** Ders 9’un “sihirli API + pixel gradyanı” sezgisi burada gerçek koda dönüşür — U-Net + scheduler + CFG. **Pixel gradyanı** fikri hem CFG’de hem **özel guidance**’ta somutlaşacak.
- **İleriye (Part 2 / NYU §4.J):** Bu loop, Part 2 Ders 19-22’de (DDPM/DDIM) sıfırdan kurulacak; CFG ve guidance, enerji-tabanlı modellerle (NYU §4.J) kavramsal akraba.
- **Tek cümle:** Whitaker, `pipe()`’ı VAE + CLIP + U-Net + scheduler + classifier-free guidance parçalarına ayırıp gürültü-giderme döngüsünü elle kurar.

17.2 Sampling Loop’u Replike Etmek

Whitaker hedefi koyar: hazır pipeline’ın yaptığını **kendi kodumuzla** yapmak — bir dizi sampling adımında latent’i gürültüden arındırıp görüntü üretmek. Önce parçaları (VAE, tokenizer, `text_encoder`, U-Net, `scheduler`) ayrı ayrı yükler; sonra **sampling döngüsünde** hepsini birleştirir. Yani **konsept haritasının** “hazır `pipe()` → beş

bileşen → sampling loop” akışını gerçek kodla yürütür.

“You’ll see that we’re going to be basically replicating this code, going through a number of sampling time steps and regenerating an image.” — Whitaker, 0:43

💡 Builder Notu — Replikasyon: Düşük Seviye Taslağın Çalışır Hâli

- **Geriye (Ders 9):** Ders 9’daki [düşük seviye pipeline taslağı](#) — beş adım — burada satır satır çalışır koda dönüşür.
- **Geriye (Ders 8):** “Framework sihir değil; bizim kuracağımızın temiz hâli” disiplini ([Ders 8](#)) burada da geçerli — `pipe()`’ı söküp parçalarından yeniden kurarız.

17.3 Auto-Encoder (VAE) ve Latent Uzay

İlk bileşen VAE. Whitaker vurgular: Stable Diffusion bir **latent diffusion** modelidir — piksellerle değil, bir autoencoder’ın latent uzayında çalışır. VAE büyük görüntüyü 4×64×64’lük bir latent’e sıkıştırır. Tüm gürültü ekleme/çıkarma bu küçük uzayda olur; piksele yalnızca en sonda, [decode](#) ile geri çıkarız.

“This is a latent diffusion model, which means it doesn’t operate on pixels, it operates in the latent space of an autoencoder. We go from this big image down to this 4 by 64 by 64 latent representation.” — Whitaker, 1:21

💡 Builder Notu — Latent Diffusion: 786K → 16K’nın Kod Karşılığı

- **Geriye (Ders 9):** Ders 9’daki VAE sıkıştırması (786K → 16K) burada `pil_to_latent/latents_to_pil` koduyla somutlaşır; latent uzay diffusion’ı ~48× ucuzlatır.
- **Geriye (Ders 8):** VAE bir autoencoder’ın ([Ders 8](#)) olasılıksal (dağılımdan örnekleyen) hâlidir — encode/decode yapısı aynı, çıkışı bir dağılımdan örnekler.

17.4 Latent’i Kodlama ve Çözme

Whitaker iki yardımcı yazar: `pil_to_latent` (görüntü → latent, VAE encode) ve `latents_to_pil` (latent → görüntü, VAE decode). Dikkat: latent’ler **0.18215** ile ölçeklenir — modelin yazarlarının seçtiği, latent’leri U-Net’in eğitildiği büyüklüğe getiren bir normalleştirme sabiti; encode’da çarpılır, decode’da bölünür.

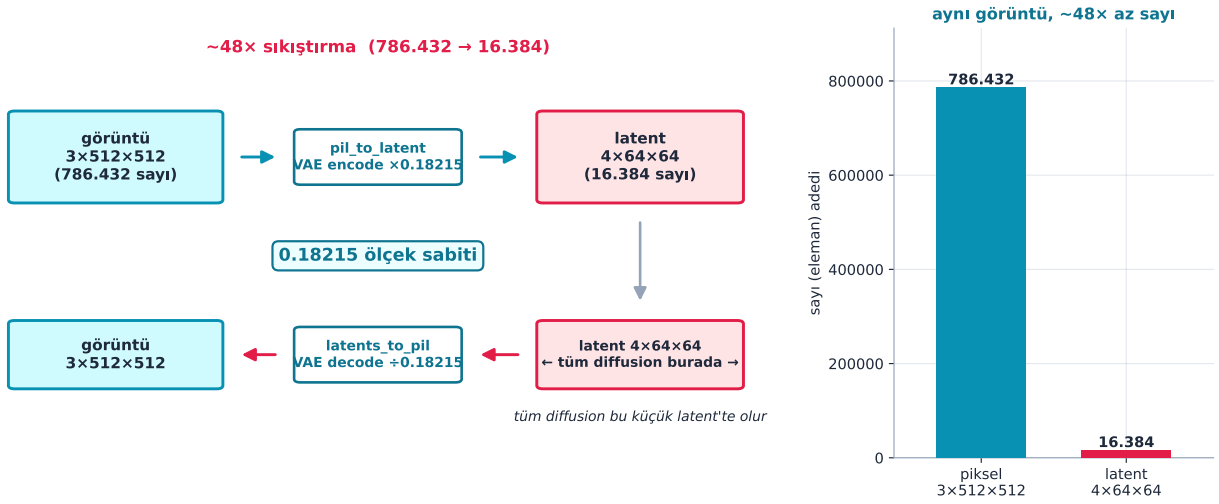
```
def pil_to_latent(input_im):
    with torch.no_grad():
        latent = vae.encode(tfms.ToTensor()(input_im).unsqueeze(0).to(torch_device))*2-1)
    return 0.18215 * latent.latent_dist.sample() # olcekleme sabiti

def latents_to_pil(latents):
    latents = (1 / 0.18215) * latents
    with torch.no_grad():
```

```
image = vae.decode(latents).sample
return image # (normalize + PIL'e çevirme adımları izler)
```

“The decoded version — very impressive compression, this is a factor of [48]; we get nice high-resolution results even though we’re only working with these 64 by 64 latents.” — Whitaker, 1:21

Şekil 17.2 bu gidiş-dönüşü gerçek sayılarla gösterir: sol panel akış şemasıyla görüntü ($3 \times 512 \times 512 = 786.432$ sayı) $\rightarrow$ pil_to_latent ($\times 0.18215$) $\rightarrow$ latent ($4 \times 64 \times 64 = 16.384$ sayı) ve geri dönüşte latents_to_pil ($\div 0.18215$); sağ panel iki büyüklüğü bar olarak karşılaştırır — $\sim 48\times$ az sayı. “Tüm diffusion bu küçük latent’te olur” mesajı dikey köprüyle vurgulanır.



Şekil 17.2: VAE latent roundtrip: tüm diffusion bu küçük latent uzayda olur (gerçek sayılar: $3 \times 512 \times 512 = 786.432 \rightarrow 4 \times 64 \times 64 = 16.384$, $\sim 48\times$; 0.18215 ölçek sabiti).

Builder Notu — encode/decode: Arama Tablosu Değil, Öğrenilmiş Sıkıştırma

- **Geriye (Ders 8):** Autoencoder encode/decode (Ders 8); VAE bunun olasılıksal (dağılımdan örnekleyen) hâli — `latent_dist.sample()` tam bu örneklemedir.
- **Builder ipucu:** 0.18215 unutulursa latent’lerin büyüklüğü U-Net’in beklediğiyle uyuşmaz; encode’da çarp, decode’da böl — ölçeği **roundtrip** boyunca tutarlı tut.

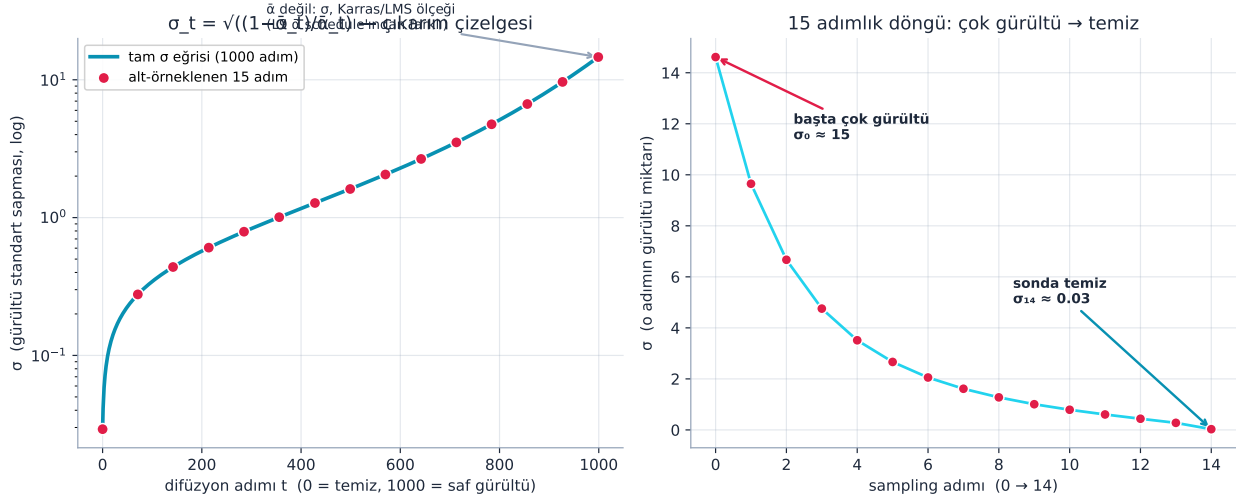
17.5 Gürültü Ekleme ve Noise Schedule

Eğitimde görüntüye gürültü eklenir; model bu gürültüyü tahmin etmeyi öğrenir. Ne kadar gürültü eklendiğini **noise schedule** belirler. `scheduler.sigmas` her adımdaki gürültü miktarını verir: başta çok yüksek, sona doğru sıfıra iner. Çıkarımda 1000 değil, az adım (örn. 15-50) kullanılır — bu, **konsept haritasındaki** σ -çizelgesinin alt-örneklemeidir.

```
scheduler.set_timesteps(15)
print(scheduler.sigmas) # her adimdaki gürültü miktarı (yüksek -> düşük)
# gürültü ekleme: noisy = latent + noise * sigma
noisy = scheduler.add_noise(encoded, noise, timesteps)
```

“This sigma is the amount of noise added; we’re going to slowly slowly try and reduce this down until ideally we get a clean image.” — Whitaker, 8:44

Şekil 17.3 çizelgeyi gerçek hesaplamayla gösterir (SD v1 scaled_linear β : $\beta_{\text{start}}=0.00085$, $\beta_{\text{end}}=0.012$): sol panel tam σ eğrisini (1000 adım, $\sigma_t = \sqrt{(1 - \bar{\alpha}_t)/\bar{\alpha}_t}$, log eksen) ve ondan alt-örneklenen 15 adımı; sağ panel bu 15 sampling adımının σ değerlerinin $\sigma_0 \approx 14.6$ ’dan $\sigma_{14} \approx 0$ ’a inişini. Bu tepe değer scheduler.sigmas’ın tepesiyle (init_noise_sigma) eşleşir. Whitaker’ın “yavaş yavaş σ ’yu sıfıra indir” dediği eğri tam budur — Ders 9’un $\bar{\alpha}$ çizelgesinden farklı bir ölçek (σ , Karras/LMS).



Şekil 17.3: Çıkarımda gürültü çizelgesi σ — sol: tam σ eğrisi (1000 adım) 15 adıma alt-örneklenir; sağ: 15 sampling adımının σ değerleri ~ 14.6 ’dan ~ 0 ’a iner. (gerçek hesaplama: SD v1 scaled_linear β , $\sigma_t = \sqrt{(1 - \bar{\alpha}_t)/\bar{\alpha}_t}$, 15 adıma alt-örneklenmiş; L9’un scheduler’ından farklı ölçek)

Builder Notu — σ Çizelgesi: DDPM/Karras’ın Çekirdeği

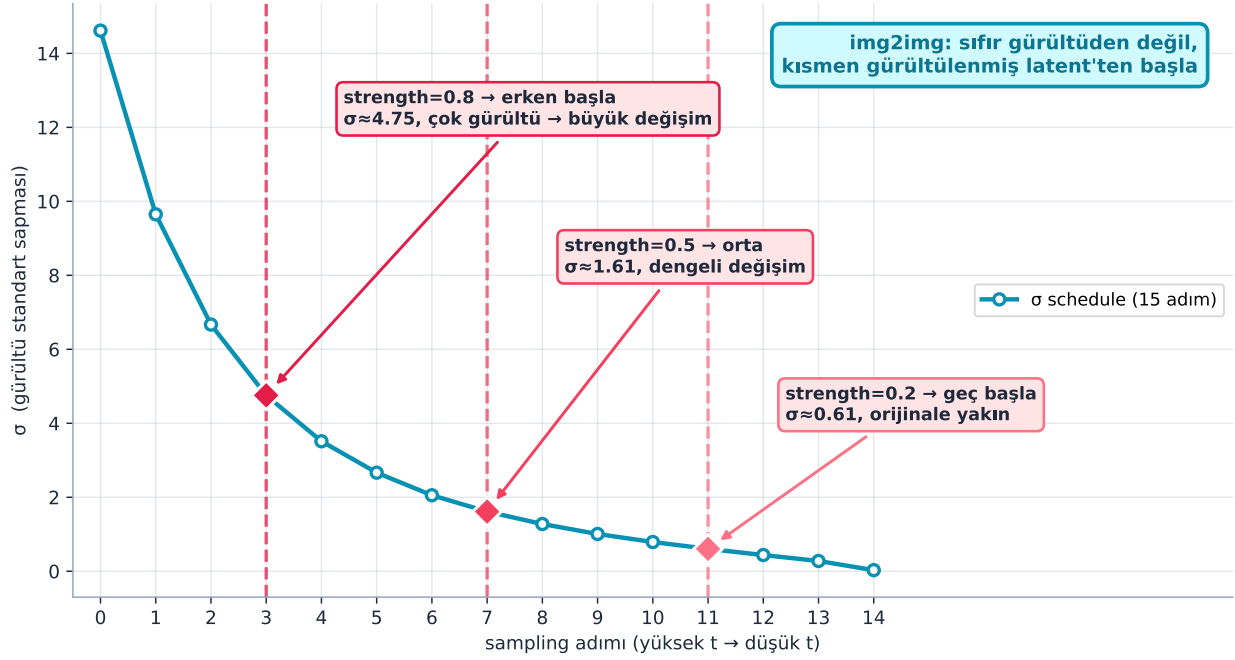
- **İleriye (Ders 19-22):** β/σ noise schedule, DDPM (Ders 19) ve Karras (Ders 22) ile Part 2’de sıfırdan kurulur (§4.I); Şekil 17.3’in σ ölçeği orada $\bar{\alpha}$ çizelgesiyle karşılaştırılır.
- **Geriye (Ders 9):** Bu σ , Ders 9’un **forward diffusion** formülü $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \varepsilon$ ’nin çıkarım-zamanı yeniden parametrisasyonudur.

17.6 img2img: Gürültüleyip Yeniden Çizme

img2img basit bir uygulamadır: bir görüntüyü **latent’e çevir**, **belirli bir adıma kadar** gürültüle (sıfıra değil), sonra oradan **sampling’e** devam et. Ne kadar gürültülersen o kadar değişir — bunu strength belirler. Yani **σ çizelgesine** baştan değil, ortadan gireriz.

“I’m now going to start with this noisy version of my input image and I’m going to [continue sampling from there].” — Whitaker, 3:55

Şekil 17.4 bu girişi σ çizelgesi üstünde gösterir: cyan eğri 15 adımlık σ inişi; her strength için rose dikey çizgi + işaretçi başlangıç noktasını işaretler — strength=0.8 erken başla ($\sigma \approx 4.75$, çok gürültü → büyük değişim), strength=0.5 ortadan ($\sigma \approx 1.61$, dengeli), strength=0.2 geç başla ($\sigma \approx 0.61$, orijinale yakın). Çekirdek sezgi: sıfır gürültüden değil, kısmen gürültülenmiş latent’ten başla.



Şekil 17.4: img2img: σ schedule eğrisi üstünde her strength'ın başlangıç noktası. strength → sampling'in hangi σ 'dan başlayacağını belirler; yüksek strength = çok gürültü = büyük değişim. (gerçek schedule: strength → sampling'in hangi σ 'dan başlayacağı; yüksek strength = çok gürültü = büyük değişim)

💡 Builder Notu — strength: Yörüngeye Hangi σ 'dan Girersin

- **Geriye (Ders 9):** Ders 9'daki img2img strength'ının iç mekanizması budur: “ne kadar adımdan / hangi σ 'dan başlanacağı”.
- **Builder ipucu:** strength orijinalin ne kadarının korunacağını ayarlar — düşük strength stil aktarımı/ince düzenleme, yüksek strength neredeyse sıfırdan üretim; aynı mekanizma inpainting ve DiffEdit'in temelidir.

17.7 Text Encoding: Tokenizer → CLIP

Prompt önce token'lara bölünür (tokenizer), sonra CLIP text encoder bunları embedding dizisine çevirir. Whitaker hazır pipe'ın atladığı bu adımları açıkça gösterir; bu embedding dizisi, [sampling döngüsünde](#) U-Net'i koşullar.


```
text_input = tokenizer(prompt, padding="max_length",
    max_length=tokenizer.model_max_length, truncation=True, return_tensors="pt")
output_embeddings = text_encoder(text_input.input_ids.to(torch_device))[0]
```

“How do we get from this sequence of tokens to [embeddings]? We look at the text_encoder embeddings.” — Whitaker, 8:44

💡 Builder Notu — CLIP Koşullama: Tokenize + Embedding’in Çok-Modal Hâli

- **Geriye (Ders 4/7):** Tokenization (Ders 4) + embedding (Ders 7); CLIP, metin embedding’lerini U-Net’e koşullama olarak verir.
- **İleriye:** Bu embedding dizisinin tek tek satırlarına müdahale [token oynama](#) ve [textual inversion](#)’ın kapısını açar.

17.8 Token Embeddings ile Oynama

Whitaker embedding’lere **token düzeyinde** müdahale edebileceğimizi gösterir: bir token’ın embedding’ini alıp değiştirebilir veya yenisini ekleyebiliriz. Bu, [CLIP’in](#) ürettiği dizinin tek bir satırını okuyup yazmaktır — [textual inversion](#)’ın temelidir.

```
prompt = 'skunk'
print(tokenizer(prompt))          # kelime -> token id'leri
token_emb_layer(torch.tensor([8797])) # token id -> embedding vektörü
```

💡 Builder Notu — Embedding = Arama Tablosu Satırı

- **Geriye (Ders 7/8):** Embedding bir arama tablosudur (Ders 7); burada o tablonun tek bir satırını (token_emb_layer[8797]) okuyup değiştiriyoruz.
- **Builder ipucu:** Token id → embedding eşlemesi türevlenebilir olduğundan, bu tek satır eğitilebilir bir parametredir — [textual inversion](#) tam bu satırı öğretir.

17.9 Textual Inversion

Textual inversion: yeni bir kavram için (örn. özel bir stil) yeni bir embedding eğitilir ve token sözlüğüne eklenir. Whitaker, eğitilmiş bir embedding’i prompt’un token embedding’leri arasına ekleyerek o kavramı çağırır — yani [token oynama](#) fikrini öğrenilmiş bir satırla yapar.

“I’m inserting them into my set of token embeddings for my prompt.” — Whitaker, 15:03

💡 Builder Notu — Tek Satır Eğit, Kavramı Çağır

- **Geriye (Ders 9):** Ders 9’da textual inversion/DreamBooth kavramsal verildi; burada eğitilmiş embedding’i [prompt dizisine](#) elle enjekte ederek mekanizmasını görüyoruz.
- **İleriye (production):** Textual inversion (tek embedding) + DreamBooth (model ağırlıkları),

kişiselleştirilmiş üretimin (özel stil/nesne) standart yöntemleridir.

17.10 U-Net ve Classifier-Free Guidance

En kritik teknik: **classifier-free guidance (CFG)**. Her sampling adımında U-Net **iki tahmin** yapar — biri prompt’la (koşullu), biri boş prompt’la (koşulsuz). Sonuç:

$$\text{noise} = \text{uncond} + s \cdot (\text{text} - \text{uncond})$$

Yani “prompt’un yönüne” ittiririz; s (guidance_scale) 1’den büyükse prompt etkisini abartırız. Bu yön farkını [sampling döngüsünde](#) her adımda hesaplarız.

“We get out two predictions for the noise... guidance_scale times the difference. If I predict without the prompt [and with], I’d like to move more in that direction, push it even further.” —

Whitaker, 18:38

Şekil 17.5 bu iki-tahmin yapısını vektör aritmetiğiyle gösterir: noise_uncond (boş prompt) ile noise_text (prompt) iki nokta; aralarındaki rose ok fark = text - uncond (prompt yönü). Her scale için ($s = 1, 7, 20$) cyan oklar pred = uncond + $s \cdot (\text{text} - \text{uncond})$ ’u çizer — s büyüdükçe tahmin koşullu noktanın ötesine taşar. Whitaker’ın “farkı daha da abart” dediği şey tam bu uzayan oktur.

💡 Builder Notu — CFG: İki Tahminin Farkını Abartmak

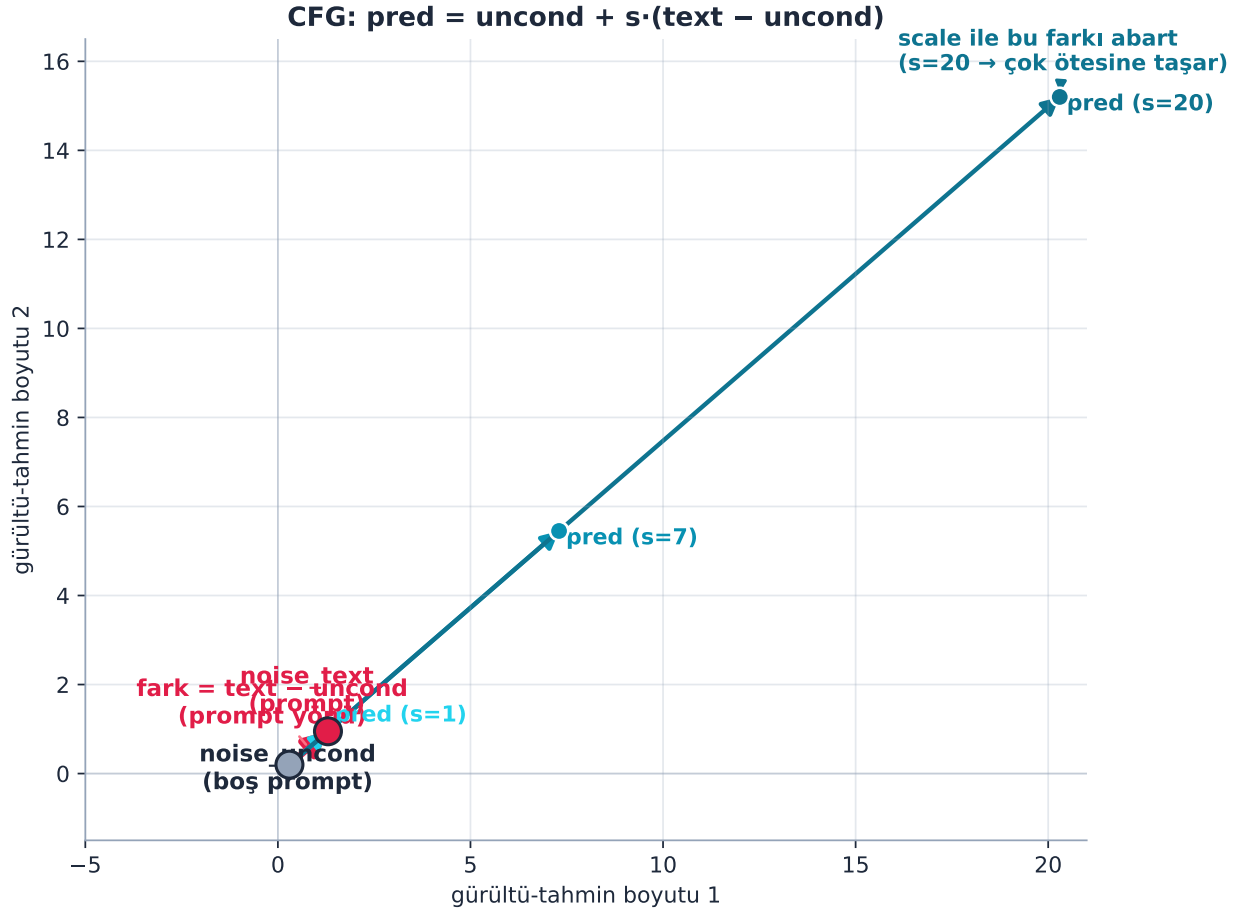
- **Geriye (Ders 9):** Ders 9’daki [guidance_scale](#)’in iç mekanizması; “iki tahminin farkını (s ile) abartmak” — sınıflandırıcı eğitmeden (classifier-free) prompt etkisini güçlendirmek. Maliyeti: adım başına iki U-Net geçişi.
- **Geriye (NYU §4.J):** CFG, koşullu/koşulsuz enerji farkını kullanır — enerji-tabanlı modellerle (NYU §4.J) kavramsal akraba; “fark = yön” sezgisi ortak.

17.11 Sampling Döngüsü

Her şey burada birleşir: rastgele gürültüden başla, her timestep’te [CFG](#) ile gürültüyü tahmin et, [scheduler](#) ile kısmen çıkar, tekrarla. [Konsept haritasının](#) sampling-loop kolu, satır satır budur.

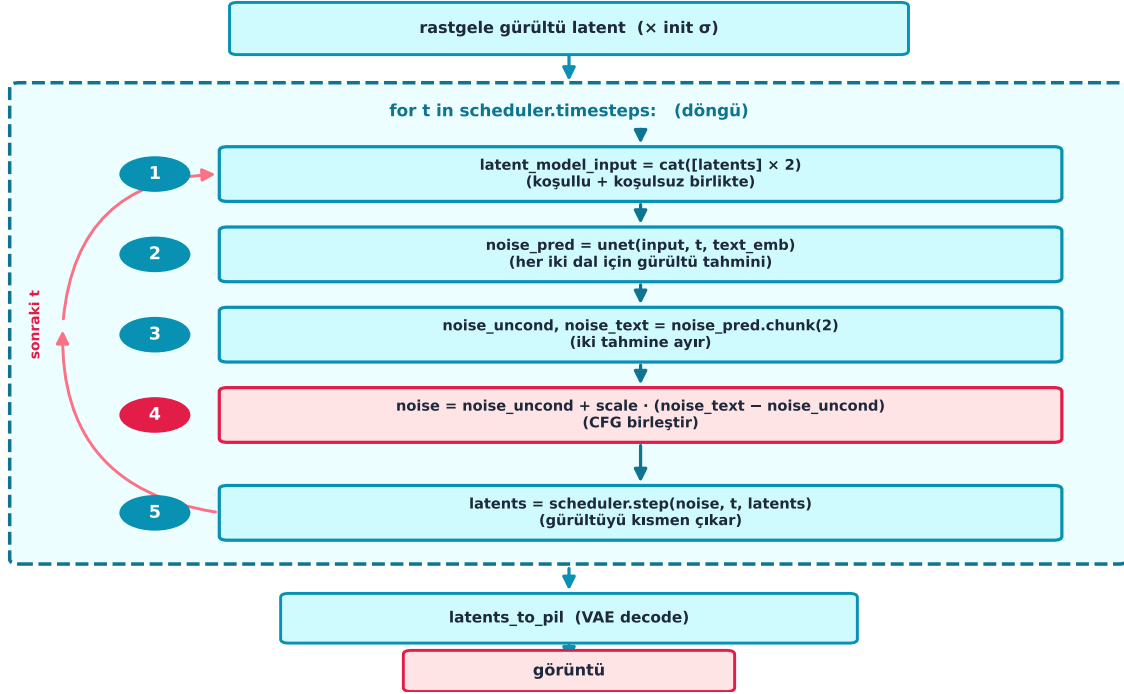
```
for i, t in enumerate(scheduler.timesteps):
    latent_model_input = torch.cat([latents] * 2) # koşullu + koşulsuz
    noise_pred = unet(latent_model_input, t, text_embeddings).sample
    noise_uncond, noise_text = noise_pred.chunk(2)
    noise_pred = noise_uncond + guidance_scale * (noise_text - noise_uncond) # CFG
    latents = scheduler.step(noise_pred, t, latents).prev_sample # gürültüyü kısmen c
```

“Current noisy latents minus sigma times the model prediction.” — Whitaker, 19:38



Şekil 17.5: Classifier-free guidance iki-tahmin yapısı: U-Net her adımda noise_uncond (boş prompt) ve noise_text (prompt) üretir; birleştirme $\text{noise\_uncond} + s \cdot (\text{noise\_text} - \text{noise\_uncond})$. Scale (Whitaker 1/7/20) prompt yönü farkını abartır. (gerçek hesaplama: CFG iki-tahmin vektör aritmetiği, Whitaker scale 1/7/20)

Şekil 17.6 bu döngüyü uçtan uca şematize eder — bu dersin **flagship** diyagramıdır. Üstte rastgele gürültü latent ($\times \text{init } \sigma$); büyük kesikli çerçeve içinde beş sıralı adım: (1) `cat([latents] \times 2)` (koşullu + koşulsuz birlikte), (2) `unet(input, t, text_emb)` (her iki dal için gürültü tahmini), (3) `chunk(2)` (iki tahmine ayır), (4) rose vurgulu CFG birleştirme `uncond + s \cdot (\text{text} - \text{uncond})`, (5) `scheduler.step` (gürültüyü kısmen çıkar); rose geri-besleme oku “sonraki t”ye döner. Döngü bitince `latents_to_pil` (VAE decode) $\rightarrow$ görüntü.



Şekil 17.6: (şematik: Whitaker’ın sampling loop’u — her adım CFG’li U-Net tahmini + scheduler.step ile gürültü giderme)

💡 Builder Notu — scheduler.step = Diffusion’ın Gradient Adımı

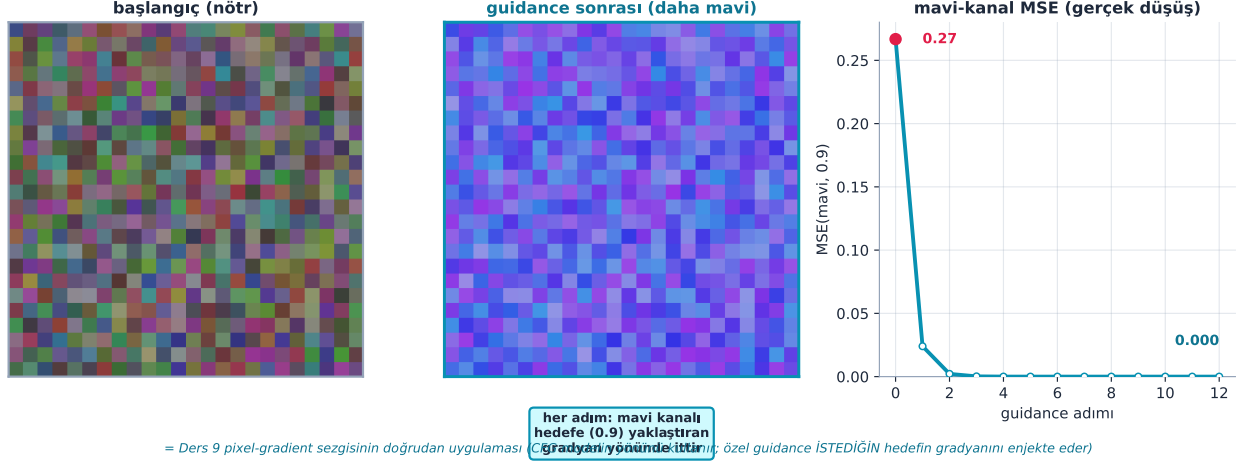
- **Geriye (Ders 5):** `scheduler.step` = Ders 5’teki “gradient adımı”nın diffusion karşılığı — latent’i biraz daha temize çeker; Whitaker’ın “current latents – $\sigma \times$ prediction” cümlesi tam bu adımdır.
- **Geriye (Ders 9):** Bu döngü, Ders 9’un **düşük seviye pipeline** beş adımının (CLIP $\rightarrow$ uncond emb $\rightarrow$ latent $\rightarrow$ U-Net+scheduler döngüsü $\rightarrow$ VAE decode) çalışan hâlidir.

17.12 Özel Guidance: Kendi Guidance Fonksiyonun

Whitaker’ın güçlü gösterisi: sampling’e **kendi** guidance fonksiyonunu ekleyebilirsin. Örnek olarak “görüntü daha mavi olsun” hedefi koyar — her adımda latent’i, mavi kanal hatasının gradyanı yönünde de ittirir. Bu, Ders 9’daki “pixel gradyanı” sezgisinin doğrudan uygulamasıdır: CFG modelin öğrendiği yönü kullanırken, özel guidance **istediğin herhangi bir** türevlenebilir hedefi enjekte eder.

“My error is going to be the difference between the blue channel [and target], and [I push the latents by its gradient].” — Whitaker, 36:32

Şekil 17.7 bunu gerçek hesaplamayla gösterir: sol panel başlangıç (nötr) görüntüsü, orta panel guidance sonrası (gözle görülür daha mavi), sağ panel mavi-kanal MSE kayıp eğrisinin gerçek düşüşü ($\sim 0.27 \rightarrow \sim 0$). Her adımda mavi kanalı hedefe (0.9) yaklaştıran gradyan yönünde itme; alt not bunun Ders 9’un pixel-gradient sezgisinin doğrudan uygulaması olduğunu vurgular.



Şekil 17.7: Özel guidance — mavi-kanal MSE’sinin gradyanı ile görüntüyü hedefe doğru ittir (gerçek hesaplama: mavi-kanal MSE’sinin gradyanı ile görüntüyü ittir — özel guidance = istediğin hedefin gradyanı)

💡 Builder Notu — Özel Guidance: Herhangi Bir Loss’un Gradyanıyla Yönlendir

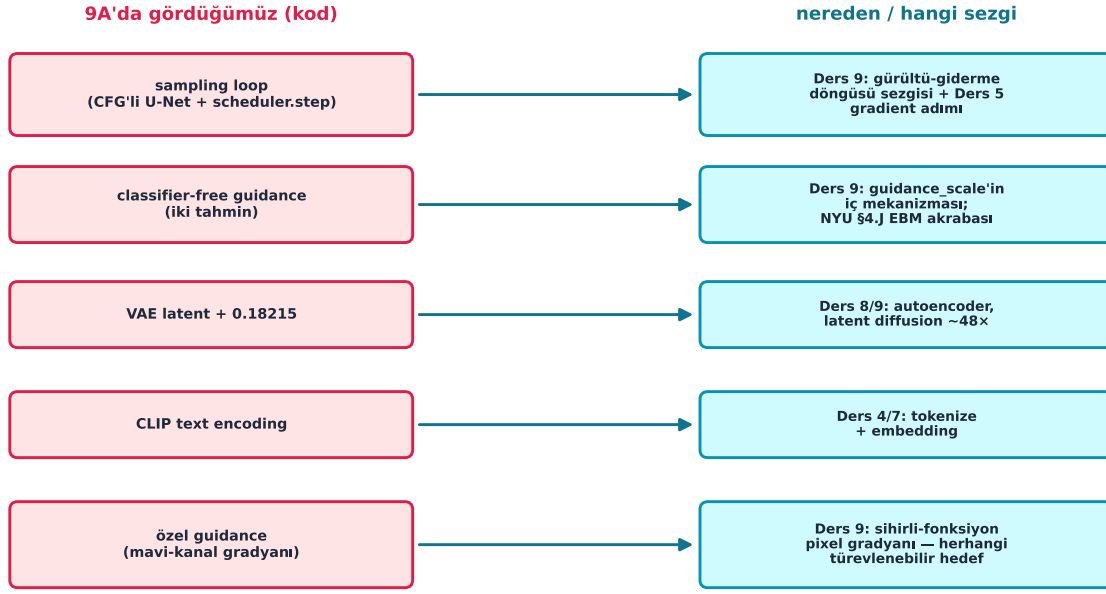
- **Geriye (Ders 3/9):** Bu tam Ders 9’un “sihirli fonksiyon gradyanı” fikri — istediğin herhangi bir hedef fonksiyonun (renk, CLIP benzerliği, kenar haritası) gradyanıyla üretimi yönlendirme; `f.backward()` aynı autograd (Ders 3).
- **İleriye:** Özel guidance, CLIP-guided diffusion ve ControlNet gibi yönlendirme tekniklerinin temel mekanizmasıdır — üretimi bir loss’un gradyanıyla itmek.

17.13 Kapanış

Whitaker, hazır `pipe()`’ı tamamen açtı: VAE ile latent’e gir/çık, `noise schedule` ile gürültü ekle, CLIP ile metni kodla, `U-Net + classifier-free guidance` ile her adımda gürültüyü tahmin et, `scheduler` ile çıkar. Ders 9’un sezgisi (pixel gradyanı) burada hem CFG’de hem özel guidance’ta somutlaştı.

“In Part 2 we’re going to build our own everything from scratch.” — Howard, 0:45 (Ders 9)

Şekil 17.8 dersin sentezidir: solda 9A’da yazdığımız kod parçaları (rose), sağda her parçanın hangi dersin / hangi sezginin uygulaması olduğu (cyan). Sampling loop ↔ Ders 9 döngü sezgisi + Ders 5 gradient adımı; CFG ↔ Ders 9 guidance_scale + NYU §4.J EBM; VAE latent + 0.18215 ↔ Ders 8/9 autoencoder; CLIP encoding ↔ Ders 4/7 tokenize + embedding; özel guidance ↔ Ders 9 pixel gradyanı. Yeni olan parçalar değil, **çalışan koda dönüşleridir**.



Şekil 17.8: 9A = Ders 9'un sezgisinin KODA dönüşü. Solda 9A'da yazdığımız kod parçaları (rose), sağda her parçanın hangi dersin / hangi sezginin uygulaması olduğu (cyan). (sentez: 9A, Ders 9'un sezgisini çalışan koda çevirir — CFG, latent, sampling loop, özel guidance)

💡 Builder Notu — 9A → 9B/10: Sezgiden Matematiğe ve Temellere

- **İleriye (Ders 9B):** Ders 9B (Waseem + Tanishq, Stability AI) bu sürecin **matematiğini** açar — forward/reverse process, noise schedule'ın türetimi, score ve olasılıksal çerçeve.
- **İleriye (Ders 10):** Ders 10 “from the foundations”ı başlatır — matmul'dan backprop'a, her şeyi sıfırdan.

17.14 Bu Dersin Özeti

1. Stable Diffusion **latent diffusion**'dır: VAE görüntüyü 4×64×64 latent'e sıkıştırır, tüm iş orada olur (0.18215 ölçekleme) ([VAE/latent](#)).
2. `pil_to_latent/latents_to_pil` ile latent'e gir/çık; ~48x sıkıştırmaya rağmen yüksek kalite ([en-code/decode](#)).
3. **Noise schedule** (σ) her adımda eklenen/çıkarılan gürültüyü belirler; `img2img` bunun bir uygulamasıdır ([noise schedule](#), `img2img`).
4. Prompt **tokenizer** → **CLIP** ile embedding'e çevrilir; token embedding'lerine müdahale textual inversion'ı sağlar ([text encoding](#), [textual inversion](#)).
5. **Classifier-free guidance**: her adımda iki tahmin (koşullu + koşulsuz); sonuç = koşulsuz + $s \times$ (koşullu – koşulsuz) ([CFG](#)).
6. **Sampling loop**: rastgele gürültü → her adımda CFG ile gürültü tahmini → `scheduler.step` ile çıkar → tekrarlar ([sampling döngüsü](#)).

7. **Özel guidance** ile herhangi bir hedef fonksiyonun gradyanı sampling’e eklenebilir (örn. “daha mavi”) (**özel guidance**).
8. Bu loop, Ders 9’un “pixel gradyanı” sezgisinin tam kod karşılığıdır (**kapanış**).

! Tek Bir Cümle

Stable Diffusion’ın hazır pipeline’ı; VAE ile latent uzaya inip, CLIP ile metni kodlayıp, her adımda classifier-free guidance’lı bir U-Net tahminiyle gürültüyü scheduler üzerinden adım adım gideren bir döngüden ibarettir — ve istediğin herhangi bir hedefin gradyanıyla yönlendirilebilir.

17.15 Kontrol Soruları

i Soru 1: Classifier-free guidance her adımda neden iki tahmin yapar ve sonucu nasıl birleştirir?

Cevap:

U-Net her adımda iki gürültü tahmini üretir: biri **prompt’la koşullu**, biri **boş prompt’la (koşulsuz)**. Bu ikisinin farkı, “prompt’un üretimi hangi yöne ittiği”ni gösterir. Sonuç = koşulsuz_tahmin + guidance_scale × (koşullu – koşulsuz). guidance_scale = 1 ise sade koşullu tahmin; 1’den büyükse prompt’un yönü **abartılır** (daha sadık ama bazen aşırı). Böylece ayrı bir sınıflandırıcı eğitmeden (classifier-free) prompt etkisi güçlendirilir — hesaplama maliyeti adım başına iki U-Net geçişidir. (Şekil 17.5 iki-tahmin vektör aritmetiğini, Şekil 17.6 adım-4 birleştirmesini gösterir.)

i Soru 2: Stable Diffusion neden ‘latent diffusion’dır ve 0.18215 ne işe yarar?

Cevap:

“Latent diffusion” demek, diffusion’ın piksel uzayında değil, VAE’nin sıkıştırdığı **latent uzayda** (4×64×64) yapılması demektir — bu ~48× daha az veri, dolayısıyla çok daha ucuz/hızlı. `pil_to_latent` VAE ile görüntüyü latent’e çevirir, `latents_to_pil` geri açar. 0.18215, latent’leri U-Net’in eğitildiği ölçeğe getiren bir **normalleştirme sabitidir** (modelin yazarlarının seçtiği); encode’da çarpılır, decode’da bölünür. Olmazsa latent’lerin büyüklüğü U-Net’in beklediğiyle uyuşmaz. (Şekil 17.2 786.432 → 16.384 roundtrip’ini gösterir.)

i Soru 3: `img2img` mekanizması nasıl çalışır? ‘strength’ neyi kontrol eder?

Cevap:

`img2img`, sıfır gürültüden değil, bir **başlangıç görüntüsünden** başlar: görüntü VAE ile latent’e çevrilir, sonra noise schedule’da **belirli bir adıma kadar** gürültülenir (tamamen değil), ve sampling oradan devam eder. `strength` bu “ne kadar gürültüleneceğini” (hangi σ ’dan başlanacağını) belirler: düşük strength → az gürültü → çıktı orijinale yakın; yüksek strength → çok gürültü → prompt’a göre büyük değişim. Yani strength, orijinal görüntünün ne kadarının korunacağını ayarlar. (Şekil 17.4 her strength’in σ çizelgesindeki başlangıç noktasını gösterir.)

i Soru 4: Whitaker’ın ‘özel guidance’ gösterisi Ders 9’un hangi fikrini somutlaştırır? (builder bağlantısı)

Cevap:

Ders 9’daki “sihirli fonksiyonun pixel gradyanı yönünde ittir” fikrini. Whitaker, sampling’e kendi hedef fonksiyonunu ekler (örn. “görüntü daha mavi olsun”): her adımda bu hedefin hatasının latent’lere göre **gradyanını** hesaplar ve latent’leri o yönde de ittirir. Yani CFG modelin öğrendiği yönü kullanırken, özel guidance **istediğin herhangi bir** türevlenebilir hedefi (renk, CLIP benzerliği, kenar haritası) sampling’e enjekte eder. Builder açısından: bu, CLIP-guided diffusion ve ControlNet gibi tüm yönlendirme tekniklerinin temel mekanizmasıdır — üretimi bir loss’un gradyanıyla yönlendirmek. (Şekil 17.7 mavi-kanal MSE’sinin gerçek düşüşünü gösterir.)

17.16 Egzersizler

Egzersiz 1 (Direkt uygulama). `pil_to_latent` ve `latents_to_pil` ile bir görüntüyü latent’e çevirip geri aç; sıkıştırma kaybını gözle (0.18215 ölçeği doğru uygula).

Egzersiz 2 (İki-aşamalı). `scheduler.sigmas`’ı çiz; bir görüntüye farklı timestep’lerde `add_noise` uygulayıp gürültü seviyelerini karşılaştır (Şekil 17.3’ı kendi adım sayınla yeniden üret).

Egzersiz 3 (Edge case). Sampling loop’unda `guidance_scale`’i 1, 7, 20 yap; çıktının prompt’a sadakatini ve “aşırı doygunluk” eşliğini gözlemler.

Egzersiz 4 (Kavramsal). CFG formülünü ($\text{koşulsuz} + s \times (\text{koşullu} - \text{koşulsuz})$) bir adım için elle yaz; $s = 0, 1, 10$ için sonucu açıkla.

Egzersiz 5 (Sonraki dersin habercisi — 9B). Noise schedule’ın ($\beta, \bar{\alpha}$) nereden geldiğini ve “forward process”in matematiğini araştır (Ders 9B).

17.17 Sonraki: Ders 9B İçin Hazırlık

Ders 9B: Diffusion’ın Matematiği (Waseem + Tanishq)

Ders 9 ve 9A diffusion’ı sezgi ve kodla verdi. Ders 9B (misafirler: Waseem + Tanishq, Stability AI) altındaki **matematiği** açar: forward/reverse process, noise schedule’ın türetilmesi, score ve olasılıksal çerçeve.

Ana konular (Ders 9B):

- Forward process (gürültü ekleme) ve Markov zinciri
- Reverse process (gürültü giderme) matematiği
- Noise schedule ($\beta, \bar{\alpha}$) türetimi
- Olasılıksal/score çerçevesi

⚠ Ders 9B Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 4 — latent + CFG).
- Sampling loop’unu kendi notebook’unda çalıştır.
- Ana cümleyi tekrar oku: “Sampling = CFG’li U-Net tahmini + scheduler ile adım adım gürültü

giderme.”

17.18 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Whitaker'da
Latent diffusion	Diffusion'ı piksel değil latent uzayda yapma	1:21
<code>pil_to_latent</code>	VAE encode: görüntü → latent ($\times 0.18215$)	1:21
<code>latents_to_pil</code>	VAE decode: latent → görüntü ($\div 0.18215$)	1:21
Noise schedule (σ)	Her adımdaki gürültü miktarı (yüksek → sıfır)	8:44
<code>add_noise</code>	latent + noise $\times \sigma$ (belirli timestep'te)	8:44
<code>img2img</code>	Görüntüyü kısmen gürültüleyip sampling'e devam	3:55
Text encoder (CLIP)	tokenizer → embedding dizisi	8:44
Textual inversion	Yeni embedding'i token sözlüğüne ekleme	15:03
Classifier-free guidance	uncond + $s \times (\text{cond} - \text{uncond})$	18:38
Sampling loop	CFG tahmini + <code>scheduler.step</code> ile gürültü giderme	19:38
Özel guidance	Herhangi bir hedefin gradyanını sampling'e ekleme	36:32
<code>scheduler.step</code>	Tahmin edilen gürültüyü latent'ten kısmen çıkarma	19:38

17.19 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: 9A'nın Sezgiden Koda Dönüşü

Bu ders, Ders 9'un sezgisini çalışan koda çevirir; köprülerin özeti:

1. **Latent diffusion** → VAE sıkıştırması (Ders 8/9); pahalı işlemi küçük uzayda yapma (**VAE/latent**).
2. **Classifier-free guidance** → koşullu/koşulsuz fark; üretimi prompt yönüne abartma; NYU EBM (§4.J) akrabası (**CFG**).
3. **Sampling loop** → `scheduler.step` = gradient adımı (**Ders 5**) diffusion karşılığı (**sampling döngüsü**).
4. **Özel guidance** → Ders 9'un **pixel-gradient sezgisi**; CLIP-guided/ControlNet temeli (**özel guidance**).
5. **CLIP text encoder** → tokenize (**Ders 4**) + embedding (**Ders 7**) çok-modal koşullama (**text**

encoding).

6. **Noise schedule** → DDPM/DDIM/Karras (Ders 19-22) ile Part 2’de sıfırdan kurulur ([noise schedule](#)).

! Bu dersten tek bir şey alıp gideceksen

Stable Diffusion’ın “büyülü” pipeline’ı, ayrı ayrı anlaşılabilir parçaların bir döngüsüdür — VAE latent’e indirir, CLIP yön verir, U-Net + classifier-free guidance her adımda gürültüyü tahmin eder, scheduler çıkarır. Ve en güçlüsü: istediğin herhangi bir hedefin gradyanını bu döngüye ekleyerek üretimi yönlendirebilirsin — tıpkı [Ders 9’daki](#) “geçerlilik gradyanı” sezgisi gibi.

18 Ders 9B — Diffusion’ın Matematiği (Waseem + Tanishq)

Misafir ders: Waseem (fastai) ve Tanishq (Stability AI), DDPM makalesinin korkutucu görünen notasyonunu satır satır açar. $q()$ bir olasılık yoğunluk fonksiyonudur; $q(x^0)$ veri dağılımı = Ders 9’un sihirli API’si; forward process her adımda β_t kadar Gaussian gürültü ekleyen bir Markov zinciridir; reparameterization ($x = \mu + \sigma \cdot \epsilon$) Gaussian’dan örneklemeyi kolaylaştırır. Forward kolay, reverse zor — Part 2’nin geri kalanı reverse’i öğrenen ağı sıfırdan kuracak.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 9B: Math of Diffusion](https://course.fast.ai/lesson-9b) (~51 dk, misafirler: Waseem + Tanishq)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 9B (Ders 9’un matematiksel eki)
- **Misafir hocalar:** Waseem (fastai) + Tanishq (Stability AI)
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** Yok — bu ders, DDPM makalesinin matematiksel notasyonunun yorumudur.
- **Okuma süresi:** ~28 dk
- **İLK LaTeX dersi:** İki block math formülü MathJax ile (\$\$). Prose içi semboller düz Unicode.

18.1 Bu Derste Ne Var?

Misafir ders. **Waseem** (fastai) ve **Tanishq** (Stability AI), DDPM makalesinin **matematiğini** ve ilk bakışta korkutan notasyonunu açar. Waseem matematik kökenli olmadığını açıkça söyler — ve tam da bu yüzden notasyonu “korkutmadan” yorumlamaya soyunur: her sembol aslında basit bir fikrin kompakt kısaltmasıdır. Ders 9 diffusion’ı sezgiyle, [Ders 9A](#) çalışan kodla verdi; bu ders altındaki matematiği gösterir.

Üç temel fikir bu dersin omurgasını kurar:

1. **$q()$ bir olasılık yoğunluk fonksiyonudur** — içine bir görüntü ver, sana “ne kadar olası olduğunu” söyleyen bir sayı dönsün. $q(x^0)$ veri dağılımıdır, $q(x_t | x_{t-1})$ ise koşullu geçiştir (**$q()$ yoğunluğu $\rightarrow q(x^0)$**).
2. **Forward process = Gaussian geçişli Markov zinciri** — her adımda görüntüye azıcık (β_t kadar) gürültü ekle; ~1000 küçük adımda saf gürültüye var (**forward formülü $\rightarrow$ Markov**).
3. **Reparameterization** — bir Gaussian’dan örneklemek basittir: birim gürültü al, varyansla ölçekle, ortalamayı ekle ($x = \mu + \sigma \cdot \epsilon$). Forward kolaydır; asıl zor olan tersini (reverse) öğrenmektir (**reparameterization $\rightarrow$ forward/reverse**).

“I don’t have a math background, and so I want to help describe how I think about it and how you can interpret all of these notations.” — Waseem, 1:55

Şekil 34.1 bu yapıyı tek bir yol haritasında birleştirir: $q()$ yoğunluk fonksiyonundan veri dağılımına (= Ders 9’un sihirli API’si), forward process’in Gaussian geçişli Markov zincirine, reparameterization’a ve **forward-KOLAY** $\leftrightarrow$ **reverse-ZOR** karşılığına. Tüm zincirin altında Stat 110 yatar: PDF, normal dağılım, koşullu olasılık, Markov.

💡 Builder Notu — Notasyondan Sezgiye: Korkutucu Değil, Kompakt

- **Geriye (Stat 110):** Tüm ders Stat 110’un olasılık yoğunluk fonksiyonu, normal dağılım, koşullu olasılık ve Markov zinciri kavramları üstüne kuruludur — yeni bir şey yok, tanıdık parçaların kompakt notasyonu var.
- **İleriye (Part 2 / NYU §4.J):** Bu matematik, DDPM (Ders 19) ve Karras (Ders 22) ile sıfırdan koda dökülür; reverse process, NYU’nun score/enerji-tabanlı modelleriyle (§4.J) akrabadır.
- **Tek cümle:** Diffusion’ın “korkutucu” notasyonu, “her adımda biraz Gaussian gürültü ekle, sonra geri öğren” fikrinin matematiksel kısaltmasıdır.

18.2 Notasyondan Korkmamak

Waseem’in tezi nettir: makalelerdeki yoğun matematik notasyonu, basit fikirleri **kompakt** biçimde ifade eder. İlk bakışta korkutur ama her sembolü tek tek açınca arkasındaki sezgi yalındır. Ders, DDPM makalesinin notasyonunu satır satır yorumlar — amaç, builder’ı makale okuyabilir hâle getirmek.

“Simple compact notations implying a lot more than you might [expect].” — Waseem, 4:16

💡 Builder Notu — Makale Okuma Becerisi

- **İleriye:** Makale okuyabilmek builder için kritik bir beceridir; notasyonu “fikre çevirmek” araştır-mayı erişilebilir kılar. Korktuğun her sembol, çözünce bildiğin bir kavrama iner.
- **İleriye (Ders 19):** DDPM makalesinin tüm forward/reverse denklemleri, bu dersin açtığı parça-lardan ($q()$, [Gaussian](#), [Markov](#), [reparameterization](#)) inşa edilir.

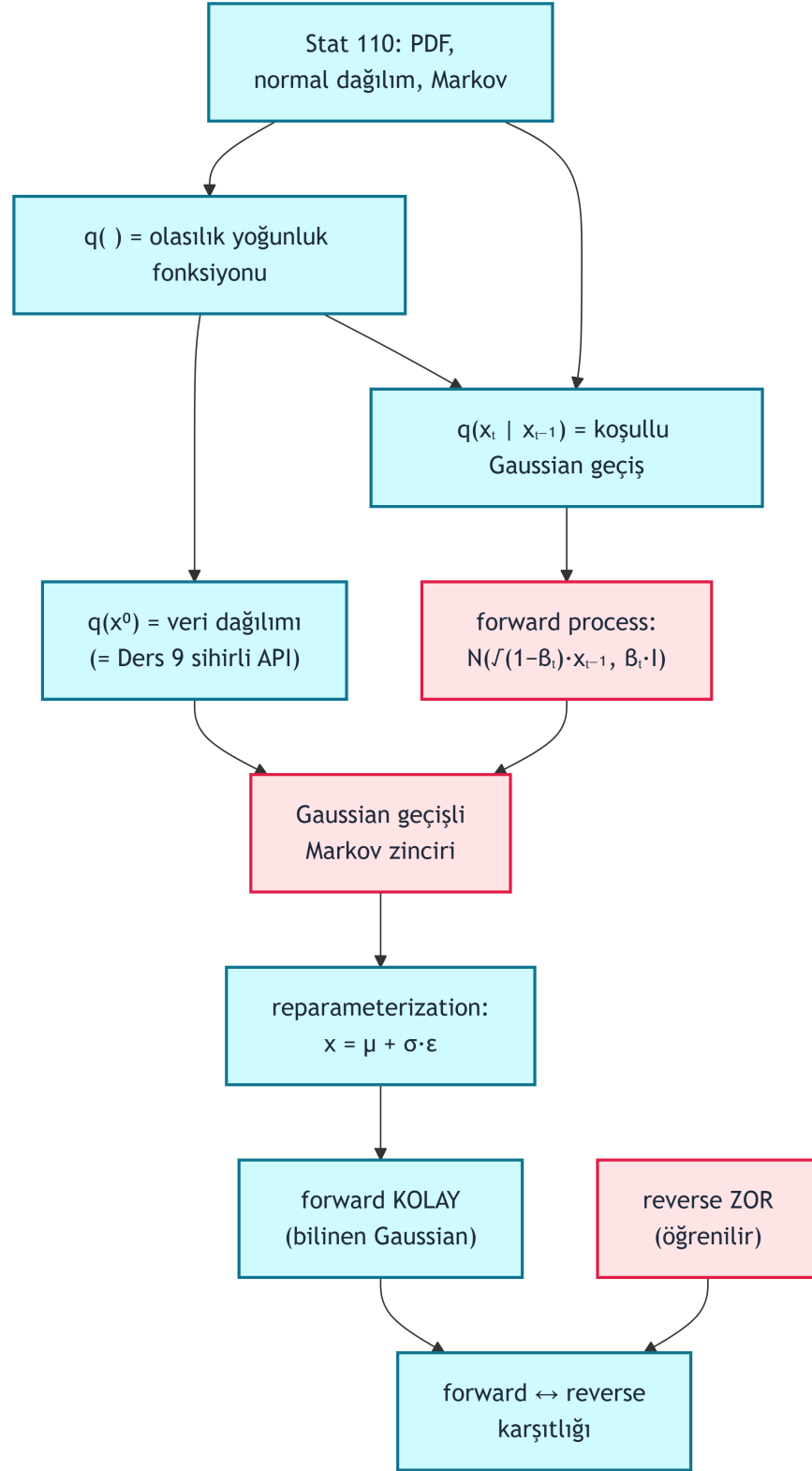
18.3 $q()$ Bir Olasılık Yoğunluk Fonksiyonudur

İlk sembol q . Waseem onu “bir kutu” (fonksiyon) gibi düşünmeyi önerir: içine bir görüntü verirsin, sana bir olasılık (yoğunluk) döner. Olasılık için genelde p kullanılır; bu makalede q tercih edilmiştir ama fikir birebir aynıdır. Yani q , bir görüntünün “ne kadar olası” olduğunu söyleyen bir ölçüdür.

“You can think of it as something that takes images... q is some function.” — Waseem, 2:25

💡 Builder Notu — PDF: Stat 110’un Temel Nesnesi

- **Geriye (Stat 110):** Olasılık yoğunluk fonksiyonu (PDF), bir değerin ne kadar “olası” olduğunu veren fonksiyondur; Stat 110’un temel nesnesi.
- **Sezgi:** q ’yu bir API gibi düşün — girdi bir görüntü, çıktı bir olasılık. Bu “API” sezgisi ders boyunca $q(x^0)$ ve [magic API bağlantısında](#) tekrar tekrar dönecek.



Şekil 18.1: Diffusion'ın matematiği kavram haritası: $q()$ yoğunluk fonksiyonu $\rightarrow$ veri dağılımı (= Ders 9 sihirli API), forward process'in Gaussian geçişli Markov zinciri, reparameterization, ve forward-KOLAY $\leftrightarrow$ reverse-ZOR karşılığı. Kaynak: Stat 110 (PDF, normal dağılım, Markov). (şematik)

18.4 Data Distribution: $q(x^0)$

İlk somut tanım $q(x^0)$ — **veri dağılımı**. Burada x^0 “orijinal görüntü”dür (örn. bir MNIST rakamı); üst-simge 0, “gürültü eklenmeden önceki, sıfıncı adım” demektir. $q(x^0)$ bir şeyin “gerçek bir görüntü olma olasılığını” verir: gerçek bir rakam yüksek değer, rastgele gürültü düşük değer alır.

“Q of x superscript 0 — they call this the data distribution.” — Waseem, 2:25

 **Builder Notu** — $q(x^0)$ = Ders 9’un Sihirli API’si

- **Geriye (Ders 9):** $q(x^0)$, Ders 9’daki “sihirli API”nin ta kendisidir — “bu geçerli bir görüntü mü?” sorusuna olasılık döndüren fonksiyon. Sezgi (Ders 9) ile matematik (bu ders) aynı nesneyi anlatır.
- **İleriye (sampling):** “Veri dağılımından örneklemek” = $q(x^0)$ ’ı yüksek yapan (yüksek olasılıklı) bir görüntü bulmaktır — üretmenin tanımı budur.

18.5 Koşullu Yoğunluk: $q(x_t | x_{t-1})$

Sonraki tanım $q(x_t | x_{t-1})$. Dikey çizgi (|) **koşullu** olasılığı gösterir: “ x_{t-1} verildiğinde x_t ’nin dağılımı”. Sezgisel olarak: bir adım önceki görüntüyü al, ona biraz gürültü ekleyerek bir sonrakini üret. Waseem bunu “başka bir sihirli API — görüntü alır, biraz daha gürültülü görüntü verir” diye tanımlar.

“This is a conditional probability density function... you can think of it as something that takes images and produces other images.” — Waseem, 9:06

 **Builder Notu** — Koşullu Olasılık: Bir Önceki Adıma Bağlı Geçiş

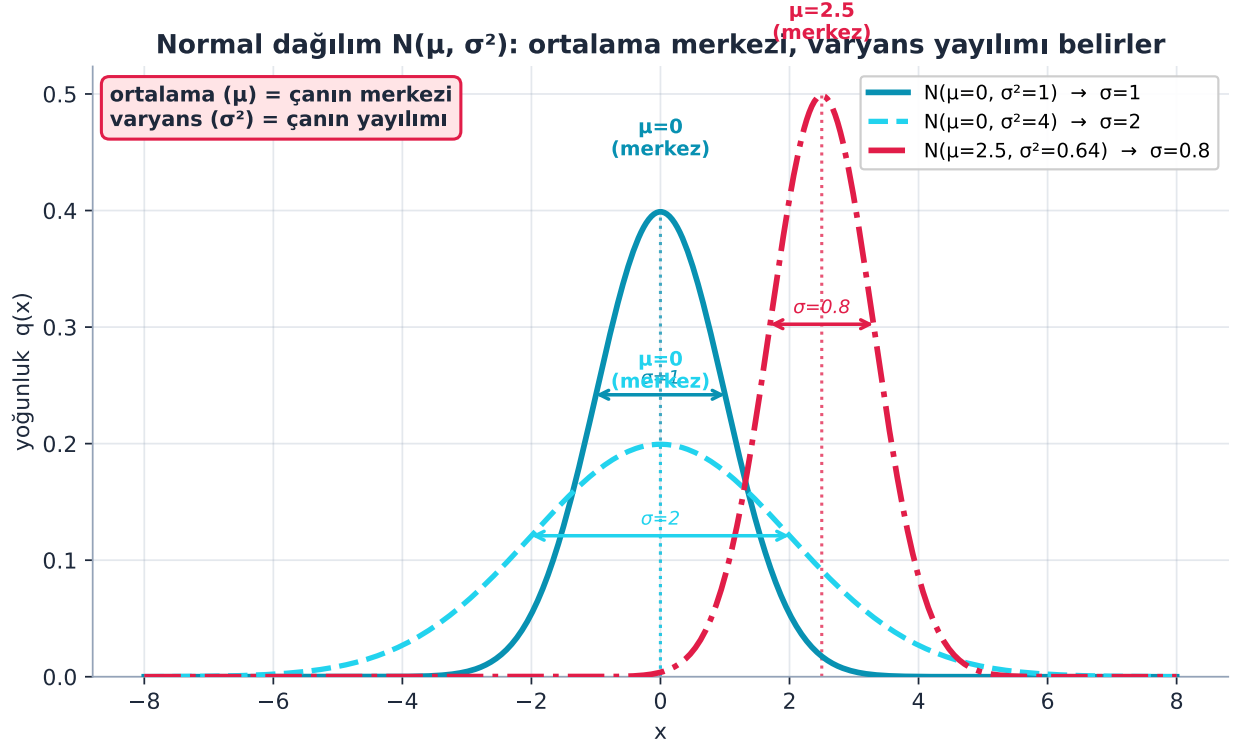
- **Geriye (Stat 110):** Koşullu olasılık $P(A | B)$; burada “bir önceki adıma bağlı” geçiş. $q(x_t | x_{t-1})$ = “ x_{t-1} ’i biliyorsan, x_t ne olur?”.
- **İleriye (forward formülü):** Bu koşullu geçiş, az sonra somut bir Gaussian formülüne ($N(\sqrt{(1-\beta_t)} \cdot x_{t-1}, \beta_t I)$) dönüşecek — şimdilik “girdi görüntü, çıktı biraz daha gürültülü görüntü” sezgisi yeter.

18.6 Normal (Gaussian) Dağılım

$q(x_t | x_{t-1})$ bir **normal (Gaussian) dağılımdır** — en bilinen olasılık dağılımı, çan eğrisi. İki parametresi vardır: **ortalama (μ , mean)** çanın merkezidir, **varyans (σ^2 , variance)** ne kadar yayıldığını söyler. Standart sapma σ , varyansın kareköküdür; varyans bazen σ^2 yazılır. Bu iki sayı dağılımı tamamen belirler.

“A Gaussian distribution... the mean and the variance of the normal distribution.” — Tanishq, 8:21

Şekil 18.2 normal dağılımı gerçek hesaplamayla gösterir: üç farklı $N(\mu, \sigma^2)$ eğrisi. Ortalama (μ) çanın merkezini kaydırır (dikey çizgi), varyans (σ^2) çanın genişliğini belirler (yarı-yükseklikteki yatay ok). $\mu=0$, $\sigma=1$ dar/yüksek; $\sigma=2$ geniş/alçak; $\mu=2.5$ kaydırılmış. Kavramsal anahtar tek satırda: **ortalama = merkez, varyans = yayılım**.

Şekil 18.2: (gerçek hesaplama: $N(\mu, \sigma^2)$ PDF; ortalama merkezi, varyans yayılımı belirler)

Builder Notu — $N(\mu, \sigma^2)$: İki Sayı, Bütün Dağılım

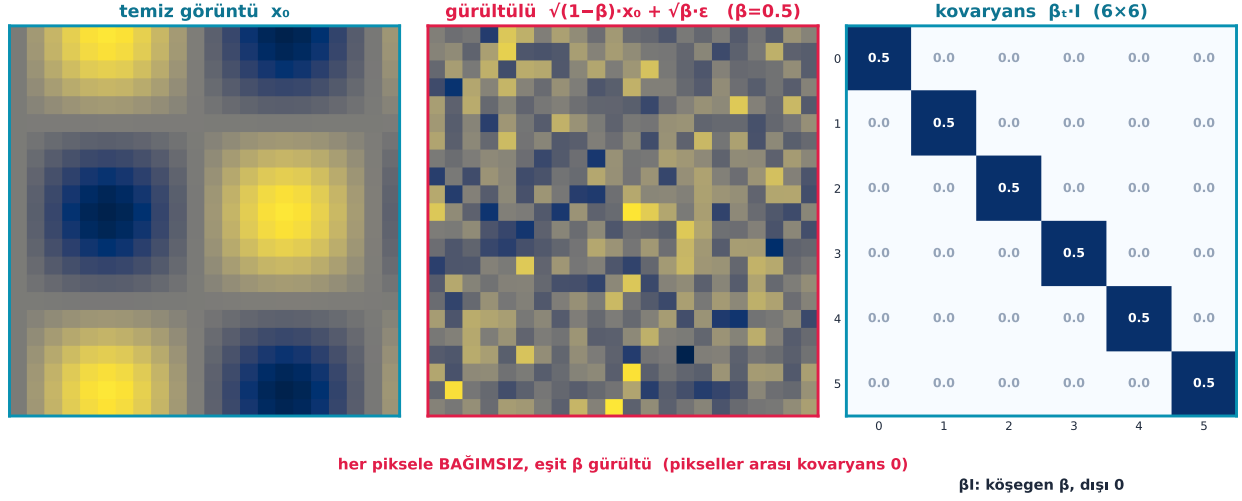
- **Geriye (Stat 110):** Normal dağılım $N(\mu, \sigma^2)$; merkezî limit teoremi ve gürültü modellemesinin temeli. μ konumu, σ^2 yayılımı verir.
- **İleriye (reparameterization):** “Sadece μ ve σ ” olması, Gaussian’dan örneklemeyi kolaylaştıran tek sebeptir — $x = \mu + \sigma \cdot \epsilon$ formülü doğrudan bu iki parametreyi kullanır.

18.7 Isotropic Kovaryans: $\beta_t \cdot I$

Görüntü tek bir sayı değil, binlerce pikseldir; bu yüzden tek bir varyans yerine bir **kovaryans matrisi** gerekir. DDPM bunu $\beta_t \cdot I$ olarak seçer: I birim matris olduğundan, her pikselin varyansı β_t , pikseller arası kovaryans **sıfırdır** — yani her piksele **bağımsız** ve aynı miktarda gürültü eklenir (isotropic Gaussian).

“We’ve now got a covariance matrix where for each individual pixel it’s beta... and the covariances between the pixels is zero.” — Waseem, 18:38

Şekil 18.3 bunu gerçek hesaplamayla gösterir: sol panel temiz görüntü x_0 , orta panel gürültülü $\sqrt{(1-\beta)} \cdot x_0 + \sqrt{\beta} \cdot \epsilon$ ($\beta=0.5$), sağ panel kovaryans matrisi $\beta_t \cdot I$ (6×6) — köşegen β , dışı 0. Mesaj net: her piksele bağımsız, eşit β gürültü; pikseller arası kovaryans sıfır.



Şekil 18.3: Isotropic gürültü $\beta_t \cdot I$: her piksele bağımsız, eşit β gürültü; kovaryans matrisi köşegen β , dışı 0 (gerçek hesaplama: $\beta_t \cdot I$ kovaryans — her piksele bağımsız eşit gürültü; forward $\sqrt{1-\beta} \cdot x_0 + \sqrt{\beta} \cdot \epsilon$).

💡 Builder Notu — $\beta_t \cdot I$: Bağımsız ve Eşit Gürültü

- **Geriye (18.06 / Stat 110):** Kovaryans matrisi (18.06); köşegen yapı = bağımsız değişkenler (Stat 110). $\beta_t \cdot I$, “her piksele bağımsız β_t gürültü” demektir — pikseller birbirinden habersiz bozulur.
- **Builder ipucu:** I (birim matris) seçimi rastgele değil; pratik bir basitleştirmedir — gürültüyü tek bir skaler β_t ile kontrol etmeyi sağlar, [noise schedule](#) tam bu β_t dizisidir.

18.8 Forward Process Formülü

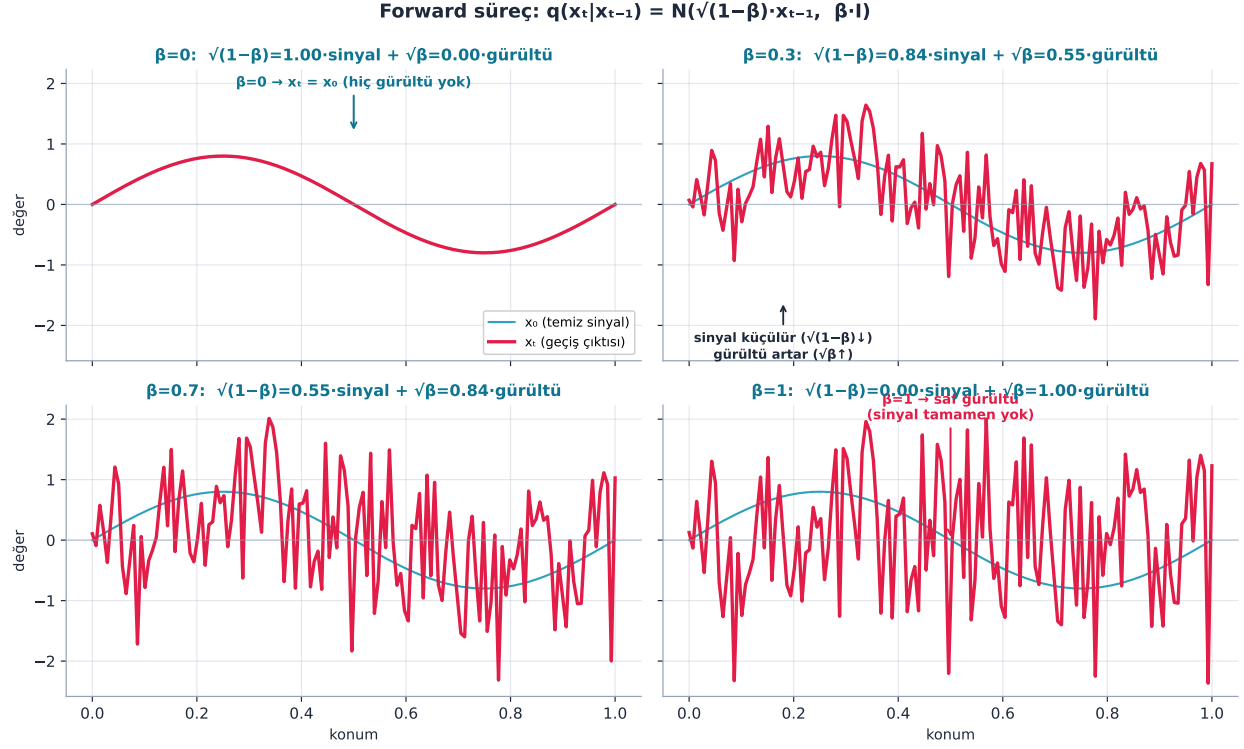
Tüm parçalar birleşince DDPM’in **forward (ileri) process** formülü çıkar — her adımda biraz gürültü ekleyen Gaussian geçiş. İşte bu dersin ilk block math’i:

$$q(x_t | x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t} x_{t-1}, \beta_t I)$$

Waseem uç durumlarla yorumlar: $\beta_t = 0$ ise $\sqrt{1-0}=1$, varyans 0 $\rightarrow$ **aynı görüntü** (hiç gürültü yok). $\beta_t = 1$ ise $\sqrt{1-1}=0$, kovaryans $I \rightarrow$ **saf gürültü** (ortalama 0, varyans 1). Arada bir yerde: görüntü + biraz gürültü karışımı. Ortalama $\sqrt{1-\beta_t} \cdot x_{t-1}$ sinyali azıcık küçültür, $\beta_t \cdot I$ gürültü ekler — her adım minik bir bozulmadır.

“If beta is one... this whole thing becomes zero and this becomes I, which implies a variance of one, so our image would just be pure noise.” — Waseem, 24:42

Şekil 18.4 bu tek adımı dört β değeri için gerçek hesaplamayla gösterir ($\beta = 0, 0.3, 0.7, 1.0$) — bu dersin **flagship** figürüdür. $\beta=0$ panelinde $x_t = x_0$ (temiz sinyal aynen); $\beta=1$ panelinde saf gürültü (sinyal tamamen yok); arada sinyal küçülür ($\sqrt{1-\beta} \downarrow$), gürültü artar ($\sqrt{\beta} \uparrow$). Her panel $\sqrt{1-\beta} \cdot \text{sinyal} + \sqrt{\beta} \cdot \text{gürültü}$ dengesini gözle gösterir.



Şekil 18.4: Forward sürecin tek adımı, β büyüdükçe sinyali söndürüp gürültüyü büyütür (gerçek hesaplama: $q(x_t|x_{t-1})=N(\sqrt{(1-\beta)} \cdot x_{t-1}, \beta \cdot I)$; $\beta=0$ aynı, $\beta=1$ saf gürültü).



Builder Notu — Her Adım Minik: Sinyal Azalır, Gürültü Artar

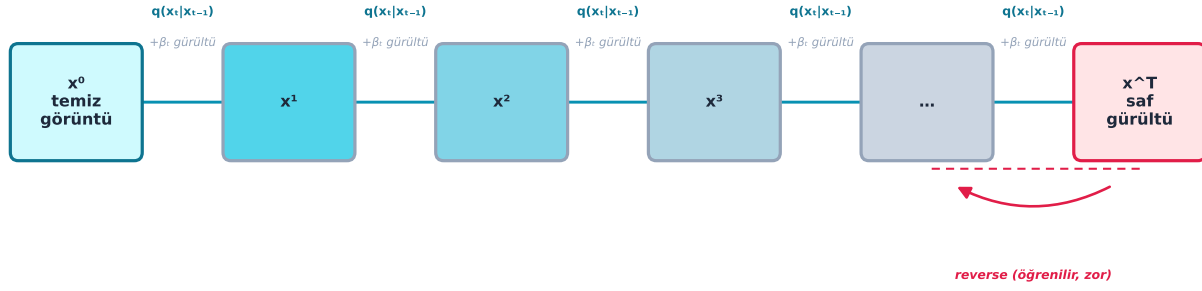
- **Geriye (Ders 5):** $\sqrt{(1-\beta_t)} \cdot x_{t-1}$ ortalamayı azıcık küçültür, $\beta_t \cdot I$ gürültü ekler; “sinyal biraz azalır, gürültü biraz artar” — her adım minik bir bozulma, ~1000 adımda saf gürültü.
- **İleriye (Ders 9A [σ çizelgesi](#)):** β_t dizisi tam Ders 9A’da gördüğün noise schedule’dır; orada σ olarak, çıkarım-zamanı yeniden parametrize edilmiş hâliyle karşılaşırısın.

18.9 Markov Process with Gaussian Transitions

Tüm forward süreç, bu adımların bir zinciridir ve adı “**Gaussian geçişli Markov süreci**”dir. Waseem terimi üç parçaya ayırır: **process** = bir dizi (sequence); **Markov** = T anındaki şey yalnızca $T-1$ anına bağlıdır (daha öncesine değil); **Gaussian** = geçişin normal dağılım olması. Üçü birlikte: her biri Gaussian olan, yalnızca bir önceki adıma bağlı geçişlerden oluşan bir zincir.

“When we say Markov it means that the thing at time T depends only on the thing at T minus one; and Gaussian is that the transition is the normal distribution.” — Waseem, 28:26

Şekil 18.5 bu zinciri şematik gösterir: x^0 (temiz görüntü, cyan) $\rightarrow x^1 \rightarrow x^2 \rightarrow \dots \rightarrow x^T$ (saf gürültü, rose). Her ileri ok bir $q(x_t|x_{t-1})$ geçişi ($+\beta_t$ gürültü); geriye doğru kesik rose ok reverse’i (öğrenilir, zor) işaretler. Altta iki anahtar: Markov (x_t yalnız x_{t-1} ’e bağlı) ve Gaussian (her geçiş $N(\sqrt{(1-\beta_t)} \cdot x_{t-1}, \beta_t \cdot I)$).



Markov: x_t yalnızca x_{t-1} 'e bağlı (geçmişe değil)

Gaussian: her geçiş normal dağılım $N(\sqrt{1-\beta_t} \cdot x_{t-1}, \beta_t \cdot I)$

Şekil 18.5: (şematik: forward = Gaussian geçişli Markov zinciri; her adım β_t gürültü, yalnız önceki adıma bağlı)

💡 Builder Notu — Markov: Gelecek Yalnız Şimdiye Bağlı

- **Geriye (Stat 110):** Markov zinciri (Stat 110); “gelecek, geçmişe değil yalnızca şimdiye bağlı” özelliği. x_t ’yi hesaplamak için yalnızca x_{t-1} yeter, $x^0 \dots x_{t-2}$ ’ye gerek yok.
- **Sezgi:** Bu “hafızasızlık” forward’ı çok ucuzlatır; her adım yerel bir Gaussian gürültü ekleme işlemidir, geçmişin tamamını taşımak gerekmez.

18.10 Reparameterization: Gaussian’dan Örneklem

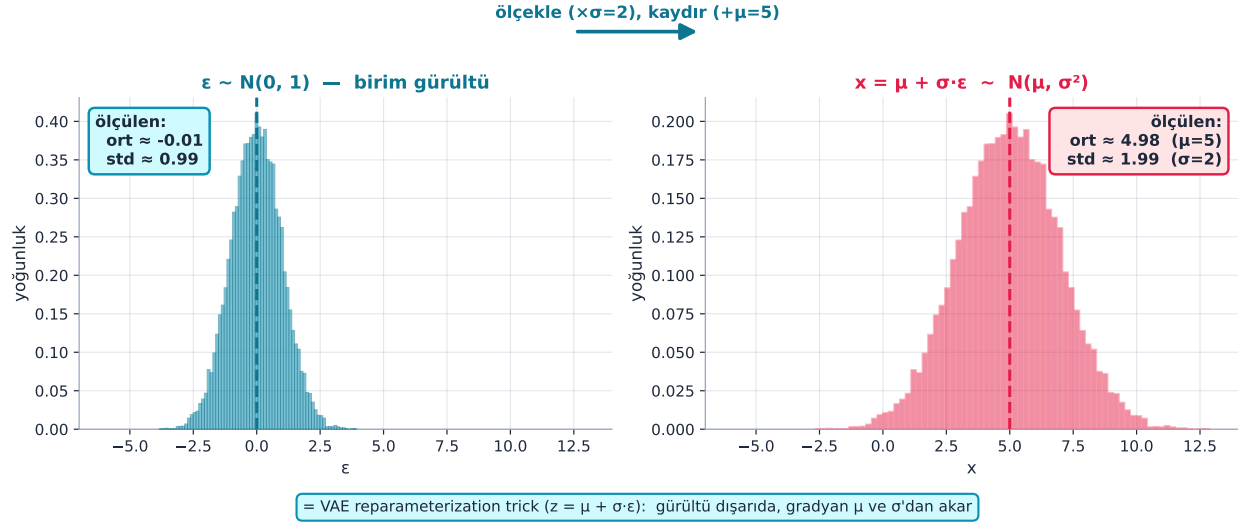
Pratik bir nokta: bir Gaussian’dan örneklemek kolaydır. **Birim gürültü** (ortalama 0, varyans 1) al, istediğin standart sapmayla ölçekle, ortalamayı ekle. İşte bu dersin ikinci block math’i:

$$x = \mu + \sigma \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Bu yüzden forward process’i kolayca örnekleyebiliriz — parametreleri (μ, σ) biliyoruz. Kod yazarken de tam böyle yapılır: kütüphaneden $N(0,1)$ örnek al, σ ile ölçekle, μ ile kaydır. Rastgeleliği tek bir ϵ terimine ayırdığı için gradyan da μ ve σ ’dan akabilir.

“Take some normal noise with a mean of zero and variance of one... multiply it by the variance and then add your mean.” — Tanishq, 30:30

Şekil 18.6 bunu gerçek hesaplamayla gösterir: sol panel birim gürültü $\epsilon \sim N(0,1)$ histogramı (ölçülen ort ≈ 0 , std ≈ 1); sağ panel $x = \mu + \sigma \cdot \epsilon \sim N(\mu, \sigma^2)$ histogramı (ölçülen ort $\approx \mu$, std $\approx \sigma$). Üstteki ok “ölçekle ($\times \sigma$), kaydır ($+\mu$)” dönüşümünü; alt not bunun **VAE reparameterization trick**’i ($z = \mu + \sigma \cdot \epsilon$) olduğunu vurgular — gürültü dışarıda, gradyan μ ve σ ’dan akar.



Şekil 18.6: Reparameterization $x = \mu + \sigma \cdot \epsilon$: birim gürültü $N(0,1)$ 'i σ ile ölçekle, μ ile kaydır $\rightarrow N(\mu, \sigma^2)$. VAE'nin reparameterization trick'inin ($z = \mu + \sigma \cdot \epsilon$) ta kendisi. (gerçek hesaplama: birim gürültüyü σ ile ölçekle + μ ile kaydır $\rightarrow N(\mu, \sigma^2)$; mean/std doğrulandı)

Builder Notu — $x = \mu + \sigma \cdot \epsilon$: VAE Trick'inin Ta Kendisi

- **Geriye (Ders 4 VAE / Stat 110):** Bu, 6.S191 VAE'deki **reparameterization trick**'in ta kendisidir ($z = \mu + \sigma \cdot \epsilon$); aynı zamanda Stat 110'daki konum-ölçek dönüşümü. Yeni bir numara değil, tanıdık bir araç.
- **Builder ipucu:** Rastgeleliği ϵ 'ya ayırmak, μ ve σ 'yı türevlenebilir bırakır — bu yüzden hem VAE hem diffusion eğitiminde gradyan akar; “stochastic node”u dışarı çıkarmanın standart yolu budur.

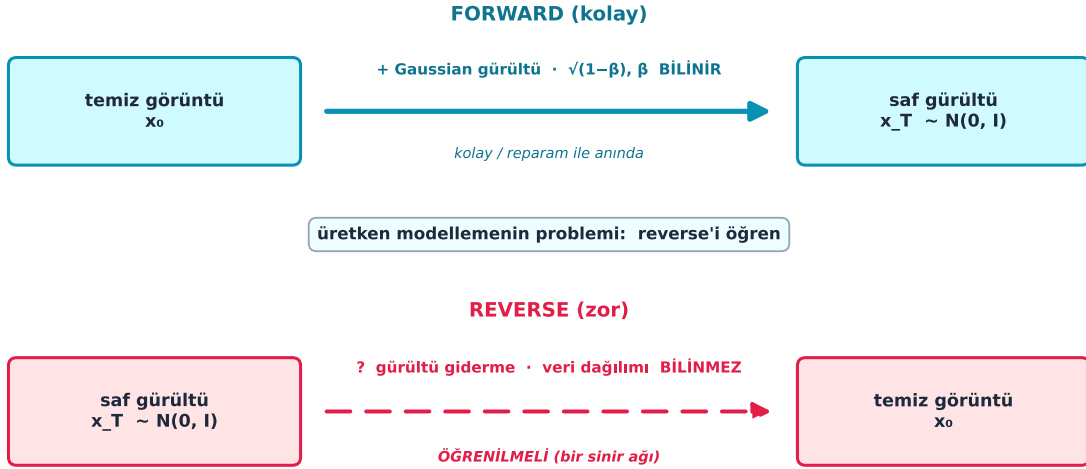
18.11 Üretken Problem: Forward Kolay, Reverse Zor

Kritik içgörü: **forward** (gürültü ekleme) kolaydır çünkü Gaussian geçişlerinin parametrelerini ($\sqrt{1-\beta_t}$, β_t) biliyoruz ve **reparameterization** ile anında örnekleriz. Ama biz **reverse**'i (gürültüden görüntü) istiyoruz — ve veri dağılımı $q(x^0)$ 'dan doğrudan örnekleyemeyiz, çünkü onu bilmiyoruz (tam da öğrenmek istediğimiz şey o). Üretken modellemenin tüm problemi budur: karmaşık veri dağılımını, kolayca örneklenebilir bir biçimde temsil etmek.

“We can't really sample from this thing — that's exactly the problem that generative modeling is trying to solve.” — Waseem, 32:20

Çözüm: çok sayıda küçük adımla (örn. 1000) görüntüyü gürültüye çeviren forward süreci, öğrenilebilir biçimde **tersine çevirmek**.

Şekil 18.7 bu asimetriyi şematize eder: üst sıra FORWARD (kolay) — temiz görüntü $x_0 \rightarrow$ saf gürültü x_T , “+ Gaussian gürültü, $\sqrt{1-\beta}$ ve β BİLİNİR, reparam ile anında”. Alt sıra REVERSE (zor) — saf gürültü $\rightarrow$ temiz görüntü, “? gürültü giderme, veri dağılımı BİLİNMEZ, ÖĞRENİLMELİ (bir sinir ağı)”. Ortada üretken modellemenin problemi: reverse'i öğren.



Şekil 18.7: (şematik: forward kolay (bilinen Gaussian), reverse zor (öğrenilir) — diffusion bu asimetriyi çözer)

Builder Notu — Reverse'i Öğrenmek: Eklenen Gürültüyü Tahmin Et

- **İleriye (Ders 19):** Reverse process'i öğrenmek = bir sinir ağına “eklenen gürültüyü tahmin et” dedirtmek; DDPM (Ders 19) tam bunu kurar. Kayıp neden MSE olur? Çünkü hedef bilinen ϵ , tahmin ağın çıktısı.
- **Geriye (Ders 9A):** [Ders 9A'nın sampling loop'u](#) bu reverse'in çalışan koduydu — her adımda gürültüyü tahmin et, kısmen çıkar. Şimdi neden öyle yaptığının matematiğini görüyorsun.

18.12 Lesson 9'un “Magic API”si ile Bağlantı

Waseem ve Tanishq sürekli Ders 9'a bağlanır: $q(x^0)$ “veri dağılımı” = Ders 9'daki **sihirli API** (bir şeyin geçerli görüntü olma olasılığı). Forward process bu dağılımı bozar; reverse onu geri kurar. Sampling = “yüksek olasılıklı (yüksek q değerli) bir görüntü bulmak”. Üç ders (9, 9A, 9B) aynı şeyi üç dilde anlatır: sezgi, kod, matematik.

“When we say we're sampling from the data distribution... we're trying to get data points that have a high value from that API.” — Tanishq, 29:25

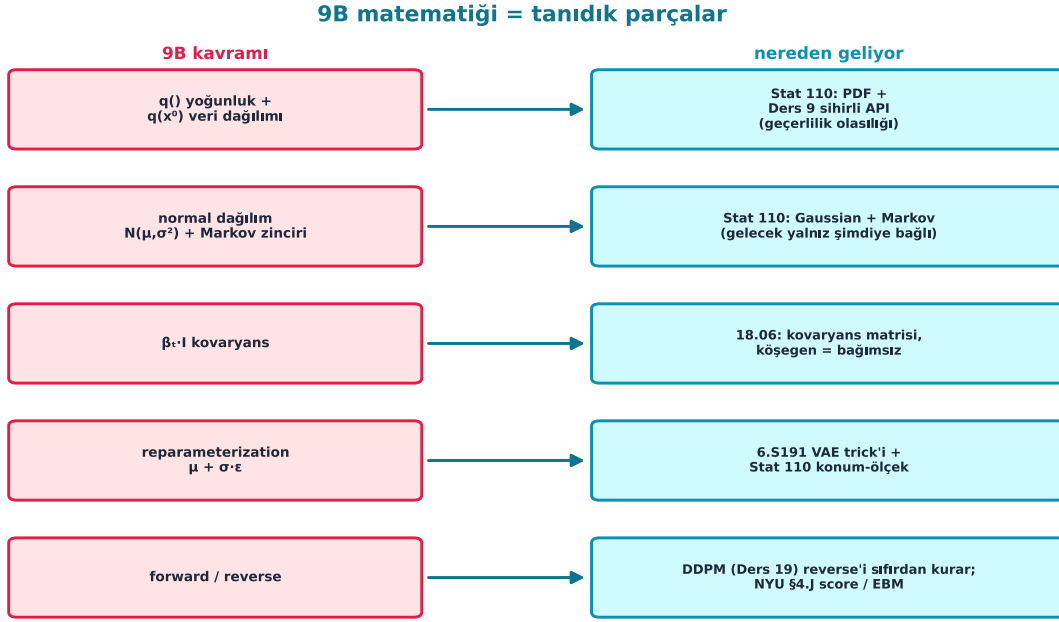
Builder Notu — Üç Dil, Tek Fikir

- **Geriye (Ders 9/9A):** [Ders 9'un sezgisi](#) (gradyanla geçerli görüntüye it) + [9A'nın kodu](#) (CFG'li sampling) + 9B'nin matematiği = aynı şeyin üç dili. $q(x^0)$ hepsinde ortak nesnedir.
- **Sezgi:** “Sampling = yüksek q değerli nokta bulmak” cümlesi, [Ders 9'un pixel gradyanı](#) sezgisiyle birebir örtüşür — geçerliliği artan yöne adım at.

18.13 Kapanış

Ders 9B, diffusion'ın “korkutucu” matematiğini sezgiye çevirdi: $q()$ bir olasılık fonksiyonu, $q(x^0)$ veri dağılımı (= magic API), forward process Gaussian geçişli bir Markov zinciri (her adımda β_t gürültü), reparameterization ile kolayca örneklenir. Forward kolay, reverse zor — ve Part 2'nin geri kalanı reverse'i öğrenen ağı sıfırdan kuracak.

Şekil 18.8 dersin sentezidir: solda 9B'nin kavramları (rose), sağda her birinin nereden geldiği (cyan). $q() + q(x^0) \leftrightarrow$ Stat 110 PDF + Ders 9 sihirli API; $N(\mu, \sigma^2) +$ Markov $\leftrightarrow$ Stat 110; $\beta_t \cdot I \leftrightarrow$ 18.06 kovaryans; reparameterization $\leftrightarrow$ 6.S191 VAE + Stat 110 konum-ölçek; forward/reverse $\leftrightarrow$ DDPM (Ders 19) + NYU §4.J score/EBM. Korkutucu notasyon, tanıdık fikirler.



Şekil 18.8: (sentez: diffusion matematiği = Stat 110 + Ders 9 + VAE parçaları; korkutucu notasyon, tanıdık fikirler)

💡 Builder Notu — 9B → 10: Matematikten Temellere

- **İleriye (Ders 10+):** Ders 10 “from the foundations”a (matmul) başlar; bu matematik Ders 19-22’de DDPM/DDIM/Karras olarak koda dökülür. Sezgi (9) → kod (9A) → matematik (9B) → temeller (10) zinciri kapanır.
- **Tek cümle:** Diffusion’ın matematiği = Stat 110 + Ders 9 + VAE parçaları; tek yeni şey, bunların kompakt notasyonda birleşmesidir.

18.14 Bu Dersin Özeti

1. Makale notasyonu korkutucu görünür ama basit fikirlerin kompakt ifadesidir; her sembol tek tek yorumlanabilir (notasyon).

2. $q()$ bir olasılık yoğunluk fonksiyonudur; $q(x^0)$ veri dağılımı (= Ders 9'un sihirli API'si).
3. $q(x_t | x_{t-1})$ koşullu geçiştir: bir önceki görüntüye biraz gürültü ekleyen Gaussian (koşullu yoğunluk).
4. Gaussian iki parametrelidir (μ, σ^2); görüntüde kovaryans $\beta_t \cdot I$ (her piksele bağımsız, eşit gürültü) (Gaussian, isotropic).
5. **Forward process:** $q(x_t | x_{t-1}) = N(x_t; \sqrt{(1-\beta_t)} \cdot x_{t-1}, \beta_t I)$; $\beta_t=0$ aynı görüntü, $\beta_t=1$ saf gürültü (forward formülü).
6. Tüm süreç **Gaussian geçişli bir Markov zinciridir** (process=dizi, Markov=yalnız önceki adıma bağlı, Gaussian=geçiş normal) (Markov).
7. **Reparameterization:** $x = \mu + \sigma \cdot \epsilon$ ($\epsilon \sim N(0, I)$); Gaussian'dan örneklemeyi kolaylaştırır (VAE trick'yle aynı) (reparameterization).
8. **Forward kolay, reverse zor;** üretken modellemenin problemi, kolay örneklenebilir bir temsil kurmaktır — Part 2 reverse'i öğrenir (forward/reverse).

! Tek Bir Cümle

Diffusion'ın matematiği özünde şudur: bir görüntüye, her biri Gaussian olan binlerce küçük adımda (β_t kadar) gürültü ekleyerek onu saf gürültüye çeviren bir Markov zinciri (forward) kurarız; asıl iş, bu kolay yönü tersine çevirip gürültüden görüntü üreten reverse'i öğrenmektir.

18.15 Kontrol Soruları

i Soru 1: Forward process formülünü ($N(x_t; \sqrt{(1-\beta_t)} \cdot x_{t-1}, \beta_t I)$) sözel açıkla. $\beta_t=0$ ve $\beta_t=1$ uçlarında ne olur?

Cevap:

Formül, bir önceki adımın görüntüsüne (x_{t-1}) biraz gürültü ekleyerek bir sonrakini (x_t) üreten bir Gaussian geçiştir: ortalaması $\sqrt{(1-\beta_t)} \cdot x_{t-1}$ (önceki görüntü hafifçe küçültülmüş), kovaryansı $\beta_t \cdot I$ (her piksele bağımsız β_t gürültü). $\beta_t=0$: $\sqrt{1}=1$, varyans 0 $\rightarrow$ görüntü aynen kalır (hiç gürültü). $\beta_t=1$: $\sqrt{0}=0$, kovaryans $I \rightarrow$ ortalama 0, varyans 1, yani **saf gürültü** (önceki görüntüden iz kalmaz). 0 ile 1 arasında: sinyal biraz azalır, gürültü biraz artar — her adım minik bir bozulmadır. (Şekil 18.4 bu dört durumu gerçek hesaplamayla gösterir.)

i Soru 2: 'Gaussian geçişli Markov süreci' teriminin üç parçası ne anlama gelir?

Cevap:

Process (süreç): bir dizi/sekans söz konusudur ($x^0 \rightarrow x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_t$). **Markov:** T anındaki durum yalnızca T-1 anına bağlıdır, daha öncesine değil ("gelecek geçmişe değil yalnızca şimdiye bağlı"). **Gaussian transitions (geçişler):** bir adımdan diğerine geçiş bir normal (Gaussian) dağılımdır — yani $q(x_t | x_{t-1})$ bir Gaussian'dır. Üçü birlikte: her biri Gaussian olan, yalnızca bir önceki adıma bağlı geçişlerden oluşan bir zincir. (Şekil 18.5 bu zinciri şematik gösterir.)

i Soru 3: Reparameterization ($x = \mu + \sigma \cdot \epsilon$) neden kullanışlıdır? Hangi önceki kavramla aynıdır?

Cevap:

Bir Gaussian'dan doğrudan örneklemek yerine, **birim gürültü** ($\epsilon \sim N(0, I)$, ortalama 0 varyans 1) alıp standart sapmayla (σ) ölçekleyip ortalamayı (μ) ekleyerek istediğin Gaussian'dan örnek üretiriz: $x = \mu + \sigma \cdot \epsilon$. Bu kullanışlıdır çünkü her kütüphane $N(0,1)$ örneği üretebilir; gerisi basit aritmetik. Ayrıca rastgeleliği tek bir ϵ terimine ayırdığı için gradyan akışına izin verir. Bu, 6.S191 Ders 4'teki **VAE reparameterization trick**'in ($z = \mu + \sigma \cdot \epsilon$) ta kendisidir ve Stat 110'daki konum-ölçek dönüşümüdür. (Şekil 18.6 mean/std doğrulamasını gerçek histogramla gösterir.)

i Soru 4: 'Forward kolay, reverse zor' ne demek? Üretken modellemenin problemi nedir? (builder bağlantısı)

Cevap:

Forward (görüntüye gürültü ekleme) kolaydır çünkü Gaussian geçişlerinin parametrelerini ($\sqrt{1-\beta_t}$, β_t) tam biliyoruz; reparameterization ile anında örnekleriz. Ama biz **reverse**'i isteriz: gürültüden gerçek bir görüntü üretmek. Veri dağılımı $q(x^0)$ 'dan doğrudan örnekleyemeyiz çünkü onu bilmiyoruz (tam da öğrenmek istediğimiz şey o). Üretken modellemenin problemi budur: karmaşık veri dağılımını, kolayca örneklenebilir bir biçimde temsil etmek. Diffusion'ın çözümü: forward'ı çok sayıda küçük adıma böl, her adımın tersini (eklenen gürültüyü tahmin etmeyi) bir sinir ağına öğret. Builder açısından: kolay yönü (gürültüleme) veri üretmek için kullan, zor yönü (gürültü giderme) modele öğret — DDPM'in özü. (Şekil 18.7 bu asimetriyi gösterir.)

18.16 Egzersizler

Egzersiz 1 (Kavramsal). Forward process formülünü $\beta_t = 0, 0.5, 1$ için elle değerlendir; her durumda ortalama ($\sqrt{1-\beta_t} \cdot x_{t-1}$) ve varyansın (β_t) ne olduğunu ve görüntünün nasıl görüldüğünü açıkla.

Egzersiz 2 (İki-aşamalı). $N(0,1)$ 'den 100 örnek çek, sonra $x = \mu + \sigma \cdot \epsilon$ ile $\mu=5, \sigma=2$ bir Gaussian'a dönüştür; histogramının ortalama ve varyansını doğru (reparameterization — Şekil 18.6'ı kendi sayınla yeniden üret).

Egzersiz 3 (Edge case). Bir MNIST görüntüsüne forward process'i 10, 100, 1000 adım uygula (her adımda β_t kadar gürültü); kaç adımda “tanınmaz” hale geldiğini gözle.

Egzersiz 4 (Notasyon). $q(x^0)$, $q(x_t | x_{t-1})$ ve $N(\mu, \sigma^2)$ sembollerini kendi cümlelerinle açıkla; “magic API” sezgisiyle eşleştir.

Egzersiz 5 (Sonraki bölümün habercisi — Ders 19). Reverse process'i bir sinir ağının “eklenen gürültüyü tahmin etmesi” olarak nasıl kurabileceğini düşün; kayıp neden MSE olur?

18.17 Sonraki: Ders 10 İçin Hazırlık

Ders 10: Daha Derine (Diving Deeper)

Ders 9/9A/9B Stable Diffusion’ı sezgi, kod ve matematikle verdi. Ders 10 hem Stable Diffusion’a biraz daha iner hem de Part 2’nin asıl omurgasını başlatır: **“from the foundations”** — her şeyi (matris çarpımından itibaren) sıfırdan yeniden kurmak.

Ana konular (Ders 10):

- Stable Diffusion’da daha ileri (DiffEdit vb.)
- “From the foundations” felsefesi
- Matris çarpımını sıfırdan (Ders 11 hazırlık)
- İzin verilen tek temel: Python + tensör oluşturma

⚠ Ders 10 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 3 — forward process’i elle uygula).
- Forward process formülünü ve reparameterization’ı kâğıt üstünde türet.
- Ana cümleyi tekrar oku: “Forward kolay (Gaussian gürültü), reverse zor (öğrenilir).”

18.18 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	9B’de
$q()$ / yoğunluk fonksiyonu	Bir görüntünün olasılığını veren fonksiyon	2:25
$q(x^0)$ / data distribution	Gerçek görüntü olma olasılığı (= magic API)	2:25
$q(x_t x_{t-1})$	Koşullu geçiş: önceki görüntüye gürültü ekle	9:06
Gaussian (normal)	Çan eğrisi; mean (μ) + variance (σ^2) ile tanımlı	8:21
$\beta_t \cdot I$ (isotropic kovaryans)	Her piksele bağımsız, eşit gürültü	18:38
Forward process	$N(x_t; \sqrt{(1-\beta_t)} \cdot x_{t-1}, \beta_t I)$; adım adım gürültüleme	27:05
Markov süreci	T anı yalnızca T-1 anına bağlı	28:26
Gaussian transition	Geçişin normal dağılım olması	28:26
Reparameterization	$x = \mu + \sigma \cdot \epsilon$; Gaussian’dan kolay örnekleme	30:30
Forward vs reverse	Gürültüleme kolay, gürültü giderme öğrenilir	32:20
Sampling	Yüksek olasılıklı (geçerli) görüntü bulma	29:25
β_t (beta / noise schedule)	Her adımda eklenen gürültü miktarı	18:38

18.19 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: 9B Matematiğinin Tanıdık Kökleri

Bu ders, diffusion'ın korkutucu notasyonunu tanıdık kavramlara indirger; köprülerin özeti:

1. **Olasılık yoğunluğu / normal dağılım / Markov** → tamamı Stat 110; diffusion matematiğinin temeli ([Gaussian](#), [Markov](#)).
2. **Forward process** → Gaussian geçişli zincir; DDPM (Ders 19) bunu koda döker ([forward formülü](#)).
3. **Reparameterization** ($\mu + \sigma \cdot \epsilon$) → 6.S191 VAE trick'i; gradyan akışına izin verir ([reparameterization](#)).
4. **Forward kolay / reverse zor** → üretken modellemenin merkez problemi; reverse bir ağla öğrenilir ([forward/reverse](#)).
5. $q(x^0) = \text{magic API}$ → [Ders 9'un sezgisi](#); reverse, "geçerlilik" yönünde adım atmak ([magic API](#)).
6. **NYU (§4.J)** → reverse/score, enerji-tabanlı modeller ve score matching ile akraba ([forward/reverse](#)).

❗ Bu dersten tek bir şey alıp gideceksen

Diffusion'ın korkutucu matematiği aslında tek bir basit fikrin kısaltmasıdır — bir görüntüye her biri Gaussian olan binlerce minik adımda gürültü ekle (forward, kolay), sonra bu süreci tersine çeviren ağı öğret (reverse, zor). $q(x^0)$ [Ders 9'un sihirli API'si](#), $\beta_t \cdot I$ her piksele bağımsız gürültü, reparameterization ise VAE'den tanıdığın $\mu + \sigma \cdot \epsilon$. Notasyon değişir; fikir basit kalır.

19 Ders 10 — Daha Derine (Diving Deeper)

Part 2'nin dönüm noktası: Howard önce Stable Diffusion'ı bitirir (distillation ile hızlandırma + tam pipeline kod yürüyüşü), sonra Part 2'nin asıl omurgasını başlatır — « from the foundations »: yalnızca Python + standart kütüphane + tensör izinli; matmul'dan U-Net'e kadar her şey sıfırdan kurulur. İlk foundation matris çarpımıdır: üç iç içe döngüden numba'ya, oradan (Ders 11'de) broadcasting'e. Kural net: bir şeyi bir kez sıfırdan kur, gerçekten anla, sonra hazır versiyonunu güvenle kullan.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 10: Diving Deeper](https://course.fast.ai/lesson/10) (~109 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 10
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/01_matmul](#)
- **Okuma süresi:** ~38 dk
- **Dönüm noktası:** Part 2'nin ilk yarısı Stable Diffusion'ı **bitirir**, ikinci yarısı « from the foundations » omurgasını **başlatır**. Bu ders ikisini birbirine bağlar.

19.1 Bu Derste Ne Var?

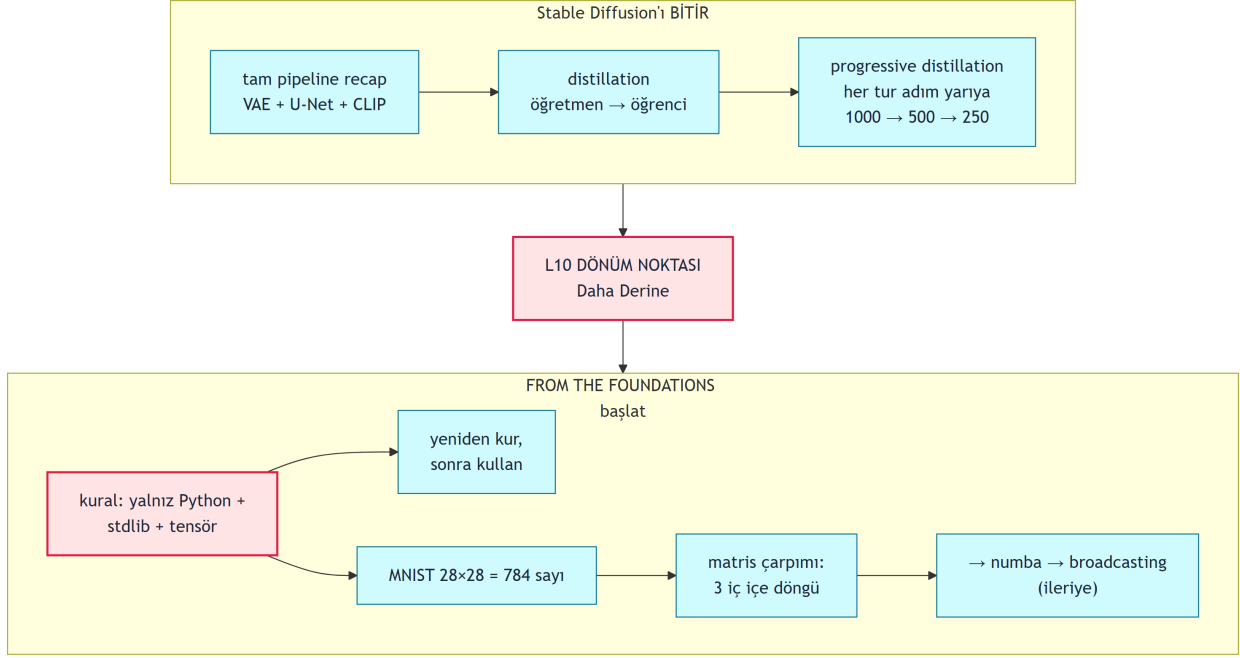
Part 2'nin ikinci dersi bir **dönüm noktasıdır**. İlk yarı Stable Diffusion'ı bitirir: hızlandırma ([distillation](#)), tam pipeline kod yürüyüşü ([pipeline](#)). İkinci yarı Part 2'nin asıl omurgasını başlatır: « **from the foundations** » — Stable Diffusion'a giden her şeyi (matris çarpımından itibaren) sıfırdan yeniden kurmak ([felsefe](#)).

Üç temel fikir bu dersin omurgasını kurar:

1. **Distillation ile hızlandırma** — bir “öğretmen” modelin birkaç adımını, bir “öğrenci” modelin tek adımına sıkıştırma; diffusion'ı az adımda çalıştırma ([distillation](#) → [progressive](#)).
2. « **From the foundations** » kuralı — yalnızca Python + standart kütüphane + tensör oluşturma izinli; geri kalan her şey (matmul, autograd, Learner, U-Net) sıfırdan kurulacak. Bir şeyi yeniden kurunca, hazır versiyonunu kullanmaya hak kazanırsınız ([felsefe](#) → [incelik](#)).
3. **Matris çarpımından başlamak** — ilk foundation matmul: önce üç iç içe Python döngüsüyle (çok yavaş), sonra adım adım hızlandırma ([matmul](#) → [üç döngü](#) → [numba](#)).

“The goal is to get to Stable Diffusion from the foundations... we're going to rebuild everything starting from this foundation.” — Howard, 53:47

Şekil 34.1 bu yapıyı tek bir yol haritasında birleştirir: solda Stable Diffusion’ı **bitir** (recap + distillation + progressive distillation), sağda « from the foundations »1 **başlat** (kural → MNIST → matmul → numba/broadcasting). Ortadaki dönüm noktası ikisini birbirine bağlar.



Şekil 19.1: Ders 10 kavram haritası: Part 2’nin dönüm noktası — Stable Diffusion’ı distillation ile BİTİR, sonra ‘from the foundations’ (yalnız Python + tensör) ile sıfırdan kurmaya başla (şematik).

💡 Builder Notu — Dönüm Noktası: Kullanmaktan Kurmaya

- **Geriye (Ders 5 + Karpathy):** “Sıfırdan kur” felsefesi Ders 5’in (Titanic from-scratch) ve tüm Karpathy serisinin ruhu; Part 2 bunu Stable Diffusion ölçeğine taşır.
- **İleriye (Ders 11-25):** Bu lesson, foundations yolculuğunu açar — matmul (11) → backprop (13) → Learner (16) → DDPM (19) → diffusion (25).
- **Tek cümle:** Stable Diffusion’a “yalnızca Python’la başlayıp” her parçayı sıfırdan kurarak varacağız; ilk parça matris çarpımı.

19.2 1. Ders 9 Özeti ve Öğrenci İşleri

Howard derse haftanın öğrenci projeleriyle ve Ders 9’un özetiyle başlar. Stable Diffusion’ın üç bileşenini (VAE, CLIP, U-Net) ve gürültü-giderme döngüsünü hatırlatır. Bu özet bir köprüdür: önceki üç ders (**Ders 9** sezgi, **Ders 9A** kod, **Ders 9B** matematik) tek bir omurgada birleşir ve Ders 10 bunu ileriye taşır.

“It’s the second lesson in Part 2, which is where we’re going from Deep Learning Foundations [to Stable Diffusion].” — Howard, 0:00

Builder Notu — Üç Dersin Sentezi

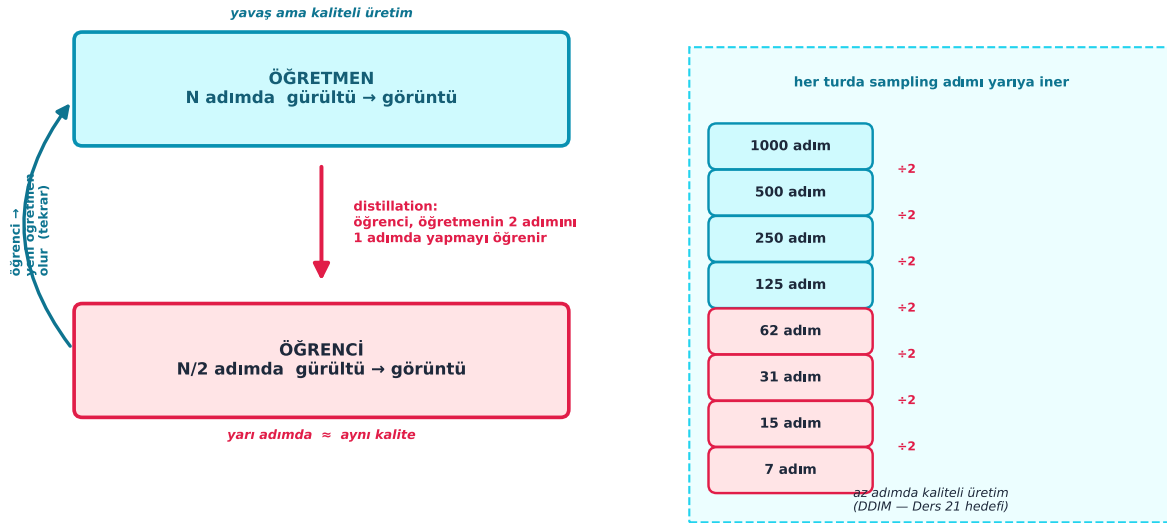
- **Geriye (Ders 9/9A/9B):** Bu ders öncekilerin üçünü (sezgi + kod + matematik) tek bir omurgada birleştirip ileriye taşır.
- **Sezgi:** Stable Diffusion'ı artık “kullanabiliyoruz”; şimdi onu altından nasıl çalıştığını bilerek “kurmaya” geçiyoruz.

19.3 2. Stable Diffusion'ı Hızlandırma: Distillation

Howard bir hızlandırma yöntemini anlatır: **distillation** (damıtma). Fikir: iyi ama yavaş bir **öğretmen (teacher)** modeli, daha hızlı bir **öğrenci (student)** modele “öğretmek”. Öğrenci, öğretmenin yaptığını daha az adımda taklit etmeyi öğrenir. Diffusion'ın temel sorunu hızdır: üretim çok sayıda (örn. 1000) sampling adımı gerektirir; distillation bu adım sayısını dramatik biçimde azaltmanın bir yoludur.

“Distillation is that you take something called a Teacher Network, which is some neural net [that works well].” — Howard, 16:56

Şekil 19.2 bu süreci şematik gösterir: üstte **ÖĞRETMEN** (N adımda gürültü → görüntü, yavaş ama kaliteli), distillation oku ile aşağıda **ÖĞRENCİ** (N/2 adımda aynı kalite); yan panelde adım sayısının her turda yarıya inişi (1000 → 500 → 250 → ...) ve öğrencinin yeni öğretmen olarak döngüye geri besleneceği işaret edilir.



Şekil 19.2: (şematik: progressive distillation — öğretmenin 2 adımı öğrencinin 1 adımı; her turda sampling adımı yarıya iner)

💡 Builder Notu — Distillation: Bilgiyi Sıkıştırma

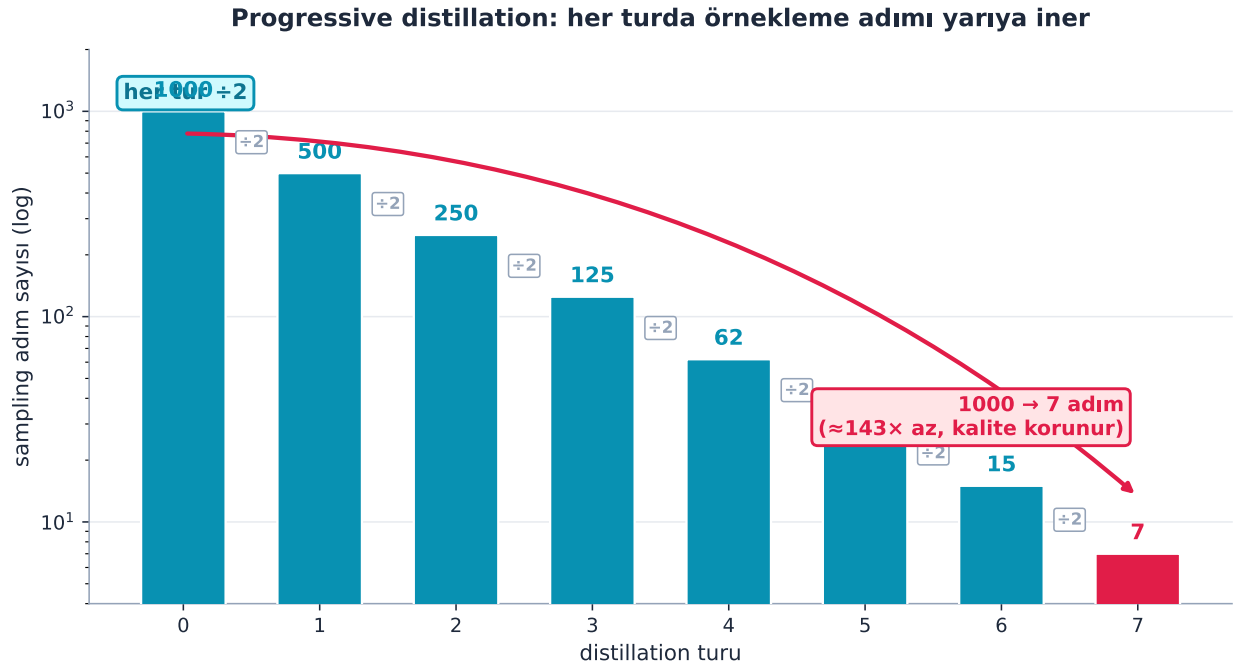
- **Mekanizma:** Öğrenci, kendi ham verisinden değil **öğretmenin çıktısından** öğrenir; öğretmenin yumuşak hedeflerini taklit ederek bilgisini daha az adıma sıkıştırır.
- **İleriye:** Distillation diffusion’a özgü değil — LLM’lerde de büyük/yavaş modeli küçük/hızlıya sıkıştırmanın production standardıdır; kalite/hız dengesini optimize eder.

19.4 3. Progressive Distillation: Adımları Yarıya İndirme

“Progressive Distillation” makalesi şunu yapar: öğrenci, öğretmenin **iki adımını tek adımda** yapmayı öğrenir. Sonra bu öğrenci yeni öğretmen olur ve süreç tekrarlanır — her turda gereken adım sayısı yarıya iner. Böylece 1000 adımlık diffusion, çok daha azına sıkışır.

“They take that student model... it learns to go directly from the noise to two goes of the student model, and then they copy that to become the next student model.” — Howard, 19:36

Şekil 19.3 bu yarılamayı **gerçek aritmetik**le gösterir: $1000 \rightarrow 500 \rightarrow 250 \rightarrow 125 \rightarrow 62 \rightarrow 31 \rightarrow 15 \rightarrow 7$ — yedi turda 1000 adım, yalnızca ~7 adıma iner ($\approx 140\times$ azalma, kalite korunur). Log-skalada her ardışık bar bir öncekinin yarısıdır; son tur (7 adım) rose ile vurgulanır.



Şekil 19.3: (gerçek hesaplama: progressive distillation adım dizisi $1000 \rightarrow 7$; her tur $\div 2$)

💡 Builder Notu — Az Adımda Üretim: Geometrik Sıkışma

- **İleriye (Ders 21):** Az adımda kaliteli üretim, DDIM (Ders 21) ve consistency modellerinin de hedefi; distillation bunun bir yolu.

- **Sezgi:** Her tur adımı $\div 2$ yaptığı için sıkışma **geometriktir** — yalnızca 7 turda $\approx 140\times$ hızlanma; üretim maliyeti doğrudan adım sayısı ile orantılı olduğundan bu pratikte büyük fark demektir.

19.5 4. CFG ve Empty Prompt (Recap)

Howard classifier-free guidance'ı (CFG) hatırlatır: her adımda prompt'lu ve **boş prompt'lu** iki tahmin yapılır; boş prompt da CLIP text encoder'dan geçer. İkisinin farkı guidance ile abartılır — bu fark, modeli “istenene yöne” daha güçlü iter ([Ders 9A](#)).

“We also put the empty prompt into our CLIP text encoder.” — Howard, 22:42

💡 Builder Notu — Boş Prompt = Koşulsuz Tahmin

- **Geriye (Ders 9A):** CFG mekaniği; boş prompt = koşulsuz tahmin. Koşullu (prompt'lu) ile koşulsuz (boş) arasındaki fark guidance ile ölçeklenir.
- **Sezgi:** Boş prompt'un da text encoder'dan geçmesi, “hiçbir şey istemiyorum” durumunu modele anlatmanın yoludur — fark bu referansa göre ölçülür.

19.6 5. Tam SD Pipeline Kod Yürüyüşü

Howard Ders 9'un düşük-seviye pipeline'ını koddan tekrar gezer: CLIP ile prompt'u embedding'e çevir, latent'i rastgele gürültüyle başlat, döngüde U-Net + CFG ile gürültüyü tahmin et, scheduler ile çıkar, VAE ile decode et. Önemli bir verimlilik vurgusu: koşullu ve koşulsuz tahminler tek bir batch'te birleştirilir — GPU aynı anda mümkün olduğunca çok iş yapsın diye.

“So we do exactly the same thing... because we like the GPU to do as many things at once as possible.” — Howard, 39:36

Pipeline'ın iskeleti, diffusers kütüphanesinin StableDiffusionPipeline parçalarıyla tam olarak şöyle görünür (kavramsal kesit):

```
# prompt -> CLIP text embedding (koşullu + koşulsuz/boş prompt birlikte)
text_input = tokenizer(prompt, padding="max_length",
                        max_length=tokenizer.model_max_length,
                        truncation=True, return_tensors="pt")
text_embeddings = text_encoder(text_input.input_ids.to("cuda"))[0]
uncond_input = tokenizer([""], padding="max_length",
                        max_length=text_input.input_ids.shape[-1],
                        return_tensors="pt")
uncond_embeddings = text_encoder(uncond_input.input_ids.to("cuda"))[0]
# GPU tek batch'te ikisini birden işlesin diye birleştir
text_embeddings = torch.cat([uncond_embeddings, text_embeddings])

# latent'i rastgele gürültüyle başlat, scheduler varyansına ölçekle
```

```

latents = torch.randn(1, unet.in_channels, height // 8, width // 8)
latents = latents.to("cuda") * scheduler.init_noise_sigma

# sampling döngüsü: U-Net gürültüyü tahmin et -> CFG -> scheduler çıkar
for t in scheduler.timesteps:
    latent_model_input = torch.cat([latents] * 2) # koşullu+koşulsuz
    noise_pred = unet(latent_model_input, t,
                      encoder_hidden_states=text_embeddings).sample
    noise_uncond, noise_text = noise_pred.chunk(2)
    noise_pred = noise_uncond + guidance_scale * (noise_text - noise_uncond) # CFG
    latents = scheduler.step(noise_pred, t, latents).prev_sample

# tüm adımlar bitince VAE decode latent -> görüntü
image = vae.decode(latents / 0.18215).sample

```

💡 Builder Notu — GPU Verimliliği: Batch'te Birleştir

- **Geriye (Ders 9/9A):** Aynı pipeline; burada GPU verimliliği (batch'te koşullu+koşulsuz birlikte) vurgulanır. `torch.cat([latents] * 2)` ile iki tahmin tek geçişte yapılır.
- **İleriye (Ders 19):** Bu döngünün her parçası (scheduler, noise tahmini, decode) Part 2'nin geri kalanında sıfırdan kurulacak; şimdi hazır kütüphane sürümüyle tanışıyoruz.

19.7 6. Rastgele Gürültüyü Ölçekleme

İnce bir nokta: başlangıç latent'i `torch.randn` ile üretilir ama scheduler'ın beklediği **varyansa** ölçeklenmelidir (`init_noise_sigma` ile çarpılır). Bu olmadan üretim bozulur — başlangıç gürültüsü yanlış ölçekteyse, döngünün matematiği tutmaz.

“Scaling random noise to ensure variance.” — Howard, 41:29

💡 Builder Notu — `init_noise_sigma`: Doğru Ölçek

- **Geriye (Ders 9B):** Reparameterization ($\mu + \sigma \cdot \epsilon$) mantığı; başlangıç gürültüsü doğru ölçekte olmalı — yanlış varyans, scheduler'ın beklediği dağılımı bozar.
- **Builder ipucu:** `latents = torch.randn(...) * scheduler.init_noise_sigma` tek satır gibi görünür ama atlanırsa üretim çöker; “küçük detay, büyük etki”.

19.8 7. Sampling Loop ve VAE Decode

Döngünün özü: her adımda gürültüyü tahmin et, latent'ten çıkar, yeni latent elde et. Tüm adımlar bitince VAE decoder latent'i görüntüye çevirir. Bu, §5'teki pipeline kodunun kavramsal özetidir: tahmin → çıkar → tekrarla → decode.

“And remove it to give us our new latents. So that’s the loop. And at the end of all that, we decode it in the VAE.” — Howard, 44:31

💡 Builder Notu — Loop = Reverse Process

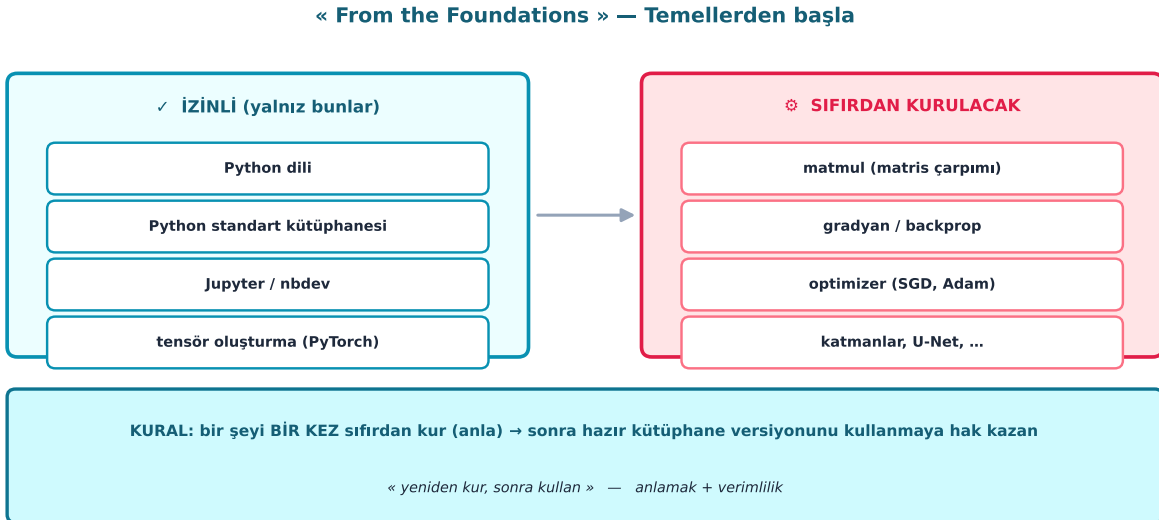
- **İleriye (Ders 19):** Bu loop, Part 2 Ders 19-22’de sıfırdan (DDPM/DDIM) kurulacak. Her adım **Ders 9B’nin** reverse process’inin bir basamağıdır.
- **Sezgi:** “Tahmin et, çıkar, tekrarla” üç kelimelik özet; bütün üretim bundan ibaret — ve sonunda VAE latent’i piksele döker.

19.9 8. « From the Foundations » Felsefesi

Dönüm noktası. Howard hedefi koyar: Stable Diffusion’a **sıfırdan** varmak. Ama “sıfırdan” ne demek? Tanımlar: yalnızca Python, Python standart kütüphanesi, Jupyter/nbdev ve tensör oluşturma iznli. Geri kalan her şey (matmul, gradyan, optimizer, katmanlar, U-Net) elle kurulacak.

“The goal is to get to Stable Diffusion from the foundations, which means we have to define what are the foundations. We’re allowed to use Python, the Python standard library, Jupyter notebooks and nbdev. We’re going to rebuild everything starting from this foundation.” — Howard, 53:47

Şekil 19.4 bu kuralı şematize eder: solda **İZİNLİ** (yalnız Python + stdlib + Jupyter/nbdev + tensör oluşturma), sağda **SIFIRDAN KURULACAK** (matmul, gradyan/backprop, optimizer, katmanlar/U-Net); altta kural bandı: bir şeyi bir kez kur (anla) → sonra hazır kütüphane versiyonunu kullanmaya hak kazan.



Şekil 19.4: (şematik: from-the-foundations kuralı — yalnız Python+tensör iznli; her parçayı bir kez sıfırdan kur, sonra hazırı kullan)

💡 Builder Notu — Top-Down Kursun En Bottom-Up Anı

- **Geriye (Karpathy):** Bu, Karpathy’nin “from scratch” kuralının ta kendisi; fast.ai top-down kursunun en bottom-up bölümü.
- **Sezgi:** “İzin verilenler” listesi kasıtlı olarak minimaldir — amaç, üst katmanların altında ne olduğunu görmezden gelmemek; her soyutlamayı bir kez elimizle kurmak.

19.10 9. Kuralın İncelikleri: Yeniden Kur, Sonra Kullan

Howard kuralı netleştirir: bir şeyi **bir kez sıfırdan kurunca**, sonrasında hazır (kütüphane) versiyonunu kullanmaya hak kazanırsınız. Yani amaç her şeyi sürekli elle yazmak değil; her parçanın altında ne olduğunu **bir kez** anlamak.

“Once we’ve reimplemented something, we’ll then be allowed to use the big pretrained versions.”
— Howard, 53:47

💡 Builder Notu — Anlamak + Verimlilik

- **İleriye:** “Bir kez sıfırdan kur, sonra kütüphaneyi güvenle kullan” — fast.ai’nin tüm pedagojisinin özeti; anlamak + verimlilik.
- **Sezgi:** Matmul’u bir kez elle yazıp anladıktan sonra PyTorch’un @’ini güvenle kullanırsın — çünkü artık altında ne olduğunu biliyorsun, “kara kutu” değil.

19.11 10. MNIST Veri Seti

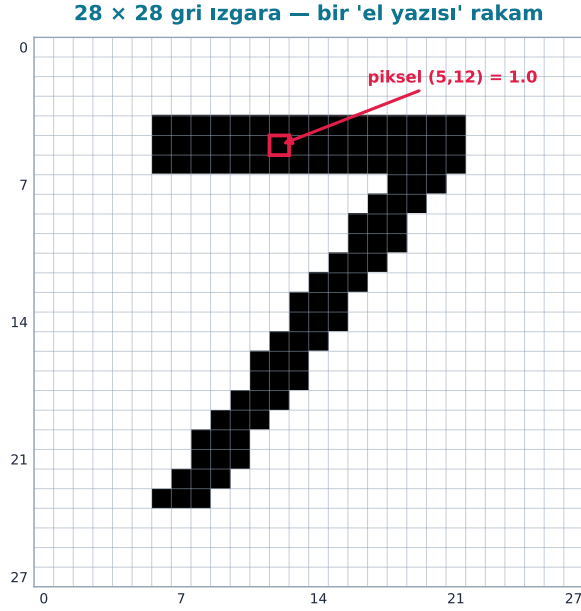
İlk foundation çalışması için klasik veri: **MNIST** — 28×28 piksel, gri tonlamalı el yazısı rakamlar. Howard veriyi yükler ve tensöre çevirir (izin verilen tek şey: Python + tensor). Her görüntü $28 \times 28 = 784$ sayıdır — sıfırdan matmul/model kurmaya tam ölçek: gerçek ama yeterince küçük.

“It’s a classic dataset. They’re 28 by 28 pixel, grayscale images.” — Howard, 57:09

Şekil 19.5 bu veriyi gösterir: solda 28×28 gri ızgara (bir “el yazısı” rakam, bir piksel değeri işaretli), sağda büyük “ $28 \times 28 = 784$ ” hesabı ve foundation notu — Ders 9’un “el yazısı rakam” sezgisinin gerçek verisi.

💡 Builder Notu — 784 Sayı: Tam Ölçek

- **Geriye (Ders 9/9B):** Ders 9’un “el yazısı rakam” örneği gerçek veriye dönüşüyor; $28 \times 28 = 784$ sayı (foundation hesapları için yeterince küçük).
- **Sezgi:** MNIST yeterince küçük olduğu için matmul’u Python döngüsüyle bile çalıştırıp ölçekbiliriz — yavaşlığı gözle görmenin de ideal boyutu.



$$28 \times 28 = 784$$

sayı (piksel)

her piksel 0-1 arası bir gri yoğunluk

foundation hesapları için yeterince küçük

(Ders 9'un 'el yazısı rakam' verisi — sıfırdan matmul/model kurmaya tam ölçek)

Şekil 19.5: (illüstratif: MNIST 28×28 gri rakam = 784 sayı; foundation çalışması için klasik küçük veri)

19.12 11. Matris Çarpımı: İlk Foundation

İlk yeniden kurulacak şey **matris çarpımı** (matmul) — sinir ağının kalbi. Bir görüntü batch'ini ($m1$) ağırlıklarla ($m2$) çarpmak gerekir. Howard çarpımın tanımıyla başlar: sonuç matrisinin her (i,j) hücresi, $m1$ 'in i . satırı ile $m2$ 'nin j . sütununun dot product'ıdır.

“We’re going to take a deep dive into matrix multiplication today.” — Howard, 53:40

💡 Builder Notu — matmul = @ Operatörünün İçi

- **Geriye (Ders 5/18.06):** matmul, Ders 5'teki @ operatörünün içi; 18.06'nın temel işlemi. Şimdi onu sıfırdan kuruyoruz.
- **Sezgi:** “Her hücre = bir satır × bir sütun dot product” tek cümlelik tanımdır; bütün matris çarpımı bu tanımın her (i,j) için tekrarıdır.

19.13 12. Üç İç İç Döngüyle Matmul

En yalın hâl: üç iç içe döngü. Dış iki döngü çıktı hücrelerini (i, j) , iç döngü dot product'ı (k) gezer.

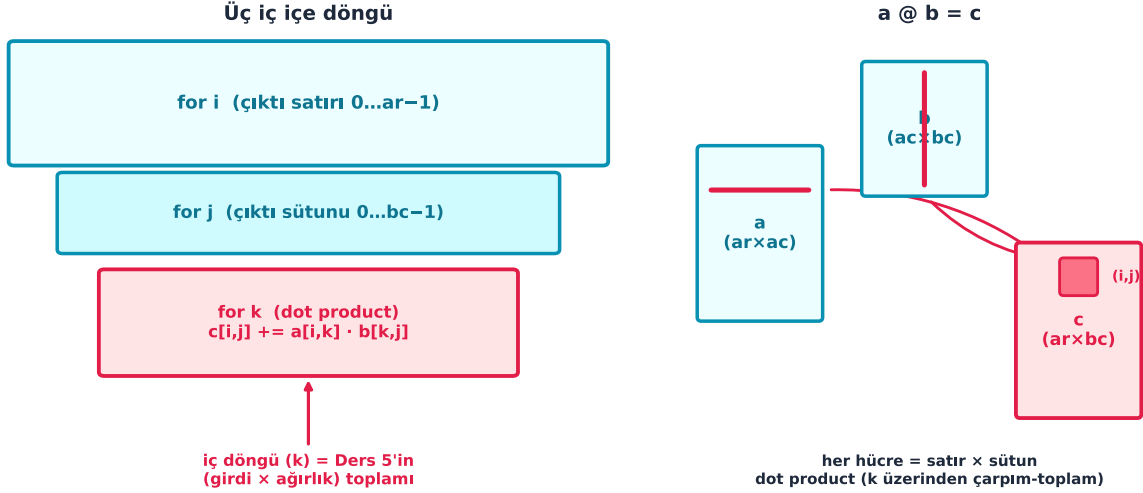
```
def matmul(a, b):
    (ar, ac), (br, bc) = a.shape, b.shape    # satir/sutun sayilari
    c = torch.zeros(ar, bc)
    for i in range(ar):
        for j in range(bc):
            for k in range(ac):
```

```

    c[i,j] += a[i,k] * b[k,j]    # dot product birikimi
return c

```

Şekil 19.6 bu üç döngüyü görselleştirir: solda iç içe kutular (i sarar j, j sarar k; en içteki $c[i,j] += a[i,k] \cdot b[k,j]$), sağda küçük bir matris çarpımı şeması — a’nın i. satırı ile b’nin j. sütununun c’nin (i,j) hücresine nasıl aktığı.



Şekil 19.6: (şematik: üç iç içe döngü matmul — i satır, j sütun, k dot product; her hücre bir satır×sütun çarpım-toplamı)

💡 Builder Notu — İç Döngü = Ders 5'in Toplamı

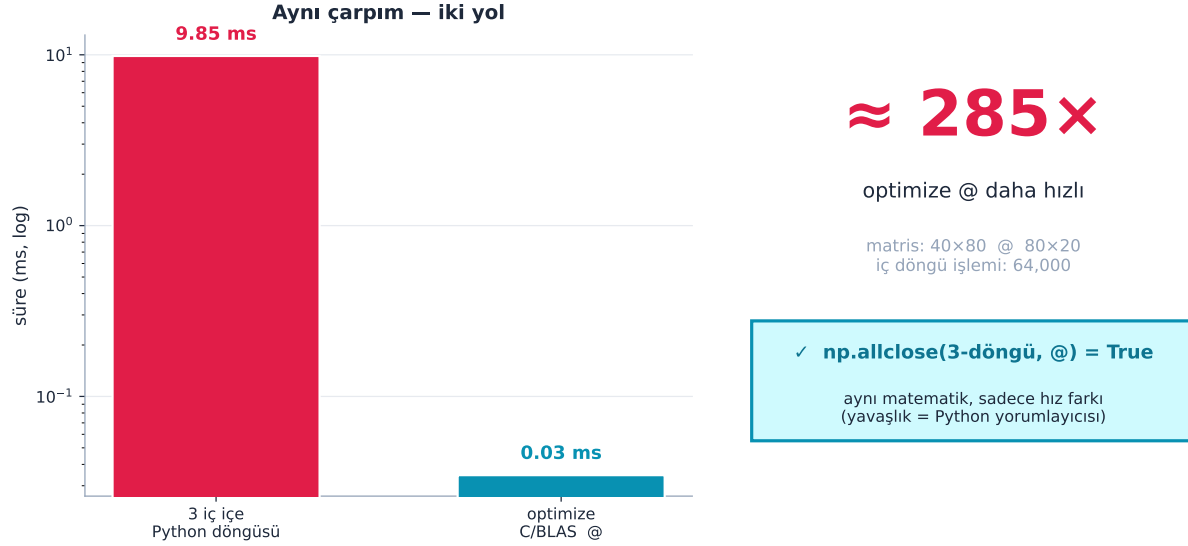
- **Geriye (Ders 5):** İç döngü (k) tam Ders 5'teki “(girdi × ağırlık) topla” işlemi; burada her hücre için tekrarlanıyor.
- **Sezgi:** Dış iki döngü “hangi çıktı hücresi”, iç döngü “o hücrenin değeri (satır×sütun çarpım-toplamı)” — matematiksel tanımın birebir koda dökülmüş hâli.

19.14 13. Python Döngüsü Çok Yavaş

Bu çalışır ama korkunç yavaştır: MNIST boyutunda matmul Python döngüsüyle saniyeler sürer ($5 \times 784 \times 784 \times 10 =$ milyonlarca işlem, her biri yorumlanan Python). Howard `%timeit` ile ölçer — bir sinir ağı için kabul edilemez. Yavaşlığın tek sebebi Python’un **yorumlanan** bir dil olması: her $c[i,j] += a[i,k] * b[k,j]$ işlemi tek tek, yavaş yorumlayıcı üzerinden çalışır.

“*[This is] really slow.*” — Howard, ~58:00 (matmul timing)

Şekil 19.7 bu yavaşlığı **gerçek bir benchmark** ile gösterir: aynı çarpımı bir kez üç iç içe Python döngüsüyle, bir kez de optimize (C/BLAS) @ ile ölçer; ikisinin sonucu `np.allclose` ile aynıdır ama süreler arasında yüzlerce kat fark vardır.



Şekil 19.7: (GERÇEK canlı benchmark: aynı matris çarpımı üç iç içe Python döngüsüyle (yavaş, yorumlanır) vs optimize C/BLAS @ ile (hızlı); sonuçlar np.allclose ile özdeş, süre farkı yüzlerce kat)

💡 Builder Notu — Yavaşlığın Kaynağı: Yorumlayıcı

- **İleriye:** Yavaşlığın sebebi Python yorumlayıcısının her işlemi tek tek yapması; çözüm, işi optimize edilmiş (C/GPU) koda devretmek — sonraki adımlar ([numba](#), [broadcasting](#)).
- **Builder ipucu:** Önemli olan benchmark'ın iki yanının **aynı sonucu** vermesi (np.allclose ✓); hız kazanırken doğruluğu kaybetmediğimizi her zaman doğrula.

19.15 14. Numba ile Hızlandırma

İlk hızlandırma: **Numba**.@njit dekoratörü bir Python fonksiyonunu (örn. dot product) makine koduna derler; iç döngü artık C hızında çalışır. Numba'nın güzelliği şudur: Python'u Python gibi yazmaya devam edersin, ama dekoratör fonksiyonu çalışma anında (JIT) derler.

```
from numba import njit
@njit
def dot(a, b):
    res = 0.
    for i in range(len(a)): res += a[i] * b[i]
    return res
# matmul artık en iç döngü yerine derlenmiş dot() kullanır -> çok daha hızlı
```

💡 Builder Notu — @njit: JIT ile C Hızı

- **İleriye (Ders 11):** Numba bir ara adım; asıl hız broadcasting ve @ (PyTorch'un GPU GEMM'i) ile gelir — Ders 11'in konusu.

- **Sezgi:** @njit ilk çağrıda derler (küçük gecikme), sonraki çağrılarda C hızında çalışır; iç döngünün yorumlama yükünü tamamen kaldırır.

19.16 15. İleriye: Broadcasting

Howard sonraki adımı işaret eder: döngüleri tamamen kaldırıp **broadcasting** ve tensör işlemleriyle matmul yapmak (Ders 5'te gördüğümüz broadcasting'in derinleşmesi). Bu, hem çok daha kısa hem optimize C/GPU koduna düşen versiyondur — döngü yazmadan, tek bir tensör ifadesiyle tüm satırı işlemek.

💡 Builder Notu — Broadcasting: Döngüsüz Tensör

- **Geriye (Ders 5/6):** Broadcasting (Ders 5) burada matmul'u hızlandırmak için tam gücüyle kullanılacak (Ders 11).
- **İleriye (Ders 11):** $a[i, :] * b$ gibi bir ifade tüm satırı tek seferde çarpar; döngüleri kaldırmak hem hızı hem okunabilirliği artırır.

19.17 16. Kapanış

Ders 10, Part 2'nin dönüm noktasıydı: Stable Diffusion'ı (distillation + tam pipeline) bitirdik ve « **from the foundations** » yolculuğunu başlattık. Kural net: yalnızca Python + standart kütüphane + tensör; her şeyi sıfırdan kur, bir kez kurunca hazır kullan. İlk foundation matris çarpımıydı — üç döngüden numba'ya, oradan (Ders 11'de) broadcasting'e.

“We'll be building our own kind of framework from scratch.” — Howard, 50:33

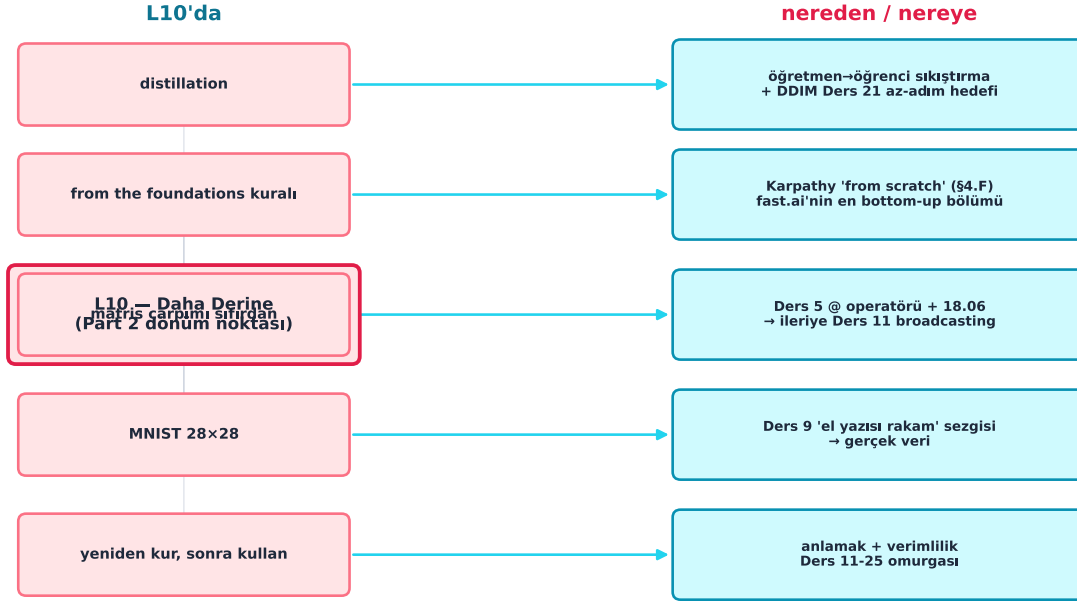
Şekil 19.8 dersin sentezidir: solda L10'un parçaları (distillation, from-the-foundations kuralı, matris çarpımı sıfırdan, MNIST 28×28, yeniden kur sonra kullan), sağda her birinin nereden/nereye köprüsü — DDIM Ders 21 az-adım hedefi, Karpathy from-scratch (§4.F), Ders 5 @ + 18.06 → Ders 11 broadcasting, Ders 9 sezgisi, Ders 11-25 omurgası.

💡 Builder Notu — Foundations Zinciri Açılıyor

- **İleriye (Ders 11+):** Sıradaki dersler matmul'u tamamlar (broadcasting/Einstein toplamı), sonra backprop, Learner ve nihayet diffusion'ı sıfırdan kurar — Karpathy ruhunun fast.ai'deki en yoğun hâli.
- **Tek cümle:** Bu ders bir kapı; arkasında Stable Diffusion'a kadar uzanan, her parçası elle kurulan bir foundations yolculuğu var.

19.18 Bu Dersin Özeti

1. Part 2 Ders 10 bir dönüm noktasıdır: Stable Diffusion'ı bitirir, « from the foundations » omurgasını başlatır (dönüm).



Şekil 19.8: (sentez: L10 = SD'yi bitir + foundations'a in; Karpathy bottom-up'ın fast.ai'deki başlangıcı) (şematik)

2. **Distillation** ile bir öğretmen modelin adımları, öğrenci modelin daha az adımına sıkıştırılır; progressive distillation'da her turda adım yarıya iner ([distillation](#), [progressive](#)).
3. SD pipeline kod olarak tekrar gezilir: CLIP → latent gürültü → U-Net+CFG döngüsü → scheduler → VAE decode; başlangıç gürültüsü doğru varyansa (`init_noise_sigma`) ölçeklenir ([pipeline](#), [ölçekleme](#)).
4. « **From the foundations** » kuralı: yalnızca Python + standart kütüphane + tensör izinli; geri kalan sıfırdan kurulur ([felsefe](#)).
5. Kural inceliği: bir şeyi **bir kez** sıfırdan kurunca, hazır versiyonunu kullanmaya hak kazanırsın ([incelik](#)).
6. İlk foundation **matris çarpımı**; sonucun her hücresi bir satır × sütun dot product'ıdır ([matmul](#)).
7. En yalın matmul **üç iç içe döngüdür** ama Python'da korkunç yavaştır (milyonlarca yorumlanan işlem) ([üç döngü](#), [yavaşlık](#)).
8. **Numba** (([njit?](#))) iç döngüyü makine koduna derleyerek hızlandırır; asıl hız (broadcasting/GPU) Ders 11'de gelir ([numba](#), [broadcasting](#)).

! Tek Bir Cümle

Ders 10, Stable Diffusion'ı kullanmayı bitirip onu sıfırdan kurmaya başladığımız andır: yalnızca Python ve tensörle başlayıp, ilk foundation olan matris çarpımını üç döngüyle kurar, sonra adım adım hızlandırırız — çünkü bir şeyi bir kez sıfırdan kurmak, onu gerçekten anlamaktır.

i Soru 1: « From the foundations » kuralı tam olarak nedir? Neden bir kez kurduktan sonra hazır versiyonu kullanmaya izin var?

19.19 Kontrol Soruları

Kural: yalnızca **Python, Python standart kütüphanesi, Jupyter/nbdev ve tensör oluşturma** kullanılabilir; matmul, gradyan, optimizör, katmanlar, U-Net dahil her şey sıfırdan kurulur. Amaç her parçanın altında ne olduğunu anlamaktır. “Bir kez kurduktan sonra hazır kullan” izni şunun içindir: hedef sürekli elle kod yazmak değil, mekanizmayı **bir kez** kavramaktır. Matmul’u bir kez sıfırdan yazıp anladıktan sonra, sonraki derslerde PyTorch’un @’ini güvenle kullanırsın — çünkü artık altında ne olduğunu biliyorsun. Bu, anlamak ile verimliliği birleştirir. (Şekil 19.4 bu kuralı şematize eder.)

i Soru 2: Üç iç içe döngülü matmul ne yapar? Hangi döngü neyi gezer?

Cevap:

Sonuç matrisinin her hücre (i,j), m1’in i. satırı ile m2’nin j. sütununun dot product’ı olarak hesaplar. Dış döngü (i) çıktının satırlarını, ortadaki döngü (j) çıktının sütunlarını gezer; en içteki döngü (k) ise o hücre için dot product’ı biriktirir: $c[i, j] += a[i, k] * b[k, j]$. Yani dış iki döngü “hangi çıktı hücresi”, iç döngü “o hücrenin değeri (satır×sütun çarpım toplamı)”. Bu, matris çarpımının matematiksel tanımının birebir koda dökülmüş hâlidir. (Şekil 19.6 bu üç döngüyü görselleştirir.)

i Soru 3: Python döngülü matmul neden bu kadar yavaştır? Numba bunu nasıl hızlandırır?

Cevap:

Python **yorumlanan** bir dildir: her $c[i, j] += a[i, k] * b[k, j]$ işlemi tek tek, yavaş yorumlayıcı üzerinden çalışır. MNIST boyutunda (5×784 ile 784×10) bu milyonlarca işlem demektir ve saniyeler sürer — bir sinir ağı için kabul edilemez. **Numba**’nın @jit dekoratörü, işaretlenen fonksiyonu (örn. dot product) çalışma anında **makine koduna derler** (JIT); böylece iç döngü artık Python yorumlayıcısı yerine C hızında çalışır. Numba bir ara adımdır; asıl hız broadcasting ve GPU matris çarpımıyla (Ders 11) gelir. (Şekil 19.7 döngü vs optimize @ farkını gerçek benchmark’la gösterir.)

i Soru 4: Distillation diffusion’ı nasıl hızlandırır? (builder bağlantısı)

Cevap:

Diffusion yavaştır çünkü üretim çok sayıda (örn. 1000) sampling adımı gerektirir. **Distillation**, iyi ama yavaş bir **öğretmen** modeli, daha az adımda aynı sonucu veren bir **öğrenci** modele “öğretir”. Progressive distillation’da öğrenci, öğretmenin **iki adımını tek adımda** yapmayı öğrenir; sonra bu öğrenci yeni öğretmen olur ve süreç tekrarlanır — her turda gereken adım sayısı yarıya iner. Builder açısından: distillation, büyük/yavaş bir modelin bilgisini küçük/hızlı bir modele aktarmanın genel bir yöntemidir (sadece diffusion değil; LLM’lerde de yaygın) — kalite/hız dengesini production için optimize eder. (Şekil 19.3 1000 → 7 sıkışmasını gerçek aritmetikte gösterir.)

19.20 Egzersizler

Egzersiz 1 (Direkt uygulama). MNIST'i yükle (28×28, tensöre çevir) ve üç iç içe döngülü matmul'u kendin yaz; bir batch ile ağırlık matrisini çarp. (§12)

Egzersiz 2 (İki-aşamalı). %timeit ile döngülü matmul'un süresini ölç; sonra Numba @njit'li dot product ile hızlandır ve süreyi kıyasla. (Şekil 19.7'i kendi sayınla yeniden üret — §13-14)

Egzersiz 3 (Edge case). matmul'da boyut uyumsuzluğu (m_1 'in sütunu $\neq$ m_2 'nin satırı) durumunda ne olduğunu gözle; neden $ac = br$ olmak zorunda açıkla.

Egzersiz 4 (Kavramsal). « From the foundations » kuralına göre hangi araçlar izinli, hangileri değil? Bir matmul'u neden NumPy ile değil elle yazıyoruz? (§8)

Egzersiz 5 (Sonraki dersin habercisi — Ders 11). Üç döngüyü broadcasting ile nasıl kaldırabileceğini araştır; $a[i, :] * b$ gibi bir ifadenin tüm satırı tek seferde nasıl çarptığını düşün. (§15)

19.21 Sonraki: Ders 11 İçin Hazırlık

Ders 11: Matris Çarpımı (Matrix multiplication)

Ders 10 matmul'u üç döngüyle kurup numba ile hızlandırdı. Ders 11 işi tamamlar: **broadcasting** ve **Einstein toplamı** ile döngüleri tamamen kaldırıp matmul'u optimize (ve sonunda GPU) hıza çıkarmak. Foundations yolculuğunun ilk tam adımı.

Ana konular (Ders 11):

- Broadcasting ile matmul (döngüsüz)
- Broadcasting kuralları (derinlemesine)
- Einstein toplamı (einsum)
- @ ve GPU hızı

⚠ Ders 11 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 2 — matmul'u kur ve numba ile hızlandır).
- « From the foundations » kuralını kendi cümlelerinle yaz.
- Ana cümleyi tekrar oku: “Bir şeyi bir kez sıfırdan kurmak, onu gerçekten anlamaktır.”

19.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
From the foundations	Yalnızca Python + standart kütüphane + tensör; gerisi sıfırdan	53:47
Distillation	Öğretmen modelin bilgisini hızlı öğrenci modele aktarma	16:56

Kavram	Tanım	Howard'da
Progressive distillation	Her turda adım sayısını yarıya indirme (1000 → 7)	19:36
init_noise_sigma	Başlangıç gürültüsünü doğru varyansa ölçekleme	41:29
SD pipeline	CLIP → latent → U-Net+CFG döngüsü → scheduler → VAE	39:36
MNIST	28×28 gri el yazısı rakam (784 sayı); klasik veri	57:09
Matris çarpımı (matmul)	Her hücre = satır × sütun dot product	53:40
Üç iç içe döngü	i (satır), j (sütun), k (dot product) gezen yalın matmul	53:40
Python yavaşlığı	Yorumlanan döngü; milyonlarca işlem yavaş	58:00
Numba ((njit?))	Python fonksiyonunu makine koduna derleme (JIT)	58:00
Yeniden kur, sonra kullan	Bir kez sıfırdan kur, sonra hazır versiyon serbest	53:47
Broadcasting (ileriye)	Döngüsüz tensör işlemi; Ders 11'de matmul hızı	50:33

19.23 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: L10'un Tanıdık Kökleri

Bu ders, Stable Diffusion'ı bitirip foundations'a inerken her parçayı tanıdık kavramlara bağlar; köprülerin özeti:

1. **From the foundations** → Karpathy “from scratch” felsefesi; anlamak için bir kez elle kur ([felsefe](#)).
2. **Distillation** → büyük/yavaş modeli küçük/hızlıya sıkıştırma; diffusion ve LLM'lerde production standardı ([distillation](#)).
3. **Matris çarpımı** → sinir ağının kalbi (Ders 5 @, 18.06); her katman bir matmul ([matmul](#)).
4. **Python yavaşlığı** → **Numba/GPU** → sayısal hesabın optimize koda devri; broadcasting (Ders 11) ([yavaşlık](#), [numba](#)).
5. **init_noise_sigma** → reparameterization ([Ders 9B](#)); doğru ölçek üretimi belirler ([ölçekleme](#)).
6. **MNIST 28×28** → foundation hesapları için yeterince küçük; Ders 9'un “el yazısı rakam” sezgisinin verisi ([MNIST](#)).

❗ Bu dersten tek bir şey alıp gideceksen

Part 2'nin asıl yolculuğu burada başlar. Stable Diffusion'ı kullanmayı öğrendik; şimdi onu yalnızca Python ve tensörle, parça parça, sıfırdan kuracağız. İlk parça matris çarpımı — üç basit döngü, sonra hızlandırma. Howard'ın kuralı net: bir şeyi bir kez sıfırdan kur, gerçekten anla, sonra kütüphaneyi

güvenle kullan. Karpathy'nin ruhu, fast.ai'nin omurgasında.

20 Ders 11 — Matris Çarpımı (Matrix multiplication)

Part 2'nin üçüncü dersi iki beceri verir. İlk yarı bir araştırma becerisi öğretir: akademik makale (DiffEdit) okuma iş akışı — arXiv'den preprint, Zotero ile not. İkinci yarı « from the foundations »a döner ve Ders 10'un yavaş üç-döngü matris çarpımını broadcasting ile döngüsüz, yüzlerce kat hızlı hâle getirir. Anahtar fikir tek cümleyle: bir döngüyü kaldırmanın yolu broadcasting'dir — iş artık optimize C/GPU koduna devredilir.

i Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 11: Matrix multiplication](https://course.fast.ai/lesson-11) (~108 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 11
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/01_matmul](#)
- **Okuma süresi:** ~38 dk
- **İki kol:** İlk yarı **araştırma becerisi** (arXiv → Zotero → DiffEdit), ikinci yarı « **from the foundations** » (matmul'u broadcasting ile döngüsüz hızlandır). Bu ders ETAP 4 (Temeller A) başıdır.

20.1 Bu Derste Ne Var?

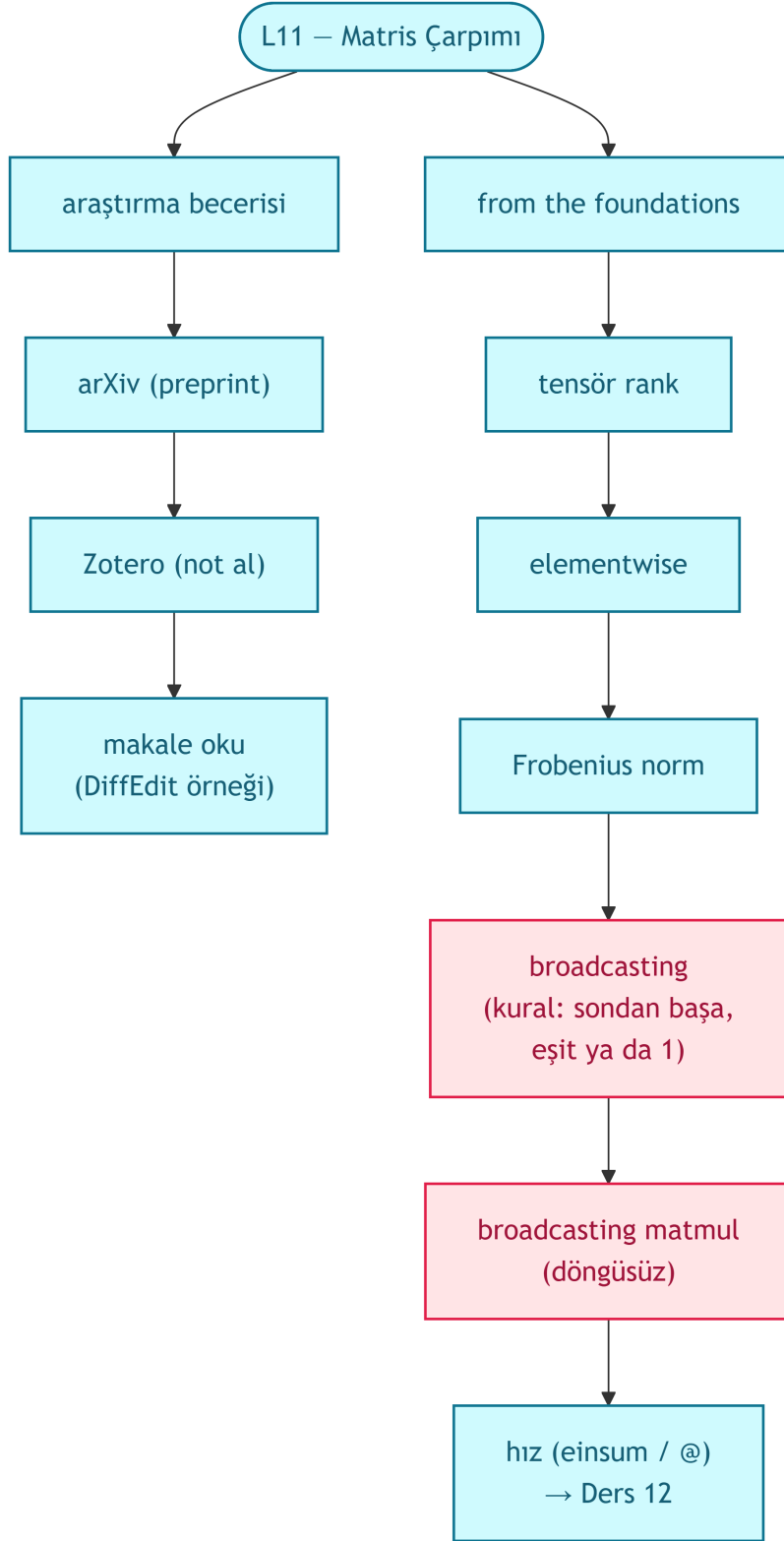
Part 2'nin üçüncü dersi iki şey yapar. İlk yarı bir **araştırma becerisi** verir: akademik makale ([DiffEdit](#)) okuma iş akışı (arXiv + Zotero). İkinci yarı “from the foundations”a döner ([dönüş](#)) ve matris çarpımını **broadcasting** ile döngüsüz, hızlı hâle getirir.

Üç temel fikir bu dersin omurgasını kurar:

1. **Makale okuma iş akışı** — arXiv'den preprint, Zotero ile not alma; DiffEdit gibi bir makaleyi adım adım anlama ve uygulama ([makale akışı](#)).
2. **Broadcasting** — farklı şekilli tensörleri otomatik hizalayıp döngüsüz işleme; matris çarpımını Python döngüsünden kurtarmanın anahtarı ([broadcasting](#) → [kurallar](#)).
3. **Döngüsüz matmul** — üç iç içe döngü yerine broadcasting ile tek döngü; optimize C/GPU koduna düşen, çok daha hızlı versiyon ([broadcasting matmul](#) → [hız](#)).

“Okay, so let's move on with our *From the Foundations* now.” — Howard, 58:52

Şekil [34.1](#) bu yapıyı tek bir yol haritasında birleştirir: solda araştırma kolu (arXiv → Zotero → makale), sağda “from the foundations” temel zinciri (tensör rank → elementwise → Frobenius → broadcasting → broadcasting matmul → hız). İki kol L11'de birleşir.



Şekil 20.1: Ders 11’in iki becerisi: solda araştırma akışı (arXiv → Zotero → makale), sağda ‘from the foundations’ temel zinciri (tensör → elementwise → Frobenius → broadcasting → broadcasting matmul → hız). İki kol L11’de birleşir. (şematik)

 Builder Notu — İki Beceri, Tek Ders

- **Geriye (Ders 5/10/18.06):** Broadcasting Ders 5'te (Titanic) tanıtılmıştı; Ders 10'daki üç-döngü matmul'u burada broadcasting ile hızlandırıyoruz. matmul = 18.06'nın kalbi.
- **İleriye (Ders 12 / NYU §4.J):** einsum ve @ (Ders 12) matmul'u tamamlar; broadcasting tüm tensör hesabının (NYU dahil) verimlilik temeli.
- **Tek cümle:** Broadcasting, döngüleri kaldırıp tensör işlemlerini optimize koda devretmenin yoludur; matris çarpımı bunun ilk büyük kazanımı.

20.2 1. Ders 10 Özeti ve Öğrenci İşleri

Howard haftanın öğrenci projeleriyle başlar ve bir SD tekniğini hatırlatır: guidance update'i, normların oranıyla ölçeklemek (çok büyük güncelleme adımı “çok ileri zıplar”). Ardından Ders 10'un “from the foundations” başlangıcını (Ders 10) özetler.

“It's the third lesson in Part 2.” — Howard, 0:00

 Builder Notu — Köprü: SD'den Foundations'a

- **Geriye (Ders 9A):** guidance ölçekleme, CFG'nin pratik bir ince ayarı; “norm oranı” sezgisi büyük adımları frenler.
- **Sezgi:** Bu özet bir köprüdür — önceki dersin SD detaylarını kapatıp temel zincire (matmul) geçişi hazırlar.

20.3 2. DiffEdit: Diffusion ile Akıllı Düzenleme

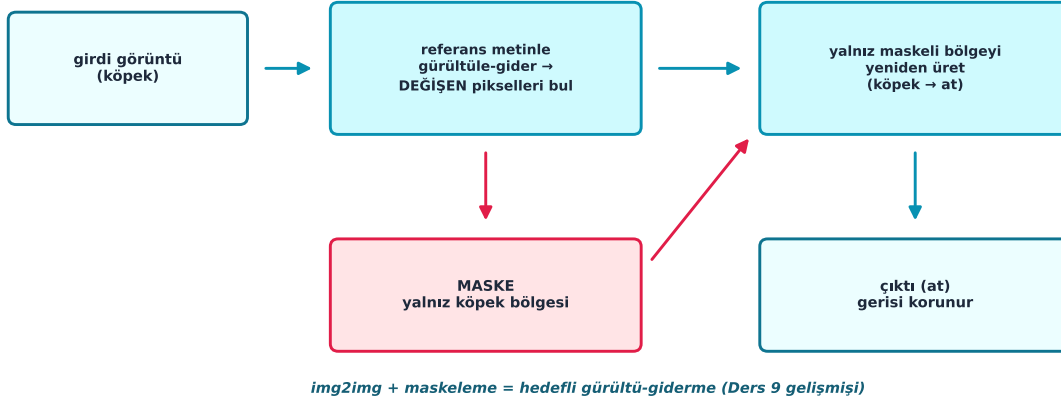
Howard bir makale örneği seçer: **DiffEdit** — bir görüntüde yalnızca prompt'un işaret ettiği bölgeyi (örn. “köpek”i “at”a) değiştiren, gerisini koruyan bir yöntem. Bir referans metinle gürültü ekleyip-çıkarak hangi piksellerin değişeceğini bulur (maske), sonra yalnızca orayı yeniden üretir.

“We add noise to it, and then we denoise it using the reference text.” — Howard, 46:36

Şekil 20.2 bu üç adımı şematik gösterir: girdi görüntü (köpek) → referans metinle gürültü-gider ve **değişen pikselleri** bul → bu farktan bir **maske** çıkar (yalnız köpek bölgesi) → yalnız maskeli bölgeyi yeniden üret (köpek → at) → çıktıda gerisi korunur.

 Builder Notu — DiffEdit: Hedefli Gürültü-Giderme

- **Geriye (Ders 9/9A):** img2img + maskeleme; DiffEdit, gürültüleme-giderme döngüsünün **hedefli** kullanımıdır — maske, değişimi yalnızca istenen bölgeye sınırlar.
- **Sezgi:** Maske, “referans metinle gürültüleyip-giderince hangi pikseller değişiyor?” sorusunun cevabıdır; değişen yerler düzenlenecek bölgedir.



Şekil 20.2: (şematik: DiffEdit — referans metinle maske bul, yalnız o bölgeyi yeniden üret; gerisini koru)

20.4 3. Akademik Makale Okuma İş Akışı

Howard değerli bir meta-beceri öğretir: makale okuma. Çoğu derin öğrenme bilgisi **arXiv** (preprint sunucusu) üzerindendir; hakem değerlendirmesini beklemek çok yavaştır. Howard’ın iş akışı: arXiv’den PDF → **Zotero** ile kaydet, üzerine not al, anla.

“arXiv is a preprint server... we use arXiv a lot. If you wait until it’s been peer reviewed, you’ll [wait too long].” — Howard, 13:22

💡 Builder Notu — Makale Okuma: Alanın Ucunda Kalmak

- **İleriye (builder):** Makale okuyabilmek, alandaki en yeni tekniklere erişimin tek yolu; hakemli yayını beklemek aylar/yıllar alır.
- **Araç:** Zotero gibi bir araçla sistemli okuma (kaydet + üzerine not al) kalıcı bilgi sağlar — “tüketici”den “üretici”ye geçişin anahtarı.

20.5 4. “From the Foundations”a Dönüş

Mola sonrası Howard foundations’a döner: matris çarpımını sıfırdan, ama bu kez **hızlı** kurmak. Ders 10’da üç döngü vardı (yavaş); şimdi döngüleri broadcasting ile kaldıracağız.

“Okay, so let’s move on with our From the Foundations now. Matrix multiplication from scratch.” — Howard, 58:52

💡 Builder Notu — Aynı matmul, Yeni Hedef: Hız

- **Geriye (Ders 10):** Aynı matmul; hedef yorumlanan Python döngülerini optimize tensör işlemlerine çevirmek.

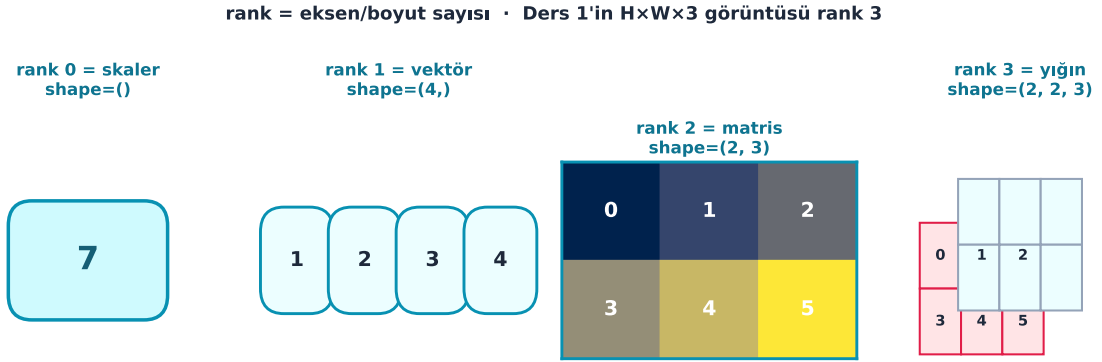
- **Sezgi:** “Sıfırdan kur” kuralı geçerli — ama bu kez amaç doğruluğu değil, **performansı** sıfırdan kazanmak; iç döngüleri tensör ifadelerine devretmek.

20.6 5. Tensör Rank'ı

Önce terminoloji: bir tensörün **rank**'ı boyut sayısıdır. Rank 0 = skaler, rank 1 = vektör, rank 2 = matris (liste listesi), rank 3 = matrisler yığını. Broadcasting'i anlamak için rank ve şekil (shape) kavramı kritiktir.

“This here is a rank one tensor — it's a vector. Whereas a rank two tensor is like a list of lists.”
— Howard, 1:01:13

Şekil 20.3 bu dört rank'ı yan yana gösterir: skaler (rank 0, tek sayı), vektör (rank 1, 1B dizi), matris (rank 2, 2B ızgara), yığın (rank 3, üst üste matrisler). Her panelin başlığında gerçek shape yazılıdır — Ders 1'in $H \times W \times 3$ görüntüsü rank 3'tür.



Şekil 20.3: Tensör rank = eksen (boyut) sayısı: skaler 0 (tek sayı), vektör 1 (1B dizi), matris 2 (2B ızgara), yığın 3 (üst üste matrisler). (gerçek: tensör rank = eksen sayısı; skaler 0 / vektör 1 / matris 2 / yığın 3)

💡 Builder Notu — Rank = Eksen Sayısı

- **Geriye (Ders 1):** Ders 1'deki “görüntü = $H \times W \times 3$ tensör” rank 3'tür; rank = eksen sayısı, shape = her eksenin uzunluğu.
- **Sezgi:** Broadcasting tam olarak shape üstünde çalışır — hangi eksenlerin “eşit ya da 1” olduğunu rank ve shape belirler; bu yüzden önce bunlar tanımlanır.

20.7 6. Elementwise İşlemler

Aynı şekilli iki tensörde toplama, çarpma, karşılaştırma **elementwise** (eleman eleman) çalışır: $a + b$, $a * b$, $a > b$ her konumu ayrı işler. Howard bunun optimize C kodunda paralel yapıldığını vurgular — Python döngüsü değil.

```

a + b    # her konumu ayrı topla
a * b    # her konumu ayrı çarp (NOT: matris çarpımı DEĞİL)
a > b    # her konumu ayrı karşılaştır -> bool tensör

```

💡 Builder Notu — Elementwise: Tensör İşlemlerinin Temeli

- **Geriye (Ders 5):** Ders 5'teki (`indep * coeffs`) elementwise çarpımdı; tensör işlemlerinin temeli.
- **Sezgi:** `a * b` elementwise çarpımdır, dot product değil — paralel C koduna düştüğü için Python döngüsünden çok hızlıdır; broadcasting bunu **farklı** şekillere genişletir.

20.8 7. Frobenius Norm

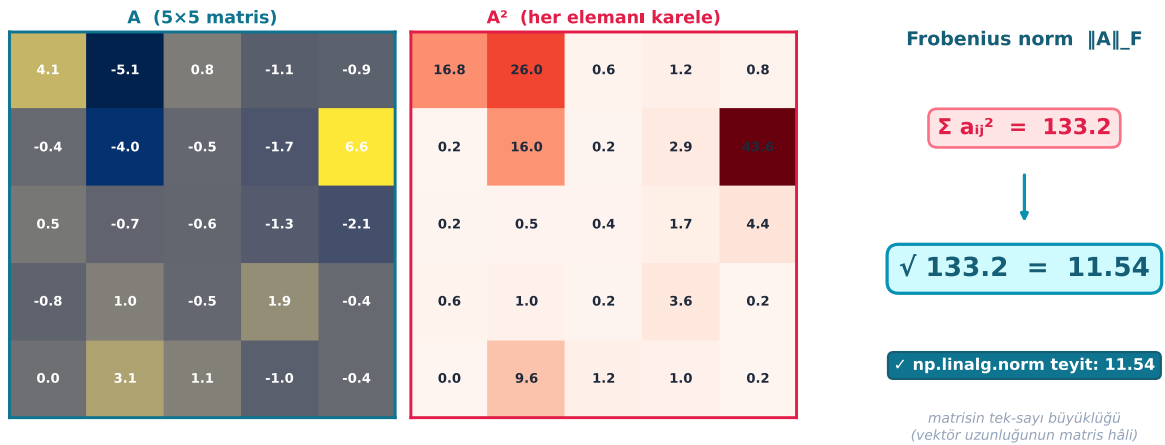
Howard üretken modellemede sık görülecek bir kavramı tanıtır: **Frobenius norm** — bir matrisin tüm elemanlarının karelerinin toplamının karekökü:

$$\|A\|_F = \sqrt{\sum_{i,j} a_{ij}^2}$$

Korkutucu görünen sembol, aslında basit bir “elemanları karele, topla, karekök al” işlemidir.

“We’re going to learn about a new thing that looks kind of crazy called Frobenius norm... we’ll use that from time to time as we’re doing generative modeling.” — Howard, 1:19:58

Şekil 20.4 bunu **gerçek hesaplama** ile gösterir: solda matris A , ortada her elemanı karelenmiş A^2 , sağda $\sum a_{ij}^2$ toplamı ve karekökü; sonucu `np.linalg.norm` ile teyit edilir.



Şekil 20.4: (gerçek hesaplama: Frobenius $\|A\|_F = \sqrt{\sum a_{ij}^2}$; elemanları karele→topla→karekök, `np.linalg.norm` ile teyit)

💡 Builder Notu — Frobenius: Matrisin Tek-Sayı Büyüklüğü

- **Geriye (18.06 / Ders 8):** Frobenius norm, bir matrisi tek bir “büyüklük” sayısına indirir; vektör uzunluğunun (Öklid normu) matris hâlidir.
- **Sezgi:** Sembol $\sqrt{\sum a_{ij}^2}$ korkutur ama tarif basittir — karele, topla, karekök al; üretken modellemede iki tensörün “ne kadar farklı” olduğunu ölçerken sık geçer.

20.9 8. Broadcasting: Skaler ile

Broadcasting, farklı şekilli tensörleri otomatik hizalayıp işleme sokmaktır. En basiti skalerle: `tensor * 2` skaleri tüm elemanlara uygular (sanki aynı şekle kopyalanmış gibi) — ama gerçek kopya oluşmaz, optimize biçimde yapılır.

“Now we’re going to get into something really fun, which is broadcasting.” — Howard, 1:24:00

💡 Builder Notu — Skaler Broadcast: En Basit Hâli

- **Geriye (Ders 5):** Ders 5’te broadcasting’i “891 kez kopyalanmış gibi” diye görmüştük; burada kuralları öğreniyoruz.
- **Sezgi:** `tensor * 2` skaleri her elemana uyguluyormuş gibidir ama bellekte kopya oluşmaz — “sanki yayıldı” yalnızca kavramsaldır; gerçek iş optimize koddadır.

20.10 9. Boyut Ekleme: unsqueeze ve None

Broadcasting’i yönlendirmek için tensöre **birim eksen** ekleriz. `unsqueeze(0)` veya `c[None, :]` başa, `unsqueeze(1)` veya `c[:, None]` ortaya bir boyut ekler. Bu, hangi eksenlerin hizalanacağını kontrol etmenin yoludur.

```
c.unsqueeze(0), c[None, :] # bas tarafa birim eksen (1, n)
c.unsqueeze(1), c[:, None] # ortaya birim eksen (n, 1)
```

💡 Builder Notu — None: Eksen Yerleştirmenin Kontrolü

- **İleriye (Ders 24):** None ile eksen ekleme, broadcasting matmul’un ve attention’ın (Ders 24) anahtar tekniğidir.
- **Sezgi:** `[None, :]` ile `[:, None]` farkı, vektörün **satır** mı **sütun** mu olarak yayılacağını belirler; matmul’da satırı (n,1) yapmak için `[:, None]` kullanılır.

20.11 10. Vektör-Matris Broadcasting

Bir vektörü bir matrisle çarparken broadcasting devreye girer: vektör, matrisin her satırına (veya sütununa, eksen ekleyişine göre) uygulanır. Bu, “her satır için ayrı işlem” döngüsünü tek bir tensör işlemine çevirir.

💡 Builder Notu — Vektör-Matris: Döngüyü Tek İşleme İndirme

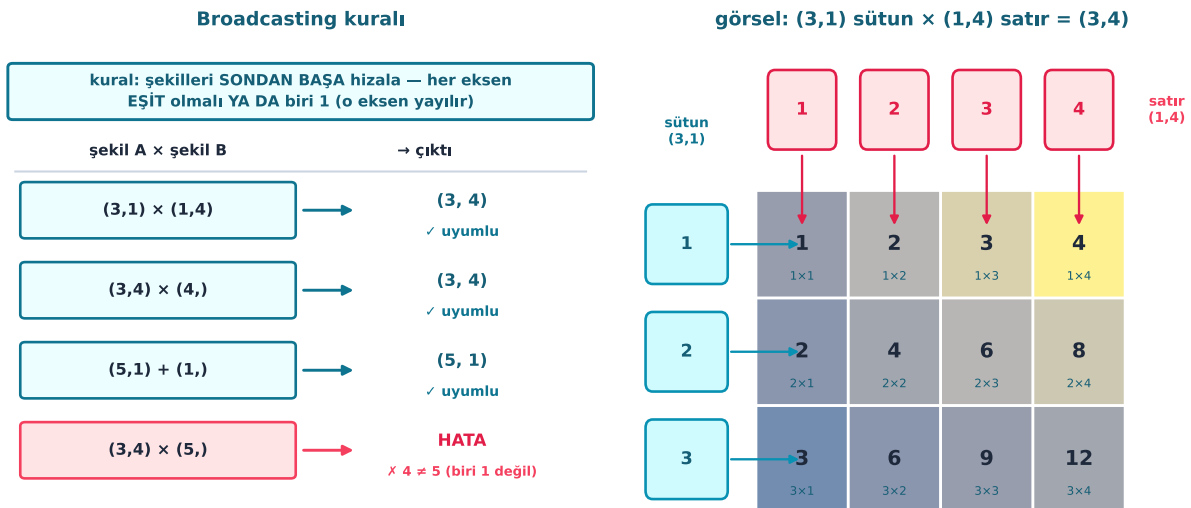
- **Geriye (Ders 5):** Ders 5’teki normalizasyon (her satırı vektöre bölme) bu broadcasting’di.
- **Sezgi:** “Her satır için ayrı uygula” bir Python döngüsü olurdu; broadcasting bu döngüyü silip işi tek tensör çağrısına devreder — okunabilirlik ve hız birlikte gelir.

20.12 11. Broadcasting Kuralları

Kural basittir: iki tensörün şekilleri **sondan başa** karşılaştırılır; her eksen boyutlar ya **eşit** olmalı ya da biri **1** olmalı (1 olan, diğerine “yayılır”). Eksen sayıları farklıysa eksik olan başa 1’lerle doldurulur.

“The rules of broadcasting are [worth knowing]; as long as the last axes match, it’ll broadcast over those axes.” — Howard, 1:24:00

Şekil 20.5 bu kuralı tablo ve görsel yayılma ile gösterir (**FLAGSHIP**): solda örnek şekiller için “uyumlu mu?” tablosu (eşit ya da biri 1 → ✓; aksi → HATA), sağda $(3, 1)$ sütun $\times$ $(1, 4)$ satır = $(3, 4)$ yayılma çarpım tablosu — sütun yatay, satır dikey yayılır.



Şekil 20.5: Broadcasting kuralı (sondan başa: her eksen eşit YA DA biri 1) ve $(3,1) \times (1,4) = (3,4)$ yayılma çarpım tablosu. (gerçek hesaplama: np.broadcast_shapes kuralları + $(3,1) \times (1,4) = (3,4)$ çarpım tablosu)

💡 Builder Notu — Kural: Sondan Başa, Eşit ya da 1

- **Geriye (Ders 5 / §4.F):** Bu kurallar NumPy/PyTorch'ta aynıdır; Karpathy §4.F'de broadcasting'i “sessiz hata kaynağı” olarak işaretler — kural bilinmezse yanlış sonuç **sessizce** çıkar.
- **Sezgi:** “Sondan başa” hizalama, eksen sayıları farklı tensörleri (örn. (784,) ile (5, 784)) otomatik eşler; eksik eksen başa 1 olarak doldurulur ve oradan yayılır.

20.13 12. Broadcasting ile Matmul: Döngüleri Kaldırma

İşte kazanım: matmul'un en içteki döngüsünü (dot product) broadcasting ile kaldırırız. $a[i, :, \text{None}] * b$ ifadesi, i . satırı tüm b sütunlarıyla aynı anda çarpar; $.sum(\text{dim}=0)$ ise toplar. Üç döngüden tek döngüye iner.

```
def matmul(a, b):
    (ar, ac), (br, bc) = a.shape, b.shape
    c = torch.zeros(ar, bc)
    for i in range(ar):
        c[i] = (a[i, :, None] * b).sum(dim=0)  # broadcast: tüm sütunlar birden
    return c
```

Burada $a[i, :, \text{None}]$ i . satırı (784, 1) yapar, b (784, 10) ile broadcast çarpılır (784, 10), $.sum(\text{dim}=0)$ ise sütunları toplar (10,) — i . satırın tüm çıktısı tek işlemde hesaplanır.

Şekil 20.6 bu akışı ve sonucu gösterir: üstte $a[i, :, \text{None}]$ (5, 1) sütun $\times$ b (5, 3) $\rightarrow$ ara (5, 3) broadcast $\rightarrow$ $sum(0) \rightarrow c[i]$ (3,); altta broadcasting sonucu ile numpy $a @ b$ yan yana — $np.allclose$ ile özdeş.

💡 Builder Notu — İç İki Döngü Broadcasting'e Devredildi

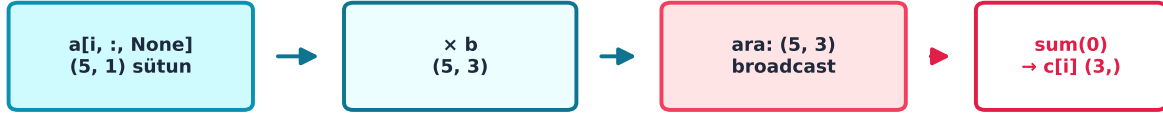
- **Geriye (Ders 10):** Ders 10'un üç döngüsü tek döngüye indi; iç iki döngü (j sütun, k dot product) broadcasting'e devredildi.
- **Sezgi:** $a[i, :, \text{None}]$ satırı dik bir sütun yapar ki tüm b sütunlarıyla aynı anda hizalansın; $.sum(\text{dim}=0)$ ile k eksenini toplanınca i . satırın bütün çıktısı tek tensör işleminde çıkar.

20.14 13. Broadcasting Matmul: Hız

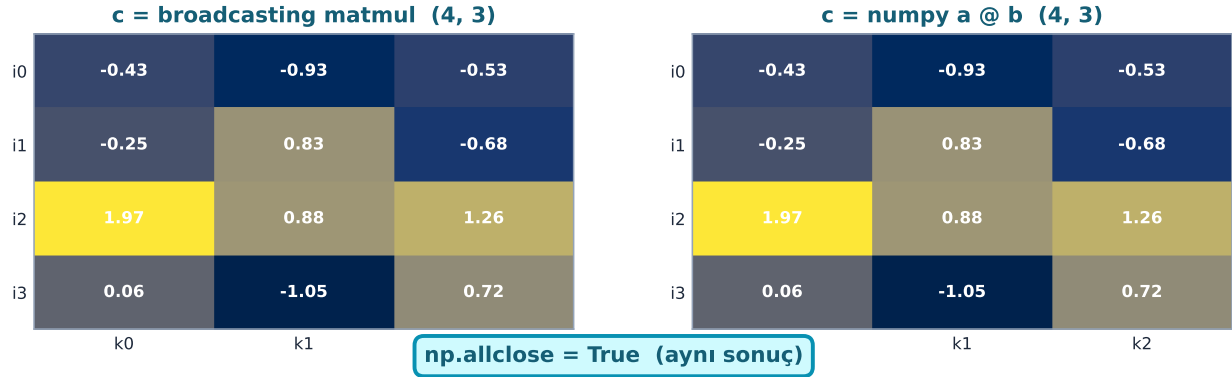
Bu versiyon, Python döngülü matmul'dan **yüzlerce kat** hızlıdır: iç iş artık optimize C koduna (ve GPU'ya) düşer. Howard `%timeit` ile farkı gösterir — milisaniyeler yerine mikrosaniyeler.

Şekil 20.7 bu ilerlemeyi **gerçek canlı benchmark** ile ölçer: aynı matris çarpımı üç yöntemle — 3 iç içe döngü $\rightarrow$ broadcasting $\rightarrow$ numpy $@$ — her adım hızlanır, sonuçlar $np.allclose$ ile özdeştir. Broadcasting döngüyü optimize koda devreder.

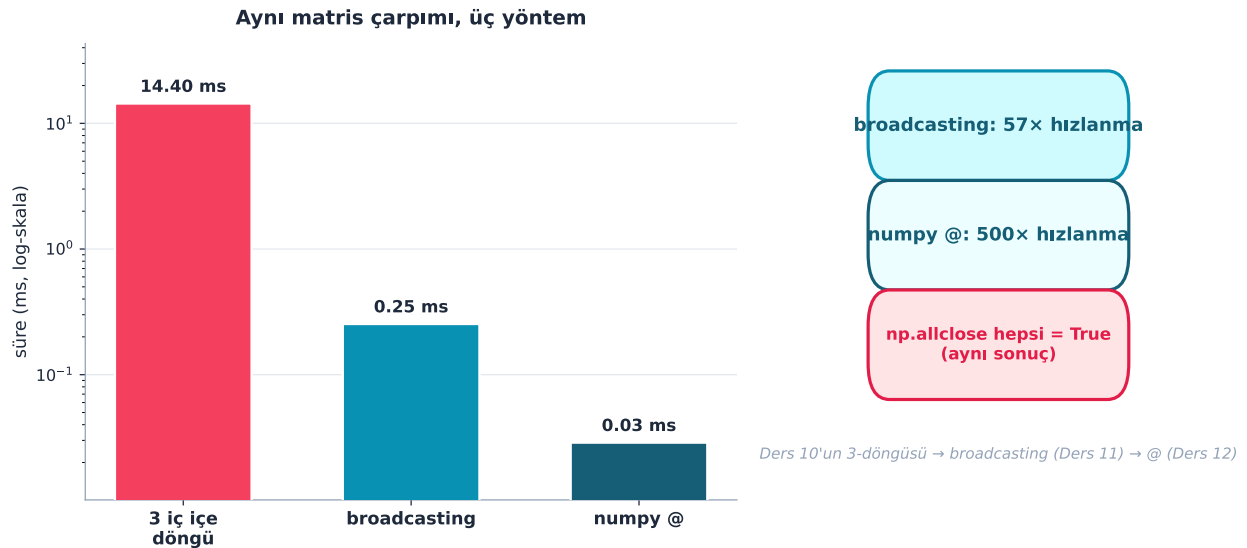
Broadcasting matmul: iç iki döngü (j, k) yok edildi



tek döngü (i, satır) kalır; iç çarpım-toplam broadcasting'e devredildi



Şekil 20.6: (gerçek hesaplama: $c[i]=(a[i,:,\text{None}]*b).\text{sum}(0)$ broadcasting matmul; numpy @ ile özdeş, np.allclose=True)



Şekil 20.7: (gerçek canlı zamanlama: 3-döngü → broadcasting → numpy @; her adım hızlanır, sonuç özdeş — broadcasting döngüyü optimize koda devreder)

💡 Builder Notu — Döngüyü Tensör İşlemine Çevir

- **İleriye:** Bu, “döngüyü tensör işlemine çevir” ilkesi; tüm performanslı sayısal kodun temel deseni-
dir.
- **Builder ipucu:** Hız kazanırken doğruluğu kaybetmediğini her zaman doğrula — benchmark’ın
her yanı `np.allclose` ✓ veriyorsa kazanım gerçektir.

20.15 14. Neden Bu Kadar Hızlı?

Sebebi: Python döngüsü her işlemi tek tek yorumlar; broadcasting ise tüm işlemi tek bir tensör çağrısında, derlenmiş ve paralelleştirilmiş (SIMD/GPU) koda devreder. Az sayıda Python çağrısı, çok sayıda paralel hesap.

“*[This is] really slow.*” — Howard, ~58:00 (döngülü matmul timing)

💡 Builder Notu — Hızın Kaynağı: Az Python, Çok Paralel Hesap

- **Geriye (Ders 10):** Numba bir fonksiyonu derliyordu; broadcasting ise PyTorch’un zaten derli
C/CUDA çekirdeklerini kullanır — kod yazmadan hız.
- **Sezgi:** Python yorumlayıcısı her `c[i, j] += ...` işlemini tek tek çalıştırır (yavaş); broadcasting
bir komutla milyonlarca hesabı SIMD/GPU’ya devreder — “az Python çağrısı, çok paralel iş”.

20.16 15. İleriye: Einsum ve @

Howard sonraki adımı işaret eder: **Einstein toplamı (einsum)** ve PyTorch’un @ operatörü matmul’u daha da kısaltır ve hızlandırır (Ders 12). Sonunda `a @ b` tek karakterle, GPU’da en hızlı hâle gelir.

```
import torch
torch.einsum('ik,kj->ij', a, b) # tekrarlanan k toplanır -> matmul
a @ b                           # en kısa: GPU'da en hızlı
```

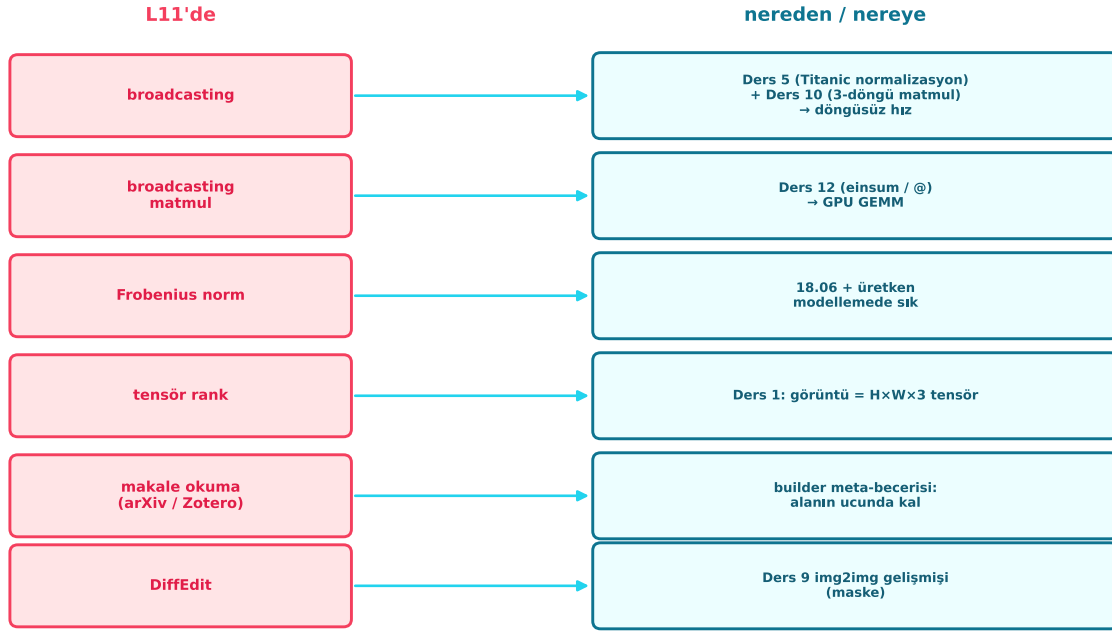
💡 Builder Notu — einsum / @: Yolculuğun Son Adımı

- **İleriye (Ders 12):** `einsum('ik,kj->ij')` ve `@`, matmul yolculuğunun son adımlarıdır; oradan
gerçek GPU GEMM’ine geçilir.
- **Sezgi:** `einsum`’da tekrarlanan indeks (k) “bu eksen üzerinden topla” demektir; `@` ise bunun en
kısa yazılışdır — broadcasting matmul’u bir kez anladıktan sonra `@`’i güvenle kullanırsın.

20.17 16. Kapanış

Ders 11 iki beceri verdi: makale okuma (arXiv + Zotero ile DiffEdit) ve broadcasting. Broadcasting, Ders 10'un yavaş üç-döngü matmul'unu optimize, döngüsüz bir tensör işlemine çevirdi — “from the foundations” omurgasının ilk büyük hız kazanımı.

Şekil 20.8 dersin sentezidir: solda L11'in parçaları (broadcasting, broadcasting matmul, Frobenius norm, tensör rank, makale okuma, DiffEdit), sağda her birinin nereden/nereye köprüsü — Ders 5+10 → döngüsüz hız, Ders 12 einsum/@ → GPU GEMM, 18.06 + üretken modelleme, Ders 1 H×W×3 tensör, builder meta-beceri, Ders 9 img2img gelişmiş (maske).



Şekil 20.8: (sentez: L11 = broadcasting hız kazanımı + makale-okuma becerisi; Ders 5/10 → 12, 18.06, Ders 9 köprüleri)

💡 Builder Notu — Foundations Zinciri İlerliyor

- **İleriye (Ders 12-14):** Ders 12 matmul'u einsum/@ ile bitirip mean shift'e geçer; Ders 13-14 backprop'u sıfırdan kurar.
- **Tek cümle:** Bir döngüyü kaldırmanın yolu broadcasting'dir; matmul bunun ilk büyük kazanımıydı, sıradaki parçalar (backprop, Learner, diffusion) aynı zincirde elle kurulacak.

20.18 Bu Dersin Özeti

1. **Makale okuma** araştırmanın temel becerisi: arXiv'den preprint, Zotero ile not alma (DiffEdit örneği) ([makale akışı](#)).
2. **DiffEdit** bir görüntüde yalnızca prompt'un işaret ettiği bölgeyi değiştirir (maske + hedefli gürültü-giderme) ([DiffEdit](#)).

3. Tensörün **rank**'ı boyut sayısıdır (skaler/vektör/matris/yığın); broadcasting şekil üstünde çalışır (**rank**).
4. **Frobenius norm** = $\sqrt{\sum a_{ij}^2}$ (tüm elemanların kareleri toplamının karekökü); matrisin “büyüklüğü” (**Frobenius**).
5. **Broadcasting** farklı şekilli tensörleri otomatik hizalar; kural: sondan başa, eksenler eşit ya da biri 1 (**kurallar**).
6. **unsqueeze/None** ile birim eksen ekleyerek broadcasting yönlendirilir (**unsqueeze**).
7. **Broadcasting ile matmul** iç döngüyü kaldırır: `c[i] = (a[i,:,None] * b).sum(dim=0)` (**broadcasting matmul**).
8. Sonuç Python döngüsünden yüzlerce kat hızlı (optimize C/GPU); einsum ve @ (Ders 12) daha da ileri gider (**hız, einsum**).

! Tek Bir Cümle

Broadcasting, farklı şekilli tensörleri otomatik hizalayıp döngüleri optimize koda devretmenin yoludur; Ders 10'un yavaş üç-döngü matris çarpımını, tek döngülü ve yüzlerce kat hızlı bir tensör işlemine dönüştürür.

20.19 Kontrol Soruları

i Soru 1: Broadcasting nedir ve kuralı nedir?

Cevap:

Broadcasting, farklı şekilli iki tensörü, küçük olanı büyük olana otomatik “yayarak” (kopyalıyormuş gibi, ama gerçek kopya oluşturmada) elementwise işleme sokmaktır. Kural: şekiller **sondan başa** karşılaştırılır; her eksenin ya boyutlar **eşit** olmalı ya da biri **1** olmalı (1 olan diğerine yayılır). Eksen sayıları farklıysa, eksik olanın başına 1'ler eklenir. Örneğin (5, 784) ile (784,) çarpılınca vektör her satıra yayılır. Bu, açık döngü yerine tek bir optimize tensör işlemi sağlar. (Şekil 20.5 bu kuralı tablo ve görsel yayılma ile gösterir.)

i Soru 2: Broadcasting matris çarpımını nasıl hızlandırır? Ders 10'un üç döngüsüne ne olur?

Cevap:

Ders 10'da matmul üç iç içe döngüyü (i, j, k) ve Python'da yavaştı. Broadcasting ile en içteki dot-product döngüsü (k) ve sütun döngüsü (j) kaldırılır: `c[i] = (a[i,:,None] * b).sum(dim=0)`. Burada `a[i,:,None]` i. satırı (784, 1) yapar, b (784, 10) ile broadcast çarpılır (784, 10), `.sum(dim=0)` ise sütunları toplar (10,) — i. satırın tüm çıktısı tek işlemde. Geriye tek döngü (i) kalır. Hesap artık optimize C/GPU koduna düştüğü için yüzlerce kat hızlanır. (Şekil 20.7 bu hızlanmayı gerçek benchmark'la gösterir.)

i Soru 3: unsqueeze / None ne işe yarar ve broadcasting'le ilişkisi nedir?

Cevap:

`unsqueeze(dim)` veya `[None]` indeksleme, tensöre **birim (1) eksen** ekler — örneğin $(n,)$ vektörünü $(1, n)$ veya $(n, 1)$ yapar. Bu, broadcasting'in hangi eksenlerde yayılacağını **kontrol etmenin** yoludur.

Matmul’da `a[i, :, None]` satırı (784, 1) yapar ki (784, 10) matrisle hizalansın ve her sütunla çarpılsın. Birim eksen eklemekten broadcasting yanlış hizalanır veya hata verir. Bu teknik attention (Ders 24) gibi ileri konularda da merkezîdir. (Şekil 20.6 `a[i, :, None]` akışını gösterir.)

i Soru 4: Akademik makale okuma neden bir builder becerisidir? (builder bağlantısı)

Cevap:

Derin öğrenme çok hızlı ilerler ve en yeni teknikler önce **arXiv**’de (preprint olarak, hakem değerlendirmesinden önce) yayımlanır; hakemli yayını beklemek aylar/yıllar alır. Bir builder olarak alanın ucunda kalmak istiyorsan makale okuyabilmen gerekir. Howard’ın iş akışı: arXiv’den PDF al, **Zotero** gibi bir araçla kaydet ve üzerine not alarak sistemli oku. DiffEdit örneğinde olduğu gibi, bir makaleyi okuyup adım adım uygulayabilmek, “tüketici”den “üretici”ye geçişin anahtarıdır. (Şekil 34.1 sol kol bu akışı gösterir.)

20.20 Egzersizler

Egzersiz 1 (Direkt uygulama). İki tensör oluştur (örn. (5, 784) ve (784,)) ve `unsqueeze/None` ile farklı eksenlere birim boyut ekleyip broadcasting sonuçlarını gözle. (§9)

Egzersiz 2 (İki-aşamalı). Ders 10’un üç-döngü `matmul`’unu broadcasting versiyonuna (`c[i] = (a[i, :, None]*b).sum(dim=0)`) çevir; `%timeit` ile hız farkını ölç. (Şekil 20.7’i kendi sayınla yeniden üret — §12-13)

Egzersiz 3 (Edge case). Şekilleri broadcast edilemeyen iki tensörü (örn. (3, 4) ve (5,)) çarpmayı dene; hatayı oku ve kuralı neden ihlal ettiğini açıkla. (§11)

Egzersiz 4 (Kavramsal). Bir matrisin Frobenius norm’unu elle hesapla (elemanları karele, topla, karekök al); `torch` ile doğrula. (§7)

Egzersiz 5 (Sonraki dersin habercisi — Ders 12). `torch.einsum('ik,kj->ij', a, b)` ifadesinin `matmul` ile aynı sonucu verdiğini doğrula; tekrarlanan indeks (k) ne anlama geliyor düşün. (§15)

20.21 Sonraki: Ders 12 İçin Hazırlık

Ders 12: Mean Shift Kümeleme (Mean shift clustering)

Ders 11 `matmul`’u broadcasting ile hızlandırdı. Ders 12 önce `matmul`’u `einsum` ve `@` ile bitirir (CLIP Interrogator demosuyla), sonra ilk tam “from foundations” algoritmasını kurar: **mean shift clustering** — etiketsiz veriyi kümelere ayıran bir yöntem.

Ana konular (Ders 12):

- Einstein toplamı (`einsum`) ve `@`
- CLIP Interrogator (görüntüden prompt)
- Mean shift clustering (Gaussian kernel)
- Batched mean shift (GPU)

⚠ Ders 12 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 2 ve 5 — broadcasting matmul + einsum).
- Broadcasting kurallarını kendi cümlelerinle yaz.
- Ana cümleyi tekrar oku: “Broadcasting = döngüyü optimize koda devret.”

20.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
arXiv	Preprint sunucusu; en yeni makaleler	13:22
Zotero	Makale kaydetme + not alma aracı	15:03
DiffEdit	Maske + hedefli gürültü-giderme ile bölgesel düzenleme	46:36
Rank	Tensörün boyut (eksen) sayısı	1:01:13
Elementwise	Aynı şekilli tensörlerde eleman eleman işlem	1:01:13
Frobenius norm	$\sqrt{\sum a_{ij}^2}$; matris büyüklüğü	1:19:58
Broadcasting	Farklı şekilli tensörleri otomatik hizalama	1:24:00
Broadcasting kuralı	Sondan başa; eksen eşit ya da biri 1	1:24:00
unsqueeze / None	Tensöre birim eksen ekleme	1:24:00
Broadcasting matmul	<code>c[i] = (a[i, :, None]*b).sum(dim=0);</code> döngüsüz	59:15
Makale okuma	arXiv + Zotero iş akışı; builder becerisi	13:09
einsum / @ (ileriye)	matmul'un en kısa/hızlı hâli (Ders 12)	58:52

20.23 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: L11'in Tanıdık Kökleri

Bu ders araştırma becerisini ve foundations hızını her parçada tanıdık kavramlara bağlar; köprülerin özeti:

1. **Broadcasting** → döngüsüz tensör hesabı; tüm performanslı sayısal kodun temeli (Ders 5/10) ([broadcasting](#)).
2. **Broadcasting matmul** → üç döngü → tek döngü; optimize C/GPU'ya devir ([broadcasting matmul](#)).

3. **Frobenius norm** → matris büyüklüğü (18.06); üretken modellemede sık ([Frobenius](#)).
4. **Makale okuma (arXiv/Zotero)** → alandaki en yeni tekniklere erişim; builder meta-becerisi ([makale akışı](#)).
5. **DiffEdit** → maske + hedefli gürültü-giderme; img2img'in (Ders 9) gelişmiş (DiffEdit).
6. **unsqueeze/None** → eksen yönetimi; attention (Ders 24) ve broadcasting'in anahtarı ([unsqueeze](#)).

! Bu dersten tek bir şey alıp gideceksen

Bir döngüyü kaldırmanın yolu broadcasting'dir. Ders 10'un yavaş üç-döngü matris çarpımını, `c[i] = (a[i, :, None]*b).sum(dim=0)` ile tek döngüye ve yüzlerce kat hıza indirdik — çünkü iş artık optimize C/GPU koduna devredildi. Ayrıca bir builder becerisi öğrendik: makaleleri arXiv'den okuyup Zotero ile not alarak alanın ucunda kalmak.

21 Ders 12 — Mean Shift Kümeleme (Mean shift clustering)

Part 2'nin dördüncü dersi iki şey yapar. Önce matris çarpımını « bitirir »: einsum'la kompakt notasyona, sonra @ operatörüyle (BLAS/cuBLAS) en kısa hâle, ardından GPU CUDA kernel'ına taşır. Sonra ilk tam « from the foundations » algoritmasını sıfırdan kurar: mean shift clustering — etiketsiz veriyi, her noktayı yakın komşularının Gaussian-ağırlıklı ortalamasına çekerek kümeleyen bir yöntem. Anahtar fikir tek cümleyle: matmul'u einsum/@ ile bitirip GPU'ya taşıdık; sonra küme sayısını önceden bilmeden veriyi kümeleyen mean shift'i sıfırdan kurduk.

i Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 12: Mean shift clustering](https://course.fast.ai/lesson/12) (~110 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 12
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/01_matmul + 02_meanshift](#)
- **Okuma süresi:** ~38 dk
- **İki kol:** İlk yarı **matmul'u bitirir** (einsum → @ → GPU kernel), ikinci yarı **ilk foundations algoritması (mean shift)**. Bu ders ETAP 4 (Temeller A) içindedir.

21.1 Bu Derste Ne Var?

Part 2'nin dördüncü dersi matris çarpımını **bitirir** (einsum + @ + GPU) ve ilk tam “from foundations” algoritmasını sıfırdan kurar: **mean shift clustering**. Açılıшта eğlenceli bir demo var: [CLIP Interrogator](#) (görüntüden prompt üretme).

Üç temel fikir bu dersin omurgasını kurar:

1. **Einstein toplamı (einsum)** — 'ik,kj->ij' gibi kompakt bir notasyonla matris çarpımı; tekrarlanan indeks çarpım+toplamı ifade eder ([einsum](#) → [matmul](#)).
2. **@ ve GPU** — matmul'un en kısa hâli a @ b; GPU'da bir CUDA kernel ile binlerce işlem paralel ([@ ve hız](#) → [CUDA kernel](#)).
3. **Mean shift clustering** — etiketsiz veriyi, her noktayı yakın noktaların (Gaussian ağırlıklı) ortalamasına doğru kaydırarak kümelere ayırma ([mean shift](#) → [döngü](#)).

“Einstein summation is a compact representation for representing products and sums.” — Howard,
12:03

Şekil 34.1 bu yapıyı tek bir yol haritasında birleştirir: üstte “matmul’u bitir” kolu (einsum → @/BLAS → CUDA kernel ile hızlandırma), altta “from the foundations”ın ilk algoritması mean shift clustering (etiketsiz veri → Gaussian kernel → ağırlıklı ortalamaya kaydır → tekrarlar → kümeler; bandwidth küme sayısını belirler, k-means’ten farkı k’yı önceden gerektirmez).

💡 Builder Notu — İki Kol, Tek Ders

- **Geriye (Ders 11/18.06):** Ders 11 broadcasting matmul’u verdi; burada einsum/@ ile bitiyor. matmul = 18.06’nın kalbi; mean shift = unsupervised learning (6.S191 yelpazesi).
- **İleriye (Ders 13 / NYU §4.J):** Mean shift, gradient-tabanlı **olmayan** bir foundations algoritması; Ders 13’ten itibaren backprop’a geçilir. Clustering, NYU’nun self-supervised/temsil öğrenme bağlamıyla akraba.
- **Tek cümle:** matmul’u einsum/@ ile bitirip GPU’ya taşıdık; sonra her noktayı komşularının ağırlıklı ortalamasına çeken mean shift’i sıfırdan kurduk.

21.2 1. CLIP Interrogator: Görüntüden Prompt

Howard eğlenceli bir demoyla açar: **CLIP Interrogator** — bir görüntü verirsin, sana onu üretebilecek bir prompt tahmini döner. CLIP image encoder görüntüyü bir embedding’e (birkaç float’lık vektör) çevirir; Interrogator bu embedding’e en uyan metin parçalarını (medium, sanatçı, stil) arar.

“The CLIP Interrogator... I upload [an image and it suggests a prompt].” — Howard, 0:52

Howard önemli bir nüans ekler: bu, görüntüyü **tam tersine çevirmez** — embedding küçük olduğu için bilgi kaybı vardır; yalnızca yaklaşık bir prompt tahminidir. Yani embedding görüntüyü “sıkıştırır”; oradan geri okunan prompt yakındır ama orijinalin kendisi değildir.

💡 Builder Notu — CLIP Interrogator: Embedding Eşleştirme

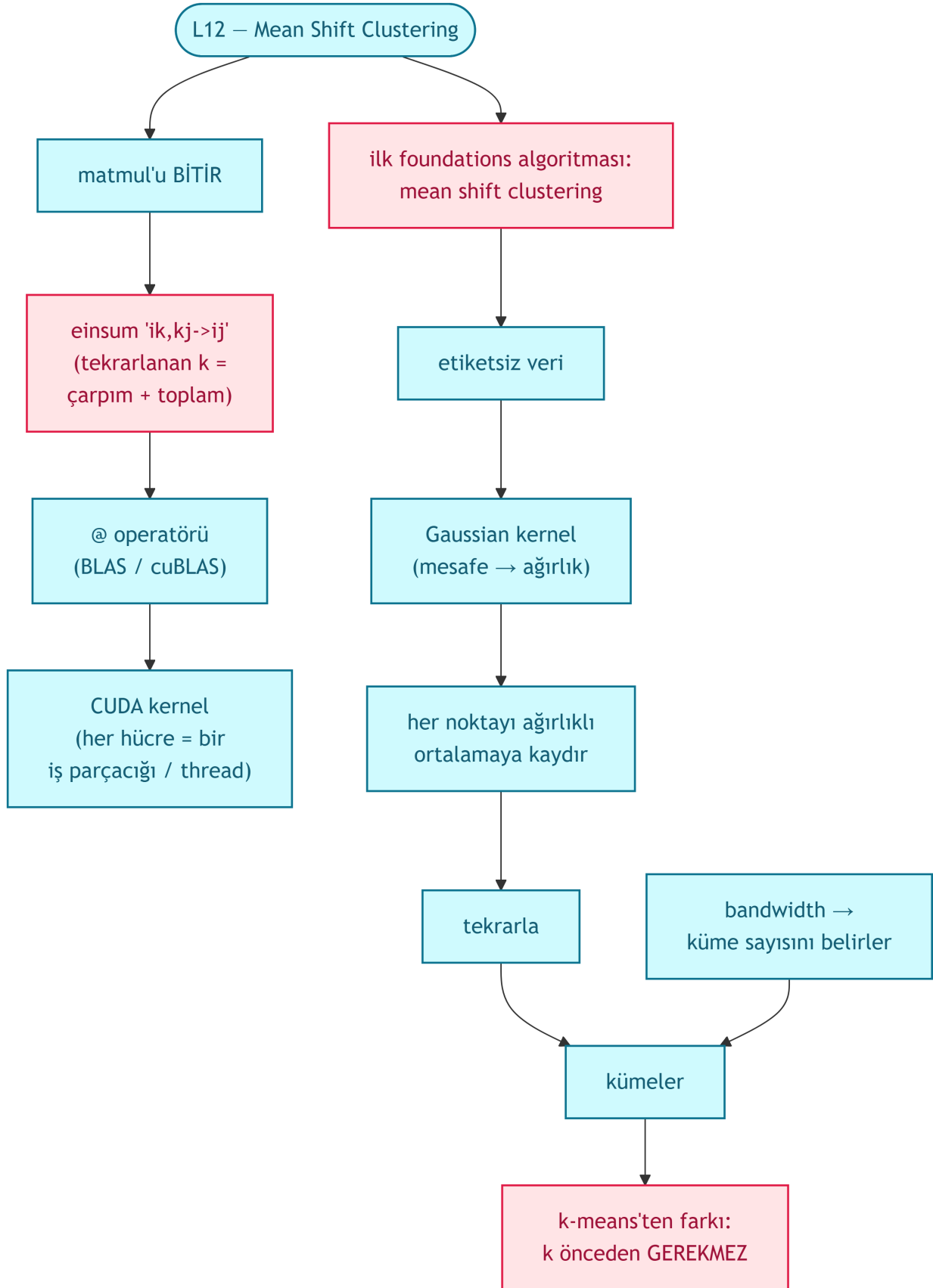
- **Geriye (Ders 8/9):** CLIP embedding’i (Ders 9) + embedding mesafesi (Ders 8); Interrogator, metin parçalarını görüntü embedding’ine en yakın olanı arayarak bulur.
- **Sezgi:** Küçük embedding bilgi kaybeder; bu yüzden Interrogator “tam tersine çevirme” değil, “en yakın metni bulma” işidir — yaklaşık bir tahmin, kesin geri-dönüşüm değil.

21.3 2. Einstein Toplamı (einsum)

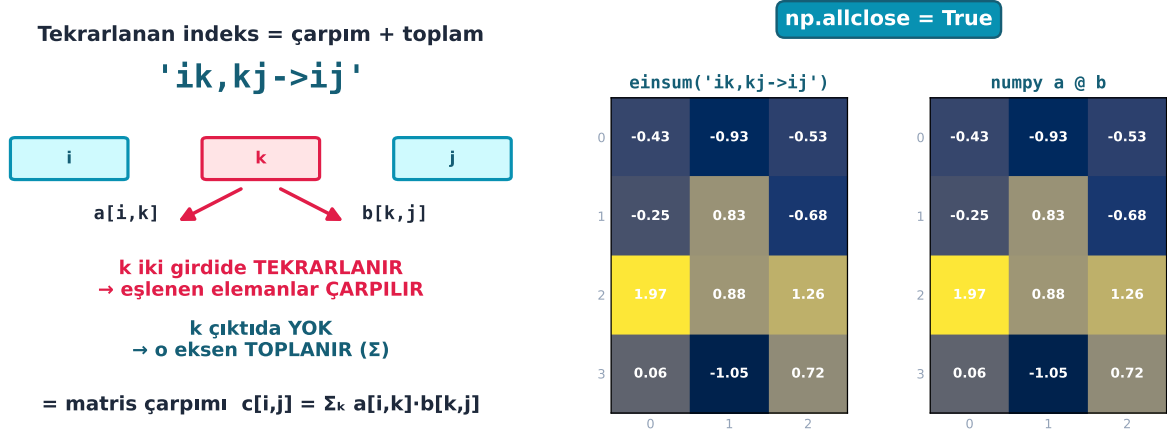
Howard matmul’u daha da kısaltan bir araç tanıtır: **Einstein toplamı**. ‘ik,kj->ij’ notasyonunda girdi matrisleri ik ve kj indeksli; çıktı ij. **Tekrarlanan indeks** (k) hem çarpımı hem toplamı ifade eder — k çıktıda olmadığı için o eksen üzerinden otomatik toplanır.

“Einstein summation is a compact representation for representing products and sums. The repeated letter [k] means those values will be summed.” — Howard, 12:03

Şekil 21.2 bu kuralı **gerçek hesaplama** ile gösterir: solda ‘ik,kj->ij’ şeması (k iki girdide tekrarlanır → eşlenen elemanlar çarpılır; k çıktıda yok → o eksen toplanır), sağda torch.einsum(‘ik,kj->ij’) ile numpy a @ b sonuçlarının np.allclose ile özdeş olduğu karşılaştırma.



Şekil 21.1: Ders 12'nin iki parçası: ÜSTTE 'matmul'u bitir' (einsum → @/BLAS → CUDA kernel ile hızlandırma), ALTTA 'from the foundations'ın ilk algoritması mean shift clustering (etiketsiz veri → Gaussian kernel → ağırlıklı ortalamaya kaydır → tekrarla → kümeler; bandwidth küme sayısını belirler, k-means'ten farkı k'yı önceden gerektirmez). (şematik)



tekrarlanan indeks = çarpım+toplam;
@ ile özdeş, daha okunabilir

Şekil 21.2: (gerçek hesaplama: torch.einsum('ik,kj->ij')=matmul; tekrarlanan k çarpım+toplam, numpy @ ile özdeş)

💡 Builder Notu — einsum: Çarpım-Toplamın Kompakt Notasyonu

- **Geriye (Ders 11):** einsum, broadcasting matmul'un kompakt notasyonu; aynı işi tek satırda yapar.
- **Sezgi:** Tek kuralı hatırla — tekrarlanan harf “bu eksen üzerinden çarp ve topla”, çıktıda olmayan harf “topla” demektir; gerisi indekslerin yeniden düzenlenmesidir.

21.4 3. einsum ile Matmul

Tüm matris çarpımı tek satıra iner. Tekrarlanan k indeksi çarpım+toplamı, ij ise sonucun şeklini verir:

```
def matmul(a, b): return torch.einsum('ik,kj->ij', a, b)
```

“We can now define our matmul as simply this torch.einsum.” — Howard, 12:03

Howard bunun broadcasting versiyonuyla **aynı hızda** olduğunu ölçer — ama çok daha okunabilir. Tek satır, tüm iç döngülerin yerini tutar.

💡 Builder Notu — einsum matmul: Tek Satır, Tüm İş

- **İleriye (Ders 24):** einsum, attention (Ders 24) gibi karmaşık tensör işlemlerini tek satırda ifade etmenin güçlü yolu; “biraz pratik ister” ama kodu sadeleştirir.
- **Sezgi:** 'ik,kj->ij' matris çarpımının matematiksel tanımının kelime kelimesi yazılışdır — kod ile formül burada birebir örtüşür.

21.5 4. PyTorch @ ve Hız

En kısa hâl: PyTorch'un @ operatörü. $a @ b$ doğrudan optimize matris çarpımını çağırır — CPU'da BLAS, GPU'da cuBLAS. einsum ve broadcasting'den de hızlıdır, çünkü onlarca yıllık optimize lineer cebir kütüphanelerini kullanır:

```
a @ b # PyTorch'un optimize matris carpimi (CPU BLAS / GPU cuBLAS)
```

💡 Builder Notu — @: Yeniden Kur, Sonra Kullan

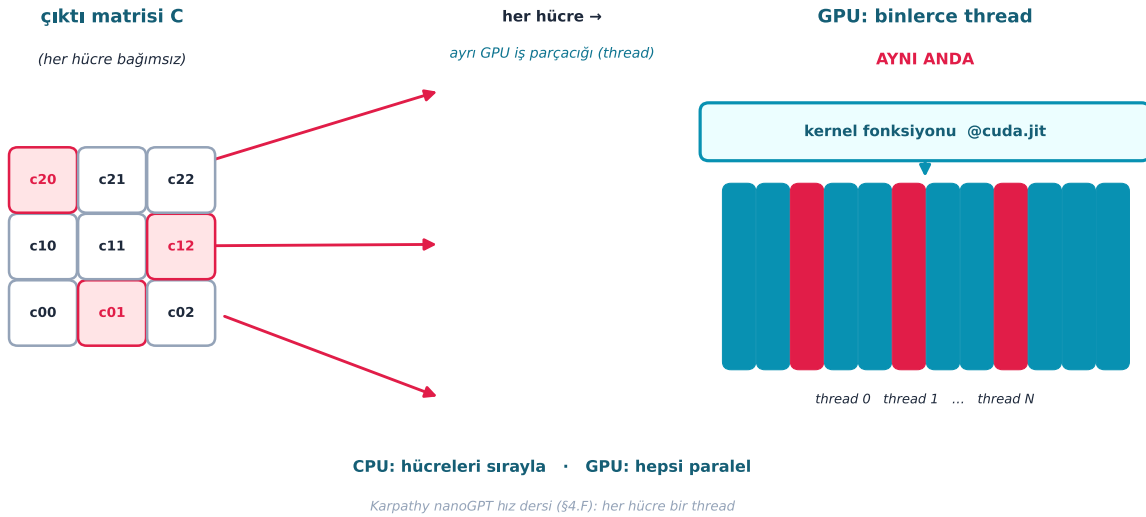
- **Geriye (Ders 5):** Ders 5'teki @'in içini (Ders 10-12 boyunca) sıfırdan kurduk; artık onu güvenle kullanabiliriz — “yeniden kur, sonra kullan” kuralı.
- **Sezgi:** @ bir “kara kutu” değil artık, “anlaşılmış kutu” — mekanizmayı kurduktan sonra optimize aracı kullanmak hem doğru hem verimlidir.

21.6 5. CUDA Kernel: GPU'da Matmul

Howard GPU'da nasıl çalışıldığını gösterir: bir **kernel** (GPU'da paralel çalışan fonksiyon) yazılır; her çıktı hücresi ayrı bir GPU iş parçacığında (thread) hesaplanır. `@cuda.jit` ile işaretlenen matmul, binlerce hücreyi aynı anda hesaplar.

“A [function we run on] a GPU is called a kernel.” — Howard, ~1:00:00

Şekil 21.3 bu fikri şematik gösterir: solda çıktı matrisi C'nin her hücresi bağımsız, ortada her hücre ayrı bir GPU iş parçacığına eşlenir, sağda GPU'nun binlerce thread'i `@cuda.jit` kernel'ı ile **aynı anda** çalıştırışı — CPU hücreleri sırayla, GPU hepsini paralel işler.



Şekil 21.3: (şematik: GPU kernel — çıktının her hücresi ayrı bir iş parçacığında paralel hesaplanır; CPU sıralı, GPU binlerce thread)

💡 Builder Notu — Kernel: Her Hücre Bir Thread

- **İleriye (Karpathy Ders 10 / §4.F):** GPU kernel programlama (Triton/CUDA), Karpathy’nin nanoGPT hız dersinin (§4.F) konusu; “her hücre bir iş parçacığı” temel fikir.
- **Sezgi:** GPU’nun gücü çiplerin tek tek hızında değil, **paralellikte**: az Python çağırısı, binlerce eşzamanlı hesap.

21.7 6. Kümeleme (Clustering) Nedir?

Şimdi ilk tam foundations algoritması. **Clustering** (kümeleme), etiketsiz veriyi benzer gruplara (kümelere) ayırır — şimdiye kadarki her şeyden farklı, çünkü **etiket yok** (unsupervised). Howard kümelemenin bazen kötüye kullanıldığını da dürüstçe ekler.

“Cluster analysis is very different to anything that we’ve [seen]... we group data and those groups we call clusters.” — Howard, 8:46

💡 Builder Notu — Clustering: Etiketsiz Yapı Keşfi

- **Geriye (Ders 4):** Şimdiye kadar supervised (etiketli) çalıştık; clustering etiketsizdir — model yapıyı kendisi bulur.
- **Sezgi:** Supervised’da “doğru cevap” (etiket) verilir; clustering’de model yalnız verinin yoğunluk yapısına bakarak grupları **kendisi** keşfeder.

21.8 7. Sentetik Veri: Centroid’ler

Algoritmayı test etmek için Howard “nasıl davranacağını bildiği” sentetik veri üretir: önce 6 rastgele **centroid** (küme merkezi, X-Y koordinatı), sonra her birinin etrafında rastgele noktalar. Amaç: algoritmanın bu 6 kümeyi yeniden bulup bulamayacağını görmek.

```
n_clusters = 6
centroids = torch.rand(n_clusters, 2) * 70 - 35 # 6 kume merkezi
# her merkez etrafında rastgele noktalar uretilir -> data
```

💡 Builder Notu — Sentetik Veri: Cevabı Bildiğin Test

- **İleriye (builder):** “Cevabını bildiğin sentetik veriyle algoritmayı doğrula” — sağlam bir test stratejisi; sonuç gerçek centroid’lere oturuyorsa algoritma çalışıyor demektir.
- **Sezgi:** Önce kontrollü, bilinen-cevaplı bir oyuncak veri; karmaşık gerçek veriye geçmeden önce mekanizmayı burada doğrula.

21.9 8. Mean Shift: Temel Fikir

Mean shift'in fikri zarif: her noktayı, yakınındaki noktaların **ağırlıklı ortalamasına** doğru kaydır; yakın noktalar daha ağır sayılır. Bunu birkaç kez tekrarlırsan, noktalar yoğun bölgelerin (kümelerin) merkezine doğru toplanır.

“Mean shift clustering, which hopefully you’ve never heard of before [so we can build it cleanly].”
— Howard, clustering bölümü

💡 Builder Notu — Mean Shift: Yoğun Bölgeye Tırman

- **Geriye (Stat 110):** Mean shift, yoğunluk tahmini (kernel density) üstüne kurulu; “yoğun bölgeye tırman” sezgisi.
- **Sezgi:** Her nokta, kendi mahallesindeki kütle merkezine doğru ufak bir adım atar; bu adımlar tekrarlanınca tüm noktalar yoğunluk tepelerine “tırmanır”.

21.10 9. Mesafe ve Gaussian Kernel

“Yakınlık” nasıl ağırlığa çevrilir? Her noktanın diğerlerine **mesafesi** hesaplanır, sonra bir **Gaussian kernel** ile ağırlığa dönüştürülür: yakın = yüksek ağırlık, uzak = sıfıra yakın. bandwidth (bant genişliği) “yakın” sayılan yarıçapı belirler.

```
import math
def gaussian(d, bw):
    return torch.exp(-0.5 * (d/bw)**2) / (bw * math.sqrt(2*math.pi))
# d = mesafe, bw = bandwidth; yakın -> büyük ağırlık
```

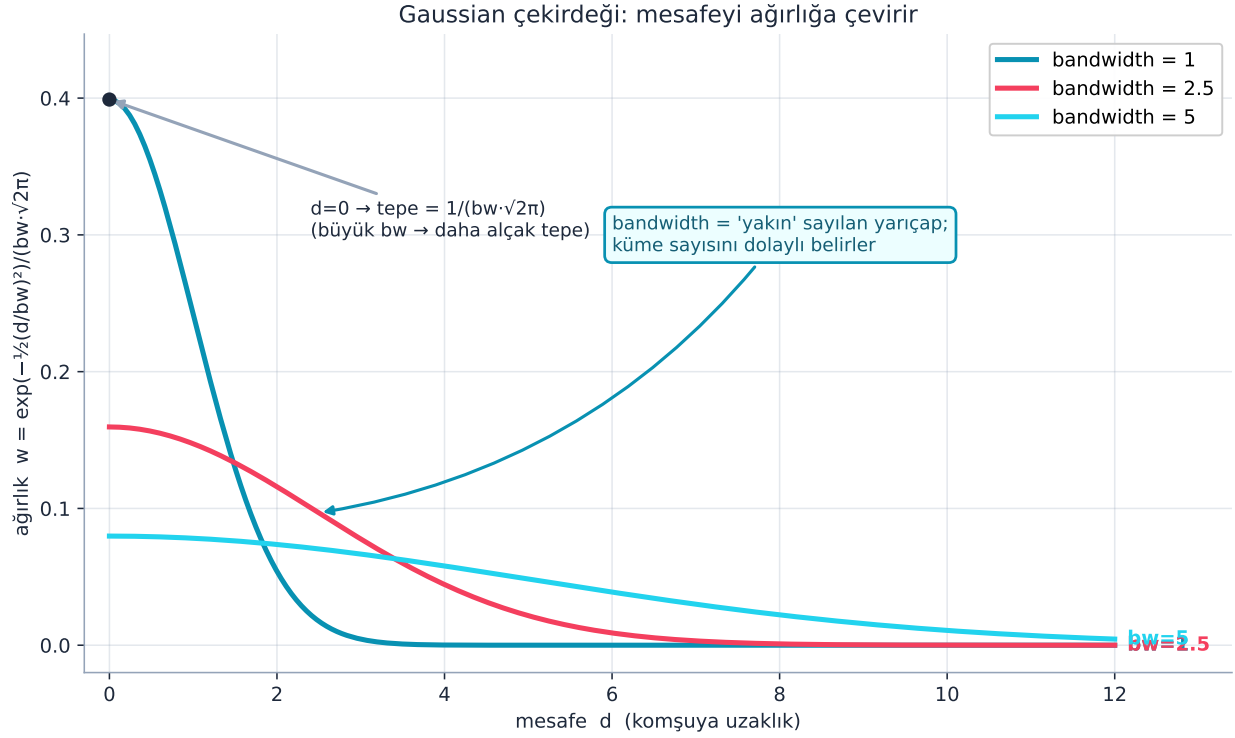
Şekil 21.4 bunu **gerçek hesaplama** ile gösterir (yukarıdaki normalize formülle birebir): mesafe (d) arttıkça ağırlık ($w = \exp(-\frac{1}{2}(d/bw)^2) / (bw\sqrt{2\pi})$) azalır; d=0’da tepe değeri $1/(bw\sqrt{2\pi})$ ’dir — yani her bandwidth için **farklıdır** (büyük bandwidth daha alçak ama daha geniş bir tepe verir, böylece daha geniş bir komşuluğu “yakın” sayar). Not: bu normalizasyon sabiti ağırlıklı ortalama ($(\sum w_i x_i) / (\sum w_i)$) sadeleştiği için mean shift sonucunu etkilemez.

💡 Builder Notu — Gaussian Kernel: Mesafeyi Ağırlığa Çevir

- **Geriye (Ders 9B / Stat 110):** Gaussian, 9B’deki normal dağılımın ta kendisi; burada “mesafeyi ağırlığa çeviren” kernel olarak kullanılır.
- **Sezgi:** Eğri yumuşak bir “yakınlık” tartısıdır — yakın komşu neredeyse tam sayılır, uzak nokta sessizce sıfıra iner; bandwidth bu yumuşaklığın genişliğini ayarlar.

21.11 10. Tek Güncelleme Adımı

Çekirdek işlem: her nokta için, tüm noktalara mesafeyi bul, Gaussian ile ağırlıkla, sonra noktayı bu ağırlıklı ortalamaya taşı. Broadcasting (Ders 11) burada işi döngüsüz yapar.



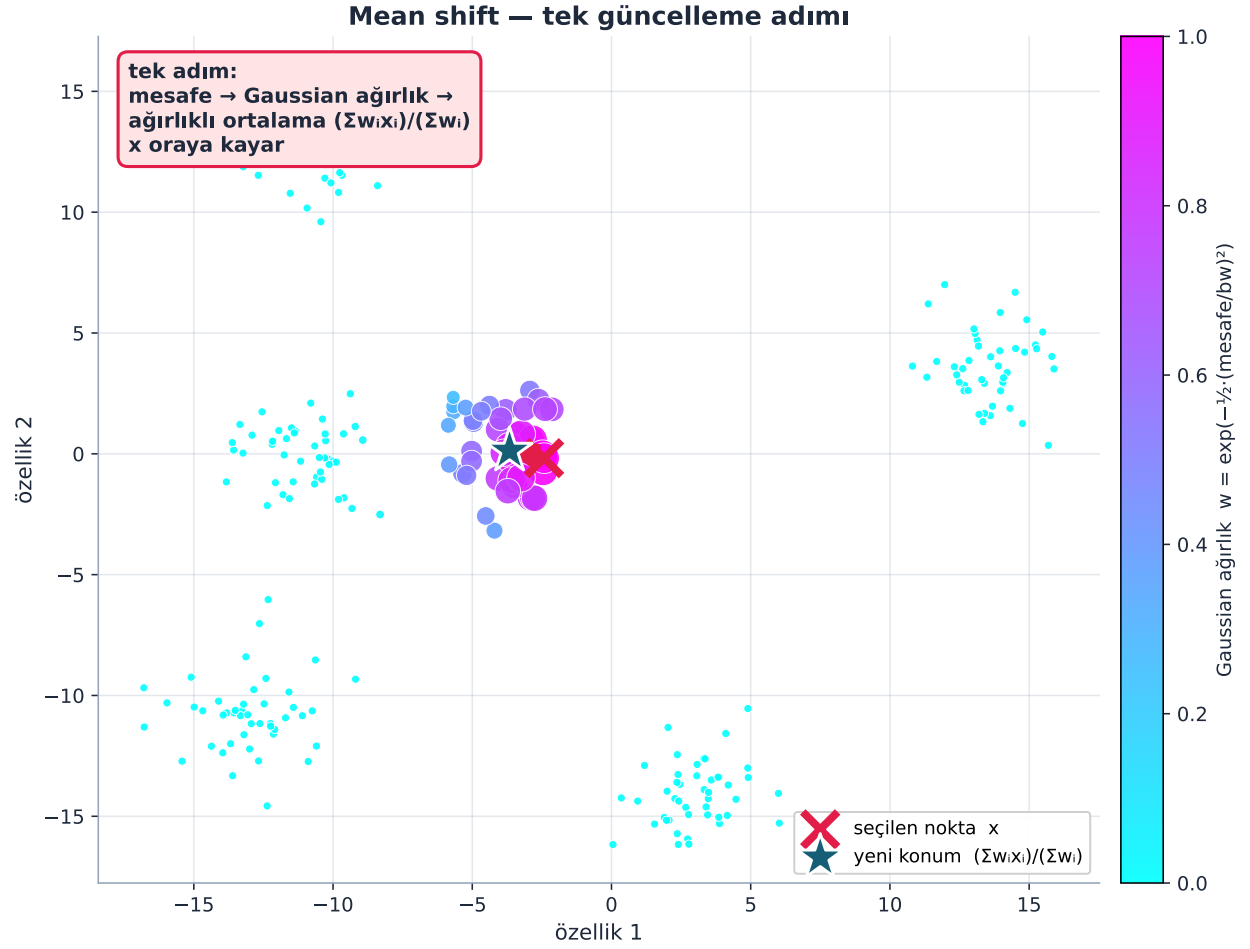
Şekil 21.4: (gerçek hesaplama: normalize Gaussian kernel $w = \exp(-\frac{1}{2}(d/bw)^2)/(bw \cdot \sqrt{2\pi})$); yakın yüksek/uzak ~ 0 , büyük bandwidth daha geniş ve daha alçak tepe)

```
def one_update(X):
    for i, x in enumerate(X):
        dist = torch.sqrt(((x - X)**2).sum(1)) # x'in tüm noktalara mesafesi
        weight = gaussian(dist, 2.5)          # mesafeyi ağırlığa çevir
        X[i] = (weight[:,None] * X).sum(0) / weight.sum() # ağırlıklı ortalama
```

Şekil 21.5 bu tek adımı **gerçek hesaplama** ile gösterir: seçilen bir x noktası (rose), tüm noktalar Gaussian ağırlığına göre renklenmiş, ve x'in yeni konumu ağırlıklı ortalama $(\sum w_i x_i) / (\sum w_i)$ (yıldız) — ok, x'in komşularının kütle merkezine kayışını gösterir.

💡 Builder Notu — Tek Adım: Ağırlıklı Ortalama

- **Geriye (Ders 11):** `weight[:,None] * X` broadcasting'dir; ağırlıklı ortalama $= (\sum w_i \cdot x_i) / (\sum w_i)$.
- **Sezgi:** Bir adım sadece “komşularının ağırlıklı kütle merkezine git”tir; Gaussian sayesinde yakın komşular bu merkezi belirler, uzaklar neredeyse hiç söz sahibi değildir.



Şekil 21.5: (gerçek hesaplama: tek mean shift adımı — x noktası yakın komşuların ağırlıklı ortalamasına kayar)

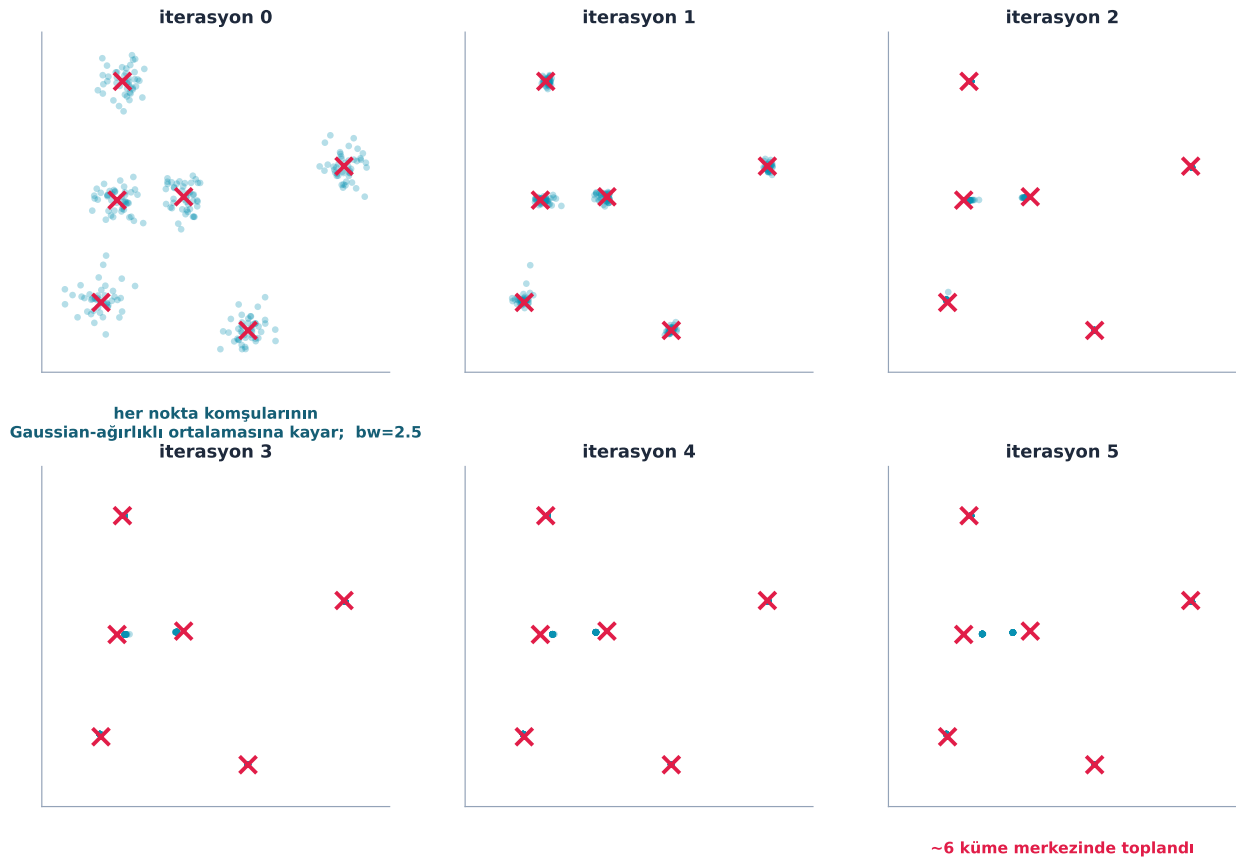
21.12 11. Mean Shift Döngüsü

Tek güncelleme bir adımdır; algoritma bunu birkaç kez (örn. 5) tekrarlar. Her turda noktalar küme merkezlerine biraz daha yaklaşır; sonunda her küme tek bir noktada toplanır.

```
def meanshift(data):
    X = data.clone()
    for it in range(5): one_update(X)
    return X
```

Şekil 21.6 bu yakınsamayı **gerçek hesaplama** ile gösterir (**FLAGSHIP**): 270 nokta, iterasyon 0'da (dağınık ham veri) başlar ve birkaç turda gerçek centroid'lere (rose) doğru toplanarak ~6 küme merkezinde yakınsar — her panel bir iterasyon, aynı pencerede kayış görünür.

Mean shift yakınsaması: dağınık veri → küme merkezleri



Şekil 21.6: (gerçek hesaplama: mean shift clustering — 270 nokta iterasyonlarla ~6 küme merkezinde toplanır)

💡 Builder Notu — Döngü: k Önceden Gerekmez

- **İleriye:** Mean shift, k-means'in aksine küme sayısını **önceden bilmeyi gerektirmez**; bandwidth küme sayısını dolaylı belirler.

- **Sezgi:** Her tur tüm noktaları biraz daha yoğunluk tepelerine iter; birkaç tur sonra noktalar birkaç ayrı tepede yığılır ve bu tepelerin sayısı küme sayısını **kendiliğinden** verir.

21.13 12. Triangular Kernel (Alternatif)

Howard Gaussian yerine başka kernel'lar da denenebileceğini gösterir, örneğin **triangular** (üçgen) kernel: mesafe arttıkça ağırlık doğrusal azalır, belli bir noktadan sonra sıfır olur.

```
def tri(d, i): return (-d + i).clamp_min(0) / i # ucgen kernel
```

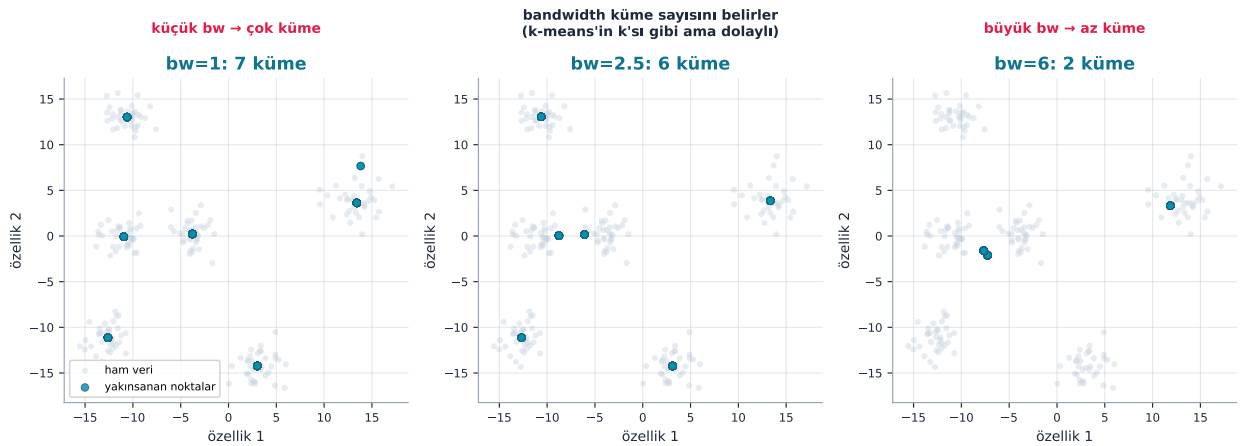
💡 Builder Notu — Kernel Seçimi: Deneysel Ayar

- **İleriye:** Kernel seçimi sonucu etkiler; farklı kernel'lar farklı yoğunluk tahminleri verir (deneysel ayar).
- **Sezgi:** Gaussian “yumuşak ve sonsuz kuyruklu”, triangular “sert kesimli” bir yakınlık tartısıdır; ağırlık fonksiyonunu değiştirmek kümelenmenin şeklini değiştirir.

21.14 13. Kaç Küme?

Mean shift'in güzelliği: küme sayısını **otomatik** bulur. Noktalar yoğun bölgelere toplandığında, kaç ayrı toplanma noktası olduğu küme sayısını verir — k-means gibi “k'yı önceden ver” gerekmez. Küme sayısını dolaylı olarak bandwidth belirler.

Şekil 21.7 bunu **gerçek hesaplama** ile gösterir: aynı veride üç bandwidth (1.0 / 2.5 / 6.0) için mean shift çalıştırılır — küçük bandwidth çok sayıda küçük küme, büyük bandwidth az sayıda büyük küme verir; bandwidth, k-means'in k'sı gibi ama dolaylı çalışır.



Şekil 21.7: Mean shift'te bandwidth, küme sayısını dolaylı belirler (gerçek hesaplama: bandwidth → küme sayısı; küçük bw çok/küçük küme, büyük bw az/büyük küme — mean shift k'yı önceden gerektirmez).

💡 Builder Notu — Kaç Küme: Veriden Keşfet

- **Geriye (Ders 6):** Random forest gibi mean shift de az varsayım ister; “veriden yapıyı keşfet” yaklaşımı.
- **Sezgi:** k’yı önceden vermek yerine “yakın ne demek?” sorusunu (bandwidth) yanıtlarsın; küme sayısı bu yarıçaptan kendiliğinden doğar.

21.15 14. Batched Mean Shift (GPU)

Tek tek nokta döngüsü yavaştır. Howard işi **batch**’lerde yapar: bir grup noktanın tüm mesafelerini tek tensör işleminde (broadcasting ile) hesaplar. Bu, GPU’da binlerce noktayı paralel işlemeyi sağlar.

```
def dist_b(a, b): return (((a[None] - b[:,None])**2).sum(2)).sqrt() # batch mesafe matrisi
```

💡 Builder Notu — Batched: Broadcasting’in Gücü

- **Geriye (Ders 11):** Batched mesafe = broadcasting’in ($a[None] - b[:,None]$) gücü; döngüsüz, GPU-dostu.
- **Sezgi:** Nokta-nokta döngüsü yerine tüm ikili mesafeleri tek tensör çağrısıyla üretirsin; Ders 11’in broadcasting deseni burada doğrudan hesabı GPU’ya devreder.

21.16 15. İleriye: Foundations Devam

Mean shift, gradient kullanmayan bir foundations algoritmasıydı. Howard sonraki adımı işaret eder: sinir ağlarına dönüp **backpropagation**’ı sıfırdan kurmak (Ders 13). Buraya kadarki araçlar (matmul, broadcasting) orada kullanılacak.

💡 Builder Notu — Foundations Zinciri İlerliyor

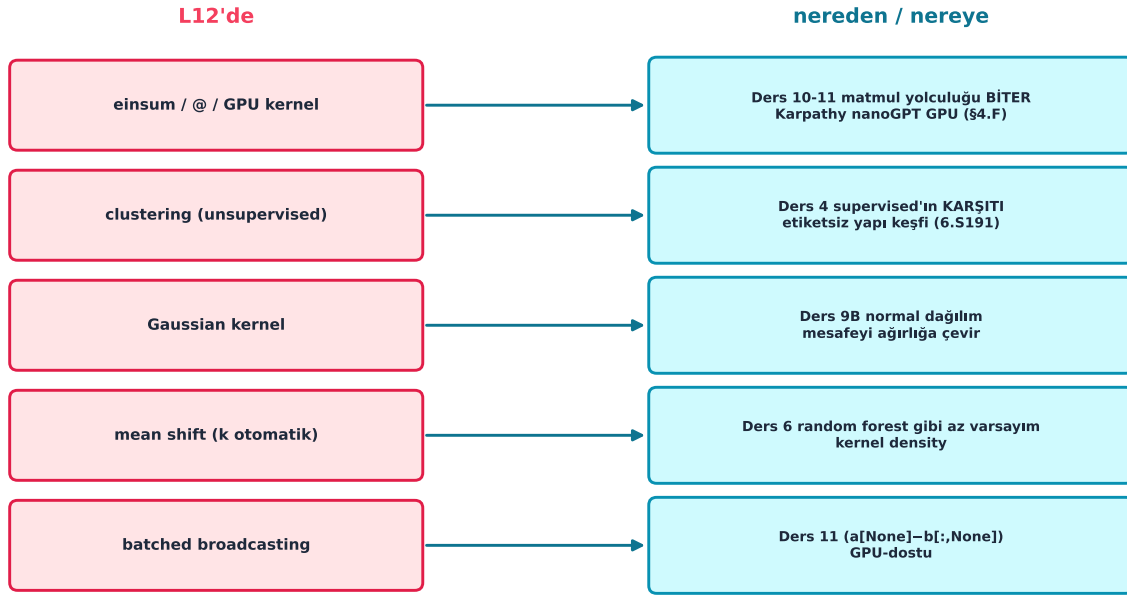
- **İleriye (Ders 13):** matmul + broadcasting artık hazır; Ders 13 bunlarla bir MLP’yi forward+backward sıfırdan kurar.
- **Sezgi:** Aynı “sıfırdan kur, sonra kullan” disiplini devam eder; mean shift gradyansız bir aradı, sıradaki parça (backprop) bütün derin öğrenmenin motorudur.

21.17 16. Kapanış

Ders 12 matris çarpımını bitirdi ($\text{einsum} \rightarrow @ \rightarrow \text{GPU kernel}$) ve ilk tam foundations algoritmasını kurdu: mean shift clustering. Etiketsiz veriyi, her noktayı komşularının Gaussian-ağırlıklı ortalamasına çekerek kümeledik — broadcasting sayesinde döngüsüz ve GPU’da.

Şekil 21.8 dersin sentezidir: solda L12’nin parçaları ($\text{einsum}/@/\text{GPU kernel}$, clustering, Gaussian kernel, mean shift, batched broadcasting), sağda her birinin nereden/nereye köprüsü — Ders 10-11 matmul yolculuğu biter

+ Karpathy nanoGPT GPU, Ders 4 supervised'ın karşıtı, Ders 9B normal dağılım, Ders 6 random forest gibi az varsayım, Ders 11 broadcasting.



Şekil 21.8: (sentez: L12 = matmul yolculuğu biter (einsum/@/GPU) + ilk unsupervised algoritma (mean shift); Ders 9B/11/6 köprüleri)

💡 Builder Notu — L12 Sentezi: İki Kol Birleşti

- **İleriye (Ders 13-14):** Sıradaki iki ders backpropagation'ı sıfırdan kurar — Karpathy micrograd'ının fast.ai karşılığı; matmul/broadcasting temeli hazır.
- **Tek cümle:** matmul yolculuğu einsum/@/GPU ile bitti; mean shift ilk unsupervised foundations algoritmasıydı, sıradaki parçalar (backprop, Learner, diffusion) aynı zincirde elle kurulacak.

21.18 Bu Dersin Özeti

1. **CLIP Interrogator** bir görüntüden yaklaşık prompt üretir (embedding'e en yakın metin parçaları); tam tersine çeviremez (bilgi kaybı) (**CLIP**).
2. **Einstein toplamı (einsum)** matmul'u 'ik,kj->ij' ile kompakt ifade eder; tekrarlanan indeks (k) çarpım+toplamı (**einsum**).
3. **@** en kısa ve hızlı matmul'dur (BLAS/cuBLAS); GPU'da bir **kernel** her hücreyi paralel hesaplar (**@** ve **hız** → **CUDA kernel**).
4. **Clustering** etiketsiz (unsupervised) veriyi benzer gruplara ayırır (**clustering**).
5. **Mean shift:** her noktayı yakın noktaların Gaussian-ağırlıklı ortalamasına doğru kaydır; tekrarla (**mean shift** → **döngü**).
6. **Gaussian kernel** mesafeyi ağırlığa çevirir; **bandwidth** “yakın” yarıçapını belirler (**Gaussian**).
7. Mean shift küme sayısını **otomatik** bulur (k-means'in aksine k gerekmez) (**kaç küme**).

8. **Batched + broadcasting** ile mesafeler döngüsüz, GPU’da paralel hesaplanır ([batched](#)).

! Tek Bir Cümle

Matris çarpımını einsum ve @ ile bitirip GPU’ya taşıdıktan sonra, etiketsiz veriyi her noktayı komşularının Gaussian-ağırlıklı ortalamasına çekerek kümeleyen mean shift’i sıfırdan kurduk — küme sayısını önceden bilmeden, broadcasting sayesinde döngüsüz.

21.19 Kontrol Soruları

i Soru 1: Einstein toplamında (‘ik,kj->ij’) tekrarlanan indeks (k) ne anlama gelir?

Cevap:

einsum notasyonunda her harf bir eksendir. Girdi matrisleri ik (5×784) ve kj (784×10), çıktı ij (5×10). **k harfi iki girdide tekrarlanıyor ama çıktıda yok** — bu iki şey demek: (1) ortak eksen (k) boyunca elemanlar **çarpılır**; (2) k çıktıda olmadığı için o eksen üzerinden **toplanır**. Yani ‘ik,kj->ij’ tam olarak matris çarpımıdır: her ij hücresi, k boyunca $a[i,k] \cdot b[k,j]$ çarpımlarının toplamıdır. einsum, çarpım-ve-topla işlemlerini tek bir kompakt string’le ifade eder. (Şekil 21.2 bu kuralı ve @ ile özdeşliğini gösterir.)

i Soru 2: Mean shift clustering nasıl çalışır? Tek bir güncelleme adımı ne yapar?

Cevap:

Mean shift, her veri noktasını yakınındaki noktaların **ağırlıklı ortalamasına** doğru kaydırır ve bunu birkaç kez tekrarlar; noktalar yoğun bölgelerin (kümelerin) merkezine toplanır. Bir güncelleme adımı: her nokta x için (1) tüm noktalara **mesafe** hesapla, (2) mesafeyi bir **Gaussian kernel** ile **ağırlığa** çevir (yakın=yüksek, uzak≈0), (3) x’i bu ağırlıklı ortalamaya taşı: $(\sum w_i \cdot x_i) / (\sum w_i)$. Broadcasting sayesinde `weight[:,None]*X` ile bu döngüsüz yapılır. Birkaç tur sonra her küme tek noktada toplanır. (Şekil 21.5 tek adımı, Şekil 21.6 tüm döngüyü gösterir.)

i Soru 3: Mean shift’in k-means’e göre avantajı nedir? bandwidth ne işe yarar?

Cevap:

k-means küme sayısını (k) **önceden** vermeyi gerektirir; mean shift ise küme sayısını **otomatik** bulur — noktalar yoğun bölgelere toplandığında kaç ayrı toplanma noktası oluştuğu küme sayısını verir. **bandwidth** (bant genişliği), Gaussian kernel’in “yakın” saydığı yarıçapı belirler: küçük bandwidth → çok sayıda küçük küme, büyük bandwidth → az sayıda büyük küme. Yani küme sayısını dolaylı olarak bandwidth kontrol eder (k yerine). Bu, veriyi keşfetmek için (kaç küme olduğunu bilmediğinde) avantajlıdır. (Şekil 21.7 üç bandwidth’in küme sayısını nasıl değiştirdiğini gösterir.)

i Soru 4: ‘@’ operatörünü artık neden güvenle kullanabiliyoruz? (builder bağlantısı)

Cevap:

Çünkü “from the foundations” kuralı gereği matris çarpımını **sıfırdan kurduk** ve nasıl çalıştığını

anladık: Ders 10’da üç döngü + numba, Ders 11’de broadcasting, Ders 12’de einsum. @’in altında ne olduğunu artık biliyoruz. Howard’ın kuralı: “bir şeyi bir kez sıfırdan kur, sonra hazır versiyonunu kullan.” a @ b PyTorch’un onlarca yıllık optimize lineer cebir kütüphanelerini (BLAS/cuBLAS) çağırır — bizim kodumuzdan çok daha hızlı. Builder açısından: mekanizmayı anladıktan sonra optimize aracı kullanmak hem doğru hem verimli; “kara kutu” değil, “anlaşılmış kutu”. (Şekil 21.3 @’in GPU’da nasıl paralelleştirdiğini gösterir.)

21.20 Egzersizler

Egzersiz 1 (Direkt uygulama). `torch.einsum('ik,kj->ij', a, b)` ile matmul yap; a @ b ve broadcasting versiyonuyla sonuçların aynı, hızların farklı olduğunu `%timeit` ile gör. (Şekil 21.2’i kendi sayınla yeniden üret — §2-3)

Egzersiz 2 (İki-aşamalı). 6 centroid’li sentetik veri üret, `meanshift` ile kümele; sonucu orijinal centroid’lerle karşılaştır. (Şekil 21.6’ı kendi verinde yeniden üret — §7, §11)

Egzersiz 3 (Edge case). `bandwidth`’i çok küçük (örn. 0.5) ve çok büyük (örn. 20) yap; küme sayısının nasıl değiştiğini gözlemler. (§13)

Egzersiz 4 (Kavramsal). Gaussian yerine `tri` (üçgen) kernel kullan; kümeleme sonucundaki farkı açıkla. (§12)

Egzersiz 5 (Sonraki dersin habercisi — Ders 13). Bir MLP’nin ileri geçişini (`lin` → `ReLU` → `lin`) matmul ile kur; çıktının şeklini ve neden `ReLU` gerektiğini düşün. (§15)

21.21 Sonraki: Ders 13 İçin Hazırlık

Ders 13: Geri Yayılım ve MLP (Backpropagation & MLP)

Ders 12 matmul’u bitirdi ve gradient’siz bir algoritma (mean shift) kurdu. Ders 13 sinir ağına döner: bir **MLP’yi sıfırdan** kurar (`lin` + `ReLU`), MSE loss’u tanımlar ve **backpropagation**’a başlar — Karpathy `micrograd`’ının `fast.ai` karşılığı.

Ana konular (Ders 13):

- Lineer katman + `ReLU` (matmul ile)
- MLP forward pass from scratch
- MSE loss from scratch
- Backpropagation’a giriş (chain rule)

⚠ Ders 13 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 5 — `einsum` + MLP forward).
- Mean shift’i kendi sentetik verinde çalıştır.
- Ana cümleyi tekrar oku: “matmul bitti; sıradaki backprop’u sıfırdan kuracağız.”

21.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
CLIP Interrogator	Görüntüden yaklaşık prompt üretme (embedding eşleştirme)	0:17
Einstein toplamı (einsum)	'ik,kj->ij'; tekrarlanan indeks çarpım+toplam	12:03
@ (matmul)	En kısa/hızlı matris çarpımı (BLAS/cuBLAS)	12:03
CUDA kernel	GPU'da paralel çalışan fonksiyon (her hücre bir iş parçacığı)	1:00:00
Clustering	Etiketsiz veriyi benzer gruplara ayırma (unsupervised)	8:46
Centroid	Küme merkezi (sentetik veri üretiminde)	9:21
Mean shift	Her noktayı komşuların ağırlıklı ortalamasına kaydırma	8:46
Gaussian kernel	Mesafeyi ağırlığa çevirir (yakın=yüksek)	9:21
Bandwidth	"Yakın" yarıçapı; küme sayısını dolaylı belirler	9:21
Ağırlıklı ortalama	$(\sum w_i \cdot x_i) / (\sum w_i)$; güncelleme adımı	9:21
Triangular kernel	Doğrusal azalan ağırlık (Gaussian alternatifi)	9:21
Batched mesafe	Broadcasting ile döngüsüz mesafe matrisi (GPU)	9:21

21.23 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: L12'nin Tanıdık Kökleri

Bu ders matmul yolculuğunu bitirir ve ilk unsupervised algoritmayı tanıdık kavramlara bağlar; köprülerin özeti:

1. **einsum** → çarpım-toplam işlemlerinin kompakt notasyonu; attention (Ders 24) gibi karmaşık işlemlerde güçlü ([einsum](#)).
2. **@ / GPU kernel** → matmul'un production hâli; Karpathy nanoGPT hız dersi (§4.F) GPU kernel'ı derinleştirir ([CUDA kernel](#)).
3. **Clustering (unsupervised)** → etiketsiz yapı keşfi; supervised öğrenmenin (Ders 4) karşıtı ([clustering](#)).
4. **Mean shift** → kernel density'ye dayalı; küme sayısını otomatik bulur (k-means'ten farkı) ([mean shift](#)).
5. **Gaussian kernel** → 9B normal dağılımı; mesafeyi ağırlığa çeviren temel araç ([Gaussian](#)).

6. **Batched broadcasting** → GPU-dostu paralel hesap; Ders 11'in doğrudan uygulaması ([batched](#)).

! Bu dersten tek bir şey alıp gideceksen

Matris çarpımı yolculuğunu bitirdik (einsum → @ → GPU) ve artık @'i güvenle kullanabiliriz çünkü altını biliyoruz. Sonra ilk tam foundations algoritmasını kurduk: mean shift, her noktayı komşularının Gaussian-ağırlıklı ortalamasına çekerek — küme sayısını önceden bilmeden, broadcasting sayesinde döngüsüz veriyi kümeledik.

22 Ders 13 — Geri Yayılım ve MLP (Backpropagation & MLP)

Part 2'nin beşinci dersi sinir ağının kalbine iner: bir MLP'yi (çok katmanlı algılayıcı) sıfırdan kurar — lineer katman + ReLU + lineer katman — MSE loss'u tanımlar ve backpropagation'ı elle hesaplar. Anahtar fikir tek cümleyle: bir MLP matris çarpımı + ReLU yığıdır; onu eğitmek, kaybın her parametreye göre türevini chain rule ile en dıştan içe katman katman geriye yaymaktır. Bu ders Karpathy micrograd'ının fast.ai karşılığıdır — manuel .g gradyanları = micrograd'ın _backward kapanışları; iki « from scratch » yolu burada buluşur.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 13: Backpropagation & MLP](https://course.fast.ai/lesson13) (~106 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 13
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/03_backprop](#)
- **Okuma süresi:** ~38 dk
-  **Karpathy köprüsü MERKEZİ:** Bu ders Karpathy micrograd'ının (Zero to Hero) fast.ai karşılığıdır — Howard'ın elle yazdığı .g gradyanları = micrograd'ın _backward kapanışları. İki “from scratch” yolu aynı yerde buluşur. Bu ders ETAP 4 (Temeller A) içindedir.

22.1 Bu Derste Ne Var?

Part 2'nin beşinci dersi sinir ağının kalbine iner: bir **MLP'yi (çok katmanlı algılayıcı)** sıfırdan kurar (lineer katman + ReLU), **MSE loss'u** tanımlar ve **backpropagation'a** başlar. Bu, Karpathy'nin micrograd'ının (Zero to Hero serisi) fast.ai karşılığıdır — iki “from scratch” yolu aynı yerde buluşur.

Üç temel fikir bu dersin omurgasını kurar:

1. **MLP = lineer + ReLU + lineer** — matris çarpımı (Ders 10-12) ile katmanlar, aralarında ReLU; “ReLU toplamı” istenen kadar karmaşık fonksiyonu kurar (Ders 3'ün somut hâli) ([ReLU toplamı](#) → [MLP mimari](#)).
2. **Loss ve gradient** — MSE ile hatayı ölç; her parametreyi hangi yönde değiştireceğini **gradient** (eğim) söyler ([MSE](#) → [gradyanlar](#)).
3. **Backpropagation = chain rule** — kaybın her parametreye göre türevini, zincir kuralıyla katman katman geriye yayarak hesaplama ([chain rule](#) → [forward_and_backward](#)).

“Welcome to Lesson 13, where we're gonna start talking about back propagation.” — Howard, 0:00

Şekil 34.1 bu yapıyı tek bir yol haritasında birleştirir: üstte MLP’nin ileri yayılımı (lineer katman 1 → ReLU → lineer katman 2 → çıktı), ardından MSE kaybı ve gradient; altta backpropagation = chain rule kolu (lin_grad + ReLU gradyanı), hepsi **Karpathy köprüsünde** birleşir — manuel .g = micrograd _backward.

💡 Builder Notu — İki « From Scratch » Yolu Aynı Yerde

- **Geriye (Karpathy — merkezî köprü):** Bu ders Karpathy micrograd’ının ta kendisidir: manuel .g gradyanları = micrograd’ın _backward kapanışları. Howard “from scratch”, Karpathy “from scratch” — iki ayrı yol, aynı yer.
- **Geriye (Ders 3/5/Calculus):** Ders 3’ün “ReLU toplamı” + Ders 5’in gradient descent’i; chain rule = Calculus zincir kuralı (3Blue1Brown).
- **Tek cümle:** Bir MLP, matris çarpımı + ReLU yığımıdır; onu eğitmek, kaybın gradyanını chain rule ile geriye yaymaktır.

22.2 1. Notebook 3 ve ReLU ile Esneklik

Howard 03_backprop notebook’una geçer. Önce Ders 3’ü hatırlatır: tek bir lineer model **düz bir çizgidir**, karmaşık şekilleri temsil edemez. Çözüm **ReLU** (rectified line): bir doğrunun negatif kısmını sıfıra kırpıp. Negatifi kırılmış bir doğru artık bir “bükülme noktası” olan parçalı bir fonksiyondur.

“If we just have a linear model, then it’s gonna look like this [a straight line].” — Howard, 4:28

💡 Builder Notu — ReLU: Düz Çizgiyi Bükmek

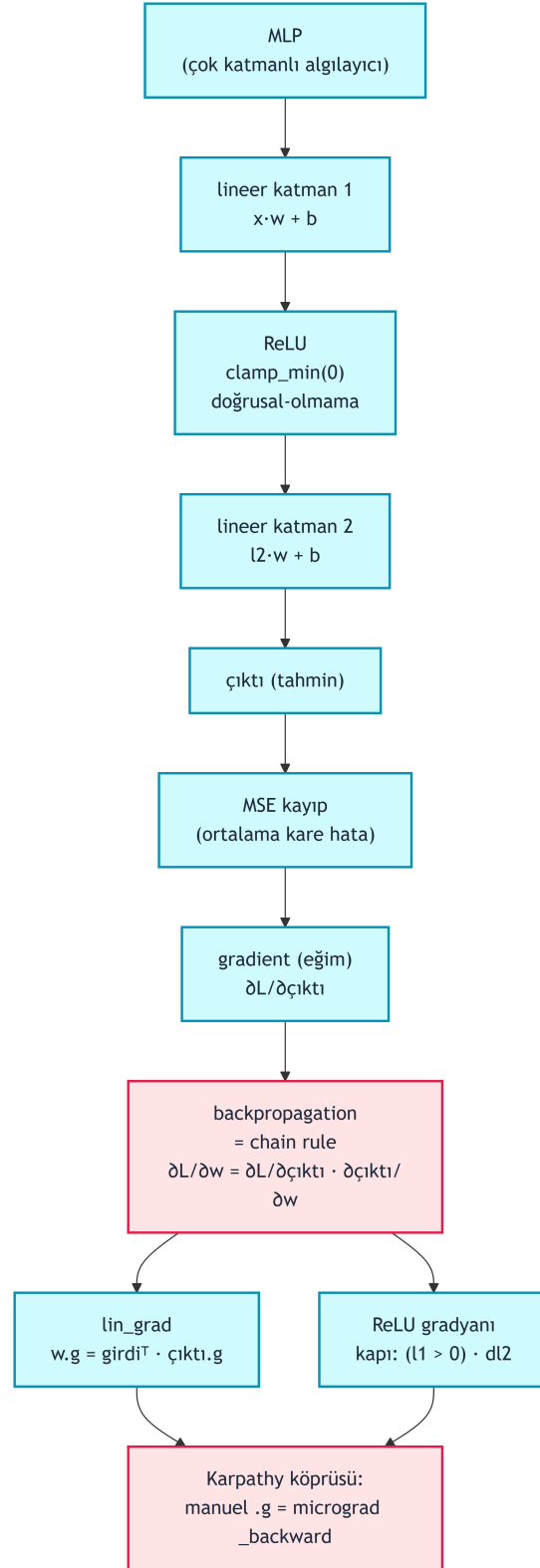
- **Geriye (Ders 3):** Ders 3’teki ReLU tanımı ($\max(0, x)$); burada onu bir MLP’nin yapı taşı olarak kullanacağız.
- **Sezgi:** Tek bir doğru karmaşık şekli kuramaz; ama doğruyu negatifte sıfıra kırıp bir “köşe” eklersek, bu köşeleri toplayarak istenen kadar karmaşık eğriler kurabiliriz.

22.3 2. Lineer Modeller ve Rectified Lines

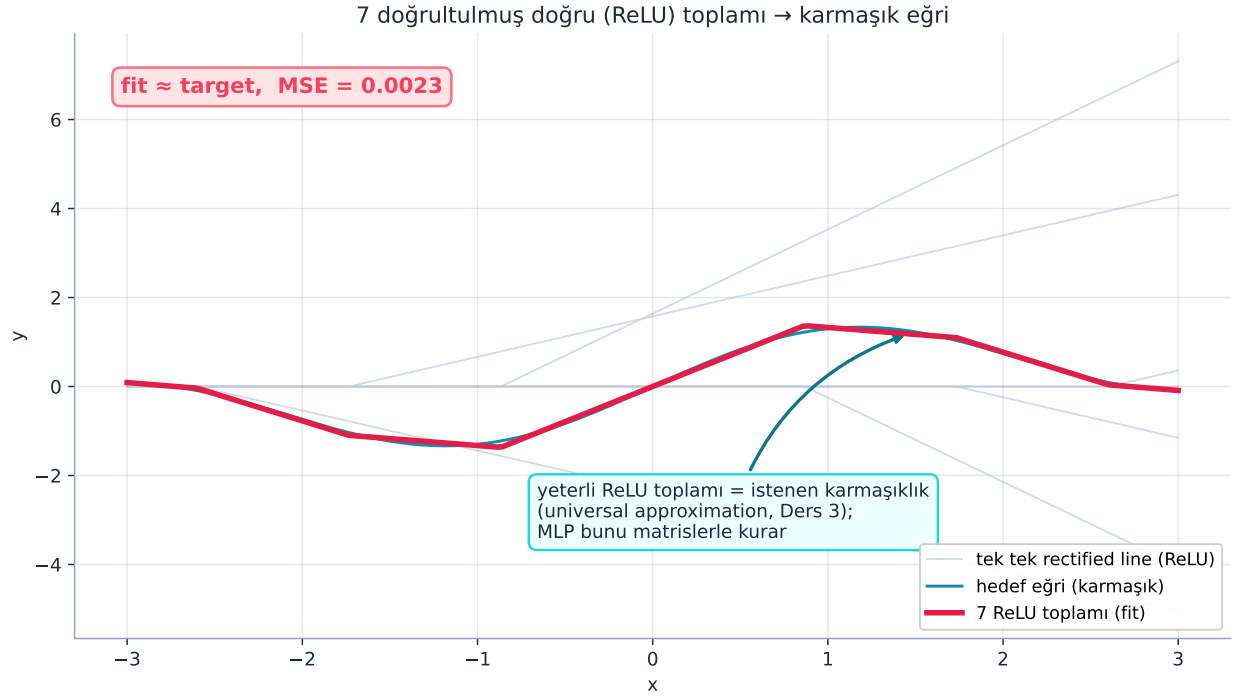
Howard bir diyagramla gösterir: tek doğru yetersiz, ama doğruları ReLU’dan geçirip **toplarsak** inişli-çıkışlı, karmaşık eğriler kurabiliriz. Yeterince rectified line toplamı, istenen kadar karmaşık bir şekil verir — bu, Ders 3’ün “universal approximation” fikrinin somut hâlidir.

“Using this approach, we could add up lots of these rectified lines.” — Howard, 6:03

Şekil 22.2 bunu **gerçek hesaplama** ile gösterir: tek tek soluk rectified line (ReLU) bileşenleri toplandığında, kalın rose eğri (ReLU toplamı / fit) ince cyan hedef eğriye oturur — lstsq ile fit edilen MSE çok küçüktür. Yani yeterli sayıda ReLU toplamı istenen karmaşıklığı kurar (universal approximation, Ders 3); MLP bunu matrislerle yapar.



Şekil 22.1: L13'ün kavram haritası: MLP (lin $\rightarrow$ ReLU $\rightarrow$ lin) ileri yayılımı, MSE kaybı, ve backpropagation = chain rule. Karpathy köprüsü: manuel .g = micrograd _backward (şematik).



Şekil 22.2: (gerçek hesaplama: 7 rectified line (ReLU) toplamı karmaşık eğriyi kurar — lstsq fit MSE küçük; Ders 3 'ReLU toplamı'nın somut hâli)

💡 Builder Notu — ReLU Toplamı: Universal Approximation

- **Geriye (Ders 3):** Bu, Ders 3'ün “ReLU toplamı = universal approximation” fikrinin görsel tekrarı; şimdi matrislerle kuracağız.
- **Sezgi:** Her ReLU bir “köşe” katar; köşeleri uygun ağırlıklarla toplayınca herhangi bir sürekli eğriyi istenen hassasiyetle yaklaşıklayabilirsin — sinir ağının gücü buradan gelir.

22.4 3. ReLU Toplamı ile Yüzeyler

Tek girdide rectified lines topluyorduk; çok girdide aynı fikir **rectified planes** (düzlemler) toplamına genişler. Böylece çok boyutlu girdiler için karmaşık karar yüzeyleri kurulur — bu bir sinir ağının yaptığı şeydir. Fikir birebir aynı; sadece doğrular düzlemlere dönüşür.

“But now they’re going to be kind of rectified planes. It’s going to be exactly the same [idea].” — Howard, 9:35

💡 Builder Notu — Rectified Planes: Aynı Fikir, Çok Boyut

- **Geriye (Ders 8):** Çok boyutlu girdi = matris; her ReLU bir doğrusal dönüşüm + kırpma (Ders 8 convolution’la akraba).
- **Sezgi:** Tek girdide “kırpılmış doğru”, çok girdide “kırpılmış düzlem” olur; düzlemleri toplayınca

karmaşık karar yüzeyleri (sinir ağının öğrendiği şey) çıkar. Şekil 22.2'daki tek-girdi sezgisi doğrudan çok boyuta genişler.

22.5 4. MLP from Scratch: Mimari

Howard MNIST için bir **MLP** kurar. Girdi: 50.000 satır $\times$ 784 sütun (28 $\times$ 28 piksel). Mimari: **784 girdi $\rightarrow$ nh gizli birim $\rightarrow$ 1 çıktı** (şimdilik tek çıktı; MSE basitliği için — gerçek 10-sınıf cross-entropy'ye Part 2'de geçilecek). Ağırlıklar rastgele başlar; eğitim onları gradyanlarla düzeltecek.

```
n, m = x_train.shape      # 50000, 784
nh = 50                   # gizli birim sayısı
w1 = torch.randn(m, nh); b1 = torch.zeros(nh)    # 784 -> 50
w2 = torch.randn(nh, 1); b2 = torch.zeros(1)     # 50 -> 1
```

💡 Builder Notu — MLP Mimarisi: İki Katman, Bir Gizli Kat

- **Geriye (Ders 5):** Ders 5'teki Titanic MLP'siyle aynı yapı (layer1, layer2); burada MNIST'te, daha sistematik.
- **Sezgi:** w1 girdiyi gizli kata (784 $\rightarrow$ 50) taşır, w2 gizli katı çıktıya (50 $\rightarrow$ 1); aradaki ReLU olmadan bu iki çarpım tek bir çarpıma çökerdi. Ağırlıkların randn ile rastgele başlaması, eğitimin onları kademeli düzeltmesi içindir.

22.6 5. Lineer Katman

İlk yapı taşı: **lineer katman** = girdi $\times$ ağırlık + bias. Tek satır, ama her sinir ağının temeli (Ders 10-12'de kurduğumuz matmul burada işler).

```
def lin(x, w, b): return x@w + b    # matris carpimi + bias
```

💡 Builder Notu — Lineer Katman: $x@w + b$

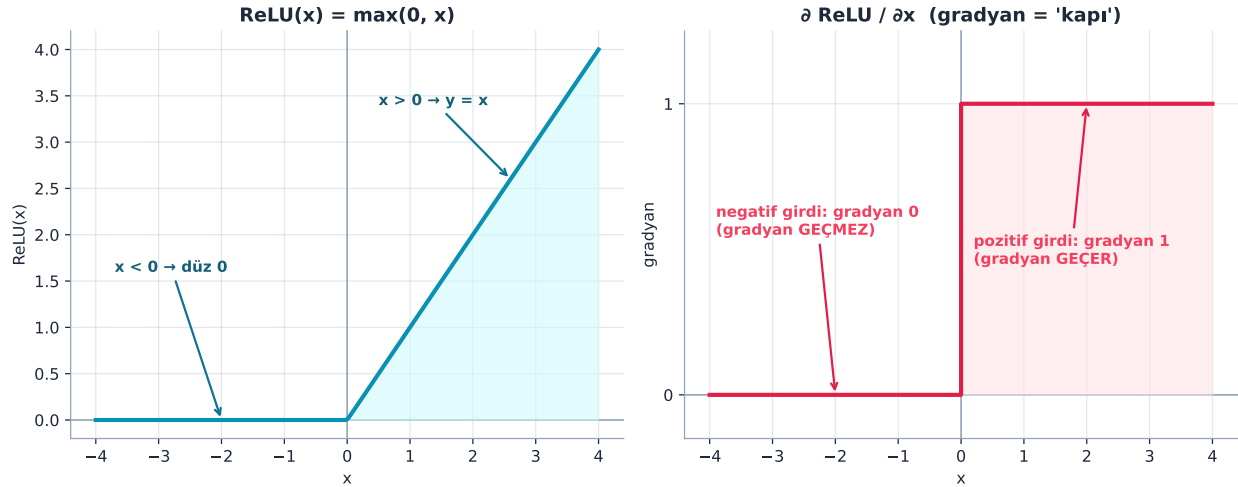
- **Geriye (Ders 5/10):** $x@w + b$ = Ders 5'teki lineer model; @ artık güvenle kullanılıyor (Ders 10-12'de sıfırdan kuruldu).
- **Sezgi:** Lineer katman her girdiyi ağırlıklı toplamına çevirir; “yeniden kur, sonra kullan” disiplini sayesinde @'in altını bildiğimiz için onu burada gönül rahatlığıyla kullanırız (Ders 12).

22.7 6. ReLU Katmanı

İkinci yapı taşı: **ReLU** = negatifleri sıfıra kırp. Bu, katmanlar arası **doğrusal-olmamayı** sağlar; olmazsa iki lineer katman tek lineer katmana çöker.

```
def relu(x): return x.clamp_min(0.) # negatifleri sifira cek
```

Şekil 22.3 ReLU'yu ve gradyanını **gerçek hesaplama** ile gösterir: solda $\text{ReLU}(x) = \max(0, x)$ ($x < 0$ 'da düz 0, $x > 0$ 'da $y=x$), sağda gradyanı — pozitif girdide 1, negatif girdide 0. Bu adım-fonksiyonu gradyanı ReLU'nun “kapı” davranışdır; backprop'ta ($l1 > 0$) maskesi olarak görünecek (§12).



Şekil 22.3: (gerçek hesaplama: $\text{ReLU}=\max(0,x)$ ve gradyanı — pozitifte 1, negatifte 0; ReLU'nun ‘kapı’ davranışı, backprop'ta ($l1 > 0$) maskesi)

💡 Builder Notu — ReLU: Doğrusal-Olmamanın Anahtarı

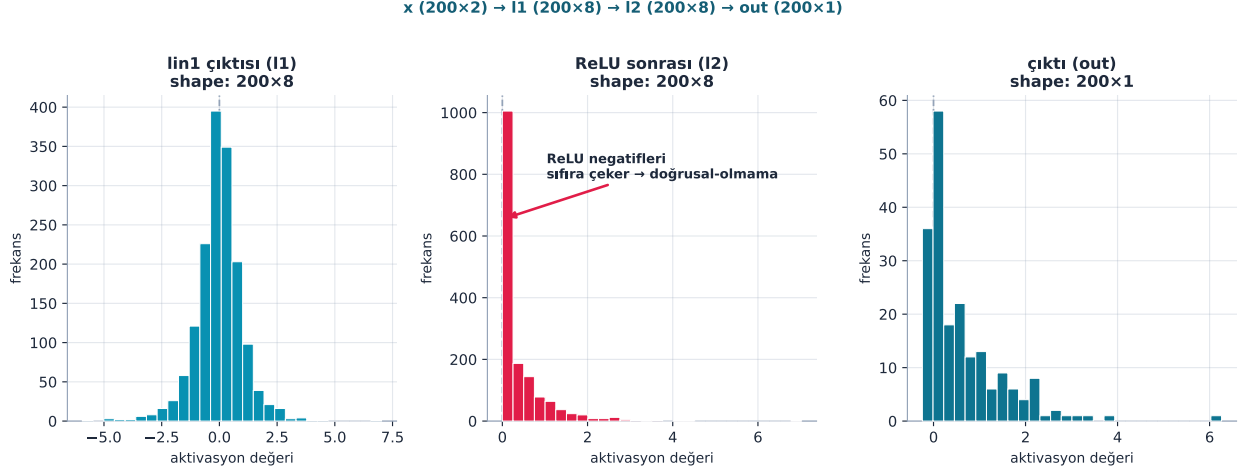
- **Geriye (Ders 3):** ReLU olmadan ağ “derin” olamaz — iki matris çarpımı tek bir matris çarpımına indirgenir (doğrusallık).
- **İleriye (§12):** ReLU'nun gradyanı (pozitifte 1, negatifte 0) backprop'un kritik parçasıdır: $l1.g = (l1 > 0).float() * l2.g$. Negatif girdili nöronlardan gradyan **geçmez** — “kapı” kapalıdır.

22.8 7. MLP Forward Pass

İki yapı taşını birleştir: **linear** → **ReLU** → **linear**. Bu, ileri geçiştir (forward pass) — girdiden tahmine.

```
def model(xb):
    l1 = lin(xb, w1, b1) # 1. lineer katman
    l2 = relu(l1)         # ReLU (dogrusal-olmama)
    return lin(l2, w2, b2) # 2. lineer katman -> cikti
```

Şekil 22.4 bu ileri geçişi **gerçek hesaplama** ile gösterir: $x \rightarrow l1 \rightarrow l2 \rightarrow \text{out}$ şekil akışı ve her aşamanın aktivasyon dağılımı. Kritik gözlem orta panelde: ReLU sonrası l2 dağılımında **sol yığılma** vardır — negatif aktivasyonlar 0'a kırılmıştır; işte doğrusal-olmama tam burada devreye girer.



Şekil 22.4: (gerçek hesaplama: MLP forward lin→ReLU→lin; ReLU negatif aktivasyonları 0’a kırpar — I2 dağılımında sol yığılma)

💡 Builder Notu — Forward Pass: lin → relu → lin

- **Geriye (Ders 5):** Ders 5’teki `calc_preds`’in aynısı; “matris çarpımı + ReLU + matris çarpımı” deseni.
- **Sezgi:** Forward pass sadece üç satır — girdiyi gizli kata taşı, doğrusal-olmama ekle, çıktıya taşı; ReLU’nun negatifleri kırpması (orta paneldeki sol yığılma) ağın “derinliğini” anlamlı kılan tek adımdır.

22.9 8. MSE Loss from Scratch

Tahminin ne kadar iyi olduğunu ölçmek için **MSE** (ortalama karesel hata). Şimdilik (sınıflandırma için ideal olmasa da, basitlik için) çıktı ile hedefin farkının karesinin ortalaması.

```
def mse(output, targ): return (output[:,0] - targ).pow(2).mean()
```

“Loss function from scratch — Mean Squared Error.” — Howard, 18:26

💡 Builder Notu — MSE: Basitlik İçin Şimdilik

- **Geriye (Ders 3/4):** MSE Ders 3’te tanımlandı; Ders 4’te cross-entropy’nin sınıflandırma için daha uygun olduğunu görmüştük (Part 2’de ona geçilecek).
- **Sezgi:** MSE rakam sınıflandırması için ideal değil (ona Part 2’de geçeceğiz), ama backprop mekanizmasını öğrenmek için en sade loss; önce mekanizmayı kur, doğru loss’a sonra geç.

22.10 9. Gradyanlar ve Backpropagation

Modeli eğitmek için her parametreyi hangi yönde değiştireceğimizi bilmeliyiz — **gradient** (eğim). Howard hatırlatır: gradyan, bir parametreyi azıcık değiştirince kaybın ne kadar değiştiğidir. Eğer bir ağırlığı artırınca kayıp artıyorsa, onu azaltırız.

“Gradients are slopes. If we can get the derivative of the loss with respect to one particular weight... if it makes the loss go up, then I want to do the opposite.” — Howard, 18:26

💡 Builder Notu — Gradient: Hangi Yöne Adım?

- **Geriye (Ders 3/5):** Gradient descent (Ders 3/5); fark: burada gradyanı **elle**, katman katman hesaplayacağız (PyTorch autograd yerine).
- **Sezgi:** Gradyan bir pusuladır: işareti hangi yöne adım atacağını söyler — kaybı artıran ağırlığı azalt, azaltanı artır. Backprop’un tüm derdi, bu pusulayı **her parametre** için verimli üretmektir.

22.11 10. Chain Rule: Zincir Kuralı

Bir ağ, fonksiyonların bileşkesidir (lin → relu → lin → loss). Kaybın bir ağırlığa göre türevini bulmak için **zincir kuralı** gerekir: bileşke fonksiyonun türevi, parçaların türevlerinin çarpımıdır.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w}$$

Burada $\hat{y}$ ağın çıktısı (tahmin); kaybın türevini, en dıştan (loss) en içe (ağırlık) doğru, her katmanın yerel türeviyle çarparak geriye yayarız. $\partial L / \partial w$ değerini doğrudan değil, parçaların çarpımı olarak biriktiririz (prose’da $\partial L / \partial \text{çıktı} \cdot \partial \text{çıktı} / \partial w$ olarak da okunur).

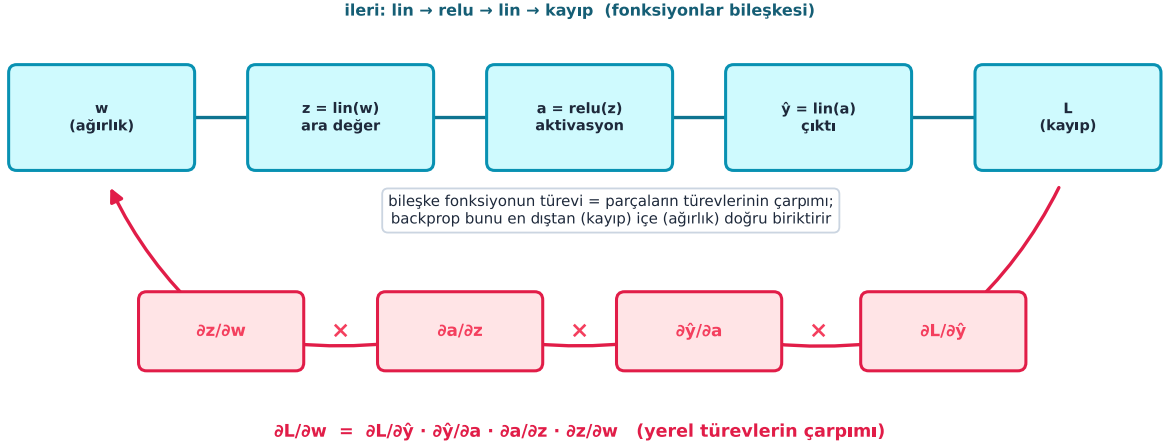
Şekil 22.5 bu fikri **şematik** gösterir: üstte ileri akış ($w \rightarrow z = \text{lin}(w) \rightarrow a = \text{relu}(z) \rightarrow \hat{y} = \text{lin}(a) \rightarrow L$), altta her bağlantının yerel türevi ($\partial L / \partial \hat{y}$, $\partial \hat{y} / \partial a$, $\partial a / \partial z$, $\partial z / \partial w$) ve bunların çarpımı. Bileşke fonksiyonun türevi = parçaların türevlerinin çarpımı; backprop bunu en dıştan (kayıp) içe (ağırlık) doğru biriktirir.

💡 Builder Notu — Chain Rule: Backprop’un Kalbi

- **Geriye (Calculus / Karpathy):** Chain rule, Calculus’un (3Blue1Brown) kalbi; Karpathy micrograd’ı bunu her düğümde `_backward` ile uygular. Howard 3Blue1Brown’u izlemeyi önerir.
- **Sezgi:** “Bileşke fonksiyonun türevi = parçaların türevlerinin çarpımı.” Backprop bu çarpımı sağdan (loss) sola (ilk ağırlık) doğru biriktirir; her katman kendi yerel türevini bilir, gerisi çarpmadır.

22.12 11. Lineer Katmanın Gradyanı

Backprop’un parçası: bir lineer katmanın gradyanı. Çıktının gradyanı (`out.g`) verildiğinde, girdiye, ağırlığa ve bias’a göre gradyanlar zincir kuralıyla hesaplanır. Howard her birini elle yazar.



Şekil 22.5: (şematik: chain rule — $\partial L / \partial w$ = yerel türevlerin çarpımı; backprop kaybın türevini katman katman geriye yayar)

```
def lin_grad(inp, out, w, b):
    inp.g = out.g @ w.t()                    # girdiye göre (zincir kuralı)
    w.g = (inp.unsqueeze(-1) * out.g.unsqueeze(1)).sum(0) # ağırlığa göre
    b.g = out.g.sum(0)                       # bias'a göre
```

💡 Builder Notu — lin_grad: Transpoze Gradyanı

- **Geriye (18.06):** Lineer katmanın girdiye göre türevi = ağırlık matrisinin transpozu ($w.t()$); bu, matris kalkülüsünün temel sonucu (18.06).
- **Sezgi:** Üç satır, üç gradyan: girdiye göre = $\text{out.g} @ w^T$ (gradyanı geri taşı), ağırlığa göre = $\text{girdi} \otimes \text{çıktı.g}$ (her ağırlığın katkısı), bias'a göre = çıktı.g 'nin toplamı. out.g 'yi alır, üç yöne dağıtır — micrograd'ın çarpma/toplama düğümünün matris hâli.

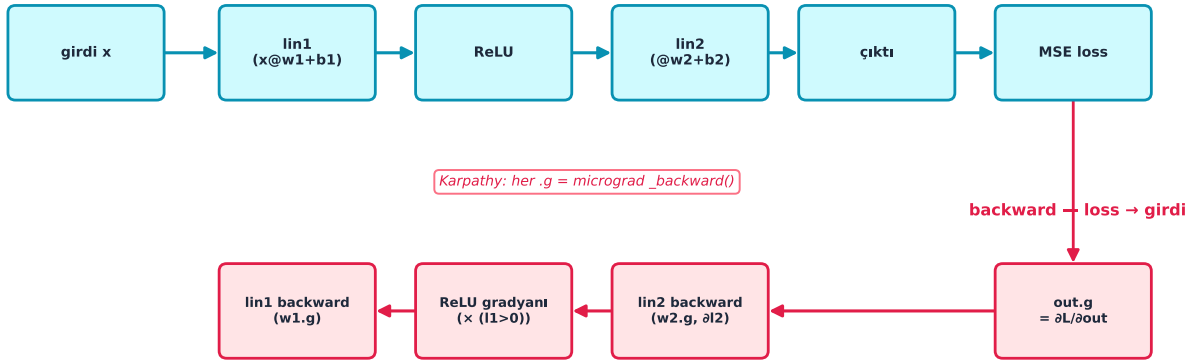
22.13 12. forward_and_backward: Tam Manuel Backprop

Howard tüm ileri ve geri geçişi tek fonksiyonda birleştirir. İleri: $\text{lin} \rightarrow \text{relu} \rightarrow \text{lin} \rightarrow \text{loss}$. Geri: en dıştan (loss 'un gradyanı) başlayıp her katmanın gradyanını zincir kuralıyla geriye yayar. **ReLU'nun gradyanı:** girdi pozitifse 1, değilse 0.

```
def forward_and_backward(inp, targ):
    # forward:
    l1 = lin(inp, w1, b1); l2 = relu(l1); out = lin(l2, w2, b2)
    diff = out[:,0] - targ; loss = diff.pow(2).mean()
    # backward (zincir kuralı, en distan ice):
    out.g = 2.*diff[:,None] / inp.shape[0] # loss'un ciktiya gore turevi
    lin_grad(l2, out, w2, b2)                # 2. katman geri
    l1.g = (l1>0).float() * l2.g           # ReLU gradyanı
    lin_grad(inp, l1, w1, b1)               # 1. katman geri
```

Şekil 22.6 bu tam akışı **şematik** gösterir (**FLAGSHIP**): üstte forward geçiş (girdi → lin1 → ReLU → lin2 → çıktı → MSE loss, cyan soldan sağa), altta manuel backward (loss → girdi, rose **sağdan sola**) — her backward kutusu yukarıdaki forward kutusunun altında hizalı. out.g’den başlar, lin2 backward → ReLU gradyanı ($\times (11>0)$) → lin1 backward ile geriye yayar. Ortadaki not Karpthy köprüsünü işaret eder: her .g = micrograd_backward().

forward — girdi → loss



chain rule: en dıştan içe, her katman yerel türeviyle çarpılır

Şekil 22.6: (şematik: MLP forward (girdi→loss) + manuel backward (loss→girdi); chain rule’u katman katman geriye yay — Karpthy micrograd’ının fast.ai hâli)

💡 Builder Notu — forward_and_backward: Her Satır Bir _backward

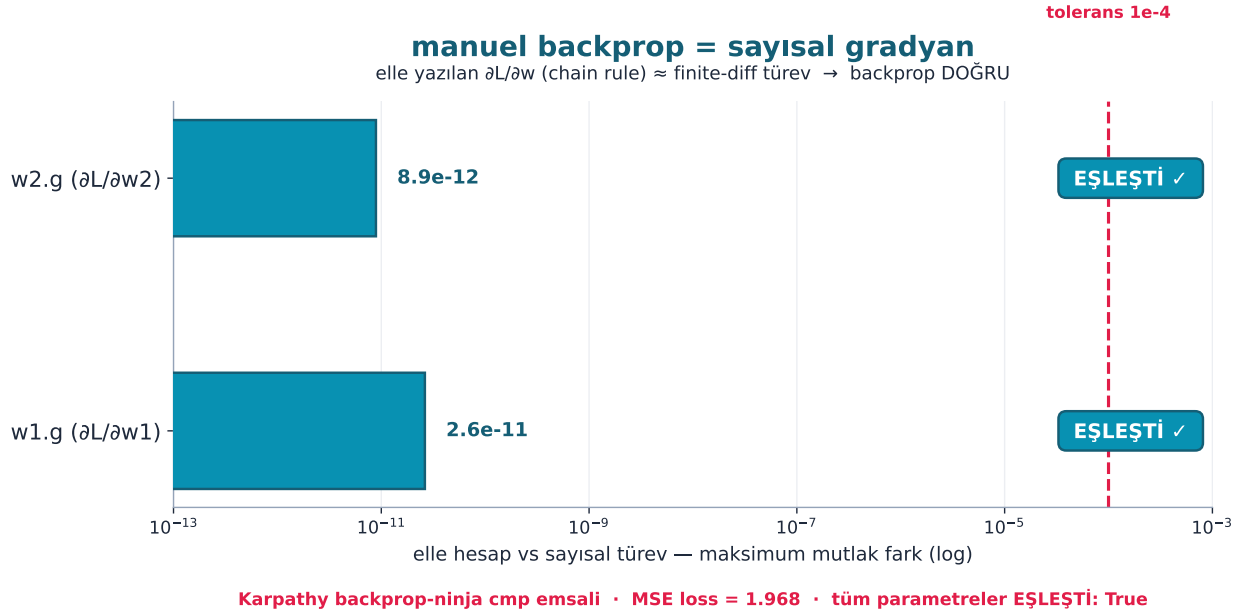
- **Geriye (Karpthy):** Bu, micrograd’ın backward()’ının tam karşılığı; her satır bir _backward adımı. out.g, l1.g = her düğümün gradyanı.
- **Sezgi:** out.g = $2 \cdot \text{diff} / n$ MSE’nin türevidir (en dış katman); sonra lin_grad ile katman 2’yi, (11>0) maskesiyle ReLU’yu, tekrar lin_grad ile katman 1’i geriye yayar. Backward kodu forward kodunun **tersten** okunuşudur — her ileri işlemin bir geri eşi vardır.

22.14 13. Karpthy Köprüsü

Bu ders, fast.ai ile Karpthy serisinin **en net bulunduğu** yerdir. Karpthy micrograd’da her işlem için bir _backward kapanışı yazar ve topolojik sırayla çağırır; Howard burada her katman için elle .g gradyanı hesaplar. İki yöntem aynı şeyin iki ifadesidir: **chain rule’u katman katman geriye uygulamak**.

“That’s how what’s happening in our code is mapping to the math.” — Howard, 6:28

Şekil 22.7 bunu **gerçek hesaplama** ile kanıtlar (**FLAGSHIP**): elle yazılan backprop gradyanları, sayısal türevle (finite-difference) **~1e-11 hassasiyetle EŞLEŞİR** — yani her .g doğru. Log-skala barlar tüm parametreler için maksimum mutlak farkı 1e-4 tolerans çizgisinin çok solunda gösterir. Bu, Karpthy’nin backprop-ninja dersindeki cmp emsalidir: elle yazdığın gradyanı referans gradyanla karşılaştır, eşleşiyorsa backprop doğru.



Şekil 22.7: (gerçek hesaplama: elle yazılan backprop gradyanları sayısal türevle $\sim 1e-11$ hassasiyetle EŞLEŞİR — backprop doğru; Karpthy micrograd cmp emsali)

💡 Builder Notu — Karpthy Köprüsü: İki Yol, Aynı Matematik

- **Geriye (Karpthy 1-5):** Karpthy Ders 1 (micrograd) + Ders 5 (backprop ninja, elle gradyan) = bu dersin birebir akrabası; ikisini birlikte yapan backprop’u asla unutmaz. Howard’ın manuel .g’si = Karpthy’nin _backward kapanışı; aynı chain rule, farklı kod stili.
- **İleriye:** Bir kez elle kurduktan sonra loss.backward() (PyTorch autograd) güvenle kullanılır — altında bu var. Şekil 22.7 elle hesabın sayısal türevle eşleştiğini gösterdiği an, autograd artık “büyü” değil, anladığın bir araçtır.

22.15 14. Refactoring’e Doğru

Howard manuel backprop’u temizlemeye hazırlanır: tekrar eden “ileri sonucu sakla, geri gradyanı hesapla” desenini **sınıflara sarmak** (Ders 14). Her katman kendi forward ve backward’ını bilen bir nesne olacak — PyTorch’un nn.Module’üne giden yol.

💡 Builder Notu — Refactoring: Her Katman Bir Sınıf

- **İleriye (Ders 14):** “Her katman = forward + backward bilen sınıf” deseni; nn.Module’ün özü.
- **Sezgi:** Şu an forward_and_backward tek devasa fonksiyon; aynı desen (“ileriye sakla, geriye hesapla”) her katmanda tekrarlanıyor. Onu sınıflara sarınca her katman bağımsız, yeniden kullanılabilir bir nesne olur — bu, nn.Module’ün doğuşudur (Ders 14).

22.16 15. Neden Elle Kurmak?

Howard'ın “from the foundations” felsefesi burada zirve yapar: backprop'u **elle kurmak**, sonra PyTorch'un otomatik türevini (autograd) güvenle kullanmamızı sağlar. Mekanizmayı anlamadan kütüphaneye güvenmek, hata ayıklamayı imkânsız kılar.

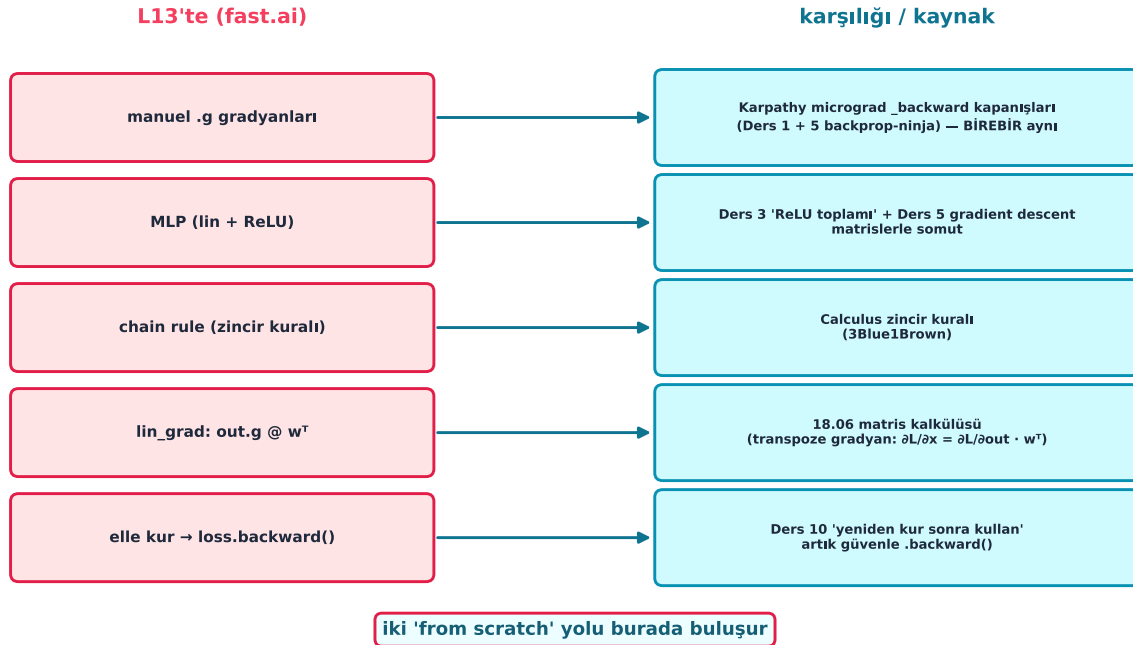
💡 Builder Notu — Neden Elle: Kara Kutu Olmasın

- **Geriye (Ders 10):** “Bir kez sıfırdan kur, sonra kullan” kuralı; backprop bunun en kritik örneği.
- **Sezgi:** `loss.backward()` bir kara kutu olarak kalırsa, gradyan patlaması/sönmesi gibi sorunları teşhis edemezsin. Bir kez elle kurarsan, altında tam olarak ne olduğunu bilirsin — hata ayıklayabilir, özel katmanlar yazabilirsin.

22.17 16. Kapanış

Ders 13, sinir ağının kalbini sıfırdan kurdu: **MLP** (lin + ReLU + lin), **MSE loss** ve **manuel backpropagation** (chain rule ile katman katman geriye gradyan). Bu, Karpathy micrograd'ının fast.ai karşılığı — iki “from scratch” yolu burada buluşur. Ders 14 bunu `nn.Module`'e refactor edecek.

Şekil 22.8 dersin sentezidir (**Karpathy köprüsü merkezde**): solda L13'ün parçaları (manuel `.g` gradyanları, MLP, chain rule, `lin_grad`, “elle kur → `loss.backward()`”), sağda her birinin kaynağı/karşılığı — manuel `.g` = Karpathy `micrograd_backward` kapanışları (Ders 1 + 5 backprop-ninja, **BİREBİR** aynı), MLP = Ders 3 + Ders 5, chain rule = Calculus, `lin_grad` = 18.06 matris kalkülüsü, “elle kur → kullan” = Ders 10. Ortada vurgu: **iki “from scratch” yolu burada buluşur**.



Şekil 22.8: (sentez: L13 = Karpathy micrograd'ının fast.ai karşılığı; manuel `.g` = `_backward`, MLP = Ders 3 + 5, chain rule = Calculus)

💡 Builder Notu — L13 Sentezi: Backprop'un Doğuşu

- **İleriye (Ders 14):** Manuel `.g` gradyanları $\rightarrow$ `forward/backward` sınıfları $\rightarrow$ `nn.Module`; aynı `backprop`, daha temiz kod.
- **Tek cümle:** Bir MLP, matris çarpımı + ReLU yığınıdır; onu eğitmek, kaybın gradyanını `chain rule` ile en dıştan içe geriye yaymaktır — ve bunu bir kez elle kurmak, Karpathy'nin `micrograd`'ıyla aynı şeyi öğretir.

22.18 Bu Dersin Özeti

1. Tek lineer model düz çizgidir; **ReLU toplamı** karmaşık şekiller (ve çok boyutta yüzeyler) kurar (Ders 3'ün somut hâli) ([ReLU toplamı](#)).
2. **MLP** = lineer katman ($x@w+b$) $\rightarrow$ **ReLU** (`clamp_min(0)`) $\rightarrow$ lineer katman; ReLU olmadan ağ tek lineer katmana çöker (**MLP** $\rightarrow$ [forward](#)).
3. **MSE loss** tahmin-hedef farkının karesinin ortalaması; modelin iyiliğini ölçer ([MSE](#)).
4. **Gradient** her parametreyi hangi yönde değiştireceğini söyler (eğim); artıran ağırlığı azaltırız ([gradyanlar](#)).
5. **Backpropagation = chain rule:** $\partial L / \partial w = (\partial L / \partial \text{çıktı}) \cdot (\partial \text{çıktı} / \partial w)$; kaybın türevini en dıştan içe katman katman yayma ([chain rule](#)).
6. **lin_grad** lineer katmanın girdi/ağırlık/bias gradyanlarını zincir kuralıyla verir (girdi gradyanı = `out.g @ wT`) ([lin_grad](#)).
7. **forward_and_backward** tüm ileri+geri geçişi elle kurar; ReLU gradyanı: pozitifse 1, değilse 0 ([forward_and_backward](#)).
8. Bu, Karpathy `micrograd`'ının `_backward` kapanışlarının `fast.ai` karşılığıdır; elle kurunca `loss.backward()` güvenle kullanılır ([Karpathy köprüsü](#)).

❗ Tek Bir Cümle

Bir MLP, matris çarpımı + ReLU katmanlarının yığınıdır; onu eğitmek, kaybın her parametreye göre türevini `chain rule` ile en dıştan içe katman katman geriye yaymaktır (backpropagation) — ve bunu bir kez elle kurmak, Karpathy'nin `micrograd`'ıyla aynı şeyi öğretir.

22.19 Kontrol Soruları

i Soru 1: Bir MLP'nin forward pass'i hangi adımlardan oluşur? ReLU neden zorunludur?

Cevap:

Forward pass: girdi $\rightarrow$ **lineer katman** ($x@w_1+b_1$) $\rightarrow$ **ReLU** (`clamp_min(0)`) $\rightarrow$ **lineer katman** ($@w_2+b_2$) $\rightarrow$ çıktı. ReLU zorunludur çünkü **doğrusal-olmamayı** sağlar: iki lineer katmanı arka arkaya koyarsan (ReLU olmadan), matematiksel olarak tek bir lineer katmana indirgenir (iki matris çarpımı = bir matris çarpımı). ReLU araya girince ağ gerçekten “derin” olur ve karmaşık (doğrusal-olmayan) fonksiyonları temsil edebilir — Ders 3'teki “ReLU toplamı” fikri. (Şekil 22.4 ileri geçişi, Şekil 22.2 ReLU toplamının karmaşık eğri kurmasını gösterir.)

i Soru 2: Backpropagation chain rule’u nasıl kullanır? $\partial L/\partial w$ neyin çarpımıdır?

Cevap:

Bir ağ fonksiyonların bileşkesidir ($\text{lin} \rightarrow \text{relu} \rightarrow \text{lin} \rightarrow \text{loss}$). Bir ağırlığın kayba etkisini bulmak için **zincir kuralı** gerekir: $\partial L/\partial w = (\partial L/\partial \text{çıktı}) \cdot (\partial \text{çıktı}/\partial w)$. Yani kaybın o ağırlığa göre türevi = “kaybın katman çıktısına göre türevi” çarpı “katman çıktısının ağırlığa göre türevi”. Backprop bunu en dıştan (loss) en içe (ilk katman ağırlığı) doğru yapar: önce loss’un gradyanı (`out.g`), sonra her katmanın yerel türeviyle çarpılarak geriye yayılır (`lin_grad`, ReLU gradyanı). Her adım bir çarpım; zincir boyunca biriktirilir. (Şekil 22.5 yerel türevlerin çarpımını, Şekil 22.6 tam geri akışı gösterir.)

i Soru 3: ReLU’nun gradyanı nedir ve kodda nasıl ifade edilir?

Cevap:

$\text{ReLU}(x) = \max(0, x)$. Türevi basittir: girdi **pozitifse 1, negatifse 0** (sıfırda tanımsız ama pratikte 0 alınır). Backprop’ta, ReLU’dan gelen gradyanı bir önceki katmana geçirirken bu türevle çarpılırız: `11.g = (11>0).float() * 12.g`. Yani `(11>0).float()` pozitif girdiler için 1, negatifler için 0 maskesi üretir; bu maske, sonraki katmanın gradyanını (`12.g`) süzer — negatif girdili nöronlardan gradyan geçmez. Bu, ReLU’nun “kapı” davranışındır. (Şekil 22.3 gradyanın adım-fonksiyonu/kapı şeklini gösterir.)

i Soru 4: Bu ders Karpathy’nin micrograd’ıyla nasıl bağlanır? Neden backprop’u elle kuruyoruz? (builder bağlantısı)

Cevap:

Karpathy micrograd’da her işlem için bir `_backward` kapanışı yazar ve topolojik sırayla çağırarak gradyanları geriye yayar; Howard burada her katman için elle `.g` gradyanı hesaplar (`lin_grad`, ReLU gradyanı). İkisi de **chain rule’u katman katman geriye uygulamanın** farklı ifadeleridir — aynı matematik, farklı kod stili. Backprop’u elle kurarız çünkü: (1) “from the foundations” kuralı (mekanizmayı bir kez anla); (2) `loss.backward()` (PyTorch autograd) bir kara kutu olmasın — altında tam olarak bunun olduğunu bilsek hata ayıklayabilir, özel katmanlar yazabiliriz. Builder açısından: backprop’u bir kez elle kuran, onu asla “büyü” olarak görmez. (Şekil 22.7 elle hesabın sayısal türevle $\sim 1e-11$ eşleştiğini, Şekil 22.8 iki “from scratch” yolunun buluşmasını gösterir.)

22.20 Egzersizler

Egzersiz 1 (Direkt uygulama). MNIST için `lin`, `relu`, `model` ve `mse`’yi yaz; rastgele ağırlıklarla bir forward pass yapıp çıktı şeklini ve loss’u gözle. (Şekil 22.4’ı kendi sayınla yeniden üret — §5-8)

Egzersiz 2 (İki-aşamalı). `forward_and_backward`’ı kur; elle hesapladığın gradyanları PyTorch’un `requires_grad_()` + `backward()` sonucuyla karşılaştır (aynı çıkmalı). (Şekil 22.7’i kendi modelinde yeniden üret — §12, §13)

Egzersiz 3 (Edge case). ReLU’yu çıkar (sadece $\text{lin} \rightarrow \text{lin}$); modelin neden tek lineer katmana çöktüğünü ve karmaşık şekilleri öğrenemediğini açıkla. (§6)

Egzersiz 4 (Kavramsal). Bir lineer katmanın girdiye göre gradyanının neden `out.g @ w.t()` olduğunu (matris kalkülüsü) elle bir örnekle türet. (§11)

Egzersiz 5 (Sonraki dersin habercisi — Ders 14). Her katmanı forward ve backward metodu olan bir sınıfa nasıl dönüştürebileceğini düşün (`nn.Module`'e giden yol). (§14)

22.21 Sonraki: Ders 14 İçin Hazırlık

Ders 14: Geri Yayılım (Backpropagation devamı)

Ders 13 manuel backprop'u kurdu. Ders 14 önce backprop matematiğini ve kodunu gözden geçirir (chain rule'u math ile koda eşler), sonra kodu **refactor** eder: her katmanı forward/backward bilen bir sınıfa, oradan PyTorch'un `nn.Module`'üne.

Ana konular (Ders 14):

- Backprop math ve kod eşlemesi (chain rule)
- Katmanları sınıflara refactor etme
- Kendi `nn.Module`'ümüzü kurma
- PyTorch `nn.Module`'e geçiş



Ders 14 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 2 ve 4 — manuel backprop + matris kalkülüsü).
- 3Blue1Brown'ın zincir kuralı / backprop videolarını izle (Howard'ın önerisi).
- Ana cümleyi tekrar oku: “Backprop = chain rule'u katman katman geriye yaymak.”

22.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
ReLU toplamı	Rectified line/plane toplamı; karmaşık şekiller	6:03
MLP	lin $\rightarrow$ ReLU $\rightarrow$ lin; çok katmanlı algılayıcı	10:20
Lineer katman (lin)	$x@w + b$; matris çarpımı + bias	10:20
ReLU	<code>clamp_min(0)</code> ; doğrusal-olmama	2:54
Forward pass	Girdiden tahmine: lin $\rightarrow$ relu $\rightarrow$ lin	10:20
MSE loss	$(\text{çıktı} - \text{hedef})^2$ ortalaması	18:26
Gradient	Eğim; parametreyi hangi yönde değiştirmeli	18:26
Chain rule	$\partial L / \partial w = (\partial L / \partial \text{çıktı}) \cdot (\partial \text{çıktı} / \partial w)$	9:35
Backpropagation	Chain rule'u en dıştan içe katman katman yayma	18:26
lin_grad	Lineer katman gradyanı (girdi = <code>out.g @ w.T</code>)	18:26
ReLU gradyanı	Pozitifse 1, değilse 0 <code>((1 > 0).float())</code>	18:26

Kavram	Tanım	Howard'da
Karpathy köprüsü	Manuel <code>.g = micrograd</code> <code>_backward</code> kapanışı	—

22.23 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: L13'ün Tanıdık Kökleri

Bu ders sinir ağının kalbini sıfırdan kurar ve Karpathy serisiyle en net bulunduğu yerdir; köprülerin özeti:

1. **MLP (lin+ReLU)** → Ders 3'ün “ReLU toplamı”nın matrislerle somut hâli; her ağın temeli ([MLP](#)).
2. **Backprop = chain rule** → Calculus zincir kuralı (3Blue1Brown); Karpathy micrograd'ının özü ([chain rule](#)).
3. **Manuel gradyan** → `loss.backward()`'in altında ne olduğu; bir kez kur, sonra autograd'ı güvenle kullan ([neden elle](#)).
4. **lin_grad (out.g @ w^T)** → matris kalkülüsü (18.06); lineer katmanın transpoze gradyanı ([lin_grad](#)).
5. **ReLU gradyanı (kapı)** → pozitif girdiden gradyan geçer, negatiften geçmez ([ReLU katmanı](#)).
6. **Karpathy köprüsü** → fast.ai “from scratch” + Karpathy “from scratch” aynı yerde; backprop'u unutmadan için ikisi ([Karpathy köprüsü](#)).

❗ Bu dersten tek bir şey alıp gideceksen

Bir sinir ağı, matris çarpımı + ReLU katmanlarından ibarettir ve “öğrenmesi”, kaybın gradyanını chain rule ile en dıştan içe katman katman geriye yaymaktır (backpropagation). Howard bunu fast.ai'de, Karpathy micrograd'da elle kurar — iki yol aynı yere çıkar. Bir kez elle kurarsan, `loss.backward()` artık büyü değil, anladığın bir araçtır.

23 Ders 14 — Geri Yayılım: nn.Module’ü Sıfırdan (Backpropagation)

Part 2’nin altıncı dersi backpropagation’ı tamamlar ve temizler. Önce Ders 13’ün manuel backprop’unu chain rule matematiğine satır satır eşler, sonra kodu adım adım refactor eder: tekrar eden ‘girdiyi sakla, gradyanı hesapla’ desenini önce katman sınıflarına, sonra bir Module taban sınıfına, en sonunda PyTorch’un nn.Module’üne taşır. Bu, ‘kendi PyTorch’unu sıfırdan kur’ yolculuğunun zirvesidir — ve Karpathy köprüsünü tamamlar: bizim Module taban sınıfımız, micrograd’ın Neuron/Layer/MLP Module’üyle aynı yapıdır. Tek cümleyle: manuel backprop’u her katmanı forward/backward bilen sınıflara refactor ederek PyTorch’un nn.Module’ünü yeniden kurduk; tek fark, PyTorch’un backward’ı autograd ile otomatik yapması — ama altında tam olarak Ders 13’ün chain rule’u var.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 14: Backpropagation](https://course.fast.ai/lesson-14-backpropagation) (~109 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 14
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/03_backprop](#)
- **Okuma süresi:** ~38 dk
-  **Karpathy köprüsü TAMAMLANIR:** Bu ders, Ders 13’te merkeze aldığımız Karpathy köprüsünü zirveye taşır. Karpathy micrograd’ın sonunda bir Module taban sınıfı (Neuron/Layer/MLP) kurar; Howard burada aynı Module desenini kurup PyTorch nn.Module’üne bağlar. İki “from scratch” yolu nn.Module’de **buluşur**. Bu ders ETAP 4 (Temeller A) finalidir.

23.1 Bu Derste Ne Var?

Part 2’nin altıncı dersi backpropagation’ı **tamamlar ve temizler**. Önce Ders 13’ün manuel backprop’unu gözden geçirir (chain rule’u matematikle koda eşler), sonra kodu adım adım **refactor** eder: tekrar eden deseni sınıflara, oradan bir Module taban sınıfına, en sonunda PyTorch’un nn.Module’üne. Bu, “kendi PyTorch’unu sıfırdan kurma” yolculuğunun **zirvesidir**.

Üç temel fikir bu dersin omurgasını kurar:

1. **Chain rule** ↔ **kod** — Ders 13’ün manuel .g gradyanlarının her satırı, zincir kuralının bir parçasıdır; math ile kod birebir eşleşir ([chain rule eşlemesi](#) → [refactor](#)).
2. **Refactoring = soyutlama** — her katmanı forward + backward bilen bir sınıfa sarmak; tekrar eden “girdiyi sakla, gradyanı hesapla” deseni bir Module taban sınıfında toplanır ([katman sınıfları](#) → [Module taban](#)).

3. **nn.Module’e varış** — kendi Module’ümüz, PyTorch’un nn.Module’üyle aynı yapıdadır; “yeniden kur, sonra kullan” kuralıyla artık onu güvenle kullanırız (nn.Module → [Karpathy köprüsü](#)).

“We had a lesson last time talking about calculus and how we implement the chain rule.” — Howard, 0:36

Şekil 34.1 bu yolculuğu tek bir yol haritasında birleştirir: manuel .g gradyanlarından (Ders 13) katman sınıflarına, Model’e, Module taban sınıfına ve nihayet PyTorch nn.Module’üne (backward’ı **otomatik** yapar). Karpathy köprüsü tamamlanır: nn.Module = micrograd Neuron/Layer/MLP.

💡 Builder Notu — İki « From Scratch » Yolu nn.Module’de Buluşur

- **Geriye (Karpathy / Calculus):** Module taban sınıfı = Karpathy micrograd’ının Module’ü; chain rule = Calculus (3Blue1Brown). İki “from scratch” yolu nn.Module’de buluşur — Ders 13’te açtığımız köprü burada **kapanır**.
- **İleriye (Ders 15-18):** nn.Module hazır olunca, Learner (Ders 16), init/norm (Ders 17), SGD (Ders 18) bunun üstüne kurulur.
- **Tek cümle:** Manuel backprop’u her katmanı forward/backward bilen sınıflara refactor ederek PyTorch’un nn.Module’ünü sıfırdan yeniden kurduk.

23.2 1. Ders 13’ün Gözden Geçirilmesi

Howard Ders 13’ün manuel backprop kodunu ve matematiğini tekrar eder. Amaç: chain rule’un her parçasının kodda nereye karşılık geldiğini netleştirmek. Ders 13’te kurduğumuz forward_and_backward (lin → relu → lin + geri gradyanlar) burada satır satır matematiğe bağlanır.

“Review of code and math from Lesson 13.” — Howard, 0:36

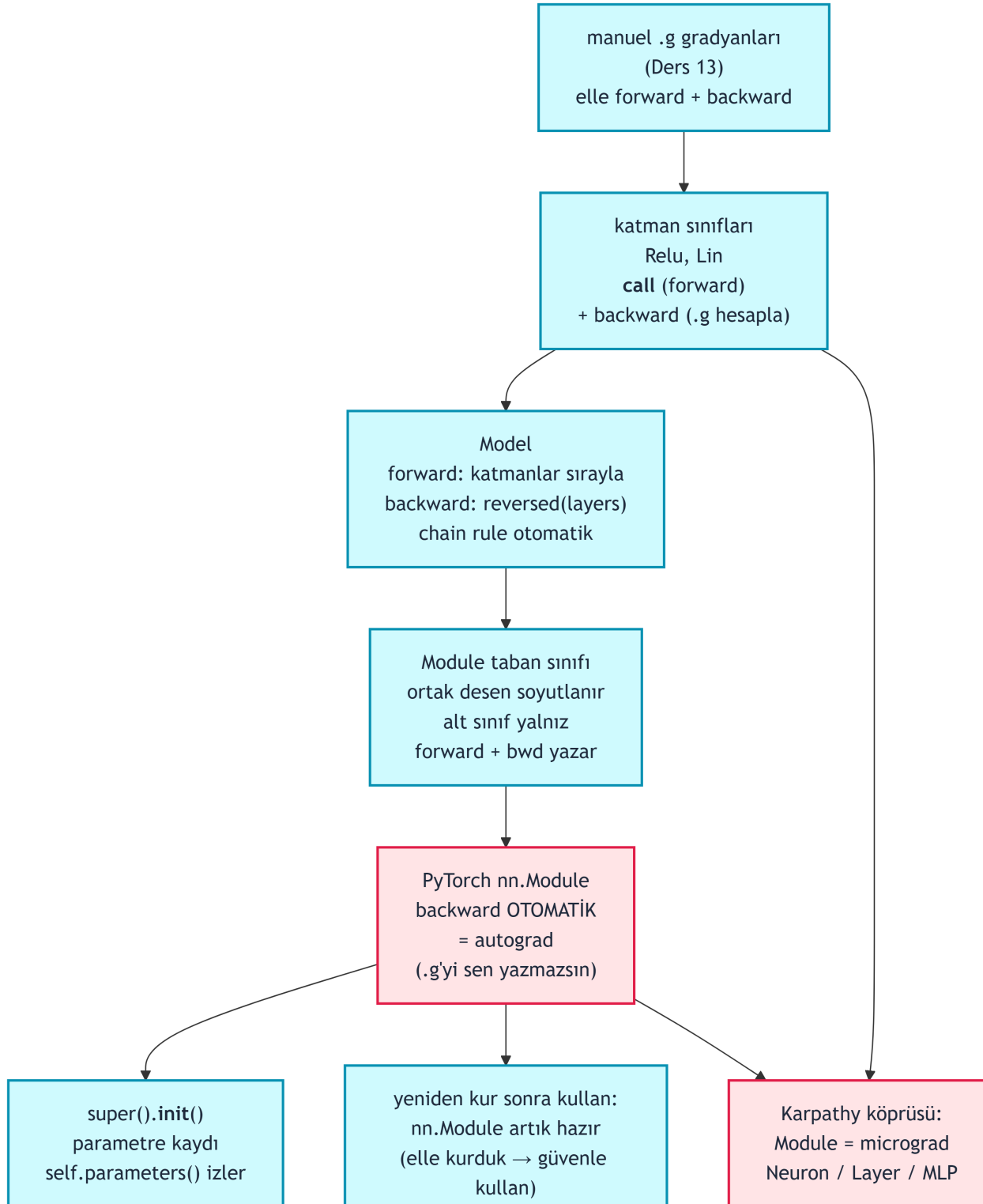
💡 Builder Notu — Geri Dönüş: Manuel Backprop’tan Devam

- **Geriye (Ders 13):** Ders 13’ün forward_and_backward’ı (manuel .g gradyanları) bu dersin başlangıç noktası; refactor onun üstüne kurulur.
- **Sezgi:** Bir kodu temizlemeden önce onu tam anlamak gerekir; Howard önce eski kodu tekrar edip her satırın “neden orada” olduğunu netleştirir — temiz refactor’un ön koşulu, mevcut davranışı tam kavramaktır.

23.3 2. Chain Rule ↔ Kod Eşlemesi

Howard kritik bağı kurar: kaybın ağırlığa göre türevi, zincir kuralıyla parçaların çarpımıdır.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w}$$



Şekil 23.1: L14'ün kavram haritası: 'kendi PyTorch'unu kur' refactor yolculuğu — manuel .g gradyanlarından (Ders 13) katman sınıflarına, Model'e, Module taban sınıfına ve nihayet nn.Module'e (autograd backward'ı OTOMATİK yapar). Karpthy köprüsü tamamlanır: nn.Module = micrograd Neuron/Layer/MLP (şematik).

Burada $\hat{y}$ ağıın çıktısı (tahmin); kaybın bu çıktıya göre türevini, çıktının ağırlığa göre türeviyle çarpacağız. Kodda `out.g` “loss’un çıktıya göre türevi”dir ($\partial L / \partial \text{çıktı}$); `lin_grad` ise bunu ağırlığa/girdiye taşır — lineer katman için $\partial \text{çıktı} / \partial \text{girdi}$ = ağırlık (transpoze). Yani Ders 13’ün her `.g` satırı, bu çarpımın bir parçasıdır.

“The derivative of the loss with respect to the weights is going to be equal to the derivative of the loss [w.r.t. output] times — this is the chain rule — the derivative of the outputs [w.r.t. weights].”
— Howard, 1:43

💡 Builder Notu — Math’ten Koda: Her `.g` Bir Türev Parçası

- **Geriye (Calculus):** Howard chain rule için 3Blue1Brown’u önerir; “temel türev ve zincir kuralı bilgisi” `backprop`’u anlamının ön koşulu.
- **Geriye (18.06):** Lineer katmanın girdiye göre türevi = ağırlık matrisinin transpozu; bu yüzden `inp.g = out.g @ w.t()` — matris kalkülüsünün temel sonucu (18.06).

23.4 3. Refactoring Neden Gerekli?

Manuel `backprop` çalışır ama tekrar doludur: her katman için “girdiyi sakla, ileri hesapla; sonra gradyanı hesapla”. Howard bu tekrarı temizler — **refactoring**, kodu daha sade ve yeniden kullanılabilir yapmaktır.

“So let’s start refactoring it. Refactoring is all about making it simpler.” — Howard, 11:02

Şekil 23.2 bu yolculuğu soldan sağa gösterir (**FLAGSHIP**): **manuel backprop** (~15 satır, dağınık) → **katman sınıfları** (~13 satır, tekrarlı desen) → **Module taban** (~7 satır, tekrar kalktı) → **nn.Module** (`backward = 0` satır, `autograd`). Her ok bir sadeleşmeyi işaret eder: “tekrarı kaldır”, “ortak deseni topla”, “PyTorch’a bırak”. Alt şerit Karpathy + fast.ai yollarının `nn.Module`’de buluştuğunu vurgular.

💡 Builder Notu — Refactoring: Tekrar Kanıtlandıktan Sonra Soyutla

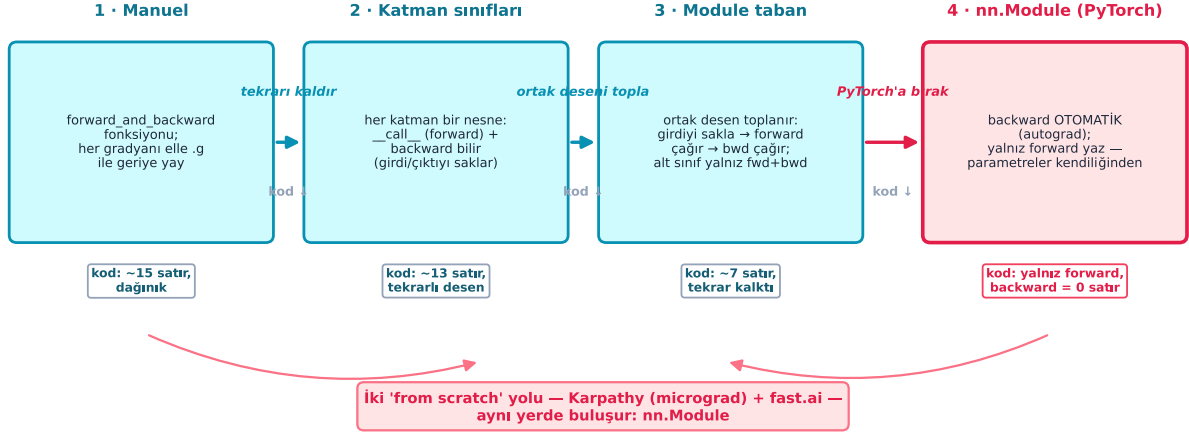
- **İleriye (coding-standards):** Refactoring, “üç benzer satırı erken soyutlama” değil; tekrar **kanıtlandıktan sonra** ortak deseni çıkarmaktır.
- **Sezgi:** Önce kodu yaz ve çalıştır, sonra tekrarı gör, sonra soyutla — bu sıra, erken soyutlamanın kuracağı yanlış sınırlardan korur. Howard tekrarı önce gözle gösterir, sonra `Module` tabanına taşır.

23.5 4. Katmanları Sınıflara Çevirmek

İlk refactor: her katman bir **sınıf** olsun, `__call__` (forward) ve `backward` bilsin. `__call__` girdiyi saklar ve çıktıyı üretir; `backward` saklı girdi/çıktıyı kullanıp gradyanı hesaplar.

```
class Relu():
    def __call__(self, inp):
        self.inp = inp
        self.out = inp.clamp_min(0.)
        return self.out
    def backward(self): self.inp.g = (self.inp>0).float() * self.out.g
```

Refactor yolculuğu — kendi PyTorch'unu kur, sonra bırak



Şekil 23.2: (şematik: manuel backprop → katman sınıfları → Module taban → nn.Module; her adım tekrarı kaldırır, kod sadeleşir — kendi PyTorch'unu kurma)

💡 Builder Notu — Katman = Nesne: **call** + backward

- **Geriye (Ders 7):** PyTorch'ta model = sınıf; `__call__`/forward deseni Ders 7'de (collab DotProduct) görülmüştü.
- **Sezgi:** Bir katmanı sınıfa sarmak, “ileri sonucu sakla, geride gradyanı hesapla” mantığını **kendi içinde** tutar; her katman bağımsız, yeniden kullanılabilir bir nesne olur — Ders 13'ün tek devasa fonksiyonu parçalanır.

23.6 5. Lineer Katman Sınıfı

Aynı desen lineer katman için: `__call__` ileri ($x@w+b$), backward ise girdi/ağırlık/bias gradyanlarını (Ders 13'teki `lin_grad`) hesaplar.

```
class Lin():
    def __init__(self, w, b): self.w, self.b = w, b
    def __call__(self, inp):
        self.inp = inp
        self.out = lin(inp, self.w, self.b)
        return self.out
    def backward(self):
        self.inp.g = self.out.g @ self.w.t()
        self.w.g = self.inp.t() @ self.out.g
        self.b.g = self.out.g.sum(0)
```

💡 Builder Notu — Lin Sınıfı: lin_grad Artık Katmanın İçinde

- **Geriye (Ders 13):** backward içindeki üç satır = Ders 13’ün lin_grad’ı; artık katmanın kendi içinde — girdiye, ağırlığa ve bias’a göre üç gradyan.
- **Sezgi:** $\text{inp.g} = \text{out.g} @ \mathbf{w}^T$ gradyanı geriye taşır (chain rule), $\mathbf{w.g} = \text{inp}^T @ \text{out.g}$ her ağırlığın katkısını biriktirir, $\mathbf{b.g} = \text{out.g.sum}(\emptyset)$ bias gradyanıdır. Aynı matematik, ama artık Lin’in sorumluluğunda — bu, sorumluluğun katmana yerleşmesidir.

23.7 6. Model Sınıfı

Katmanları bir listede toplayan **Model** sınıfı: `__call__` katmanları sırayla çalıştırır (forward); backward ise **ters sırada** her katmanın backward’ını çağırır — chain rule’un “en dıştan içe” yayılımı.

```
class Model():
    def __init__(self, w1, b1, w2, b2):
        self.layers = [Lin(w1,b1), Relu(), Lin(w2,b2)]
        self.loss = Mse()
    def __call__(self, x, targ):
        for l in self.layers: x = l(x)    # forward: sırayla
        return self.loss(x, targ)
    def backward(self):
        self.loss.backward()
        for l in reversed(self.layers): l.backward()    # backward: ters sıra
```

💡 Builder Notu — reversed(layers): Chain Rule’un Yönü

- **Geriye (Karpathy):** `reversed(self.layers)` = Karpathy’nin topolojik sıralamasının basit hâli; gradyan en dıştan içe akar.
- **Sezgi:** Forward girdiden çıktıya akar, backward ise loss’tan ilk katmana — çünkü bir katmanın gradyanı, kendisinden **sonraki** katmanın gradyanına (`out.g`) bağlıdır. `reversed(layers)` tam olarak bu bağımlılık sırasını kodlar.

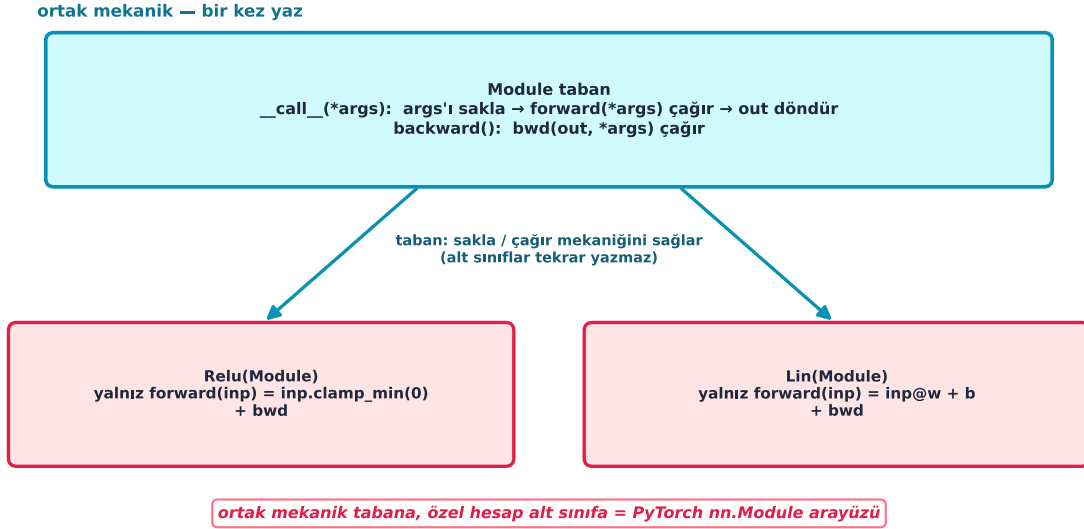
23.8 7. Module Taban Sınıfı

Hâlâ tekrar var: her katman “girdiyi sakla, forward çağır, backward’da gradyanı hesapla” yapıyor. Howard ortak deseni bir **Module taban sınıfında** toplar: `__call__` girdiyi saklayıp forward’ı çağırır; backward ise bwd’yi çağırır. Alt sınıflar yalnızca forward ve bwd yazar.

```
class Module():
    def __call__(self, *args):
        self.args = args
        self.out = self.forward(*args)
        return self.out
```

```
def forward(self): raise Exception('not implemented')
def backward(self): self.bwd(self.out, *self.args)
def bwd(self): raise Exception('not implemented')
```

Şekil 23.3 bu deseni şematik gösterir: üstte **Module taban** sınıfı (cyan) ortak mekaniği — `__call__` args'ı saklar → `forward` çağırır → `out` döndürür; `backward` ise `bwd`'yi çağırır. Altta iki alt sınıf (rose) yalnızca `forward` + `bwd` yazar. Taban “sakla/çağır” mekaniğini sağlar; alt sınıflar onu tekrar yazmaz — bu tam olarak PyTorch `nn.Module` arayüzüdür.



Şekil 23.3: (şematik: Module taban sınıfı ortak mekaniği (sakla/çağır) toplar; alt sınıflar yalnız `forward`+`bwd` yazar — `nn.Module` arayüzü)

💡 Builder Notu — Module Taban: Karpathy micrograd Module'ü

- **Geriye (Karpathy):** Bu, Karpathy micrograd'ının `Module`'üyle aynı fikir; ortak mekaniği taban sınıfa koyup tekrarı kaldırmak. İki “from scratch” yolu burada **somut** olarak buluşur.
- **Sezgi:** Ortak mekanik (sakla/çağır) tabana, özel hesap (her katmanın `forward`/`bwd`'si) alt sınıfa — bu ayırım, hem tekrarı kaldırır hem de yeni katman eklemeyi (yalnız `forward`+`bwd` yaz) kolaylaştırır.

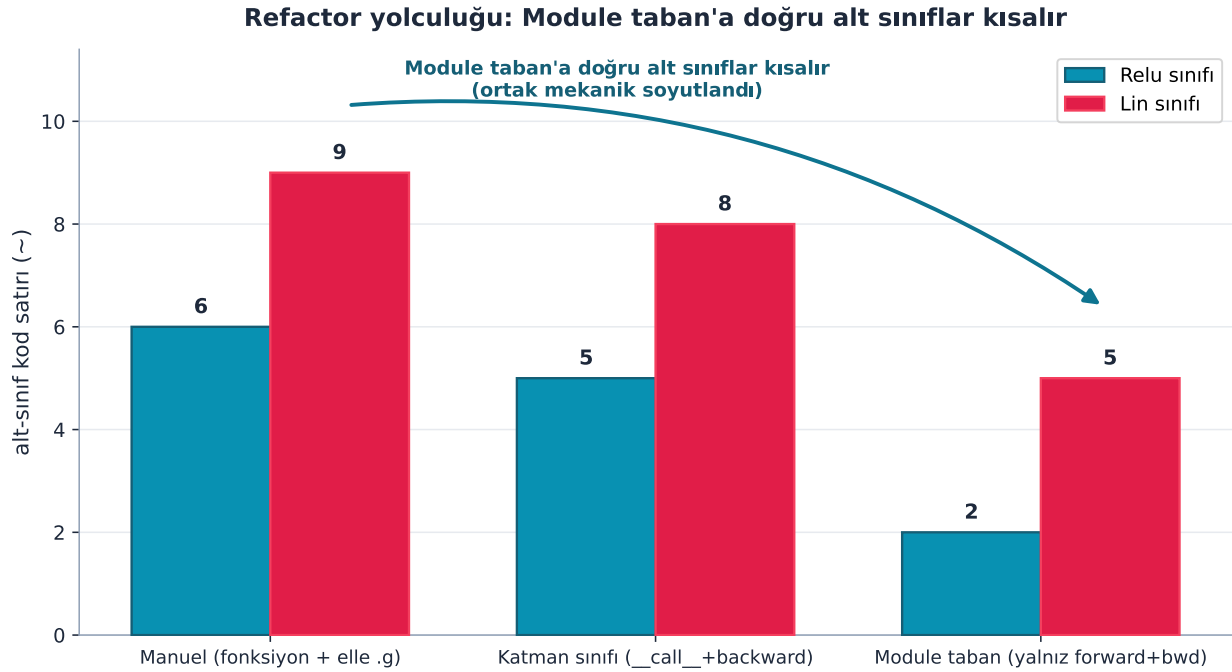
23.9 8. Temiz Alt Sınıflar

Taban sınıf sayesinde katmanlar çok kısalmış: yalnızca `forward` ve `bwd`.

```
class Relu(Module):
    def forward(self, inp): return inp.clamp_min(0.)
    def bwd(self, out, inp): inp.g = (inp>0).float() * out.g
```

```
class Lin(Module):
    def __init__(self, w, b): self.w, self.b = w, b
    def forward(self, inp): return inp@self.w + self.b
    def bwd(self, out, inp):
        inp.g = out.g @ self.w.t()
        self.w.g = inp.t() @ out.g
        self.b.g = out.g.sum(0)
```

Şekil 23.4 bu sadeleşmeyi sayıyla gösterir: refactor aşamaları boyunca alt-sınıf kod satırı **azalır** — Module taban ortak deseni soyutlayınca Relu ve Lin sınıfları kısalır. Tekrar (saklama/çağırma mantığı) tabana çıktığı için alt sınıflar yalnızca kendi işlerini (forward + bwd) yazar.



Şekil 23.4: (illüstratif: refactor aşamalarında alt-sınıf kod-satırı azalır — Module taban ortak deseni soyutlar, Builder Notu — Alt Sınıflar: forward + bwd = nn.Module Arayüzü (tekrar kalkar))

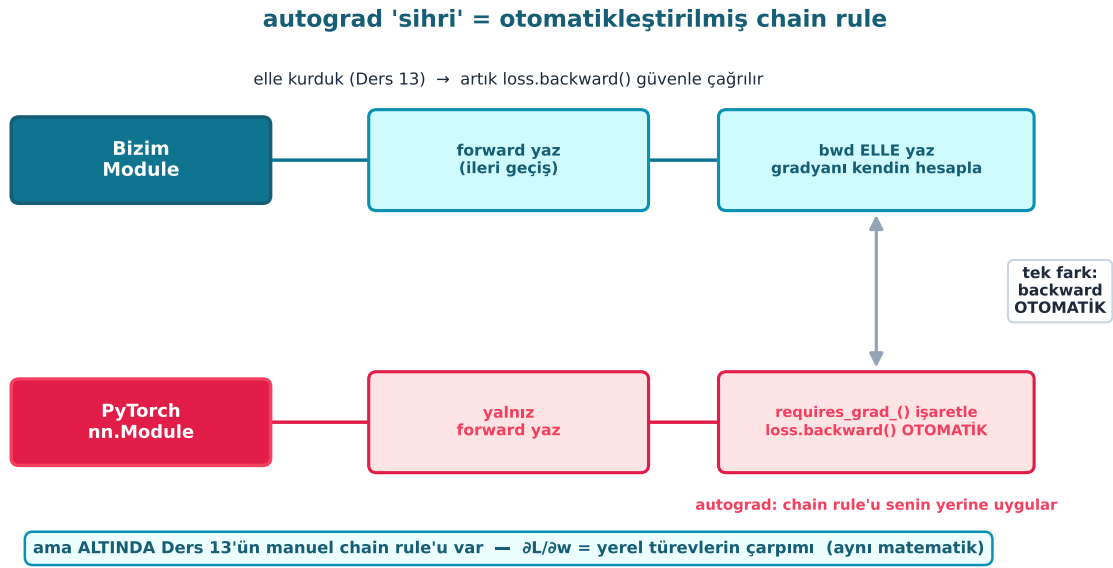
- **İleriye:** Her katman = forward + bwd; bu tam olarak PyTorch'un nn.Module arayüzü (forward + autograd backward).
- **Sezgi:** Kodun kısalması yan ürün değil amaç: her katman artık yalnızca “ne hesaplar” (forward) ve “gradyanı nasıl geçirir” (bwd) sorularını yanıtlar; mekanik altyapı görünmez olur — okunabilirlik ve bakım kolaylaşır.

23.10 9. PyTorch nn.Module

Howard son adımı atar: kendi Module'ümüz zaten PyTorch'un **nn.Module**'üyle aynı yapıda. Tek fark: PyTorch backward'ı **otomatik** yapar (autograd) — bwd yazmamıza gerek yok, sadece forward yeter.

“If we create a model in PyTorch, you have to make Module your super class. This is actually fastai's version of Module, but it's nearly the same as PyTorch's.” — Howard, 11:02

Şekil 23.5 bu “tek farkı” görselleştirir: üstte **bizim Module** (cyan) — forward yaz + bwd'yi **elle** yaz; altta **PyTorch nn.Module** (rose) — yalnız forward yaz, requires_grad_() işaretle, loss.backward() **otomatik**. İki “backward” kutusu dikey bağlanır: tek fark backward'ın otomatik olması. Ama altında Ders 13'ün manuel chain rule'u var ($\partial L / \partial w$ = yerel türevlerin çarpımı, aynı matematik).



Şekil 23.5: (şematik: bizim Module bwd'yi elle yazar; nn.Module backward'ı autograd ile OTOMATİK yapar — altında aynı chain rule)

💡 Builder Notu — nn.Module: Yeniden Kur, Sonra Kullan (Zirve)

- **Geriye (Ders 10):** “Yeniden kur, sonra kullan” kuralı zirvede: nn.Module'ü sıfırdan kurduk, artık güvenle kullanabiliriz.
- **Sezgi:** PyTorch nn.Module'ünü “büyü” gibi kullanmak yerine onu sıfırdan kurmak, tek fark olan autograd'ı (otomatik backward) bir kara kutu olmaktan çıkarır — altında ne olduğunu (Ders 13 chain rule) tam bilirsin.

23.11 10. nn.Module ile Linear ve Model

PyTorch nn.Module versiyonu: `super().__init__()` çağrılır, ağırlıklar `requires_grad_()` ile işaretlenir (autograd izlesin), yalnızca forward yazılır. backward PyTorch'a bırakılır.

```

class Linear(nn.Module):
    def __init__(self, n_in, n_out):
        super().__init__()
        self.w = torch.randn(n_in, n_out).requires_grad_()
        self.b = torch.zeros(n_out).requires_grad_()
    def forward(self, inp): return inp@self.w + self.b

class Model(nn.Module):
    def __init__(self, n_in, nh, n_out):
        super().__init__()
        self.layers = [Linear(n_in,nh), nn.ReLU(), Linear(nh,n_out)]
    def __call__(self, x, targ):
        for l in self.layers: x = l(x)
        return F.mse_loss(x, targ[:,None])

```

💡 Builder Notu — nn.Module: Manuel bwd Artık Yok

- **Geriye (Ders 13):** Manuel bwd artık yok; `requires_grad_()` + `loss.backward()` (autograd) onu otomatik yapar — ama altında Ders 13’ün chain rule’u var.
- **Sezgi:** Yalnızca forward yazarsın; `requires_grad_()` ile işaretlenen ağırlıklar için PyTorch grafi izler ve `loss.backward()`’ta tüm zincir kuralını uygular. “Gradyanlar bedava gelir” — ama bedavalığın bedeli bir kez elle kurmuş olmaktır.

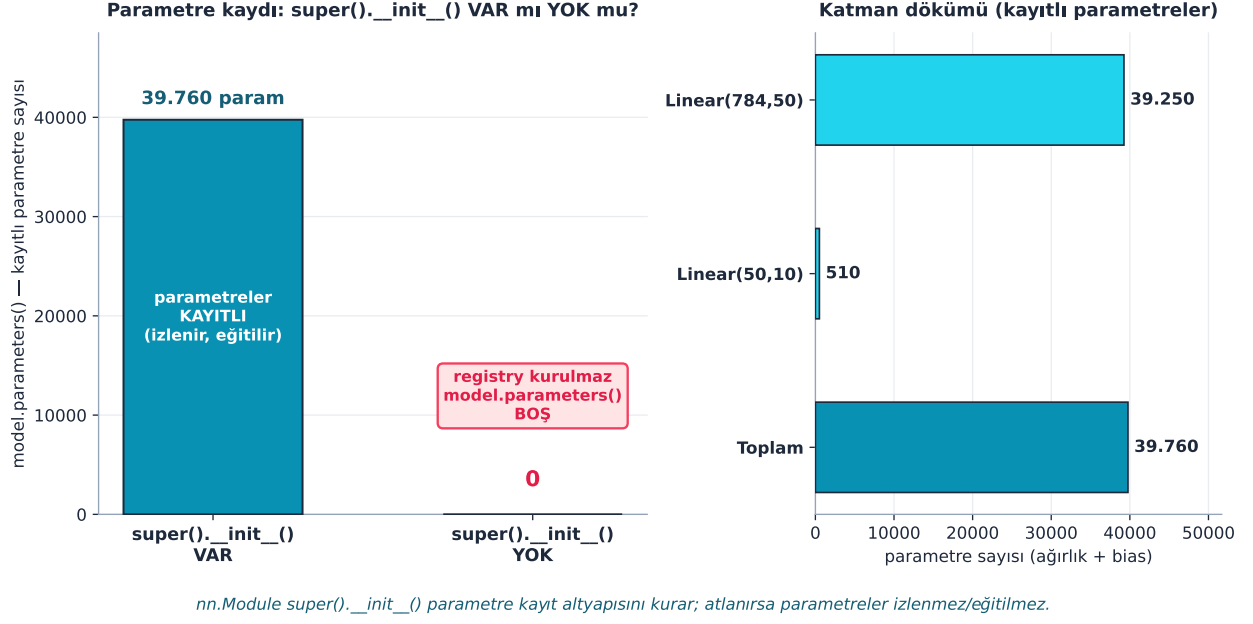
23.12 11. `super().__init__()` {#sec-super}

Howard PyTorch nesne-yönelimli detayına değinir: `nn.Module`’den miras alan bir sınıf, `__init__`’inde mutlaka `super().__init__()` çağırmalıdır — bu, PyTorch’un parametreleri izlemesini sağlayan altyapıyı kurar. Atlanırsa parametreler kaydedilmez.

Şekil 23.6 bunu gerçek hesaplama ile kanıtlar: `super().__init__()` **VAR** ise 39.760 parametre kayıtlı ($\text{Linear}(784,50) = 39.250 + \text{Linear}(50,10) = 510$); **YOK** ise `model.parameters()` boş (0) — registry hiç kurulmaz. Yani bu çağrı, parametre kayıt altyapısını kuran zorunlu satırdır; atlanırsa parametreler izlenmez ve eğitilmez.

💡 Builder Notu — `super().__init__()`: Parametre Kaydının Anahtarı

- **Geriye (Python):** `super()` = üst sınıfın kurucusu; `nn.Module`’ün parametre kayıt mekanizması için zorunlu.
- **Sezgi:** `nn.Module.__init__` bir parametre registry’si (`_parameters` sözlüğü) kurar; `super().__init__()` atlanırsa bu registry hiç oluşmaz, `self.w = ...` ataması hiçbir yere “kaydedilmez” — `model.parameters()` boş döner, optimizer hiçbir şeyi eğitemez. Şekil 23.6 bu sessiz başarısızlığı sayıyla gösterir.



Şekil 23.6: `super().__init__()` etkisi (gerçek hesaplama: `super().__init__()` VAR → 39.760 parametre kayıtlı; YOK → 0; parametre kaydı için zorunlu).

23.13 12. requires_grad_ ve Autograd

`requires_grad_()`, bir tensöre “senin gradyanını izle” der. Sonra `loss.backward()` çağrıldığında PyTorch, Ders 13’te elle yaptığımız tüm chain rule adımlarını **otomatik** yapar ve gradyanları `.grad`’a yazar.

“Just call `.backward()` [and PyTorch does the chain rule for us].” — Howard

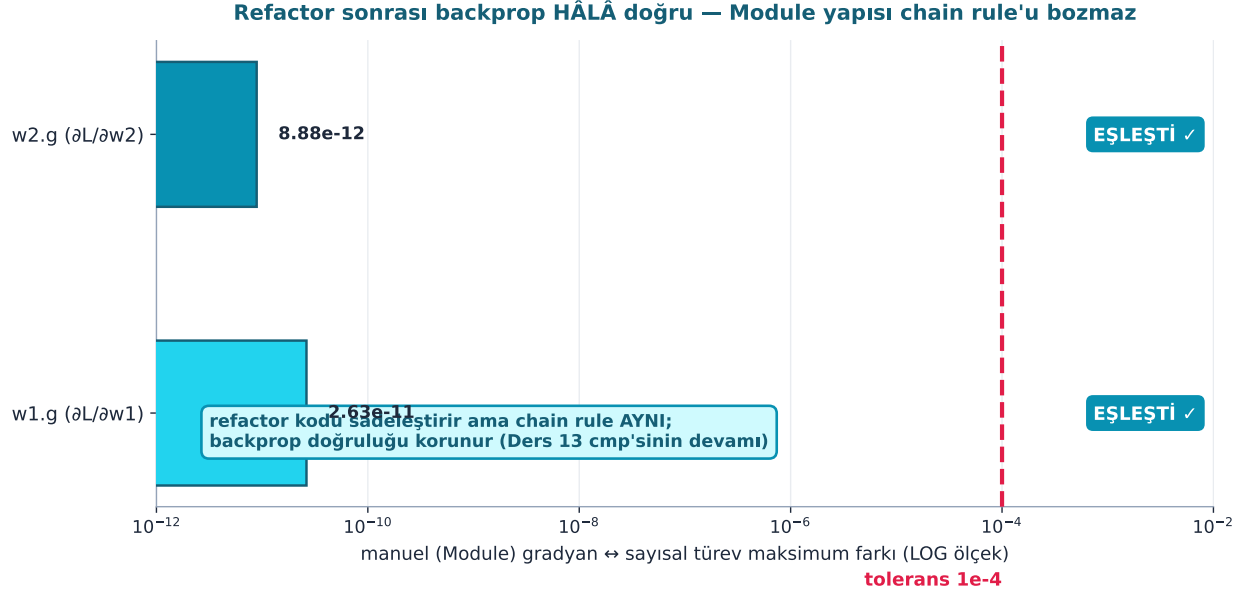
Şekil 23.7 refactor’un doğruluğu bozmadığını gerçek hesaplama ile kanıtlar: Module yapısına geçtikten **sonra** da backprop gradyanları sayısal türevle **~1e-11 hassasiyetle EŞLEŞİR** — yani yapı değişti, chain rule aynı kaldı. Log-skala barlar tüm parametreler için maksimum farkı $1e-4$ tolerans çizgisinin çok solunda gösterir. Bu, Ders 13’ün `cmp` emsalinin devamıdır: refactor kodu sadeleştirir ama gradyan matematiğini değiştirmez.

💡 Builder Notu — Autograd: Otomatikleşmiş Chain Rule

- **Geriye (Ders 5/13):** Ders 5’te `requires_grad_()` + `backward()`’ı kullanmıştık; şimdi altında ne olduğunu (Ders 13 manuel backprop) biliyoruz.
- **Sezgi:** Autograd “sihir” değil, otomatikleştirilmiş chain rule; Şekil 23.7 elle hesabın refactor sonrası bile sayısal türevle eşleştiğini gösterdiği an, `loss.backward()` artık anladığın bir araçtır — kara kutu değil.

23.14 13. Hazır Parçalar: F.mse_loss, nn.ReLU

`nn.Module`’e geçince hazır parçaları da kullanabiliriz: `nn.ReLU()` (ReLU katmanı), `F.mse_loss` (MSE kaybı). Hepsini sıfırdan kurduğumuz için artık güvenle kullanırız.



Şekil 23.7: (gerçek hesaplama: Module refactor sonrası backprop hâlâ sayısal türevle ~1e-11 EŞLEŞİR — yapı değişti, chain rule aynı; Ders 13 cmp devamı)

💡 Builder Notu — Hazır Parçalar: Kurduktan Sonra Kullanma Hakkı

- **Geriye (Ders 10 kuralı):** Relu, MSE, Linear, backward — hepsini elle kurduk; şimdi PyTorch'un optimize versiyonlarını kullanmaya hak kazandık.
- **Sezgi:** `nn.ReLU()` ve `F.mse_loss`'u sıfırdan kurduğumuz için içlerinde ne olduğunu biliriz; bir gün beklenmedik davranış görürsek (sayısal kararlılık, kenar durumlar) nereye bakacağımızı biliriz — kütüphane bizim için artık “okuyabildiğimiz kod”.

23.15 14. Minibatch Eğitime Doğru

Howard sonraki adımı işaret eder: model `nn.Module` olunca, bir eğitim döngüsü (minibatch, optimizer, dataloader) kurmak gerekir. Şu anki kod tüm veriyi tek batch'te işliyor; sonraki notebook (04) minibatch eğitimi kurar.

💡 Builder Notu — Minibatch: `nn.Module`'ün Üstündeki Katman

- **İleriye (Ders 16):** Eğitim döngüsü → callback'ler → Learner (Ders 16); `nn.Module` bunların temeli.
- **Sezgi:** `nn.Module` bir modeli temsil eder ama henüz “eğitmez”; eğitim döngüsü (veriyi minibatch'lere böl, ileri geçiş, `loss.backward()`, optimizer adımı) bir sonraki soyutlama katmanıdır — ve hepsi `nn.Module`'ün üstüne kurulur.

23.16 15. Karpathy Köprüsü: nn.Module

Bu ders, Karpathy köprüsünü **tamamlar**. Karpathy micrograd’da bir Module taban sınıfı (Neuron/Layer/MLP) kurar; Howard burada aynı Module desenini kurup PyTorch nn.Module’üne bağlar. İki seri de “katman = forward + backward bilen nesne” fikrine varır.

Ders 13’te köprüyü açmıştık: Howard’ın manuel .g gradyanları = Karpathy’nin _backward kapanışları. Bu derste köprü kapanır: Howard’ın Module taban sınıfı = Karpathy’nin Module’ü (Neuron/Layer/MLP). İki “from scratch” yolu, **nn.Module** ortak noktasında buluşur — ikisi de aynı yapıya, farklı bir API’den varır.

💡 Builder Notu — Karpathy Köprüsü: İki Yol, Aynı Module

- **Geriye (Karpathy 1):** Karpathy Ders 1’in son bölümü (Neuron/Layer/MLP + PyTorch karşılaştırması) = bu dersin birebir akrabası; nn.Module ikisinin buluşma noktası.
- **Sezgi:** Karpathy “tek skaler değerden” (Value), Howard “tensörden” (matris) yola çıkar; ama ikisi de aynı Module desenine (forward + backward bilen, çağrılabilir katman) ulaşır. Farklı ölçek, aynı soyutlama — bu, deseni gerçekten anladığının kanıtıdır.

23.17 16. Kapanış

Ders 14 backprop’u tamamladı ve “kendi PyTorch’unu kurma” yolculuğunu zirveye taşıdı: manuel .g gradyanları → forward/backward sınıfları → Module taban sınıfı → PyTorch nn.Module. Artık nn.Module’ü, autograd’ı ve loss.backward()’ı güvenle kullanabiliriz çünkü altında tam olarak ne olduğunu sıfırdan kurduk.

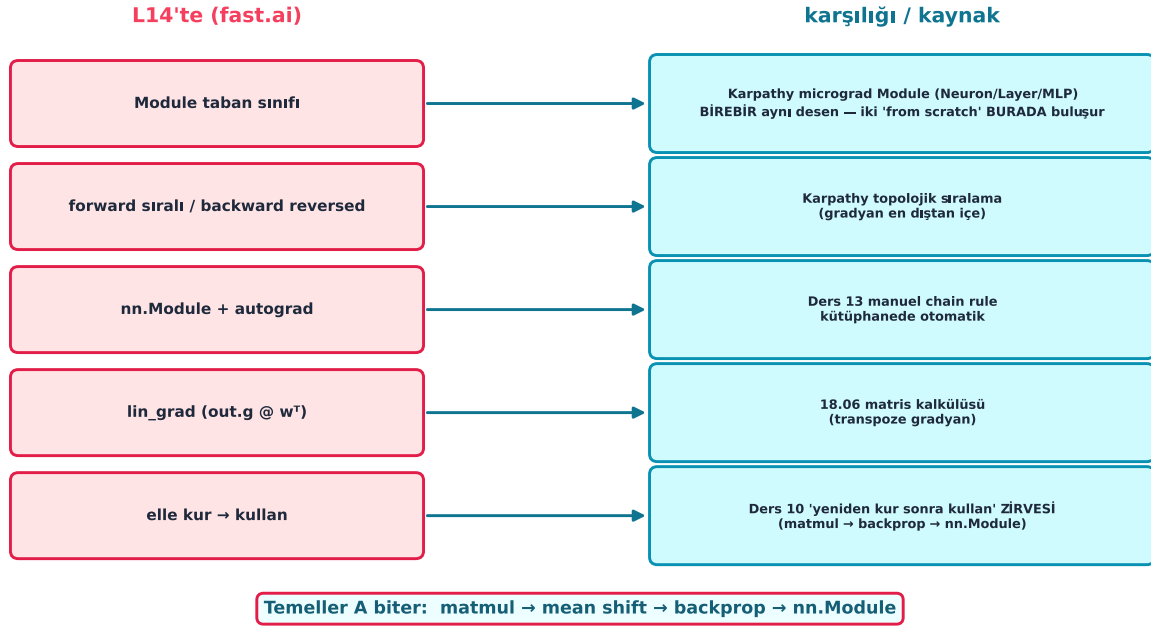
Şekil 23.8 dersin sentezidir (**Karpathy köprüsü merkezde, Temeller A kapanır**): solda L14’ün parçaları (Module taban sınıfı, forward sıralı/backward reversed, nn.Module + autograd, lin_grad, “elle kur → kullan”), sağda her birinin kaynağı/karşılığı — Module taban = Karpathy micrograd Module (Neuron/Layer/MLP, **Bİ-REBİR aynı**, iki “from scratch” burada buluşur), reversed = Karpathy topolojik sıralama, nn.Module+autograd = Ders 13 manuel chain rule (kütüphanede otomatik), lin_grad = 18.06 matris kalkülüsü, “elle kur → kullan” = Ders 10 zirvesi. Ortada vurgu: **Temeller A biter** (matmul → mean shift → backprop → nn.Module).

💡 Builder Notu — L14 Sentezi: Temeller A Zirvesi

- **İleriye (Ders 15-18):** Temeller A bitti (matmul → mean shift → backprop → nn.Module). Temeller B (Ders 15-18): autoencoder, Learner, init/norm, hızlandırılmış SGD — hepsi nn.Module üstüne.
- **Tek cümle:** Manuel backprop’u forward/backward sınıflarına, ortak deseni bir Module taban sınıfına refactor ederek PyTorch’un nn.Module’ünü sıfırdan yeniden kurduk — ve Karpathy köprüsü burada kapandı.

23.18 Bu Dersin Özeti

1. Ders 13’ün manuel backprop’u **chain rule’a eşlenir**: $out.g = \partial L / \partial çıktı$, $lin\_grad = \partial çıktı / \partial (girdi, ağırlık)$ (**chain rule eşlemesi**).
2. **Refactoring** ile her katman `__call__` (forward) + backward bilen bir sınıf olur (**katman sınıfları**).



Şekil 23.8: (sentez: L14 Karpathy köprüsünü tamamlar — Module = micrograd Module; Temeller A (matmul→backprop→nn.Module) biter)

3. **Model** sınıfı katmanları listede tutar; forward sırayla, **backward ters sırada** (chain rule) ([Model](#)).
4. **Module taban sınıfı** ortak deseni (girdiyi sakla, forward çağır, bwd çağır) toplar; alt sınıflar yalnızca forward+bwd yazar ([Module taban](#)).
5. Bu yapı PyTorch'un **nn.Module**'üyle aynıdır; tek fark backward'ın **otomatik** (autograd) olması ([nn.Module](#)).
6. **nn.Module**'de `super().__init__()` zorunlu (parametre kaydı); ağırlıklar `requires_grad_()` ile işaretlenir (`super().__init__()`).
7. `loss.backward()` + autograd, Ders 13'te elle yaptığımız chain rule'u **otomatik** yapar; refactor doğruluğu bozmaz (~1e-11 eşleşme) ([autograd](#)).
8. Sıfırdan kurduğumuz için artık `nn.Module`, `nn.ReLU`, `F.mse_loss` gibi hazır parçaları güvenle kullanırız ([hazır parçalar](#)).

! Tek Bir Cümle

Manuel backprop'u, her katmanı forward/backward bilen bir sınıfa ve ortak deseni bir Module taban sınıfına refactor ederek PyTorch'un `nn.Module`'ünü sıfırdan yeniden kurduk — ve artık autograd'ı, altında tam olarak Ders 13'ün chain rule'u olduğunu bilerek, güvenle kullanabiliriz. Karpathy köprüsü burada kapanır; bizim Module = micrograd Module.

23.19 Kontrol Soruları

i Soru 1: Bir Model sınıfında forward sırayla, backward neden ters sırada çalışır?

Cevap:

Forward pass girdiden çıktıya doğru akar: katmanlar **sırayla** çalışır (lin $\rightarrow$ relu $\rightarrow$ lin $\rightarrow$ loss). Backward pass ise chain rule gereği **en dıştan (loss) en içe (ilk katman)** doğru akar — çünkü bir katmanın gradyanı, kendisinden **sonraki** katmanın gradyanına (out.g) bağlıdır. Bu yüzden `for l in reversed(self.layers): l.backward()`: önce loss'un gradyanı, sonra son katman, sonra ReLU, en son ilk katman. Her katman, bir sonrakinden gelen gradyanı alıp kendi yerel türeviyle çarpar (chain rule) ve bir öncekine geçirir. (Şekil 34.1 yolculuğu, Şekil 23.2 sadeleşmeyi gösterir.)

i Soru 2: Module taban sınıfı hangi tekrarı kaldırır? Alt sınıflar ne yazar?

Cevap:

Refactor öncesi her katman aynı deseni tekrarlıyordu: `__call__`'da girdiyi sakla + ileri hesapla, `backward`'da saklı değerlerle gradyanı hesapla. **Module taban sınıfı** bu ortak mekaniği toplar: `__call__` argümanları saklar ve `forward`'ı çağırır; `backward` ise `bwd`'yi saklı argümanlarla çağırır. Böylece alt sınıflar (Relu, Lin) yalnızca **forward** (ileri hesap) ve **bwd** (gradyan hesabı) yazar — saklama/çağırma mantığını tekrarlamazlar. Bu, kodu hem kısaltır hem de PyTorch `nn.Module` arayüzüne yaklaştırır. (Şekil 23.3 deseni, Şekil 23.4 kod azalmasını gösterir.)

i Soru 3: Kendi Module'ümüz ile PyTorch `nn.Module` arasındaki temel fark nedir?

Cevap:

Yapı aynıdır: ikisi de bir `forward` metodu olan, çağrılabilir (`__call__`) katman nesneleridir. Temel fark **backward**'tadır: bizim `Module`'ümüzde her katmanın `bwd`'sini elle yazıp gradyanları kendimiz hesaplıyoruz; PyTorch `nn.Module`'de ise `backward` **otomatiktir** (autograd) — ağırlıkları `requires_grad_()` ile işaretler, `loss.backward()` çağırırsın, PyTorch chain rule'u tüm graf üzerinde otomatik uygular. Yani `nn.Module`'de yalnızca `forward` yazarsın; gradyanlar bedava gelir. Ayrıca `nn.Module` `super().__init__()` ile parametre kaydı, GPU taşıma gibi altyapı sağlar. (Şekil 23.5 tek farkı, Şekil 23.6 parametre kaydını gösterir.)

i Soru 4: Bu ders Karpathy serisiyle ve « from the foundations » kuralıyla nasıl bağlanır? (builder bağlantısı)

Cevap:

Karpathy `micrograd`'ın sonunda bir `Module` taban sınıfı (Neuron/Layer/MLP) kurar ve PyTorch ile karşılaştırır; Howard burada aynı `Module` desenini kurup PyTorch `nn.Module`'üne bağlar. İki “from scratch” yolu `nn.Module`'de buluşur. “From the foundations” kuralı da burada zirve yapar: `matmul` (Ders 10-12), `backprop` (Ders 13), `nn.Module` (Ders 14) — hepsini sıfırdan kurduk. Artık `loss.backward()`, `nn.Module`, `nn.ReLU` bizim için kara kutu değil; altında tam olarak ne olduğunu bildiğimiz, güvenle kullandığımız araçlar. Builder açısından: bir framework'ü sıfırdan kurmak, onu ustaca (ve gerektiğinde özelleştirerek) kullanmanın ön koşuludur. (Şekil 23.8 iki yolun buluşmasını ve Temeller A'nın kapanışını gösterir.)

23.20 Egzersizler

Egzersiz 1 (Direkt uygulama). Relu ve Lin katmanlarını sınıf olarak (`__call__` forward + backward) yaz; bir Model ile forward+backward yapıp gradyanları gözle. (§4-6)

Egzersiz 2 (İki-aşamalı). Ortak deseni bir Module taban sınıfına çıkar; Relu/Lin'i ondan türetip kodun ne kadar kısalacağını gör (Şekil 23.4'ü kendi satır sayınla yeniden üret — §7-8).

Egzersiz 3 (Edge case). PyTorch `nn.Module`'de `super().__init__()`'i kasıtlı olarak sil; `model.parameters()`'ın neden boş döndüğünü açıkla (Şekil 23.6'ı kendi modelinde doğrula — §11).

Egzersiz 4 (Kavramsal). Kendi bwd'lerinle hesapladığın gradyanları, PyTorch `autograd (requires_grad_ + backward())` sonucuyla karşılaştır; aynı çıktığını doğrula (Şekil 23.7'i kendi modelinde yeniden üret — §12).

Egzersiz 5 (Sonraki dersin habercisi — Ders 15). Bir eğitim döngüsünün (minibatch + optimizer adımı) `nn.Module` üstüne nasıl kurulacağını düşün (Learner'a giden yol — §14).

23.21 Sonraki: Ders 15 İçin Hazırlık

Ders 15: Autoencoder'lar (Autoencoders)

Temeller A bitti (matmul → mean shift → backprop → `nn.Module`). Ders 15, Temeller B'yi başlatır: convolution'larla bir **autoencoder** kurmak — görüntüyü sıkıştırıp geri açan ağ (Stable Diffusion'ın VAE'sinin temeli).

Ana konular (Ders 15):

- Eğitim döngüsü ve minibatch (`nn.Module` üstüne)
- Convolution'lar (sıfırdan)
- Autoencoder mimarisi
- VAE'ye köprü



Ders 15 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 2 ve 4 — Module refactor + autograd karşılaştırma).
- Kendi `nn.Module` modelini kurup `loss.backward()` ile eğit.
- Ana cümleyi tekrar oku: “`nn.Module`'ü sıfırdan kurduk; artık güvenle kullanırsınız.”

23.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Chain rule ↔ kod	$out.g = \partial L / \partial \text{çıktı}; \text{lin\_grad} = \partial \text{çıktı} / \partial (\text{girdi}, \text{ağırlık})$	1:43
Refactoring	Kodu sadeleştirme; tekrarı kaldırma	11:02

Kavram	Tanım	Howard'da
Katman sınıfı	<code>__call__</code> (forward) + backward bilen nesne	11:02
Model sınıfı	Katman listesi; forward sıralı, backward ters	13:26
<code>reversed(layers)</code>	Backward'ın en dıştan içe yayılımı (chain rule)	13:26
Module taban sınıfı	Ortak deseni toplar; alt sınıf forward+bwd yazar	17:59
<code>nn.Module</code>	PyTorch katman temeli; backward otomatik (autograd)	11:02
<code>super().__init__()</code>	<code>nn.Module</code> parametre kaydı için zorunlu	13:26
<code>requires_grad_</code>	Tensörün gradyanını izle (autograd)	—
<code>F.mse_loss</code> / <code>nn.ReLU</code>	Hazır parçalar; sıfırdan kurduktan sonra güvenli	21:42
autograd	<code>loss.backward()</code> ile chain rule'u otomatik yapma	—
Karpathy köprüsü	Module deseni = micrograd Neuron/Layer/MLP	—

23.23 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: `nn.Module` Zirvesi

Bu ders backprop'u tamamlar ve Karpathy köprüsünü kapatır; köprülerin özeti:

1. **Chain rule** ↔ **kod** → backprop'un her satırı bir türev parçası (Calculus/3B1B) ([chain rule eşlemesi](#)).
2. **Module deseni** → forward+backward bilen katman; Karpathy micrograd Module'ü (Neuron/Layer/MLP) ([Module taban](#)).
3. **nn.Module + autograd** → backward otomatik; Ders 13'ün manuel chain rule'u kütüphanede ([autograd](#)).
4. **Refactoring** → tekrar kanıtlanınca soyutla (coding-standards); erken soyutlama değil ([refactoring](#)).
5. **requires_grad_/backward** → Ders 5'te kullanmıştık, şimdi altını biliyoruz (Ders 13) (`super().__init__()`).
6. **“Yeniden kur, sonra kullan”** → `nn.Module` zirvesi; framework'ü sıfırdan kurmak ustaca kullanmanın koşulu ([Karpathy köprüsü](#)).

! Bu dersten tek bir şey alıp gideceksen

PyTorch'un `nn.Module`'ü sihir değildir — onu sıfırdan kurduk. Manuel `.g` gradyanlarından başlayıp, her katmanı `forward/backward` sınıfına, ortak deseni bir `Module` taban sınıfına refactor ederek tam olarak `nn.Module`'e verdik. Tek fark, PyTorch'un `backward`'ı `autograd` ile otomatik yapması — ama altında Ders 13'ün `chain rule`'u var. Artık `loss.backward()` anladığın bir araç; Karpathy köprüsü kapandı ve Temeller B (Ders 15-18) bunun üstüne kurulacak.

24 Ders 15 — Autoencoder'lar (Autoencoders)

Temeller B'nin ilk dersi iki şey kurar: convolution'ı sıfırdan (Part 1 Ders 8'de kavramsal görmüştük; şimdi koddan) ve onun üstüne bir convolutional autoencoder — görüntüyü sıkıştırıp geri açan ağ. Önce convolution'ın ne olduğunu `apply_kernel` ile en yalın hâlde kurar, sonra onu `im2col` ile bir matris çarpımına indirger (`F.unfold + matmul = F.conv2d`), en sonunda bir autoencoder inşa eder: encoder conv'la küçültür, decoder deconv'la büyütür, kayıp ise çıktı ile girdinin kendisi arasındadır (reconstruction, self-supervised). Tek cümleyle: convolution kaydırılan bir matris çarpımıdır; autoencoder ise conv'la sıkıştırıp deconv'la geri açan, girdisini yeniden kurmayı öğrenen ve Stable Diffusion'ın VAE'sine giden ağıdır.

i Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 15: Autoencoders](https://course.fast.ai/lesson/15) (~97 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 15
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/07_convolutions + 08_autoencoder](#)
- **Okuma süresi:** ~38 dk
- **🔗 Temeller B başlar (ETAP 5):** Ders 14 Temeller A'yı kapadı (matmul → mean shift → backprop → nn.Module). Ders 15 Temeller B'yi açar: convolution'ı sıfırdan kurup üstüne bir convolutional autoencoder inşa eder — Stable Diffusion'ın VAE'sinin (Ders 9 / 25) temeli.

24.1 Bu Derste Ne Var?

Temeller B'nin ilk dersi iki şey kurar: **convolution'ı sıfırdan** (Part 1 Ders 8'de kavramsal görmüştük; şimdi koddan) ve onun üstüne bir **convolutional autoencoder** — görüntüyü sıkıştırıp geri açan ağ. Bu, Stable Diffusion'ın VAE'sinin (Ders 9) temelidir.

Üç temel fikir bu dersin omurgasını kurar:

1. **Convolution = kaydırılan dot product** — küçük bir kernel'i görüntü üzerinde gezdirip her konumda dot product; mekânsal yapıyı yakalar, az parametre kullanır (**kernel ve sliding window** → `apply_kernel`).
2. **im2col** — convolution'ı bir **matris çarpımına** çevirme; yamaları “açıp” (unfold) matmul yapma; PyTorch `F.unfold/F.conv2d` bunu optimize eder (`im2col` → `F.conv2d`).
3. **Autoencoder** — encoder (conv ile küçült) + decoder (deconv ile büyüt); girdiyi bir darboğazdan (bottleneck) geçirip yeniden kurmayı öğrenir (**autoencoder nedir** → **VAE köprüsü**).

“Today, let's start by talking, before we can create a convolutional autoencoder, [about] what are convolutions.” — Howard, 0:53

Şekil 34.1 bu yolculuğu tek bir yol haritasında birleştirir: üstte **convolution sıfırdan** (kaydırılan kernel → dot product → im2col → tek matmul → F.unfold/F.conv2d → nn.Conv2d), altta **autoencoder** (encoder küçültür → darboğaz → decoder büyütür → reconstruction loss → Stable Diffusion VAE temeli).

💡 Builder Notu — Temeller B Başlar: Convolution + Autoencoder

- **Geriye (Ders 8/9/10-12):** Convolution Ders 8'de (Part 1) kavramsal; burada Ders 10-12'nin matmul'uyla sıfırdan. Autoencoder = Ders 9'un VAE'sinin temeli.
- **İleriye (Ders 25 / NYU §4.J):** Autoencoder, latent diffusion'ın (Ders 25) VAE'sine; NYU §4.J'de autoencoder enerji-tabanlı model olarak işlenir.
- **Tek cümle:** Convolution kaydırılan bir matris çarpımıdır; autoencoder ise conv'la sıkıştırıp deconv'la geri açan, VAE'ye giden ağıdır.

24.2 1. Hedef: Convolutional Autoencoder

Howard hedefi koyar: bir **convolutional autoencoder** kurmak. Ama önce convolution'ın ne olduğunu (Part 1'de kavramsal görmüştük) koddan, “from the foundations” ruhuyla yeniden kurar. Hedefi baştan ortaya koymak, her ara adımın (apply_kernel, im2col, F.conv2d) neye hizmet ettiğini netleştirir.

“We will create a convolutional autoencoder. And in the process, we will see why [convolutions matter].” — Howard, 0:53

💡 Builder Notu — Hedef Önce: Neden Convolution Kuruyoruz?

- **Geriye (Ders 8):** Part 1 Ders 8 convolution'ı tanıttı; bu ders onu sıfırdan kodlayıp autoencoder'a bağlar.
- **Sezgi:** Hedefi (autoencoder) baştan koymak, “neden convolution'ı sıfırdan kuruyoruz” sorusunu yanıtlar — her ara adım bir amaca hizmet eder, “yan iş” değildir.

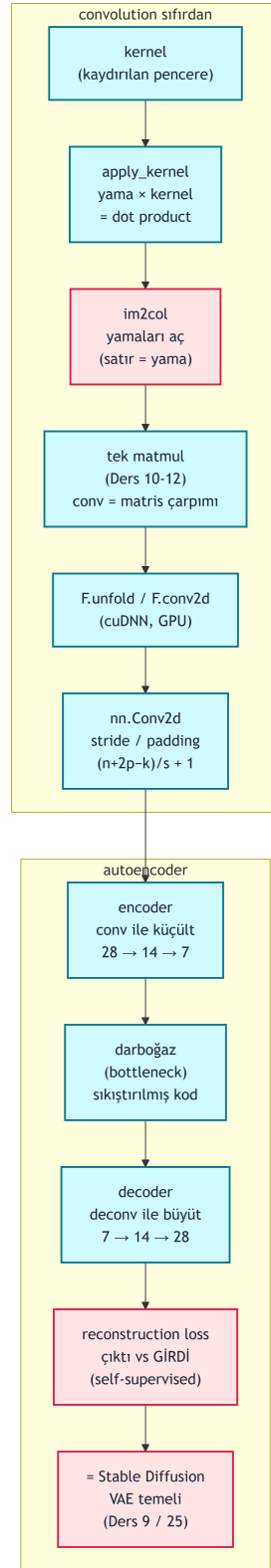
24.3 2. Convolution Nedir, Ne İşe Yarar?

Convolution, ağa görüntünün **mekânsal yapısını** söyler: yakın pikseller ilişkilidir, bir desen (kenar, köşe) görüntünün her yerinde aynıdır. Bu, **Convolutional Neural Network (CNN)**'in doğal olarak yakaladığı şeydir. Yalnızca görüntü değil, 1B convolution dil için de kullanılır.

“Convolutions are something that allows us to tell our neural network a little bit about [the spatial structure].” — Howard, 0:53

💡 Builder Notu — Inductive Bias: Mekânsal Yapı + Paylaşım

- **Geriye (Ders 8):** “Mekânsal yapı + parametre paylaşımı” = convolution'ın doğru inductive bias'ı; görüntü için ideal.
- **Sezgi:** Tam bağlı katman her pikseli bağımsız sanır; convolution ise “yakın pikseller ilişkilidir, bir kenar her yerde kenardır” bilgisini ağa **gömer** — bu doğru ön-yargı, daha az veriyle daha iyi



Şekil 24.1: L15'in kavram haritası — İKİ parça. ÜST 'convolution sıfırdan': kaydırılan kernel → her kolumda yama × kernel dot product → yamaları aç (im2col) → tek matmul (Ders 10-12) → F.unfold/F.conv2d (cuDNN) → nn.Conv2d (stride/padding). ALT 'autoencoder': encoder conv ile küçültür (28→14→7) → darboğaz (bottleneck) → decoder deconv ile büyütür → reconstruction loss (çıkı vs GİRDİ = self-supervised) → Stable Diffusion VAE temeli (Ders 9/25). (şematik)

genelleme demektir.

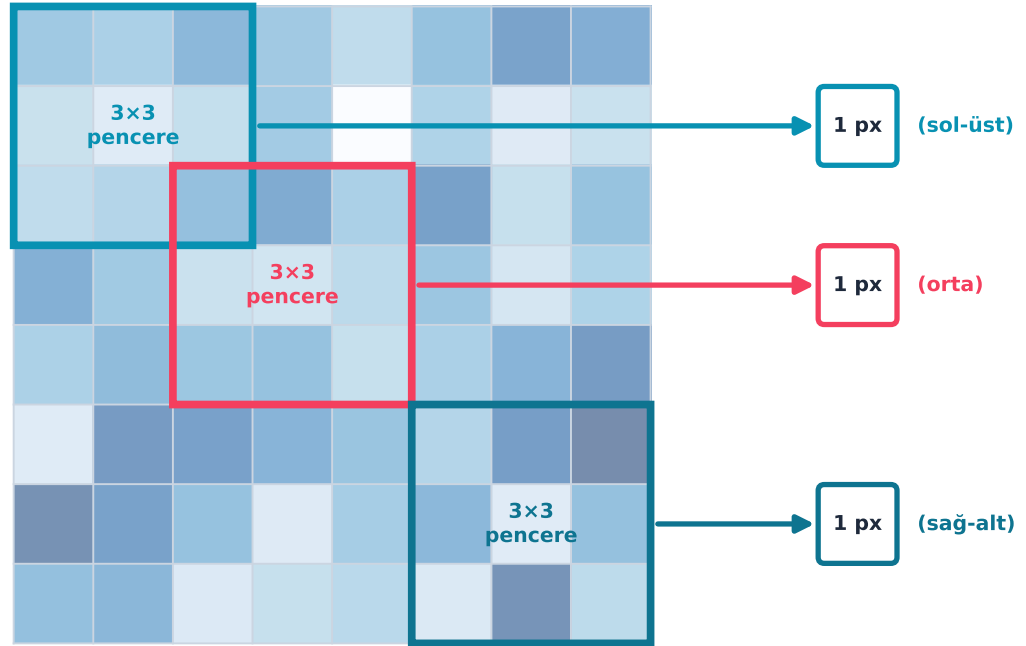
24.4 3. Kernel ve Sliding Window

Convolution bir **kernel** (küçük matris, örn. 3×3) kullanır. Kernel görüntü üzerinde **kaydırılan bir pencere** gibi gezer; her konumda kernel ile o yamanın elemanları çarpılıp toplanır — yani bir dot product.

“We’re going to slide — it’s like a sliding window — we’re going to slide our kernel over [the image].” — Howard, 5:27

Şekil 24.2 bu kayan pencereyi gösterir: aynı 3×3 kernel görüntünün üç farklı konumunda (sol-üst, orta, sağ-alt) aynı işlemi yapar — her konumda yama $\times$ kernel toplamı bir tek çıktı pikseli üretir. Aynı kernel her yerde kaydığı için bu **parametre paylaşımıdır**: az parametre, mekânsal yapı korunur.

kernel = öğrenilen öznelik dedektörü; AYNI kernel her konumda kayar
(parametre paylaşımı $\rightarrow$ az parametre + mekânsal yapı korunur) **öznelik haritası**
(çıktı pikselleri)



her konumda: yama (3×3) $\times$ kernel (3×3) topla = 1 çıktı pikseli (dot product)

Şekil 24.2: (şematik: convolution = kaydırılan pencere; kernel her konumda yama ile dot product alır — parametre paylaşımı + mekânsal yapı)

💡 Builder Notu — Kernel = Öğrenilen Öznelik Dedektörü

- **Geriye (Ders 8/11):** Ders 8'deki “kaydırılan mini dot product” + Ders 11'in dot product'ı; kernel = öğrenilen öznelik dedektörü.

- **Sezgi:** Bir kenar-kernel'i (örn. dikey kenar dedektörü) görüntünün her yerinde aynı deseni arar; öğrenme sırasında bu kernel'in 9 sayısı, ağın hangi öznelikleri arayacağına karar verir — ve aynı 9 sayı tüm görüntüyü tarar.

24.5 4. *apply_kernel*: Sıfırdan Convolution

Howard convolution'ı en yalın hâlde kurar: bir *apply_kernel* fonksiyonu, görüntünün (row, col) merkezli 3×3 yamasını kernel ile çarpıp toplar. Bunu tüm konumlara uygulayınca öznelik haritası (feature map) çıkar.

```
def apply_kernel(row, col, kernel):
    return (im3[row-1:row+2, col-1:col+2] * kernel).sum() # 3x3 yama . kernel

rng = range(1, 27)
top_edge3 = tensor([[apply_kernel(i, j, top_edge) for j in rng] for i in rng])
```

💡 Builder Notu — *apply_kernel*: Ders 8'in Aynısı, Koddan

- **Geriye (Ders 8):** Bu, Ders 8'deki *apply_kernel*'in aynısı; her konumda yama × kernel toplamı.
- **Sezgi:** İç içe iki list comprehension, kernel'i tüm (i, j) konumlarına kaydırır — yani saf Python'da kayan pencere döngüsüdür. Çalışır ama yavaştır; bu yavaşlık sonraki adımın (im2col + matmul) motivasyonudur.

24.6 5. Convolution = Dot Product

Howard kritik bağı kurar: her konumdaki “yama × kernel topla” işlemi, yamayı ve kernel'i vektöre açıp **dot product** almakla aynıdır. Bu içgörü, convolution'ı matris çarpımına çevirmenin (im2col) yolunu açar.

“Flattening rank 1 tensors into vectors and then doing a dot product would be one way of thinking [about it].” — Howard, 13:24

💡 Builder Notu — Dot Product: matmul'a Açılan Kapı

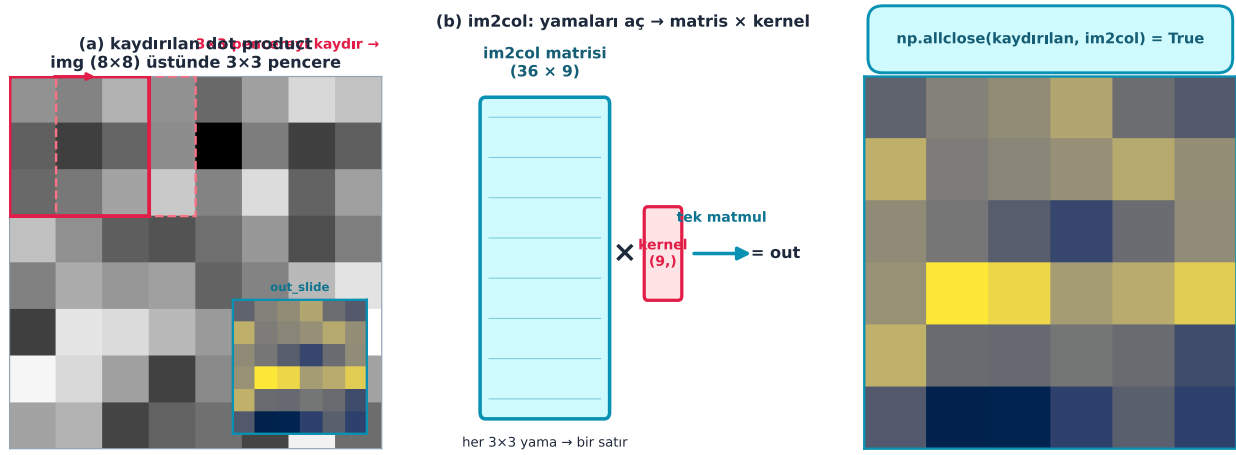
- **Geriye (Ders 11):** Dot product (Ders 11); convolution = mekânsal olarak tekrarlanan dot product.
- **Sezgi:** “Yama × kernel topla” = iki 9'luk vektörün dot product'ı. Bu eşitlik fark edilince, convolution'ı bir matris çarpımına dönüştürmek (her yama bir satır, kernel bir vektör) doğal bir adım olur — bu, im2col'un tohumudur.

24.7 6. im2col: Convolution'ı Matmul'a Çevirme

Büyük içgörü: convolution bir **matris çarpımı** olarak yazılabilir. **im2col** tekniği, her yamayı bir sütuna “açar” (unroll); kernel'i de vektöre açar. Sonra tek bir matmul tüm konumları hesaplar — Ders 10-12'de kurduğumuz hızlı matmul'u kullanır.

“im2col is a way of converting a convolution into a matrix multiply.” — Howard, 19:30

Şekil 24.3 bu dönüşümü **gerçek hesaplamayla** kanıtlar (**FLAGSHIP**): solda kaydırılan dot product (3×3 pencereyi görüntü üzerinde gezdir), ortada im2col matrisi (her 3×3 yama $\rightarrow$ bir satır, sonra $\times$ kernel vektörü), sağda sonuç. `np.allclose(kaydırılan, im2col) = True` — yani iki yöntem birebir aynı çıktıyı verir; convolution sıradan bir matris çarpımıdır ve Ders 10-12'nin hızlı matmul'unu doğrudan kullanır.



convolution sıradan bir matris çarpımıdır → Ders 10-12 hızlı matmul'unu doğrudan kullanır

Şekil 24.3: (gerçek hesaplama: convolution = im2col + matmul; 36 yamayı açıp tek matmul = kaydırılan dot product, `np.allclose=True`)

💡 Builder Notu — im2col: « Her Şeyi matmul'a Çevir » Deseni

- **Geriye (Ders 10-12):** Convolution'ı matmul'a indirgemek, onu optimize C/GPU koduyla (matmul) hızlandırır; “her şeyi matmul'a çevir” deseni.
- **Sezgi:** im2col bellek harcar (yamalar tekrar tekrar kopyalanır) ama hesabı tek bir matmul'a indirger — ve matmul, on yıllardır optimize edilmiş en hızlı işlemdir. “Bellek karşılığı hız” takası, GPU çağında neredeyse her zaman kazançlıdır.

24.8 7. F.unfold + Matris Çarpımı

PyTorch'un `F.unfold`'u `im2col`'u yapar: girdiyi yamalara “açar”. Sonra açılmış yamaları kernel ile çarpacağız (matmul). Sonuç, convolution ile birebir aynıdır.

```
inp_unf = F.unfold(inp, (3,3))[0] # yamaları ac (im2col)
w = left_edge.view(-1) # kernel'i vektöre ac
out = (w @ inp_unf).view(26, 26) # matmul = convolution
```

💡 Builder Notu — F.unfold: Ders 12'nin @ İşareti

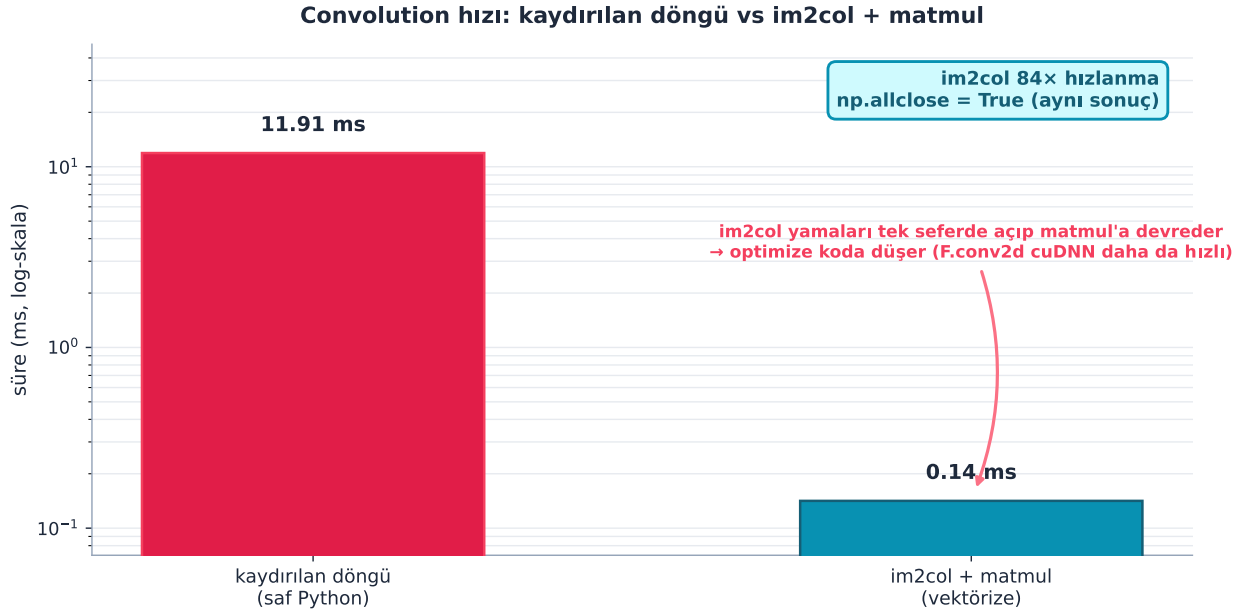
- **Geriye (Ders 12):** `w @ inp_unf` = Ders 12'nin @'i; convolution artık tek bir matris çarpımı.
- **Sezgi:** `F.unfold` el yazımı `im2col` döngüsünün yerini alır (yamaları C hızında açar); `w @ inp_unf` ise Ders 12'de sıfırdan kurduğumuz `matmul`'dur. İkisi birleşince `convolution` = tek satır matris çarpımı olur — kavramsal olarak `apply_kernel`'le aynı, ama vektörize.

24.9 8. F.conv2d ve Hız

Elle `unfold` + `matmul` yerine PyTorch'un `F.conv2d`'si bunu optimize biçimde, tek çağrıda yapar. Howard `%timeit` ile kıyaslar: elle döngü ~9 ms, `unfold`+`matmul` çok daha hızlı, `F.conv2d` en hızlı (cuDNN).

```
F.conv2d(inp, left_edge[None,None]) # optimize convolution (cuDNN)
batch_features = F.conv2d(xb, edge_kernels) # tum batch + tum kerneller
```

Şekil 24.4 bu hız farkını **gerçek canlı zamanlamayla** ölçer: kaydırılan döngü (saf Python) vs `im2col` + `matmul` (vektörize); log-skalada iki büyüklük mertebesi fark görünür. `np.allclose = True` (aynı sonuç) ama `im2col` çok daha hızlı — convolution'ı `matmul`'a indirgemenin kazancı budur (`F.conv2d` cuDNN ile daha da hızlı).



Şekil 24.4: (gerçek canlı zamanlama: kaydırılan döngü vs `im2col`+`matmul`; aynı sonuç, `im2col` çok daha hızlı — convolution'ı `matmul`'a indirgemenin kazancı)

💡 Builder Notu — F.conv2d: Kurduktan Sonra Kullanma Hakkı

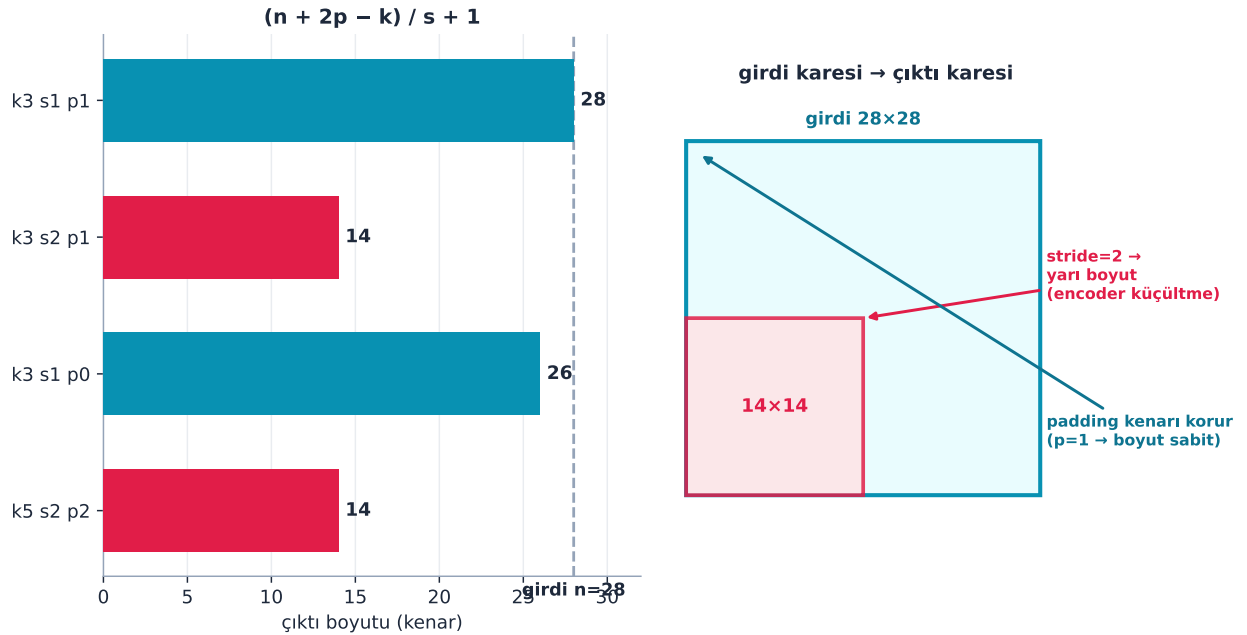
- **Geriye (Ders 10 kuralı):** Convolution'ı sıfırdan kurduk (apply_kernel → im2col → unfold); artık F.conv2d'yi güvenle kullanırız.
- **Sezgi:** “Yeniden kur, sonra kullan” kuralı: F.conv2d'nin içinde ne olduğunu (im2col + matmul) bildiğimiz için onu kara kutu olarak değil, okuyabildiğimiz optimize bir araç olarak kullanırız — beklenmedik davranışta nereye bakacağımızı biliriz.

24.10 9. nn.Conv2d ile CNN

Hazır parça: nn.Conv2d. Howard bir conv yardımcısı yazar (conv + ReLU, stride ile küçültme) ve bir CNN kurar. stride=2 görüntüyü yarıya indirir, padding=ks//2 kenarları korur.

```
def conv(ni, nf, ks=3, stride=2, act=True):
    res = nn.Conv2d(ni, nf, stride=stride, kernel_size=ks, padding=ks//2)
    if act: res = nn.Sequential(res, nn.ReLU())
    return res
```

Şekil 24.5 çıktı boyutunu **gerçek hesaplamayla** gösterir: formül $(n + 2p - k) / s + 1$. stride=2 boyutu yarılar (encoder küçültme, rose), stride=1 + padding=1 boyutu sabit tutar (kenarı korur, cyan). Sağ panel girdi karesi (28×28) ile yarılanan çıktı karesini (14×14) yan yana çizer — stride'ın “küçültme” rolü görselleşir.



Şekil 24.5: (gerçek hesaplama: çıktı boyutu $(n+2p-k)/s+1$; stride=2 yarılar (encoder küçültme), padding kenarı korur)

💡 Builder Notu — stride/padding: Boyut Kontrolü

- **Geriye (Ders 8):** Ders 8'deki `simple_cnn` deseni; stride boyutu küçültür, kanal sayısı artar.
- **Sezgi:** stride=2 her adımda iki piksel atlar → çıktı yarı boyut (encoder'ın "sıkıştırma" mekanığı); padding=ks//2 kenarda kaybolan pikselleri telafi eder → boyut sabit kalır. Bu iki düğme, bir autoencoder'da encoder'ın daraltma profilini belirler.

24.11 10. Eğitim Döngüsü ve Dataset

Modeli eğitmek için bir **Dataset** (x, y çiftleri) ve minibatch döngüsü gerekir. Howard görüntüleri (1,28,28) şekline getirir ve Dataset'e sarar — Ders 14'ün `nn.Module`'ü üstüne eğitim altyapısı.

```
x_imgs = x_train.view(-1, 1, 28, 28) # (batch, kanal, yukseklik, genislik)
train_ds = Dataset(x_imgs, y_train)
```

💡 Builder Notu — Dataset: `nn.Module` Üstündeki Katman

- **İleriye (Ders 16):** Bu eğitim döngüsü Ders 16'da callback'lerle `Learner`'a dönüşecek.
- **Sezgi:** (batch, kanal, yükseklik, genişlik) = PyTorch'un standart görüntü tensör düzeni (NCHW); MNIST tek kanallı (gri) olduğu için kanal=1. Bu şekillendirme, `nn.Conv2d`'nin beklediği girdi formatıdır — model ile veri arasındaki arayüz.

24.12 11. Autoencoder Nedir?

İkinci yarı: **autoencoder**. Fikir: girdiyi (görüntüyü) bir **darboğazdan** (bottleneck — küçük bir temsil) geçirip sonra **geri kurmak**. Ağ, girdiyi az sayıda sayıya sıkıştırıp (encoder) tekrar açmayı (decoder) öğrenir — çıktının girdiye benzemesi hedeflenir.

"To remind you about what an autoencoder is... [it compresses and reconstructs]." — Howard, 15:50

💡 Builder Notu — Autoencoder: VAE'nin Olasılıksal-Olmayan Atası

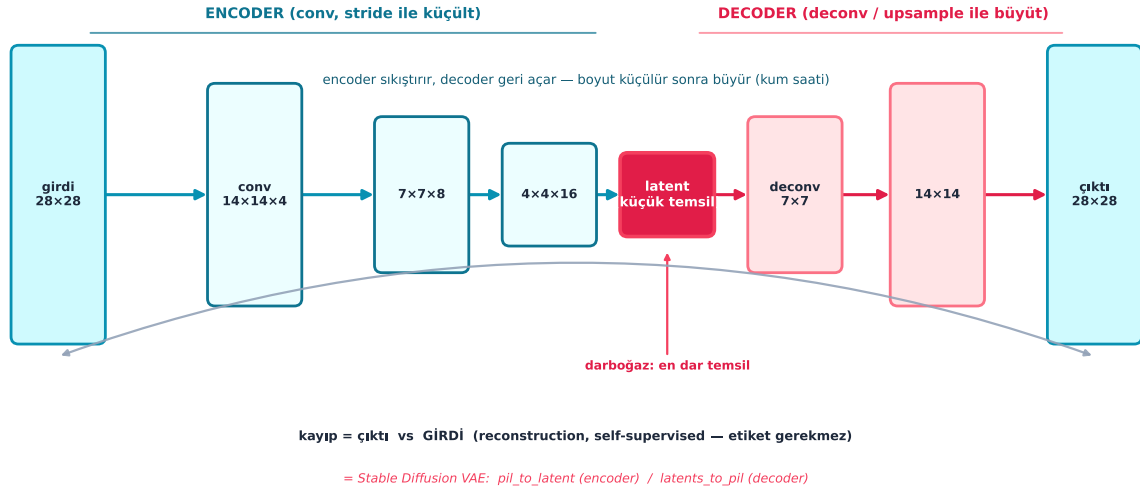
- **Geriye (Ders 8/9):** Embedding/sıkıştırma (Ders 8) + Ders 9'un VAE'si; autoencoder VAE'nin olasılıksal-olmayan atasıdır.
- **Sezgi:** Bir autoencoder'ın "anlamı keşfetmek zorunda kalması", darboğazdan gelir: ağ tüm pikselleri küçük bir temsile sığdırmak zorunda olduğu için, gürültüyü atıp yapıyı (kenarlar, şekiller) öğrenir — sıkıştırma, öğrenmeyi zorlar.

24.13 12. Encoder: Conv ile Küçültme

Encoder, stride'lı conv katmanlarıyla görüntüyü adım adım küçültür ($28 \rightarrow 14 \rightarrow 7 \rightarrow \dots$) ve kanal sayısını artırır — bilgiyi küçük bir temsile sıkıştırır.

```
cnn = nn.Sequential(
    conv(1, 4),    # 14x14
    conv(4, 8),    # 7x7
    conv(8, 16),   # 4x4
    conv(16, 16),  # 2x2
    conv(16, 10, act=False), nn.Flatten())
```

Şekil 24.6 tüm autoencoder mimarisini bir **kum saati** olarak gösterir (**FLAGSHIP**): solda encoder conv ile boyutu küçültür ($28 \times 28 \rightarrow 14 \times 14 \times 4 \rightarrow 7 \times 7 \times 8 \rightarrow 4 \times 4 \times 16 \rightarrow \text{latent}$), ortada darboğaz (en dar temsil), sağda decoder deconv ile geri büyütür ($7 \times 7 \rightarrow 14 \times 14 \rightarrow 28 \times 28$). Boyut önce küçülür sonra büyür; kayıp ise çıktı ile **girdinin kendisi** arasındadır (self-supervised). Bu yapı = Stable Diffusion VAE'si (pil_to_latent / latents_to_pil).



Şekil 24.6: (şematik: convolutional autoencoder — encoder conv ile darboğaza sıkıştırır, decoder deconv ile geri açar; reconstruction loss = girdinin kendisi; SD VAE'sinin temeli)

💡 Builder Notu — Encoder: Ders 9 VAE'sinin Basit Hâli

- **Geriye (Ders 9):** Encoder = Ders 9'daki VAE'nin “786K $\rightarrow$ 16K sıkıştırma”nın basit hâli.
- **Sezgi:** Her $\text{conv}(n_i, n_f)$ boyutu yarılarken (stride=2) kanalı artırır — mekânsal çözünürlük azalır, ama her konumdaki “anlam zenginliği” (kanal) artar. Encoder bilgiyi “geniş ve sığ”dan “dar ve derin”e taşır; latent, görüntünün özüdür.

24.14 13. Decoder: Deconv ile Büyütme

Decoder, encoder'ı tersine çevirir: küçük temsili adım adım büyütür. Howard bunu UpsamplingNearest2d (boyutu 2x büyüt) + conv ile yapar — “deconv”.

```
def deconv(ni, nf, ks=3, act=True):
    layers = [nn.UpsamplingNearest2d(scale_factor=2),          # 2x buyut
              nn.Conv2d(ni, nf, stride=1, kernel_size=ks, padding=ks//2)]
    if act: layers.append(nn.ReLU())
    return nn.Sequential(*layers)
```

💡 Builder Notu — Decoder: Upsample + Conv = « deconv »

- **Geriye (Ders 9):** Decoder = VAE'nin “latent → görüntü” decode'unun temeli; upsampling + conv.
- **Sezgi:** UpsamplingNearest2d boyutu kabaca 2x büyütür (en yakın komşu — bloklu), ardından conv bu büyütülmüş haritayı yumuşatıp detay ekler. “deconv” = “boyutu büyüten conv”; encoder'ın stride'lı küçültmesinin tersidir. İkisi birlikte simetrik kum saati kurar.

24.15 14. Autoencoder'ı Eğitmek

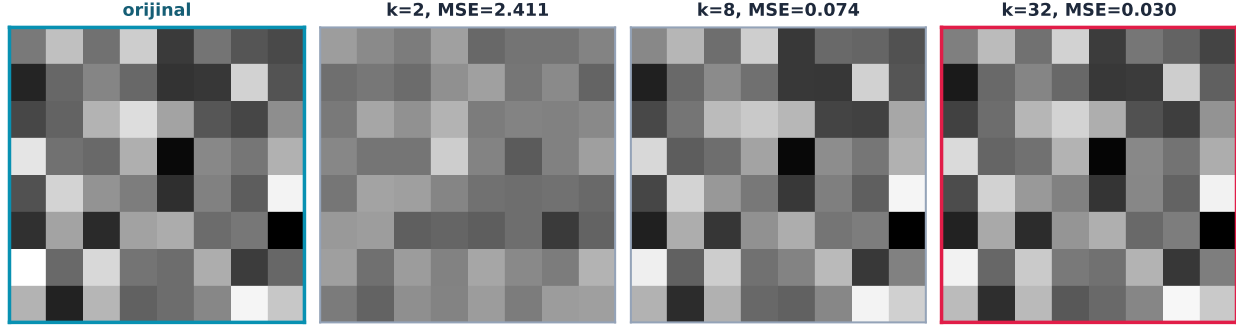
Encoder + decoder birleşince autoencoder olur. Kayıp: çıktı ile **girdinin** farkı (reconstruction loss) — etiket değil, girdinin kendisi hedeftir (self-supervised). Howard basit bir autoencoder'ı (conv + deconv) MNIST'te eğitir.

```
ae = nn.Sequential(
    nn.ZeroPad2d(2), conv(1,2), conv(2,4),    # encoder
    deconv(4,2), deconv(2,1, act=False), nn.ZeroPad2d(-2)) # decoder
# loss: cikti ile GIRDİ arasında (reconstruction)
```

Şekil 24.7 reconstruction'ı **gerçek hesaplamayla** gösterir: lineer autoencoder (= PCA) için darboğaz k büyüdükçe ($2 \rightarrow 8 \rightarrow 32$) reconstruction iyileşir (MSE $2.4 \rightarrow 0.07 \rightarrow 0.03$). Darboğaz ne kadar darsa o kadar çok bilgi atılır; ne kadar genişse o kadar sadık geri kurma — bu, sıkıştırma/kalite dengesidir. Lineer AE, SD VAE'sinin olasılıksal-olmayan atasıdır.

💡 Builder Notu — Reconstruction: Hedef = Girdinin Kendisi

- **Geriye (Ders 9 / NYU §4.J):** “Hedef = girdinin kendisi” self-supervised öğrenmedir (etiket gerekmez); Ders 9'un enerji-tabanlı model (EBM) + self-supervised akrabalığıyla aynı fikir, NYU §4.J'nin merkez teması.
- **Sezgi:** Reconstruction loss (çıkıktı - girdi)² etiket istemez — verinin kendisi denetim sinyalıdır. Bu, milyonlarca etiketsiz görüntüden öğrenmeyi mümkün kılar (etiketleme pahalıdır); autoencoder, self-supervised öğrenmenin en saf örneğidir.



darboğaz büyüdükçe reconstruction iyileşir (sıkıştırma/kalite dengesi) — lineer AE = PCA, SD VAE'sinin olasılıksal-olmayan atası

Şekil 24.7: (gerçek hesaplama: linear autoencoder (PCA) — darboğaz $k=2 \rightarrow 8 \rightarrow 32$ büyüdükçe reconstruction iyileşir (MSE $2.4 \rightarrow 0.07 \rightarrow 0.03$); sıkıştırma/kalite dengesi)

24.16 15. VAE ve Stable Diffusion'a Köprü

Howard autoencoder'ın neden önemli olduğunu vurgular: Stable Diffusion'ın **VAE**'si tam olarak budur — görüntüyü latent uzaya sıkıştıran encoder + geri açan decoder. Ders 9'daki "latent diffusion" bu autoencoder üstüne kurulur.

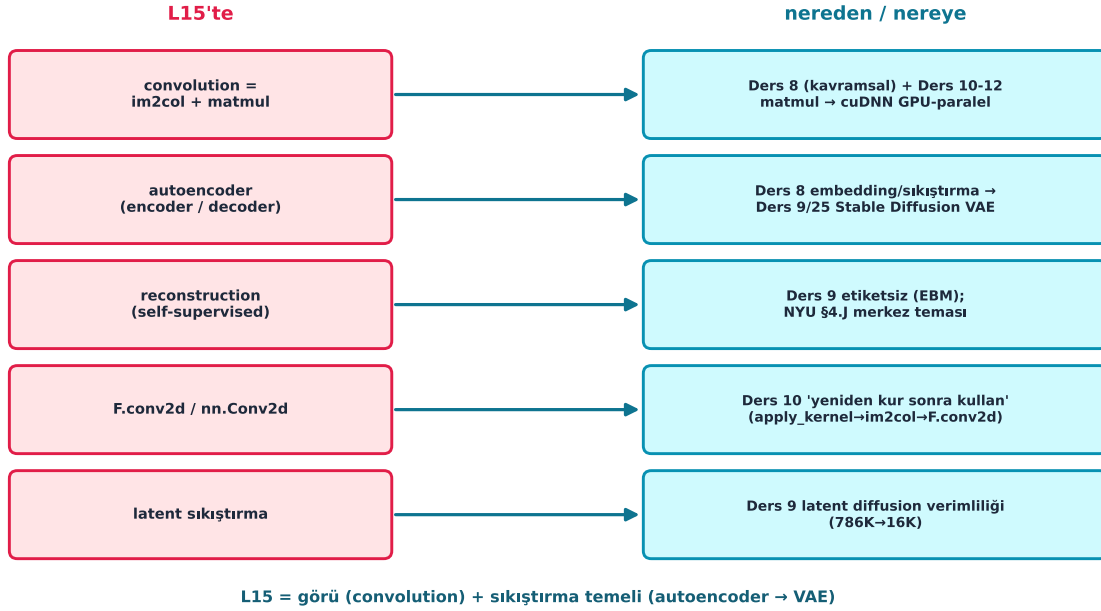
Şekil 24.8 dersin sentezidir (**VAE köprüsü merkezde**): solda L15'in parçaları (convolution = `im2col` + `matmul`, autoencoder encoder/decoder, reconstruction self-supervised, `F.conv2d`/`nn.Conv2d`, latent sıkıştırma), sağda her birinin kaynağı/hedefi — convolution = Ders 8 (kavramsal) + Ders 10-12 (`matmul` → `cuDNN`), autoencoder = Ders 8 embedding → **Ders 9/25 Stable Diffusion VAE**, reconstruction = Ders 9 etiketsiz (EBM/self-supervised) + NYU §4.J, latent sıkıştırma = Ders 9 (786K → 16K). Alt şerit: L15 = görü (convolution) + sıkıştırma temeli (autoencoder → VAE).

💡 Builder Notu — VAE Köprüsü: Autoencoder = SD'nin Kalbi

- **İleriye (Ders 25):** Bu basit autoencoder, Ders 25'te (latent diffusion) gerçek VAE'ye dönüşür; SD'nin `pil_to_latent/latents_to_pil`'i (Ders 9A) bunun ürünü.
- **Sezgi:** Stable Diffusion pahalı işlemi (diffusion) tüm pikseller üzerinde değil, VAE'nin sıkıştırdığı küçük latent uzayda yapar — bu yüzden "latent diffusion". Bu derste kurduğumuz encoder/decoder, tam olarak o sıkıştırmayı sağlayan parçadır; SD'nin verimliliği bu autoencoder sayesinde.

24.17 16. Kapanış

Ders 15, convolution'ı sıfırdan kurdu (`apply_kernel` → `im2col/unfold` → `F.conv2d` → `nn.Conv2d`) ve üstüne bir convolutional autoencoder inşa etti (encoder conv ile sıkıştırır, decoder deconv ile açar). Bu, Stable Diffusion'ın VAE'sinin temelidir — Temeller B'nin görü/sıkıştırma ayağı.



Şekil 24.8: (sentez: L15 = convolution (im2col=matmul) + autoencoder (SD VAE temeli); Ders 8/9-12 + latent diffusion köprüleri)

💡 Builder Notu — Kapanış: Sırada Eğitim Altyapısı

- **İleriye (Ders 16-18):** Sıradaki dersler eğitim altyapısını kurar: Learner (16), init/norm (17), accelerated SGD (18) — autoencoder ve sonraki diffusion modelleri bunların üstünde eğitilecek.
- **Sezgi:** Bu derste model (convolution + autoencoder) hazır, ama eğitimi hâlâ elle döngüyle yapıyoruz; sonraki dersler bu döngüyü esnek bir Learner'a (callback'lerle) dönüştürür — model ve eğitim altyapısı ayrı soyutlama katmanları olarak olgunlaşır.

24.18 Bu Dersin Özeti

1. **Convolution** mekânsal yapıyı yakalar: bir kernel'i kaydırılan pencere gibi gezdirip her konumda dot product alır (CNN'in temeli) (**kernel**).
2. **apply_kernel** convolution'ı sıfırdan kurar; her 3×3 yama × kernel toplamı (**apply_kernel**).
3. Her konumdaki işlem bir **dot product**'tır; bu, convolution'ı matris çarpımına çevirmenin (im2col) yolunu açar (**dot product**).
4. **im2col** / **F.unfold** yamaları açıp matmul yapar; **F.conv2d** bunu optimize (cuDNN) ile en hızlı yapar (**im2col**, **F.conv2d**).
5. **nn.Conv2d** + stride/padding ile CNN kurulur (stride küçültür, kanal artar) (**CNN**).
6. **Autoencoder** girdiyi bir darboğazdan geçirip yeniden kurar: **encoder** (conv ile sıkıştır) + **decoder** (deconv ile aç) (**encoder**, **decoder**).
7. **Decoder** = UpsamplingNearest + conv; encoder'ı tersine çevirir (**decoder**).
8. Kayıp **reconstruction** (çıktı vs girdi, self-supervised); bu, Stable Diffusion'ın **VAE**'sinin temelidir (**VAE köprüsü**).

! Tek Bir Cümle

Convolution, kaydırılan bir kernel'in her konumdaki dot product'ıdır ve `im2col` ile bir matris çarpımına indirgenir; autoencoder ise `conv`'la sıkıştırıp `deconv`'la geri açan, girdisini yeniden kurmayı öğrenen ve Stable Diffusion'ın VAE'sine giden bir ağıdır.

24.19 Kontrol Soruları

i Soru 1: `im2col` nedir ve convolution'ı neden matris çarpımına çevirmek isteriz?

Cevap:

Bir convolution, her konumda “yama × kernel toplamı” yani bir dot product hesaplar. **im2col**, her yamayı bir sütuna “açar” (unroll) ve kernel'i bir vektöre açar; böylece tüm konumlar tek bir **matris çarpımıyla** hesaplanır (PyTorch'ta `F.unfold + matmul`). Bunu isteriz çünkü matris çarpımını Ders 10-12'de sıfırdan kurup optimize ettik (broadcasting, @, cuDNN); convolution'ı matmul'a indirmek, onu bu yüksek optimize edilmiş, GPU-paralel koda devreder. Yani “her şeyi matmul'a çevir” deseni convolution'ı da hızlandırır. (Şekil 24.3 `allclose=True` ile, Şekil 24.4 hız kazancını gösterir.)

i Soru 2: Bir autoencoder'ın encoder ve decoder'ı ne yapar? Kayıp neye göre hesaplanır?

Cevap:

Encoder, girdiyi (görüntüyü) stride'lı conv katmanlarıyla adım adım küçültüp az sayıda sayıya (darboğaz/bottleneck) sıkıştırır. **Decoder**, bu küçük temsili `UpsamplingNearest + conv (deconv)` ile adım adım büyütüp orijinal boyuta geri açar. Kayıp **reconstruction loss**'tur: çıktı ile **girdinin kendisi** arasındaki fark (etiket değil, girdi hedefidir — self-supervised). Ağ böylece “bilgiyi küçük bir temsile sıkıştırıp geri kurmayı” öğrenir; bu, Stable Diffusion'ın VAE'sinin (görüntü ↔ latent) temelidir. (Şekil 24.6 kum saati mimarisini, Şekil 24.7 darboğaz/kalite dengesini gösterir.)

i Soru 3: Convolution neden tam bağlı katmandan daha uygun bir görüntü işlemcisidir?

Cevap:

İki sebep: (1) **Parametre paylaşımı** — aynı kernel görüntünün her yerinde kullanılır, az parametre (Ders 8: 784 yerine 9); (2) **mekânsal yapı** — convolution yakın pikselleri birlikte işler ve “bir kenar her yerde kenardır” (öteleme-değişmezlik) bilgisini gömer. Tam bağlı katman her pikseli bağımsız ele alır, mekânsal komşuluğu yok sayar ve devasa parametre ister. Convolution'ın bu “doğru inductive bias”ı, onu görüntü için tam bağlı katmandan çok daha verimli ve genelleyici yapar. (Şekil 24.2 aynı kernel'in her konumda kaymasını gösterir.)

i Soru 4: Bu dersteki autoencoder Stable Diffusion'a nasıl bağlanır? (builder bağlantısı)

Cevap:

Stable Diffusion bir **latent diffusion** modelidir (Ders 9): diffusion'ı piksel uzayında değil, bir VAE'nin sıkıştırdığı latent uzayda yapar. Bu VAE tam olarak bu dersteki autoencoder'dır — encoder görüntüyü küçük bir latent'e sıkıştırır (Ders 9A'daki `pil_to_latent`), decoder geri açar (`latents_to_pil`). Bu

derste basit bir convolutional autoencoder kurduk; Ders 25'te (latent diffusion) bu gerçek VAE'ye dönüşür. Builder açısından: pahalı işlemi (diffusion) küçük, anlamlı bir temsilde yapmak — autoencoder bu sıkıştırımayı sağlayan parçadır. (Şekil 24.8 VAE köprüsünü merkeze alır.)

24.20 Egzersizler

Egzersiz 1 (Direkt uygulama). `apply_kernel` ile bir kenar kernel'ini MNIST görüntüsüne uygula; öznetelik haritasını görselleştir. (§4)

Egzersiz 2 (İki-aşamalı). Aynı convolution'ı `F.unfold + matmul` ve `F.conv2d` ile yap; üçünün sonucunun aynı, hızlarının farklı olduğunu `%timeit` ile gör (Şekil 24.4'i kendi zamanlamanla yeniden üret — §7-8).

Egzersiz 3 (Edge case). `stride` ve `padding` değerlerini değiştir; çıktı boyutunun nasıl değiştiğini (formül: $(n + 2p - k) / s + 1$) gözlemler (Şekil 24.5'i kendi değerlerinle doğrula — §9).

Egzersiz 4 (Kavramsal). Bir convolution'un neden bir matris çarpımı olduğunu `im2col` ile bir 3×3 örnek üzerinde elle göster (Şekil 24.3'un mantığını kâğıt üzerinde izle — §6).

Egzersiz 5 (Sonraki dersin habercisi — Ders 16). Eğitim döngüsündeki tekrar eden parçaları (before/after batch, epoch) bir callback sistemine nasıl ayırabileceğini düşün (Learner'a giden yol — §10).

24.21 Sonraki: Ders 16 İçin Hazırlık

Ders 16: Learner Çerçevesi (The Learner framework)

Ders 15 convolution + autoencoder kurdu ve bir eğitim döngüsü kullandı. Ders 16, bu döngüyü esnek bir **Learner** çerçevesine dönüştürür: **callback**'ler ile eğitimin her adımına müdahale. Burada fast.ai'nin (ve "miniai"nin) çekirdeği sıfırdan doğar.

Ana konular (Ders 16):

- Esnek eğitim döngüsü (Learner)
- Callback sistemi (before/after fit/epoch/batch)
- `CancelFit/Epoch/Batch` exceptions
- miniai'nin doğuşu

⚠ Ders 16 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 2 ve 5 — conv yöntemleri + callback fikri).
- Kendi autoencoder'ını MNIST'te eğit; reconstruction'ı gözle.
- Ana cümleyi tekrar oku: "Convolution = kaydırılan matmul; autoencoder = sıkıştır + geri aç."

24.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Convolution	Kaydırılan kernel'in her konumda dot product'ı	0:53
Kernel	Küçük filtre matrisi (örn. 3×3); öznelik dedektörü	5:27
apply_kernel	Bir yama × kernel toplamı (sıfırdan conv)	13:24
im2col	Convolution'ı matris çarpımına çevirme (yamaları açma)	19:30
F.unfold	PyTorch'un im2col'u; girdiyi yamalara açar	22:27
F.conv2d	Optimize convolution (cuDNN); en hızlı	22:27
stride / padding	Adım büyüklüğü / kenar koruma; çıktı boyutunu belirler	27:26
Autoencoder	Girdiyi darboğazdan geçirip yeniden kurma	15:50
Encoder	Conv ile küçültme (sıkıştırma)	15:50
Decoder (deconv)	UpsamplingNearest + conv ile büyütme	15:50
Reconstruction loss	Çıktı vs girdi farkı (self-supervised)	15:50
VAE köprüsü	Autoencoder = SD'nin latent encoder/decoder'ı	—

24.23 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: Convolution + VAE

Bu ders convolution'ı sıfırdan kurar ve autoencoder ile Stable Diffusion'ın VAE'sine köprü atar; köprülerin özeti:

1. **Convolution = matmul (im2col)** → “her şeyi matmul'a çevir” (Ders 10-12); cuDNN ile GPU-paralel ([im2col](#)).
2. **Parametre paylaşımı** → görüntü için doğru inductive bias (Ders 8); az parametre, genelleme ([kernel](#)).
3. **Autoencoder** → encoder/decoder; Stable Diffusion VAE'sinin (Ders 9/25) temeli ([VAE köprüsü](#)).
4. **Reconstruction (self-supervised)** → etiketsiz öğrenme; Ders 9 (EBM + self-supervised) ile akraba, NYU §4.J'nin merkez teması ([reconstruction](#)).
5. **F.conv2d** → sıfırdan kurduktan sonra optimize aracı kullan (Ders 10 kuralı) ([F.conv2d](#)).
6. **Latent sıkıştırma** → pahalı işlemi küçük temsilde yapma; latent diffusion verimliliği ([VAE köprüsü](#)).

! Bu dersten tek bir şey alıp gideceksen

Convolution gizemli değildir — kaydırılan bir kernel'in dot product'ı, im2col ile sıradan bir matris çarpımına indirgenir (ve böylece Ders 10-12'nin hızlı matmul'unu kullanır). Üstüne kurduğumuz autoencoder, görüntüyü conv'la sıkıştırıp deconv'la geri açar; bu, Stable Diffusion'ın VAE'sinin ta kendisidir. Temeller B'nin görü/sıkıştırma ayağını kurduk.

25 Ders 16 — Learner Çerçevesi (The Learner framework)

Temeller B’nin kalbi: esnek bir eğitim çerçevesi olan Learner’ı sıfırdan kurmak. Howard katı bir eğitim döngüsünden başlar (çalışır ama hiçbir yerine müdahale edilemez), onu callback’lerle genişletir — eğitimin her adımına (before/after fit/epoch/batch) çekirdek koda dokunmadan müdahale edebileceğimiz bir sistem — ve Cancel exception’larla akış kontrolünü callback’lere taşır. run_cbs callback’leri order’a göre sıralayıp getattr ile metotlarını çağırır; cb_ctx context manager before/after/cleanup’ı tek yapıda toplar. Tüm bu sistem miniai’dır: Ders 2’de kara kutu kullandığımız fastai Learner’ın sıfırdan kurulmuş, anlaşılır çekirdeği ve Part 2’nin (L17-25) eğitim altyapısı. Tek cümleyle: Learner eğitim döngüsünü yürütür; callback’ler ise o döngüye çekirdeğe dokunmadan müdahale eden, sınırsız esneklik veren eklentilerdir.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 16: The Learner framework](https://course.fast.ai/lesson-16) (~86 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 16
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/09_learner](#)
- **Okuma süresi:** ~36 dk
-  **Temeller B’nin kalbi (ETAP 5):** Ders 15 convolution + autoencoder’ı kurup eğitim döngüsünü hâlâ elle yazmıştık. Ders 16 o döngüyü esnek bir **Learner**’a dönüştürür: callback’lerle her aşamaya müdahale. Burada fast.ai’nin (ve “miniai” mini-kütüphanesinin) çekirdeği sıfırdan **doğar** — L17-25’in tüm eğitim altyapısı.

25.1 Bu Derste Ne Var?

Temeller B’nin kalbi: esnek bir eğitim çerçevesi olan **Learner**’ı sıfırdan kurmak. Howard, katı bir eğitim döngüsünden başlayıp onu **callback**’lerle genişletir — eğitimin her adımına (fit/epoch/batch başında ve sonunda) müdahale edebileceğimiz bir sistem. Burada fast.ai’nin (ve “miniai” mini-kütüphanesinin) çekirdeği doğar.

Üç temel fikir bu dersin omurgasını kurar:

1. **Learner** — model + veri + loss + optimizer’ı tek nesnede toplayan ve eğitim döngüsünü (fit → epoch → batch) yürüten yapı ([basit Learner](#) → [callback’li Learner](#)).
2. **Callback** — eğitimin belirli noktalarına (before_fit, after_batch, ...) takılan, davranışı **çekirdek koda dokunmadan** değiştiren eklentiler ([callback fikri](#) → [run_cbs](#)).
3. **Cancel exceptions** — bir callback’in fit/epoch/batch’i temiz biçimde **iptal** etmesi (CancelFitException vb.); akış kontrolü ([Cancel exceptions](#)).

“Welcome to Lesson 16, where we are working on building our first flexible training framework: the Learner.” — Howard, 0:00

Şekil 34.1 bu yolculuğu tek bir yol haritasında birleştirir: üstte **çekirdek döngü** (Learner → fit → epoch → batch → callback çağrısı → run_cbs), ortada **callback’ler** (DeviceCB, CompletionCB, MetricCB — order’a göre) ve **Cancel exception’lar**, altta ise hepsinin vardıđı yer: **açık/kapalı ilkesi** (çekirdek KAPALI, davranış AÇIK) ve onun ürünü olan **miniai** — fast.ai çekirdeđi sıfırdan.

💡 Builder Notu — Temeller B’nin Kalbi: Learner + Callback Sistemi

- **Geriye (Ders 2/14):** Ders 2’de fastai Learner’ı kara kutu kullandık; burada sıfırdan kuruyoruz. Ders 14’ün nn.Module + eğitim döngüsü üstüne.
- **İleriye (Ders 17-25):** miniai Learner, init/norm (17), accelerated SGD (18) ve tüm diffusion eğitimlerinin (19-25) altyapısı; callback’ler her şeyi mümkün kılar.
- **Tek cümle:** Learner eğitim döngüsünü yürütür; callback’ler ise çekirdeđe dokunmadan o döngüye müdahale eden, sonsuz esneklik veren eklentilerdir.

25.2 1. Hedef: Esnek Eğitim Çerçevesi

Howard amacı koyar: ilk **esnek eğitim framework**’ünü (Learner) kurmak. Ders 14-15’te eğitim döngüsünü elle yazdık ama katıydı (her şey sabit). Şimdi her adımı özelleştirilebilir hâle getireceğiz. Hedefi baştan ortaya koymak, sonraki her adımın (callback, run_cbs, Cancel) neye hizmet ettiđini netleştirir.

“We are working on building our first flexible training framework: the Learner.” — Howard, 0:00

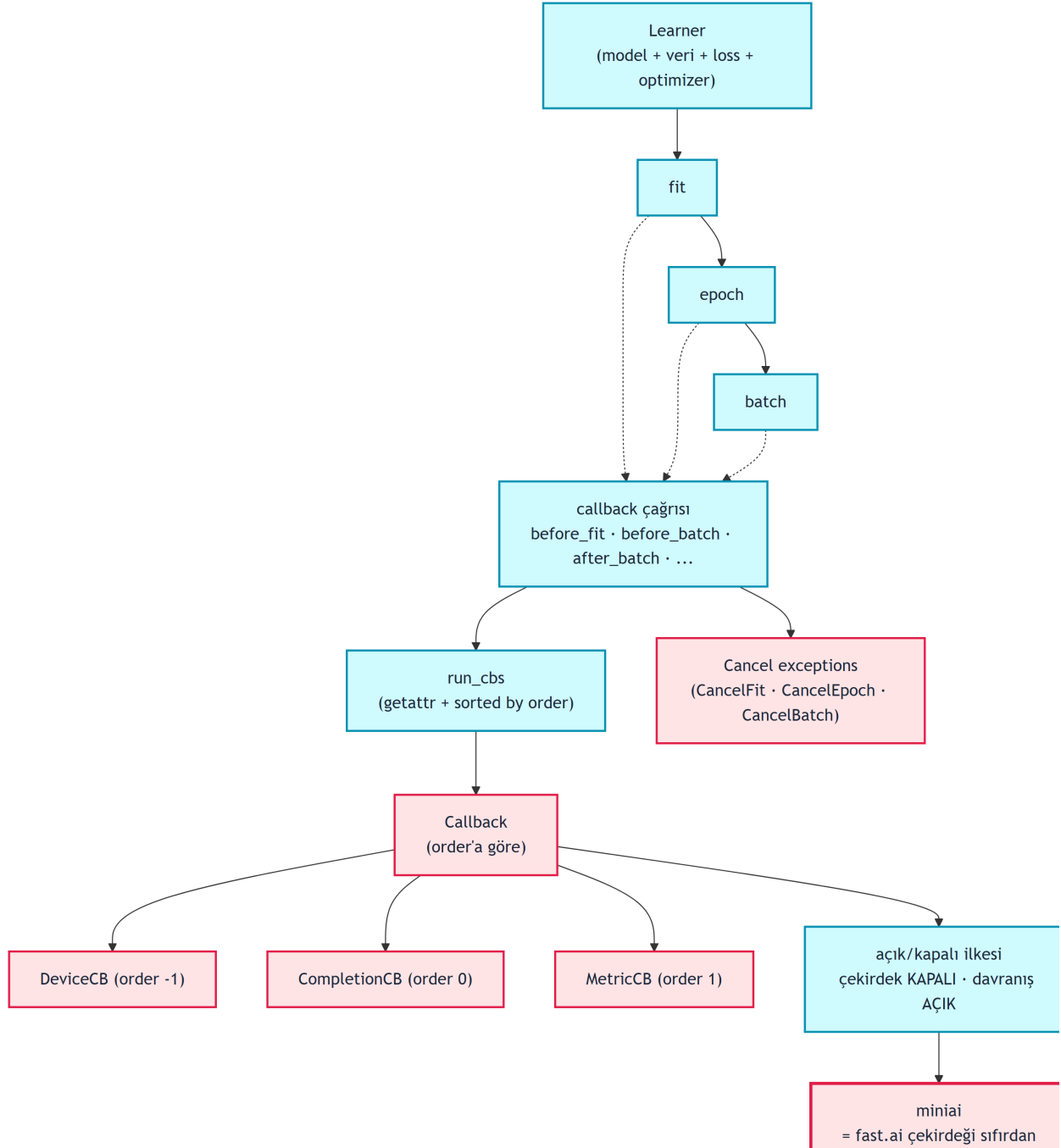
💡 Builder Notu — Hedef Önce: Esnekliđi Tasarla

- **Geriye (Ders 2):** fastai Learner’ın (Ders 2) sıfırdan, anlaşılır versiyonu — “yeniden kur, sonra kullan” kuralının doğrudan uygulaması.
- **Sezgi:** “Esneklik” burada soyut bir slogan deđil; somut bir hedef — eğitim döngüsünü deđiştirtilmeden ona yeni davranış ekleyebilmek. Bu hedef, callback desenini kaçınılmaz kılar.

25.3 2. Basit (Esnek Olmayan) Learner

Önce katı bir Learner: fit epoch’larda döner, one_epoch DataLoader’daki batch’lerde döner, one_batch tahmin yapar, loss hesaplar, eğitimdeyse backward + optimizer adımı atar. Çalışır ama hiçbir yerine müdahale edemezsin.

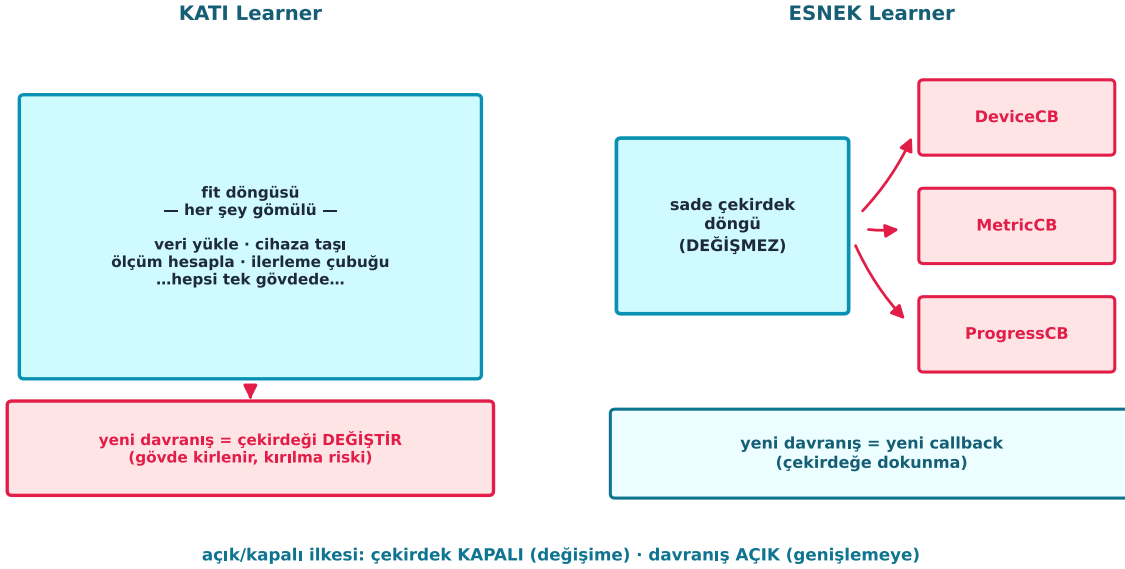
```
class Learner:
    def __init__(self, model, dls, loss_func, lr, opt_func=optim.SGD):
        fc.store_attr()
    def one_batch(self):
        self.preds = self.model(self.xb)
        self.loss = self.loss_func(self.preds, self.yb)
        if self.model.training:
            self.loss.backward(); self.opt.step(); self.opt.zero_grad()
```



Şekil 25.1: L16 kavram haritası: esnek Learner çerçevesi + callback sistemi (şematik). Çekirdek KAPALI, davranış AÇIK; miniai = fast.ai çekirdeği sıfırdan.

“We’ve seen so far this Learner which wasn’t flexible at all, but it had all the basic pieces.” — Howard, 0:45

Şekil 25.2 iki tasarımı yan yana koyar: solda **katı Learner** (her şey — veri yükleme, cihaza taşıma, ölçüm, ilerleme çubuğu — tek gövdeye gömülü; yeni davranış = çekirdeği değiştir), sağda **esnek Learner** (sade, değişmez çekirdek + ayrı ayrı takılan callback’ler; yeni davranış = yeni callback). Alt şerit dersin tasarım ilkesini özetler: açık/kapalı.



Şekil 25.2: (şematik: katı döngü (yeni davranış=çekirdeği değiştir) vs esnek Learner (yeni davranış=yeni callback); açık/kapalı ilkesi)

💡 Builder Notu — Basit Learner: Eğitim Adımı Artık Nesnede

- **Geriye (Ders 5/14):** `loss.backward()` + `opt.step()` + `opt.zero_grad()` = Ders 5/14’ün eğitim adımı; artık bir nesnenin metodu.
- **Sezgi:** Bu basit Learner “doğru parçalara” (model, veri, loss, optimizer, döngü) sahip — eksik olan tek şey **esneklik**. Çalışan ama katı bir temelden başlamak, esnekliğin tam olarak neyi çözdüğünü gösterir.

25.4 3. Neden Esnek Değil?

Sorun: bu döngüye yeni bir davranış eklemek (örn. GPU’ya taşı, metrik hesapla, erken durdur) için **çekirdek kodu değiştirmen** gerekir. Her yeni ihtiyaç kodu kirlendir ve kırılma riskini artırır. Çözüm: döngüyü değiştirmeden müdahale edebileceğin **kancalar** (hooks). Şekil 25.2’in sol panelindeki “çekirdeği DEĞİŞTİR” uyarısı tam bu sorunu işaret eder.

💡 Builder Notu — Katı Döngünün Maliyeti: Çekirdek Kirlenir

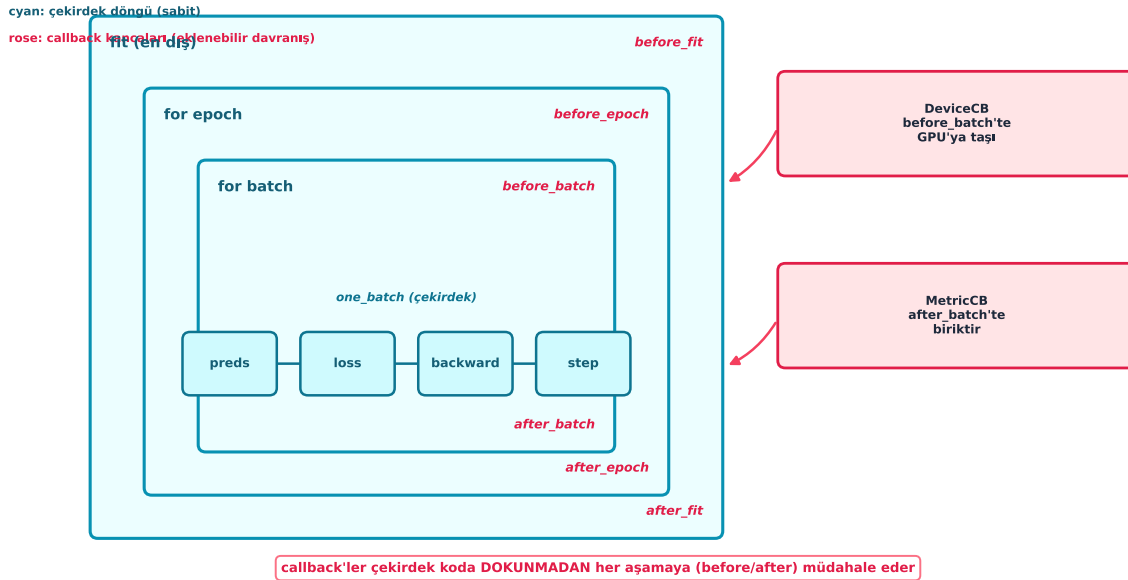
- **İleriye:** “Çekirdeğe dokunmadan genişlet” = açık/kapalı ilkesi (open-closed); callback’ler bunu sağlar — §15.
- **Sezgi:** Her yeni özelliği if’lerle çekirdeğe eklersen, döngü zamanla okunamaz hâle gelir ve bir değişiklik beklenmedik yan etkiler doğurur. “Değiştirilemez çekirdek” bir kısıt değil, bir koruma — bozulamayan bir temel.

25.5 4. Callback Fikri

Callback: eğitimin belirli noktalarına (`before_fit`, `after_batch`, `before_epoch`, ...) takılan bir nesne. Learner bu noktalara geldiğinde, kayıtlı tüm callback’lerin o isimli metodunu çağırır. Böylece davranış, çekirdek döngüye dokunmadan eklenir.

“We first of all call `self.callback` passing in ‘`before_fit`’.” — Howard, 0:45

Şekil 25.3 callback kancalarının nereye takıldığını gösterir (**FLAGSHIP**): iç içe çekirdek döngüler (`fit` → `for epoch` → `for batch` → `one_batch`) cyan ile sabittir; her seviyenin başında `before_*`, sonunda `after_*` kancası vardır; rose callback’ler (`DeviceCB` → `before_batch`, `MetricCB` → `after_batch`) bu kancalara takılır. Çekirdek döngü hiç değişmez; davranış kancalardan akar.



Şekil 25.3: (şematik: eğitim döngüsü + callback kancaları — before/after fit/epoch/batch; çekirdek döngü sabit, davranış callback’lerle eklenir)

💡 Builder Notu — Callback = Olay-Güdümlü Tasarım

- **İleriye:** Callback = “olay-güdümlü” (event-driven) tasarım; eğitim döngüsünün her aşaması bir **olaydır** (before_fit, after_batch, ...) ve callback o olaya abone olur.
- **Sezgi:** Bir callback yalnızca ilgilendiği olayları yazar — CompletionCB yalnızca before_fit/after_batch/after_fit tanımlar, diğer kancaları görmezden gelir. “Sadece ilgilendiğin yere takıl” deseni, callback’leri küçük ve bağımsız tutar.

25.6 5. Callback Taban Sınıfı ve order

Tüm callback’ler bir Callback taban sınıfından türer; bir order özniteliği vardır (çağırılma sırasını belirler, varsayılan 0). Bir callback yalnızca ilgilendiği metotları (örn. before_fit) yazar.

```
class Callback(): order = 0
```

💡 Builder Notu — order: Callback’ler Arası Bağımlılık

- **İleriye:** order, callback’ler arası bağımlılığı yönetir (örn. cihaz taşıma metrikten önce olmalı; daha küçük order daha önce çalışır). fast.ai’de explicit nonzero order taşıyan örnekler ProgressCB (order = MetricsCB.order + 1) ve SingleBatchCB’dir; çoğu callback (örn. DeviceCB, MetricsCB) taban sınıftan order = 0 miras alır.
- **Sezgi:** İki callback aynı kancaya takılıysa, hangisinin önce çalışacağı önemli olabilir (veri önce GPU’da olmalı ki metrik doğru hesaplınsın). order bu sırayı tek sayıyla, callback’lerin birbirini bilmesine gerek kalmadan çözer.

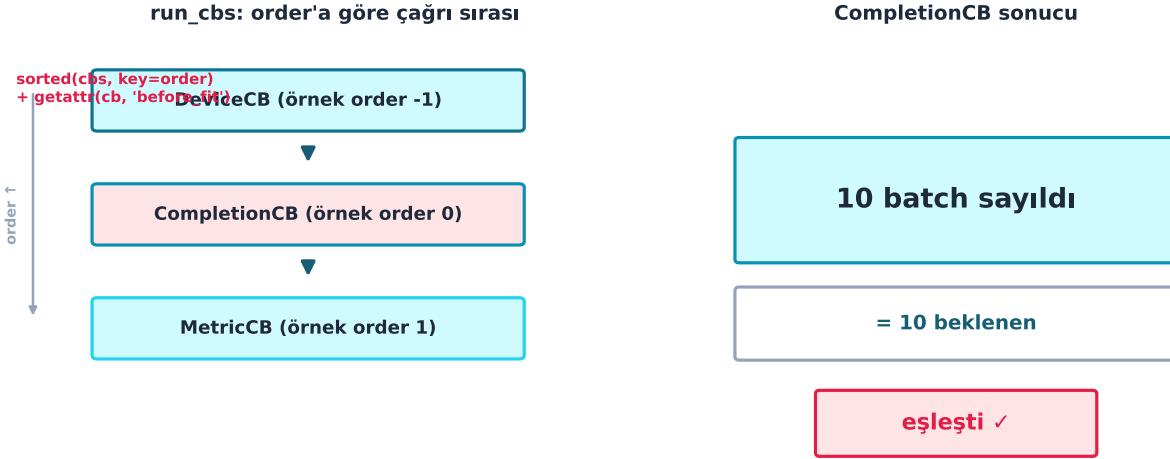
25.7 6. run_cbs: Callback’leri Çağırarak

Çekirdek mekanizma: run_cbs bir callback listesini alır, order’a göre sıralar ve her birinde verilen isimli metodu (varsa) çağırır. getattr ile metodu isimden bulur — sistemin tüm sihri bu tek fonksiyondadır.

```
def run_cbs(cbs, method_nm, learn=None):
    for cb in sorted(cbs, key=attrgetter('order')):
        method = getattr(cb, method_nm, None)
        if method is not None: method(learn)
```

“It uses getattr to find an attribute with this name, which is ‘before_fit’.” — Howard, 5:23

Şekil 25.4 run_cbs’in **çalışma akışını** gösterir: solda callback’ler order’a göre küçükten büyüğe sıralanır — liste sırası rastgele olsa da çağrı sırasını order belirler (şekildeki order değerleri illüstratiftir, fast.ai’nin gerçek atamaları değil); sağda CompletionCB’nin sonucu, saydığı batch sayısının beklenenle **eşleştiğini** (match) doğrular. getattr metodu isimden bulur; bir callback o metodu yazmamışsa (None) atlanır.



liste sırası rastgele ama order çağrı sırasını belirler; getattr metodu isimden bulur

Şekil 25.4: (çalışan kod: run_cbs callback'leri order'a göre küçükten büyüğe sıralar (order değerleri illüstratiftir), getattr ile metodu bulur; CompletionCB 10 batch saydı)

💡 Builder Notu — getattr: Callback Sisteminin Sihri

- **Geriye (Python):** `getattr(obj, 'name')` = bir niteliği isminden alma; callback sistemini mümkün kılan Python özelliği.
- **Sezgi:** run_cbs callback'lerin hangi metotlara sahip olduğunu **bilmek zorunda değil** — metot ismini bir string olarak alır, `getattr` ile arar, varsa çağırır, yoksa sessizce geçer. Bu dinamiklik, çekirdeği callback'lerin içeriğinden tamamen bağımsız kılar.

25.8 7. CompletionCB: İlk Callback Örneği

Howard basit bir örnek gösterir: CompletionCB batch'leri sayar. `before_fit` sayacı sıfırlar, `after_batch` artırır, `after_fit` yazdırır. Bu callback, çekirdek döngüye hiç dokunmadan yeni bir davranış ekler.

```
class CompletionCB(Callback):
    def before_fit(self, learn): self.count = 0
    def after_batch(self, learn): self.count += 1
    def after_fit(self, learn): print(f'Completed {self.count} batches')
```

💡 Builder Notu — CompletionCB: Tüm Callback'lerin Şablonu

- **İleriye:** Metrikler, loglama, görselleştirme — hepsi bu desende ayrı callback'ler olur (yalnızca ilgilendikleri kancayı yazar).
- **Sezgi:** CompletionCB sadece üç kancayı tanımlar; `before_epoch`, `before_batch` gibi diğerlerini yazmaz — run_cbs onları `getattr ... None` ile atlar. Bu, “en küçük callback” örneği: tek bir iş, üç satır, çekirdeğe sıfır dokunuş.

Şimdi Learner, döngünün her aşamasında `self.callback('before_fit')` gibi çağrılar yapar. `fit`, `one_epoch`, `one_batch` artık her adımda kayıtlı callback'leri tetikler. Çekirdek döngü sabit; davranış callback'lerle değişir.

```
class Learner():
    def __init__(self, model, dls, loss_func, lr, cbs, opt_func=optim.SGD):
        fc.store_attr()
    def one_batch(self):
        self.preds = self.model(self.batch[0])
        self.loss = self.loss_func(self.preds, self.batch[1])
        if self.model.training:
            self.loss.backward(); self.opt.step(); self.opt.zero_grad()
```

💡 Builder Notu — Aynı Adım, Eklenen Tek Şey: callback Çağrılar

- **Geriye (Ders 14):** Aynı eğitim adımı (`preds` → `loss` → `backward` → `step`); eklenen tek şey her aşamada `callback(...)` çağrıları.
- **Sezgi:** §2'deki katı Learner ile bu callback'li Learner arasındaki fark **minimaldir** — eğitim mantığı birebir aynı, fakat aralara serpiyen `callback('before...')` / `callback('after...')` çağrıları döngüyü sınırsız genişletilebilir kılar. Küçük bir ekleme, büyük bir esneklik.

25.10 9. Cancel Exceptions

Bir callback'in akışı **iptal** etmesi gerekebilir (örn. tek batch çalıştır, erken dur). Howard bunu özel istisnalarla yapar: `CancelFitException`, `CancelEpochException`, `CancelBatchException`. Bir callback bunları fırlatınca, Learner ilgili döngüyü temiz biçimde keser.

```
class CancelFitException(Exception): pass
class CancelBatchException(Exception): pass
class CancelEpochException(Exception): pass
```

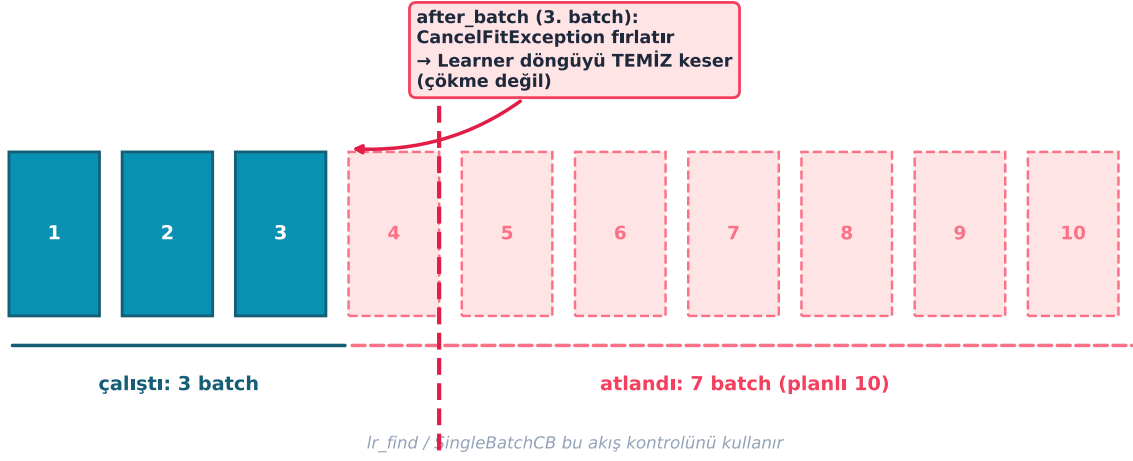
“*CancelFitException, CancelEpochException, and CancelBatchException.*” — Howard, 8:18

Şekil 25.5 bu mekanizmayı **gerçek çalışan kodla** gösterir: 10 batch planlanmış; bir callback 3. batch'in `after_batch`'inde `CancelFitException` fırlatır; Learner döngüyü **temiz** keser (çökme değil) — yalnızca 3 batch çalışır, 7'si atlanır. Akış kontrolü artık çekirdekte değil, callback'lerdedir; `lr_find` ve `SingleBatchCB` tam olarak bunu kullanır.

💡 Builder Notu — Cancel: Akış Kontrolü Callback'lere Taşınır

- **İleriye (Ders 17):** `lr_find` (Ders 17) bir callback'tir; learning rate çok büyüyüp loss patlayınca `CancelFitException` ile eğitimi durdurur.
- **Sezgi:** İstisnaları “hata” için değil, **kontrollü çıkış** için kullanmak güçlü bir desendir: bir callback “buraya kadar yeter” diyebilir, Learner'ın context manager'ı (`cb_ctx`) onu yakalar ve ilgili döngüyü

Cancel exception ile erken durdurma — 10 batch planlı, 3 batch çalıştı



Şekil 25.5: (gerçek çalışan kod: callback 3. batch'te CancelFitException → döngü temiz durur (10 yerine 3 batch); akış kontrolü callback'lerde)

sessizce keser. Erken durdurma, tek-batch testi, lr_find — hepsi bu üç istisnadan doğar.

25.11 10. DeviceCB: Pratik Bir Callback

Gerçek bir örnek: DeviceCB modeli ve her batch'i GPU'ya taşır. before_fit'te modeli, before_batch'te veriyi cihaza gönderir. GPU mantığı çekirdek döngüde değil, ayrı bir callback'te.

```
class DeviceCB(Callback):
    def before_fit(self, learn): learn.model.to(self.device)
    def before_batch(self, learn): learn.batch = to_device(learn.batch, self.device)
```

💡 Builder Notu — DeviceCB: Her Pratik Özellik = Bir Callback

- **İleriye:** Aynı desen: mixed precision (Ders 20), gradient clipping, logging — hepsi bağımsız callback.
- **Sezgi:** GPU'ya taşıma “her eğitimde lazım” görünse de çekirdeğe gömülmez — çünkü CPU'da çalışmak isteyebilirsin veya farklı cihaza taşıyabilirsin. Onu callback yapmak, “varsayılan ama zorunlu değil” davranışı temiz biçimde isteğe bağlı kılar.

25.12 11. Esnek Learner: Context Manager

Howard kodu daha da temizler: her aşamayı bir **context manager** (cb_ctx) ile sarar. before_<nm> çağrılır, iş yapılır, after_<nm> çağrılır; ilgili Cancel exception yakalanır; cleanup_<nm> her durumda çalışır.

```
@contextmanager
def cb_ctx(self, nm):
    try:
        self.callback(f'before_{nm}'); yield; self.callback(f'after_{nm}')
    except globals()[f'Cancel{nm.title()}Exception']: pass
    finally: self.callback(f'cleanup_{nm}')
```

💡 Builder Notu — cb_ctx: before/after/cleanup Tek Yapıda (DRY)

- **Geriye (Python):** @contextmanager + try/yield/finally; before/after/cleanup'ı tek yapıda toplar (DRY — kendini tekrar etme).
- **Sezgi:** cb_ctx üç şeyi tek desende birleştirir: olaydan önce before_, sonra after_, **her durumda** (Cancel olsa bile) cleanup_. globals()[f'Cancel{nm.title()}Exception'] ise aşama isiminden ilgili istisnayı dinamik bulur — getattr deseninin istisnalarındaki kuzeni. Bir context manager, üç callback noktasını ve istisna yakalamayı tek satıra indirir.

25.13 12. Metrics Callback

Doğruluk, kayıp gibi metrikler de birer callback'tir: after_batch'te biriktir, after_epoch'ta hesapla ve yazdır. Eğitim döngüsü metrik bilmez; metrik callback'i kendi işini yapar.

💡 Builder Notu — Metrik: Çekirdekten Ayrı, Yeniden Kullanılabilir

- **Geriye (Ders 5):** accuracy (Ders 5) hesabı; artık çekirdek döngüden ayrı, yeniden kullanılabilir bir callback.
- **Sezgi:** Metrik mantığı (biriktir → ortalama al → yazdır) eğitim döngüsünün işi değildir — döngü yalnızca “ileri-geri-adım” yapar. Metriği callback'e taşımak, aynı çekirdeği farklı metrik setleriyle (accuracy, F1, perplexity) çalıştırmayı tek satırlık değişiklikle mümkün kılar.

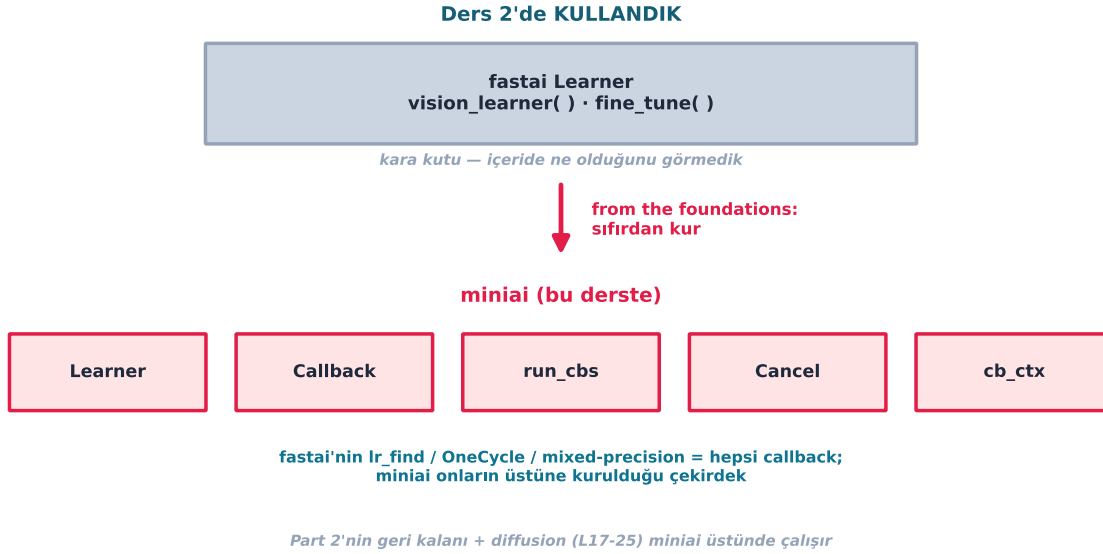
25.14 13. miniai'nin Doğuşu

Howard bu Learner + callback sistemini **miniai** adlı minik bir kütüphaneye koyar (#|export ile). miniai, Part 2'nin geri kalanında (ve diffusion'da) kullanılacak kendi mini fast.ai'mizdir — tamamen sıfırdan, anlaşılır.

Şekil 25.6 bu doğuşu görselleştirir: üstte **Ders 2'de kullandığımız** fastai Learner (vision_learner/fine_tune) — kara kutu, içeriğini görmedik; aradaki ok “from the foundations: sıfırdan kur”; altta **bu derste kurduğumuz** miniai çekirdeği (Learner · Callback · run_cbs · Cancel · cb_ctx). fastai'nin lr_find / OneCycle / mixed-precision'ı hep callback'tir; miniai onların üstüne kurulduğu çekirdektir.

💡 Builder Notu — miniai: « Yeniden Kur, Sonra Kullan »'ın Ürünü

- **Geriye (§4.1):** miniai = fastai'nin (Ders 1-8) sıfırdan, minimal versiyonu; “yeniden kur, sonra kullan” kuralının nihai ürünü.
- **İleriye (Ders 17-25):** Part 2'nin geri kalanı (init/norm, SGD) ve tüm diffusion eğitimleri (DDPM



Şekil 25.6: (şematik: miniai = fast.ai'nin sıfırdan, minimal çekirdeği — Ders 2'deki Learner'ın altındaki callback makinesi; L17-25 bunun üstüne)

→ DDIM → latent) miniai üstünde çalışır — yani şimdi kurduğumuz beş parça, kalan onlarca dersin altyapısıdır.

25.15 14. fast.ai Learner ile Köprü

Bu miniai Learner, Ders 2'de kullandığımız fastai Learner'ın çekirdeğidir. fastai'nin tüm callback ekosistemi (lr_find, OneCycle, mixed precision) bu desenin üstüne kuruludur. Artık fastai'yi kullanırken altında ne olduğunu biliyoruz.

💡 Builder Notu — Köprü: Artık Kara Kutu Şeffaf

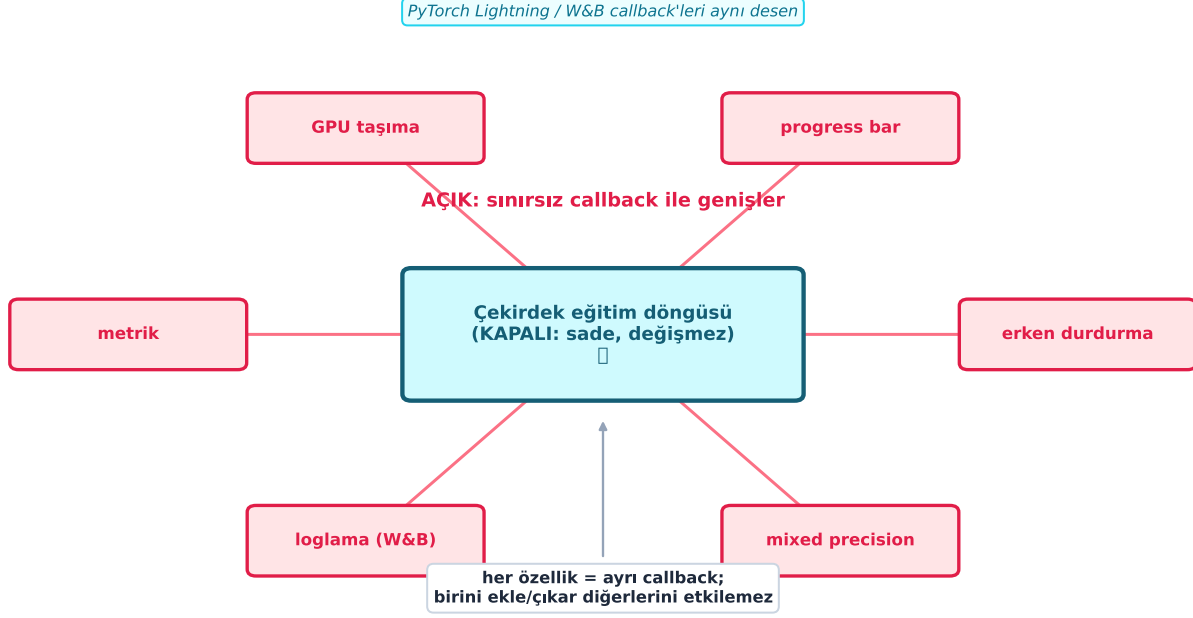
- **Geriye (Ders 2):** Ders 2'deki vision_learner/fine_tune'un altındaki callback makinesi tam budur — [Şekil 25.6](#) bu kara-kutu → çekirdek geçişini gösterir.
- **Sezgi:** fast.ai'yi “büyülü” yapan şey, onlarca yerleşik callback'tir (lr_find, OneCycle, mixed precision); ama her biri bu derste kurduğumuz Callback taban sınıfından türer. Çekirdeği bildiğimiz için, beklenmedik bir davranışta nereye bakacağımızı ve kendi callback'imizi nasıl yazacağımızı biliyoruz.

25.16 15. Neden Callback Deseni Güçlü?

Callback deseni, “açık/kapalı ilkesi”ni mükemmel uygular: çekirdek eğitim döngüsü **kapalı** (değişmez, sade), ama davranış **açık** (sınırsız callback ile genişler). Yeni bir özellik = yeni bir callback; çekirdek koda hiç

dokunmazsın. Bu, fast.ai'yi bu kadar esnek yapan tasarımıdır.

Şekil 25.7 bu ilkeyi merkez-çevre olarak çizer: ortada **çekirdek eğitim döngüsü** (KAPALI: sade, değişmez, kilitli), çevrede takılan callback'ler (GPU taşıma, metrik, loglama, mixed precision, erken durdurma, progress bar) — her özellik ayrı bir callback; birini ekle/çıkar diğerlerini etkilemez. Aynı desen PyTorch Lightning / W&B callback'lerinde de standarttır.



Şekil 25.7: (şematik: açık/kapalı ilkesi — çekirdek döngü kapalı/değişmez, davranış callback'lerle sınırsız açık; fast.ai esnekliğinin sırrı)

💡 Builder Notu — Açık/Kapalı: Production'da da Standart

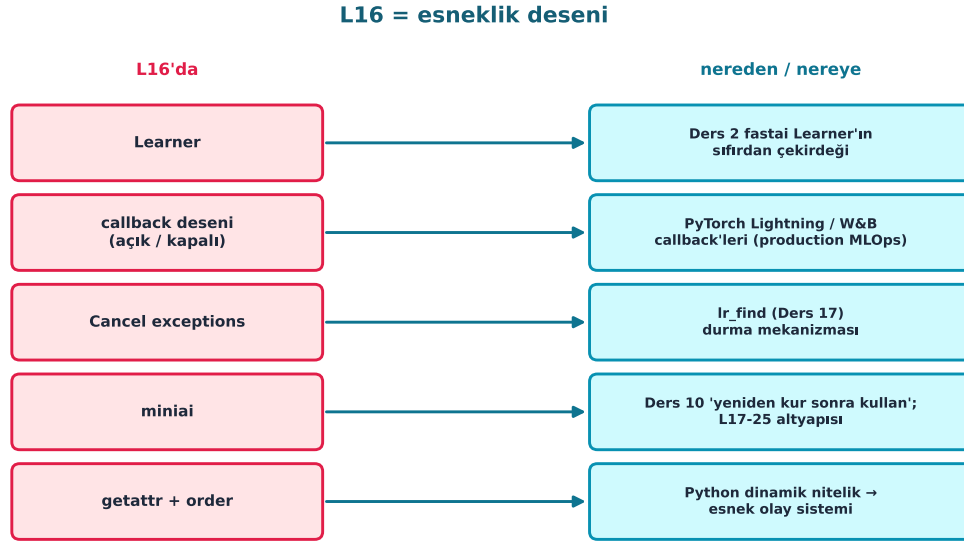
- **İleriye (production):** Aynı desen MLOps framework'lerinde (W&B, PyTorch Lightning call-backs) standarttır; öğrenmesi builder için değerli.
- **Sezgi:** “Çekirdek kapalı, davranış açık” sadece şık bir tasarım değil — bakım maliyetini düşürür: yeni özellik eklemek mevcut kodu **okumayı/değiştirmeyi gerektirmez**, sadece yeni bir callback yazarsın. Bu yüzden olgun framework'ler (Lightning, Keras, fast.ai) hep aynı deseni seçer.

25.17 16. Kapanış

Ders 16, esnek eğitim çerçevesi Learner'ı sıfırdan kurdu: katı döngü → callback'lerle her aşamaya müdahale → Cancel exceptions ile akış kontrolü → context manager ile temizlik → miniai kütüphanesi. Bu, fast.ai'nin çekirdeği ve Part 2'nin geri kalanının eğitim altyapısıdır.

Şekil 25.8 dersin sentezidir (**callback deseni merkezde**): solda L16'nın parçaları (Learner, callback deseni, Cancel exceptions, miniai, getattr + order), sağda her birinin kaynağı/hedefi — Learner = Ders 2 fastai

Learner'ın çekirdeği, callback deseni = PyTorch Lightning / W&B (production MLOps), Cancel = lr_find (Ders 17) durma mekanizması, miniai = Ders 10 "yeniden kur sonra kullan" + L17-25 altyapısı, getattr + order = Python dinamik nitelik → esnek olay sistemi.



Şekil 25.8: (sentez: L16 = callback deseni (açık/kapalı ilkesi); Ders 2 Learner çekirdeği, miniai L17-25 altyapısı, production callback köprüsü)

💡 Builder Notu — Kapanış: Sırada Eğitilmesi Zor Ağlar

- **İleriye (Ders 17-18):** miniai hazır; Ders 17 (init/norm) ve 18 (accelerated SGD) callback ve Learner üstüne kurulur; diffusion eğitimleri (19-25) de.
- **Sezgi:** Bu derste **eğitim altyapısı** olgunlaştı (model ve döngü artık ayrı soyutlama katmanları). Sıradaki dersler bu altyapıyı kullanarak asıl zorluğa döner: derin ağların neden eğitilmesinin zor olduğu (aktivasyonların patlaması/sönmesi) ve callback-tabanlı çözümleri (hook'lar, init, norm).

25.18 Bu Dersin Özeti

1. **Learner** model + veri + loss + optimizer'ı toplar ve eğitim döngüsünü (fit → epoch → batch) yürütür (**basit Learner**).
2. Katı bir döngü esnek değildir; yeni davranış için çekirdeği değiştirmek gerekir — kötü tasarım (**neden esnek değil**).
3. **Callback**, eğitimin belirli noktalarına (before_fit, after_batch, ...) takılan, çekirdeğe dokunmadan davranış ekleyen nesnedir (**callback fikri**).
4. **run_cbs** callback'leri order'a göre sıralayıp getattr ile ilgili metotlarını çağırır (**run_cbs**).
5. **Cancel exceptions** (Fit/Epoch/Batch) bir callback'in akışı temiz biçimde iptal etmesini sağlar (örn. lr_find) (**Cancel**).
6. **DeviceCB** gibi callback'ler (GPU taşıma, metrik, log) çekirdek döngüden bağımsız, yeniden kullanılabilir (**DeviceCB**).

7. **Context manager** (`cb_ctx`) `before/after/cleanup`'ı tek yapıda toplar; `Cancel exception`'ları yakalar (`cb_ctx`).
8. Tüm bu sistem **miniai**'dir — `fast.ai`'nin sıfırdan, anlaşılır çekirdeği; Part 2'nin eğitim altyapısı ([miniai](#)).

! Tek Bir Cümle

Learner eğitim döngüsünü yürütür; callback'ler ise eğitimin her aşamasına (`before/after fit/epoch/batch`) çekirdek koda dokunmadan müdahale eden eklentilerdir — ve bu desen, Ders 2'de kullandığımız `fastai` Learner'ın sıfırdan kurulmuş çekirdeği olan `miniai`'yi oluşturur.

25.19 Kontrol Soruları

i Soru 1: Callback sistemi nasıl çalışır? `run_cbs` ne yapar?

Cevap:

Learner, eğitim döngüsünün belirli noktalarında (`before_fit`, `before_batch`, `after_batch`, `after_epoch`, `after_fit`, ...) `self.callback('<isim>')` çağırır. Bu çağrı `run_cbs`'i tetikler: kayıtlı tüm callback'leri `order` özniteliğine göre sıralar ve her birinde o isimli metodu (`getattr(cb, method_nm, None)` ile bulup, varsa) çağırır. Böylece her callback yalnızca ilgilendiği aşamalara takılır (örn. `CompletionCB` yalnızca `before_fit/after_batch/after_fit` yazar). Sonuç: davranış, çekirdek döngüye hiç dokunmadan eklenir. (Şekil 25.3 kancaları, Şekil 25.4 order'a göre çağrı sırasını gösterir.)

i Soru 2: Callback deseni neden katı bir eğitim döngüsünden üstündür?

Cevap:

Katı döngüde yeni bir davranış (GPU'ya taşıma, metrik, erken durdurma, mixed precision) eklemek için **çekirdek kodu değiştirmen** gerekir; her ihtiyaç döngüyü kirletir ve hata riski artar. Callback deseni “açık/kapalı ilkesi”ni uygular: çekirdek döngü **kapalı** (sade, değişmez) kalır, davranış **açık** (yeni callback eklenerek) genişler. Her özellik bağımsız, yeniden kullanılabilir bir callback olur; birini eklemek/çıkarmak diğerlerini etkilemez. Bu, `fast.ai`'yi (ve `PyTorch Lightning` gibi framework'leri) bu kadar esnek yapan tasarımıdır. (Şekil 25.2 iki tasarımı, Şekil 25.7 açık/kapalı ilkesini gösterir.)

i Soru 3: `Cancel exceptions` ne işe yarar? Bir örnek ver.

Cevap:

`CancelFitException`, `CancelEpochException`, `CancelBatchException`, bir callback'in eğitim akışını **temiz biçimde iptal** etmesini sağlar. Bir callback bu istisnayı fırlatınca, Learner'ın context manager'ı (`cb_ctx`) onu yakalar ve ilgili döngüyü (`fit/epoch/batch`) sessizce keser — çökme değil, kontrollü çıkış. Örnek: **lr_find** (Ders 17) bir callback'tir; `learning rate`'i kademeli artırır, `loss` patlayınca `CancelFitException` fırlatıp eğitimi durdurur. Veya `SingleBatchCB`, tek batch sonrası `CancelFitException` ile hızlı test yapar. Bu, akış kontrolünü callback'lere taşır. (Şekil 25.5 10 batch'in 3'te temiz durmasını gerçek kodla gösterir.)

i Soru 4: miniai nedir ve fast.ai ile ilişkisi nedir? (builder bağlantısı)

Cevap:

miniai, bu derste sıfırdan kurduğumuz Learner + callback sistemini (ve sonraki derslerde eklenecek init, optimizer, vb.) içeren minik bir kütüphanedir — fast.ai’nin **minimal, anlaşılır çekirdeği**. Ders 2’de kullandığımız fastai Learner (vision_learner/fine_tune) tam olarak bu callback makinesinin üstüne kuruludur; fastai’nin lr_find, OneCycle, mixed precision gibi tüm özellikleri birer callback’tir. Builder açısından: “from the foundations” kuralının ürünü — fast.ai’nin eğitim çekirdeğini sıfırdan kurduk, artık onu (ve callback ekleyerek genişletmeyi) tam anlıyoruz. Part 2’nin geri kalanı ve diffusion eğitimleri miniai üstünde çalışır. (Şekil 25.6 kara-kutu → çekirdek geçişini gösterir.)

25.20 Egzersizler

Egzersiz 1 (Direkt uygulama). CompletionCB’yi yaz ve bir Learner’a ver; eğitim sonunda kaç batch işlendiğini yazdırdığını gör (Şekil 25.4’in sağ panelini kendi sayımınla doğrula — §7).

Egzersiz 2 (İki-aşamalı). Kendi callback’ini yaz (örn. her epoch sonunda loss yazan); order ile bir başka callback’ten önce/sonra çalışmasını sağla (§5-6).

Egzersiz 3 (Edge case). SingleBatchCB gibi after_batch’ta CancelFitException fırlatan bir callback yaz; eğitimin tek batch sonrası temiz durduğunu gözle (Şekil 25.5’in mantığını kendi callback’inle yeniden üret — §9).

Egzersiz 4 (Kavramsal). run_cbs’in getattr’ı neden kullandığını açıkla; bir callback yalnızca before_fit yazıp after_batch yazmazsa ne olur? (§6)

Egzersiz 5 (Sonraki dersin habercisi — Ders 17). Eğitim sırasında her katmanın aktivasyon istatistiklerini (ortalama/std) toplayan bir callback (hook) nasıl yazılır düşün (init/norm’a giriş — §16).

25.21 Sonraki: Ders 17 İçin Hazırlık

Ders 17: Başlatma / Normalizasyon (Initialization/normalization)

Ders 16 Learner + callback’leri (miniai) kurdu. Ders 17, bu altyapıyla derin ağların **neden eğitilmesinin zor** olduğunu inceler: aktivasyonların patlaması/sönmesi. Çözümler: doğru **başlatma** (Kaiming init) ve **normalizasyon** (BatchNorm/LayerNorm). Karpathy makemore 3’ün fast.ai karşılığı.

Ana konular (Ders 17):

- Hooks ile aktivasyon istatistikleri (callback)
- Aktivasyonların patlaması/sönmesi
- Kaiming (He) init — varyans koruma
- BatchNorm / LayerNorm

⚠ Ders 17 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 3 — callback + Cancel exception).
- Kendi Learner'ını callback'lerle kurup MNIST'te eğit.
- Ana cümleyi tekrar oku: “Learner döngüyü yürütür; callback'ler dokunmadan müdahale eder.”

25.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Learner	Model+veri+loss+optimizer'ı toplayan eğitim döngüsü	0:00
Callback	Eğitim noktalarına takılan, çekirdeğe dokunmayan eklenti	0:45
before_fit / after_batch	Callback'ın takıldığı aşama isimleri	0:45
run_cbs	Callback'leri order'a göre sıralayıp metotlarını çağırır	5:23
order	Callback çağırılma sırası (varsayılan 0)	5:23
getattr	Bir niteliği isminden bulma (callback sihri)	5:23
CompletionCB	İlk örnek callback (batch sayar)	4:30
Cancel exceptions	Fit/Epoch/Batch'i temiz iptal (akış kontrolü)	8:18
DeviceCB	Model/veriyi GPU'ya taşıyan callback	—
cb_ctx	before/after/cleanup'ı saran context manager	—
miniai	Sıfırdan kurulan minimal fast.ai çekirdeği	—
Açık/kapalı ilkesi	Çekirdek kapalı, davranış callback ile açık	—

25.23 ML Bağlantıları Özeti**💡 Builder Notu — 6 ML Köprüsü: Callback Deseni + miniai**

Bu ders esnek eğitim çerçevesini sıfırdan kurar ve callback deseniyle production framework'lerine köprü atar; köprülerin özeti:

1. **Learner** → Ders 2 fastai Learner'ın sıfırdan çekirdeği; eğitim döngüsünü toplar (**köprü**).
2. **Callback deseni** → açık/kapalı ilkesi; PyTorch Lightning / W&B callback'lerinin temeli (**neden**

- güçlü).
3. **Cancel exceptions** → akış kontrolü; lr_find (Ders 17) gibi callback'lerin durma mekanizması ([Cancel](#)).
 4. **miniai** → “yeniden kur, sonra kullan” (Ders 10); fast.ai'nin minimal versiyonu; L17-25 altyapısı ([miniai](#)).
 5. **getattr + order** → Python'un dinamik niteliğiyle esnek olay sistemi ([run_cbs](#)).
 6. **DeviceCB / metrik CB** → çekirdek döngüden bağımsız, yeniden kullanılabilir davranış ([DeviceCB](#)).

! Bu dersten tek bir şey alıp gideceksen

fast.ai'nin esnekliğinin sırrı **callback desenidir**. Eğitim döngüsünü (Learner) sade ve değişmez tutar, her aşamaya (before/after fit/epoch/batch) callback'lerle müdahale edersin — GPU taşıma, metrik, erken durdurma, hepsi ayrı eklenti, çekirdeğe dokunmadan. Bunu sıfırdan kurduk (miniai); artık Ders 2'deki fastai Learner'ın altında ne olduğunu biliyoruz.

26 Ders 17 — Başlatma / Normalizasyon (Initialization/normalization)

Temeller B'nin teknik zirvesi: derin ağların neden eğitilmesinin zor olduğunu ve bunu çözen iki büyük fikri kurmak — doğru başlatma (initialization) ve normalizasyon. Howard önce ağın içine hook'larla bakar (her katmanın aktivasyon istatistikleri), sorunu (aktivasyonların patlaması/sönmesi) gözle görür, sonra matematiğini (varyans/kovaryans) kurup Xavier ve Kaiming init'i türetir; ardından BatchNorm/LayerNorm'u sıfırdan yazar. Tek cümleyle: derin ağ eğitiminin sırrı aktivasyon varyansını katmandan katmana sabit tutmaktır — bunu başlatma (Kaiming) baştan, normalizasyon (BatchNorm) sürekli sağlar. Ve bu, Karpathy'nin makemore 3'te elle keşfettiği şeyin tam fast.ai karşılığıdır.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 17: Initialization/normalization](https://course.fast.ai/lesson-17-initialization-normalization) (~116 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 17
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/10_activations + nbs/11_initializing](#)
- **Okuma süresi:** ~42 dk
-  **Temeller B'nin teknik zirvesi (ETAP 5):** Ders 16'da esnek **Learner**'ı (miniai) kurduk; eğitim altyapısı hazır. Ders 17 o altyapıyla **asıl zorluğa** döner: derin ağların neden eğitilmesinin zor olduğu (aktivasyonların patlaması/sönmesi) ve callback-tabanlı çözümleri — hook'lar, init, norm. Bu ders **Karpathy makemore 3'ün birebir fast.ai karşılığıdır**.

26.1 Bu Derste Ne Var?

Temeller B'nin teknik zirvesi: derin ağların **neden eğitilmesinin zor** olduğunu ve bunu çözen iki büyük fikri kurmak — doğru **başlatma** (initialization) ve **normalizasyon**. Howard önce ağın içine **hook**'larla bakar (her katmanın aktivasyon istatistikleri), sorunu (aktivasyonların patlaması/sönmesi) gözle görür, sonra matematiğini (varyans/kovaryans) kurup Xavier ve Kaiming init'i türetir; ardından BatchNorm/LayerNorm'u sıfırdan yazar.

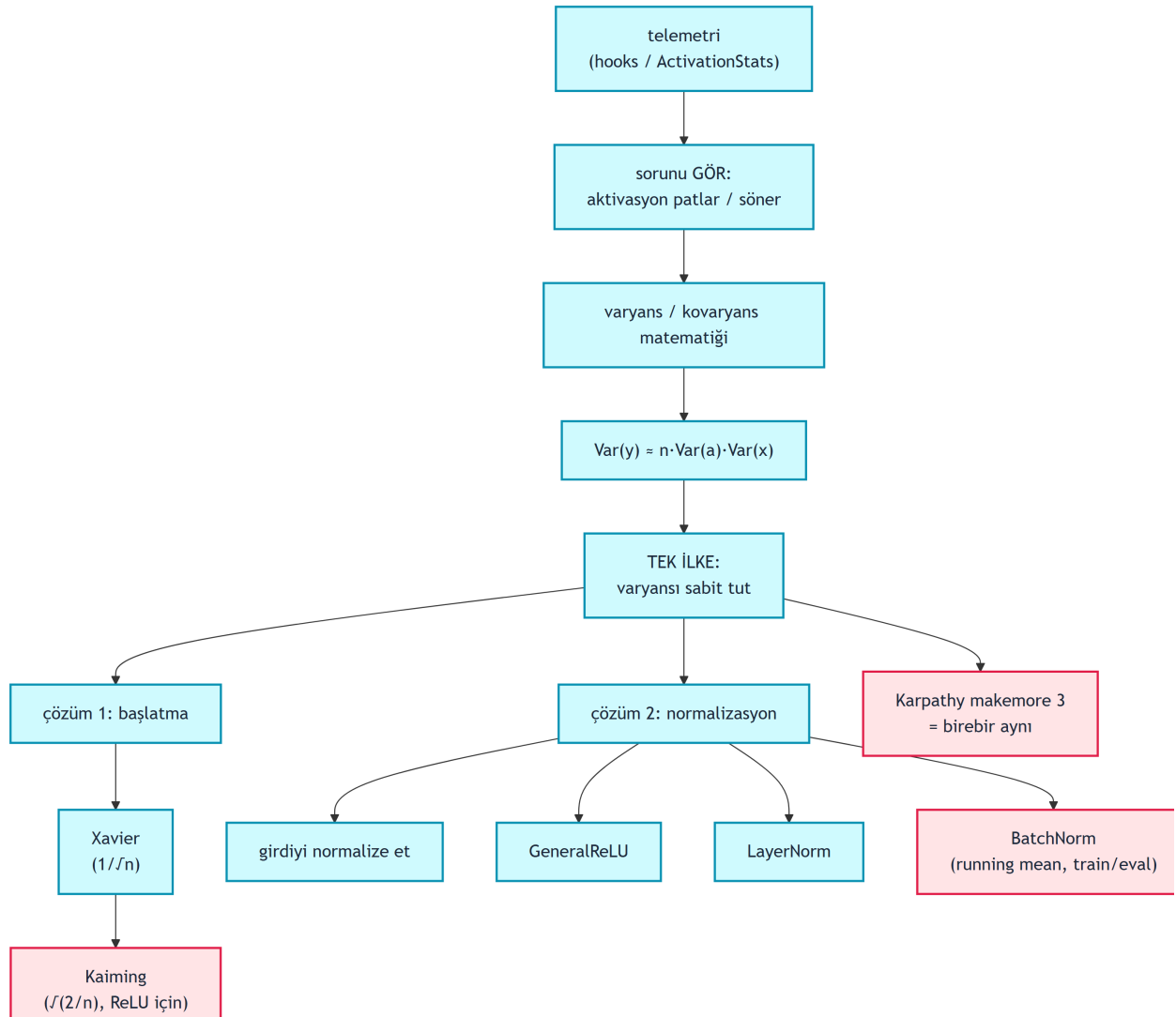
Üç temel fikir bu dersin omurgasını kurar:

1. **Telemetri (hooks)** — eğitim sırasında her katmanın aktivasyon ortalama/std'sini izlemek; sorunu *görmek* çözmenin ilk adıımıdır (**telemetri** → **hook** → **ActivationStats**).
2. **Başlatma (Kaiming init)** — ağırlıkları öyle ölçekle ki aktivasyon varyansı katmandan katmana sabit kalsın; patlama/sönme önlenir (**Xavier** → **Kaiming**).

3. **Normalizasyon (BatchNorm/LayerNorm)** — aktivasyonları her adımda yeniden merkezleyip ölçekleyerek eğitimi sağlamlaştıır (**LayerNorm** → **BatchNorm**).

“I’m really excited about what we’re going to look at over the next lesson or two.” — Howard, 0:07

Şekil 34.1 bu yolculuğu tek bir yol haritasında birleştirir: solda **telemetri** (hooks → sorunu gör → varyans/kovaryans matematiği), ortada **tek ilke** (varyansı sabit tut), sağda iki çözüm — **başlatma** (Xavier $1/\sqrt{n}$ → Kaiming $\sqrt{2/n}$) ve **normalizasyon** (girdi normalize / GeneralReLU / LayerNorm / BatchNorm). Kaiming, BatchNorm ve **Karpathy makemore 3 köprüsü** rose ile işaretlidir: bu ders, Karpathy’nin bottom-up kurduğu şeyin top-down karşılığıdır.



Şekil 26.1: L17 kavram haritası: tek ilke ‘varyansı sabit tut’ — telemetriyle sorunu gör, varyans matematiğiyle anla, başlatma (Xavier→Kaiming) ve normalizasyon (girdi/GeneralReLU/LayerNorm/BatchNorm) ile çöz. Kaiming, BatchNorm ve Karpathy makemore 3 köprüsü rose ile işaretli. (şematik)

💡 Builder Notu — Temeller B’nin Zirvesi: Telemetri → Matematik → Çözüm

- **Geriye (Karpathy makemore 3):** Bu ders, Karpathy’nin makemore 3’te elle keşfettiği her şeyin fast.ai karşılığı: aktivasyon istatistiklerini izleme, Kaiming init ile varyans koruma, BatchNorm. İki kurs burada birebir buluşur — Howard top-down (önce hook’la gör), Karpathy bottom-up (önce elle türet).
- **İleriye (Ders 18-25):** Sağlam init + norm olmadan derin ağlar (ResNet, U-Net, diffusion) eğitilemez; bu ders sonraki her şeyin önkoşulu.
- **Tek cümle:** Derin ağ eğitiminin sırrı, aktivasyon varyansını katmandan katmana sabit tutmaktır — bunu başlatma (Kaiming) baştan, normalizasyon (BatchNorm) sürekli sağlar.

26.2 1. Telemetri: Ağın İçine Bakmak

Howard’ın ilk hamlesi tedavi değil **teşhis**: eğitim sırasında her katmanın çıktısının ortalama ve standart sapmasını izlemek. “Sorunu göremezsen çözemezsin.” Bunun için miniai’ye bir HooksCallback ve ActivationStats eklendi — Ders 16’nın callback altyapısı üstüne kuruludur.

Şekil 26.2 bu izleme mekanizmasını gösterir: model katmanlarına `register_forward_hook` ile bir kanca takılır, `ActivationStats` callback’i her katman çıktısının ortalama/std/histogramını biriktirir, sonra bunları `color_dim` (aktivasyon histogramının zaman içindeki evrimi) ve `dead_chart` (ölü ünite oranı) ile görselleştirir. Karpathy makemore 3 de tam bu histogramları çizer.

Sorunu GÖRMEYEN çözemezsin: kötü init’te grafik patlamayı/çökmeyi gösterir · Ders 16 callback altyapısı üstüne



Şekil 26.2: (şematik: telemetri — `register_forward_hook` ile katman aktivasyon istatistiklerini izle (`ActivationStats` callback); sorunu görmek çözmenin ilk adımı)

💡 Builder Notu — Telemetri: Önce Teşhis, Sonra Tedavi

- **Geriye (Ders 16):** Telemetri bir callback’tir; Ders 16’da kurduğumuz Learner+callback sistemi olmadan bu izleme mümkün değildi — `ActivationStats` `HooksCallback`’ten türer.

- **Sezgi:** Howard’ın tüm dersi tek bir mühendislik refleksine dayanır: bir sorunu çözmeden önce onu **ölçülebilir** kıl. Aktivasyon std’sini her katmanda izlemek, patlama/sönmeyi soyut bir korku olmaktan çıkarıp bir grafiğe indirger — artık düzeltmenin işe yarayıp yaramadığını gözle görebiliriz.

26.3 2. Hook Nedir?

Hook, PyTorch’un bir katmana iliştirilen ve o katman her çalıştığında (forward/backward) otomatik tetiklenen bir fonksiyondur. `register_forward_hook` ile katmanın çıktısını yakalayıp kaydedebiliriz — modeli değiştirmeden içine bakarız.

```
class Hook():
    def __init__(self, m, f): self.hook = m.register_forward_hook(partial(f, self))
    def remove(self): self.hook.remove()
    def __del__(self): self.remove()

def append_stats(hook, mod, inp, outp):
    if not hasattr(hook, 'stats'): hook.stats = ([], [])
    acts = to_cpu(outp)
    hook.stats[0].append(acts.mean())
    hook.stats[1].append(acts.std())
```

💡 Builder Notu — Hook: Modeli Değiştirmeden İçine Bakma

- **İleriye:** Hook deseni her derin öğrenme framework’ünde vardır (PyTorch hooks, TF callbacks); model içini gözlemlemenin standart yolu. Aynı `register_forward_hook` ileride Grad-CAM, feature extraction, debugging için de kullanılır.
- **Sezgi:** `append_stats` yalnızca üç satır: çıktının ortalamasını ve std’sini ilgili kancanın `stats` listesine ekler. `partial(f, self)` ile kanca nesnesi fonksiyona bağlanır — böylece her katmanın kendi istatistik geçmişi olur. Modelin forward koduna tek satır eklemekten tüm aktivasyonları izleyebiliriz.

26.4 3. ActivationStats: Renkli Boyut ve Ölü Üniteler

Hook’larla topladığımız istatistikleri görselleştiririz: `color_dim` (her katmanın aktivasyon histogramının zamanla evrimi — “renkli boyut” grafiği) ve `dead_chart` (sıfır kalan, yani “ölü” ünitelerin oranı). Kötü init’te bu grafikler aktivasyonların önce patlayıp sonra çöktüğünü açıkça gösterir.

```
class ActivationStats(HooksCallback):
    def __init__(self, mod_filter=fc.noop): super().__init__(append_stats, mod_filter)
    def color_dim(self, figsize=(11,5)): ...
    def dead_chart(self, figsize=(11,5)): ...
```


💡 Builder Notu — color_dim: Karpathy ile Aynı Teşhis Grafiği

- **Geriye (Karpathy makemore 3):** Karpathy de aynı şeyi yapar — katman aktivasyonlarının histogramını çizip “ölü” nöronları (saturated tanh) gösterir. Aynı teşhis, aynı grafik; sadece Howard’ın aracı ActivationStats callback’i, Karpathy’ninki elle yazılmış döngü.
- **Sezgi:** color_dim zamanı (eğitim adımları) bir eksene, aktivasyon değerlerinin histogramını öteki eksene koyar; renk yoğunluğu o değerin ne sıklıkta görüldüğüdür. Sağlıklı bir katmanda histogram sabit ve dolu; kötü init’te önce yukarı patlar, sonra hepsi sıfıra (ölü) çöker — bozulmanın *tam ne zaman* başladığını gözle yakalarsın.

26.5 4. Sorun: Aktivasyonların Patlaması/Sönmesi

Basit bir deney sorunu çıplak gösterir: rastgele bir matrisi 50 kez çarp. Standart normal ağırlıkla aktivasyonlar nan’a patlar; çok küçük ağırlıkla ($\times 0.01$) sıfıra söner. İkisi de eğitimi imkânsız kılar.

```
x = torch.randn(200, 100)
for i in range(50): x = x @ torch.randn(100,100)
x[0:5,0:5] # nan – patladı

x = torch.randn(200, 100)
for i in range(50): x = x @ (torch.randn(100,100) * 0.01)
x[0:5,0:5] # 0 – söndü
```

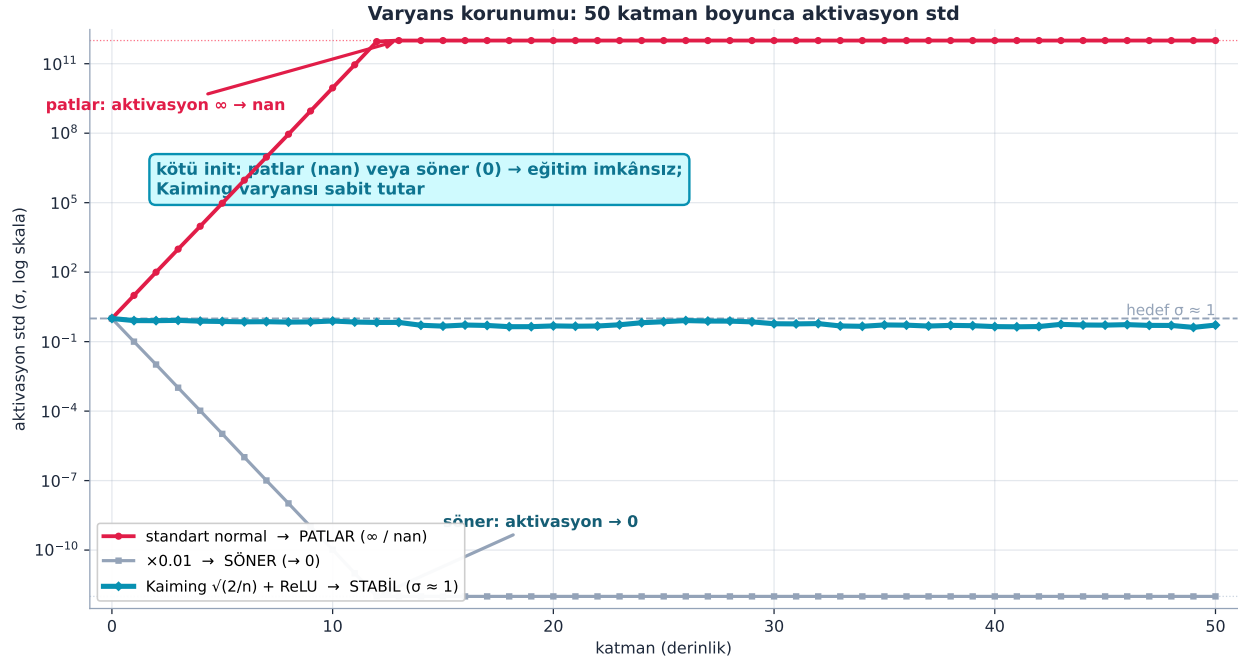
Şekil 26.3 bu deneyi **gerçek hesaplamayla** çizer (**FLAGSHIP**): 50 katman boyunca aktivasyon std’si log skalada izlenir. Standart normal ağırlık eğrisi ($\sigma \rightarrow \infty$, nan) yukarı patlar; $\times 0.01$ eğrisi ($\sigma \rightarrow 0$) tabana söner; Kaiming $\sqrt{(2/n)} + \text{ReLU}$ eğrisi ise $\sigma \approx 1$ hedefinde **stabil** kalır. Üç eğri tek bir gerçeği gösterir: kötü init eğitimi imkânsız kılar, doğru init varyansı sabit tutar.

💡 Builder Notu — Patlama/Sönme: Derin Öğrenmenin Tarihsel Duvarı

- **İleriye:** Bu, derin ağ tarihinin (2012 öncesi) eğitilememe sorununun ta kendisi; çözümü (init) derin öğrenmeyi mümkün kıldı. AlexNet öncesi derin ağların “çalışmaması” büyük ölçüde bu varyans patlaması/sönmesiydi.
- **Sezgi:** 50 katman = 50 kez çarpma; her çarpma varyansı bir çarpanla ölçekler. Çarpan 1’den büyükse 50. kuvvete üssel patlama, küçükse üssel sönme. Tam 1 olması gereken bir denge — ve bu dengeyi *tesadüfen* tutturamazsın, ağırlık ölçeğini bilinçli ayarlaman gerekir.

26.6 5. Varyans ve Standart Sapma

Sorunun matematiği **varyans**: verinin yayılımının ölçüsü. Varyans, ortalamadan farkların karesinin ortalamasıdır: $\text{Var}(X) = \text{ort}((x_i - \mu)^2)$ (burada “ort” ortalama). Standart sapma $\sigma = \sqrt{\text{Var}}$, veriyle aynı birime döner. Pratik özdeşlik: $\text{Var}(X) = \text{ort}(X^2) - \text{ort}(X)^2$ (hesabı kolaylaştırır).



Şekil 26.3: (gerçek hesaplama: 50-katman deneyi — standart normal aktivasyonu patlatır ($\rightarrow \text{inf}$), $\times 0.01$ söndürür ($\rightarrow 0$), Kaiming $\sqrt{(2/n)} + \text{ReLU}$ stabil tutar (varyans korunumu))

```
t = torch.tensor([1., 2., 4., 18])
m = t.mean()
(t-m).pow(2).mean()           # varyans
(t-m).pow(2).mean().sqrt()    # std
(t*t).mean() - (m*m)          # aynı varyans, kısa yol
```

💡 Builder Notu — Varyans: Ağırlık Ölçeklemenin Dili

- **Geriye (Stat 110):** Varyans/std olasılığın temel dağılım ölçüleri; burada ağırlık ölçeklemenin dilini verir. $\text{Var}(X) = E[X^2] - E[X]^2$ özdeşliği Stat 110'da kanıtlanan klasik sonuç.
- **Sezgi:** Karelerin ortalamasından ortalamanın karesini çıkaran kısa yol ($E[X^2] - \mu^2$), tek geçişte varyans hesaplamayı sağlar — veriyi iki kez dolaşmana gerek kalmaz. Init türetmesinde sıfır-ortalama varsayımıyla ($\mu = 0$) varyans doğrudan $E[X^2]$ 'e indirgenir, bu da formülleri sadeleştirir.

26.7 6. Kovaryans

İki değişkenin birlikte değişimini ölçen **kovaryans**: $\text{Cov}(X, Y) = \text{ort}((X - \mu_X)(Y - \mu_Y)) = \text{ort}(XY) - \mu_X \cdot \mu_Y$. İlişkisiz değişkenlerde ≈ 0 . Varyans, X'in kendisiyle kovaryansıdır: $\text{Var}(X) = \text{Cov}(X, X)$. Bu kavram, init türetmesinde “ağırlık ve girdi bağımsız” varsayımını kullanmak için gerekir.

```
cov = (t*v).mean() - t.mean()*v.mean()
cov / (t.std() * v.std()) # korelasyon = ölçeklenmiş kovaryans
```

💡 Builder Notu — Kovaryans: Bağımsızlık Varsayımının Anahtarı

- **Geriye (18.06/18.065):** Kovaryans matrisi lineer cebirin ve PCA'nın temeli; burada bağımsızlık varsayımını sağlar. Strang'ın 18.065'inde kovaryans matrisi $A^T A$ formundaydı; aynı kavram.
- **Sezgi:** Init türetmesinin can damarı şudur: ağırlık a_i ile girdi x_i **bağımsız ve sıfır ortalamalı** ise, çarpımlarının çapraz terimleri (kovaryans) ortalama ≈ 0 olur ve düşer. İşte bu yüzden $\text{Var}(y) \approx n \cdot \text{Var}(a) \cdot \text{Var}(x)$ formülü temiz çıkar — kovaryans, “neden bu terimleri atabiliyoruz?” sorusunun cevabıdır.

26.8 7. Xavier / Glorot Başlatma

Anahtar gözlem: $y = \sum a_i x_i$ (n girdi üzerinden) çıktısının varyansı, bağımsız sıfır-ortalamalı a_i, x_i için **n ile çarpılır**: $\text{Var}(y) \approx n \cdot \text{Var}(a) \cdot \text{Var}(x)$. Bir katman varyansı n katına çıkarıyorsa, 50 katmanda n^{50} — işte patlama. Çözüm: ağırlıkları $\sqrt{n}$ ile böl, yani $\text{Var}(a) = 1/n$. Bu **Xavier (Glorot) init**'tir; çıktı varyansını girdiye eşit tutar.

```
mean, sqr = 0., 0.
for i in range(100):
    x = torch.randn(100); a = torch.randn(512, 100)
    y = a @ x
    mean += y.mean().item(); sqr += y.pow(2).mean().item()
mean/100, sqr/100 # ortalama  $\approx 0$ , kare-ortalama  $\approx 100 =$  girdi sayısı

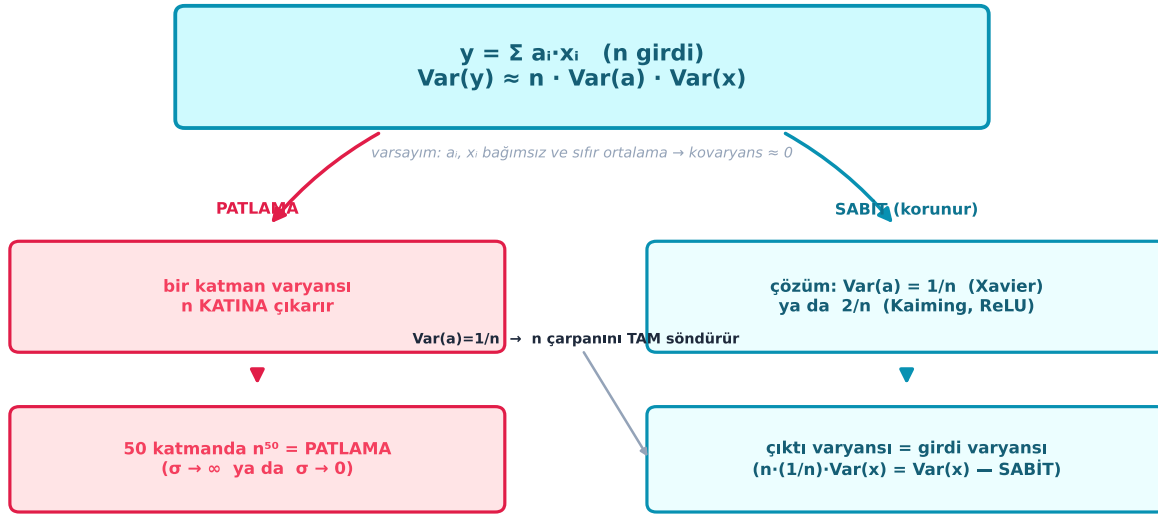
w1 = torch.randn(100, 50) / sqrt(100) # Glorot:  $\sqrt{n}$  ile böl
```

“So that’s the reason why we need this math dot square root 100.” — Howard, 32:45

Şekil 26.4 bu türetmenin mantığını şema olarak verir: merkezde $\text{Var}(y) \approx n \cdot \text{Var}(a) \cdot \text{Var}(x)$ formülü ve “ a_i, x_i bağımsız, sıfır ortalama $\rightarrow$ kovaryans ≈ 0 ” varsayımı. Sol kol (rose, sorun): her katman varyansı n katına çıkarır $\rightarrow$ 50 katmanda $n^{50} =$ patlama. Sağ kol (cyan, çözüm): $\text{Var}(a) = 1/n$ (Xavier) ya da $2/n$ (Kaiming) $\rightarrow$ çıktı varyansı = girdi varyansı, sabit kalır.

💡 Builder Notu — Glorot: Karpathy’nin Elle Bulduğu « Kar-Tanesi »

- **Geriye (Karpathy makemore 3):** Karpathy “kar-tanesi” sabitini ($\text{gain}/\sqrt{\text{fan\_in}}$) elle bulup ağırlıkları ölçekler — bu tam olarak Glorot init’in türetilmesi. Karpathy önce sezgiyle 0.2 gibi bir sabit dener, sonra $\sqrt{\text{fan\_in}}$ ’in neden doğru olduğunu görür; Howard aynı sonucu varyans cebrinden türetir.
- **Sezgi:** Formülün kalbi tek bir kelimedede: **n**. Çıktı, n tane çarpımın toplamıdır; bağımsız terimlerin toplamının varyansı, varyansların toplamıdır $\rightarrow$ n çarpanı buradan gelir. $\sqrt{n}$ ile bölmek tam olarak bu n’i geri söndürür: $n \cdot (1/n) \cdot \text{Var}(x) = \text{Var}(x)$. Girdi varyansı çıktıya değişmeden geçer.



Şekil 26.4: (şematik: $\text{Var}(y) \approx n \cdot \text{Var}(a) \cdot \text{Var}(x)$ — her katman varyansı n ile ölçekler; çözüm ağırlık varyansını $1/n$ (Xavier) ya da $2/n$ (Kaiming) yapmak)

26.9 8. ReLU Sorunu ve Kaiming (He) Başlatma

Xavier, ReLU’yu hesaba katmaz. ReLU negatifleri sıfırlar $\rightarrow$ aktivasyonların $\sim$ yarısını siler $\rightarrow$ varyansı yarıya indirir. Telafi için varyansı iki katına ölçekle: $\text{Var}(a) = 2/n$, yani $\sqrt{2/n}$ ile böl. Bu **Kaiming (He) init**’tir; ReLU’lu derin ağların standardı. Deney: 50 katman ReLU + $\sqrt{(2/100)}$ ile aktivasyonlar sağ kalır.

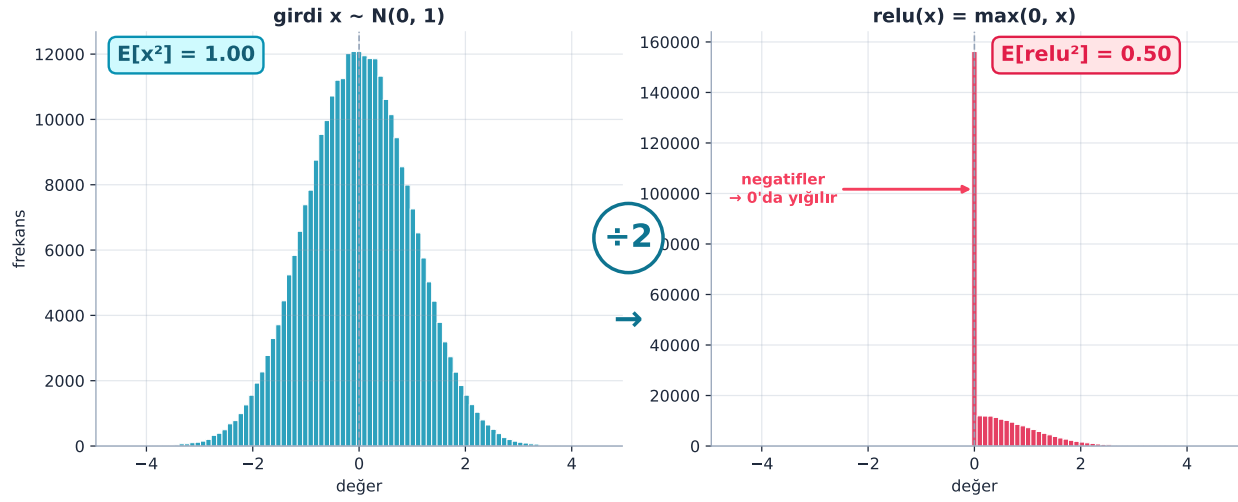
```
x = torch.randn(200, 100)
for i in range(50): x = relu(x @ (torch.randn(100,100) * sqrt(2/100)))
x[0:5,0:5] # patlamadı, sönmedi — Kaiming init işe yaradı
```

“rectified linear units, which is not something that Xavier Glorot looked at.” — Howard, 33:23

Şekil 26.5 ReLU’nun varyans üzerindeki etkisini **gerçek hesaplamayla** gösterir: solda standart normal girdi ($E[x^2] \approx 1.0$, simetrik), sağda $\text{relu}(x)$ çıktısı ($E[\text{relu}^2] \approx 0.5$ — negatifler sıfırda yığılır). ReLU karesel-ortalamayı **yarılar**; bu yüzden Kaiming $\sqrt{(2/n)}$ (Xavier’in $\sqrt{2}$ katı telafisi) varyansı korur. Şekil 26.6 ise ReLU’lu 30-katman ağda iki init’i yarıştıır: Xavier ($1/\sqrt{n}$) varyansı söndürür ($\rightarrow 0$), Kaiming ($\sqrt{(2/n)}$) sabit (~ 1) tutar.

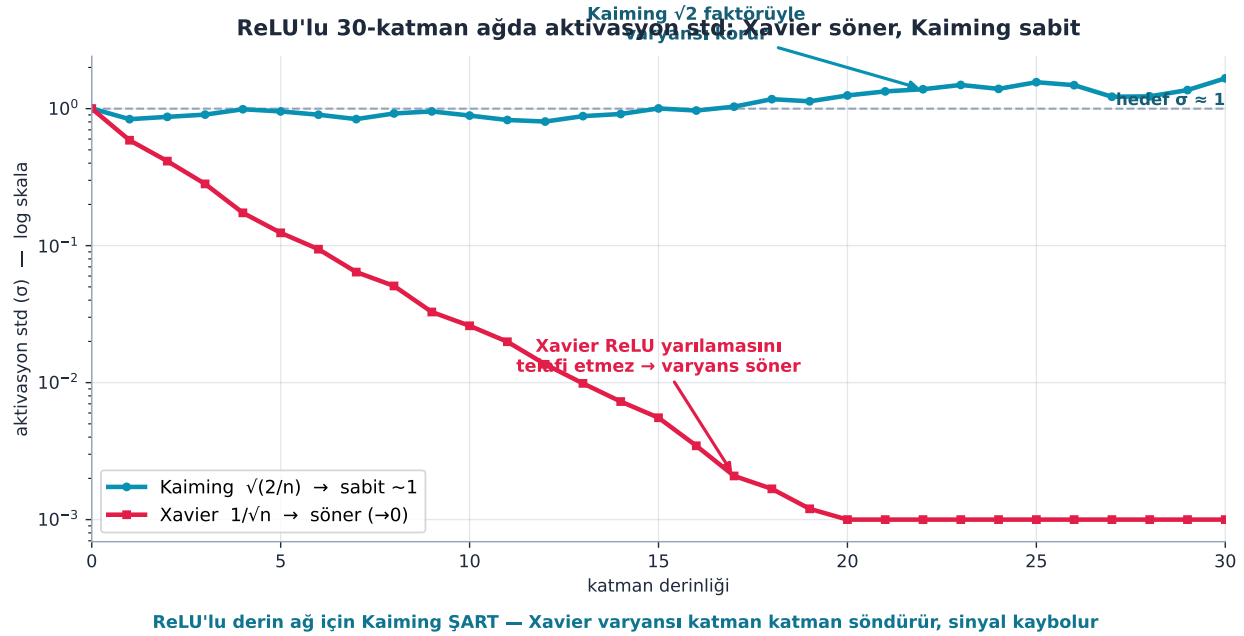
! Builder Notu — Kaiming = Karpathy makemore 3’ün TAM OMURGASI (MERKEZİ)

- **Geriye (Karpathy makemore 3) — MERKEZİ:** Karpathy makemore 3’ün tüm omurgası budur: aktivasyon istatistiklerini izle $\rightarrow$ varyansın katmandan katmana bozulduğunu gör $\rightarrow$ Kaiming gain ($\sqrt{2/n}$) ile ağırlıkları ölçekle $\rightarrow$ düzeldiğini doğrula. Karpathy tanh için gain’i $5/3$ olarak elle bulur; ReLU için bu $\sqrt{2}$ ’dir. Aynı fikir, farklı aktivasyon.
- **İki yol, tek fizik:** Howard **top-down** (önce hook’la sorunu gör, sonra matematikle türet), Karpathy **bottom-up** (önce elle deneyerek türet, sonra histogramla gör) — ikisi de aynı sonuca varır: **varyans korunumu**. Builder için bu eşitlik dersin kalbidir: `init.kaiming_normal_`’in altındaki



ReLU negatifleri sıfırlar $\rightarrow$ karesel-ortalama yarılanır ($E[x^2]: 1.0 \rightarrow 0.5$) · bu yüzden Kaiming $\sqrt{2/n}$ (Xavier'in $\sqrt{2}$ katı telafisi) varyansı korur

Şekil 26.5: (gerçek hesaplama: ReLU $E[x^2]$ 'yi $\sim 1/2$ 'ye indirir ($1.0 \rightarrow 0.5$); Kaiming init bunu $\sqrt{2/n}$ ile telafi eder)



Şekil 26.6: (gerçek hesaplama: ReLU'lu ağda Xavier ($1/\sqrt{n}$) aktivasyonu söndürür, Kaiming ($\sqrt{2/n}$) sabit tutar — Kaiming ReLU ağların standardı)

büyük = Karpathy'nin satır satır yazdığı şey.

- **Sezgi:** ReLU sinyalin yarısını (negatifleri) attığı için varyans yarıya iner; $\sqrt{2}$ ile çarpmak (yani $1/n$ yerine $2/n$) bu kaybı *önceden* telafi eder. Telafi katmanın çıkışında değil, ağırlığın *başlangıcında* yapılır — bu yüzden “başlatma” sorunudur.

26.10 9. init Fonksiyonunu Uygulamak

PyTorch'ta `model.apply(fn)` her alt-modüle bir fonksiyon uygular. Kaiming init'i tüm conv katmanlarına böyle uygularız. `init.kaiming_normal_PyTorch`'un hazır Kaiming başlatıcısıdır.

```
def init_weights(m):
    if isinstance(m, (nn.Conv1d, nn.Conv2d, nn.Conv3d)): init.kaiming_normal_(m.weight)

model.apply(init_weights)
```

💡 Builder Notu — model.apply: Ağaç Gezme Deseni

- **İleriye:** `model.apply` ağaç gezme (tree traversal) deseni; tüm katmanlara toplu işlem uygulamanın standart yolu. Aynı desenle ağırlık dondurma, ağırlık paylaşımı, custom init hep tek satırda uygulanır.
- **Sezgi:** `isinstance` kontrolü init'i yalnızca conv katmanlarına uygular — BatchNorm veya bias gibi farklı başlatma isteyen modülleri atlar. Bu seçicilik önemlidir: her katman tipi kendi doğru init'ini ister, hepsine aynı şeyi uygulamak yeni bir hata kaynağıdır.

26.11 10. Girdi Normalizasyonu

İlk katmanın girdisi de doğru ölçekte olmalı: veriyi ortalaması 0, std'si 1 olacak şekilde normalize et ($x' = (x - \mu)/\sigma$). Bunu bir callback (`BatchTransformCB`) ile her batch'e uygularız — eğitim ve doğrulamada tutarlı.

```
xmean, xstd = xb.mean(), xb.std()
class BatchTransformCB(Callback):
    def __init__(self, tfm, on_train=True, on_val=True): fc.store_attr()
    def before_batch(self, learn):
        if (self.on_train and learn.training) or (self.on_val and not learn.training):
            learn.batch = self.tfm(learn.batch)
```

💡 Builder Notu — Girdi Normalizasyonu: Zincirin İlk Halkası

- **Geriye (Ders 16):** Yine bir callback; girdi normalizasyonu çekirdek döngüden ayrı, yeniden kullanılabilir — `BatchTransformCB` Ders 16'nın `before_batch` kancasına takılır.
- **Sezgi:** Kaiming init *katmanların* varyansını korur ama bu, *ilk* girdinin zaten $\sigma \approx 1$ olduğunu

varsayar. Girdi normalizasyonu o varsayımı sağlar: zincirin başını doğru ölçeğe oturtursan, init zincirin geri kalanını korur. Eğitim/doğrulamada **aynı** μ , σ kullanmak kritiktir — yoksa model gördüğü dağılımdan farklı bir dağılımda test edilir.

26.12 11. GeneralReLU: 0-Ortalamalı Aktivasyon

ReLU çıktısı hep ≥ 0 olduğundan aktivasyon ortalaması pozitif kayar — bu da bir tür içsel kayma. Howard GeneralRelu ile çözer: küçük bir negatif eğim (leak) ekler ve çıktıyı bir sabit (sub) kadar aşağı kaydırarak ortalamayı ~ 0 'a çeker.

```
class GeneralRelu(nn.Module):
    def __init__(self, leak=None, sub=None, maxv=None):
        super().__init__()
        self.leak, self.sub, self.maxv = leak, sub, maxv
    def forward(self, x):
        x = F.leaky_relu(x, self.leak) if self.leak is not None else F.relu(x)
        if self.sub is not None: x -= self.sub
        if self.maxv is not None: x.clamp_max_(self.maxv)
        return x
```

💡 Builder Notu — GeneralReLU: Ortalamayı Sıfıra Çekmek

- **Geriye (Karpathy makemore):** Karpathy de “tanh çıktısının ortalaması sıfırdan kaymış” sorunu tartışır; aktivasyon ortalamasını 0’a çekmek ortak temadır. İki kursta da hedef aynı: sadece varyansı değil, **ortalamayı da** kontrol altında tutmak.
- **Sezgi:** ReLU’nun iki gizli problemi var: (1) negatif girdiler için gradyan tam sıfır (ölü nöron), (2) çıktı hep pozitif (ortalama kayar). *leak* birinciyi çözer (küçük negatif eğim $\rightarrow$ gradyan akar), *sub* ikinciyi (sabit kaydırma $\rightarrow$ ortalama ~ 0). *maxv* ise üstten kırparak patlamayı sınırlar. Üç küçük müdahale, üç ayrı dert.

26.13 12. LSUV: Veriye Bağlı Başlatma

Init’i formülle değil **veriyle** ayarlamak: LSUV (Layer-wise Sequential Unit Variance). Bir batch’i ağdan geçir, her katmanın çıktısının ortalama/std’sini hook’la ölç, sonra ağırlık ve bias’ı çıktı ortalaması 0 / std 1 olana dek kaydır-ölçekle. Karmaşık mimarilerde formül bilinmese bile çalışır.

💡 Builder Notu — LSUV: Formülü Bilmiyorsan Veriden Öğren

- **İleriye:** LSUV, “doğru init formülünü bilmiyorsan veriden öğren” fikri; modern mimarilerde (transformer init) hâlâ değerli. Karmaşık bloklar (attention, residual) için kapalı-form varyans formülü türetmek zordur — LSUV bunu deneysel olarak çözer.
- **Sezgi:** Kaiming bir *formül* (matematikten gelen ölçek); LSUV bir *prosedür* (veriden gelen ölçek). LSUV hook’ları (§2) kullanarak gerçek bir batch’in gerçek aktivasyon std’sini ölçer ve ağırlığı

tam o std'ye bölerek çıktıyı 1'e zorlar. Formül yanılabilceği yerde (özel aktivasyon, karmaşık blok) veri yanılmaz.

26.14 13. LayerNorm: En Basit Normalizasyon

Howard normalizasyona en sade olandan başlar: **LayerNorm**. Her örnek için, kendi aktivasyonlarının ortalama/varyansı ile normalize eder (batch'ten bağımsız). Tüm fikir tek bir küçük modüldür; öğrenilebilir mult (γ) ve add (β) ağı ölççeği geri kazanma esnekliği verir.

```
class LayerNorm(nn.Module):
    def __init__(self, dummy, eps=1e-5):
        super().__init__()
        self.eps = eps
        self.mult = nn.Parameter(tensor(1.))
        self.add = nn.Parameter(tensor(0.))
    def forward(self, x):
        m = x.mean((1,2,3), keepdim=True)
        v = x.var((1,2,3), keepdim=True)
        x = (x-m) / ((v+self.eps).sqrt())
        return x*self.mult + self.add
```

“the whole thing is this code. What is layer normalization? Well, we can create a module.” — Howard, 1:07:11

💡 Builder Notu — LayerNorm: Transformer’ların Standart Normu

- **İleriye (Ders 24, transformer):** LayerNorm, transformer’ların standart normalizasyonudur; attention dersinde tekrar karşımıza çıkar. Karpathy’nin nanoGPT’inde de her transformer bloğunda LayerNorm vardır.
- **Sezgi:** LayerNorm her örneği **kendi içinde** normalize eder — batch’teki diğer örneklerle hiç konuşmaz. Bu onu batch boyutuna duyarsız kılar (batch=1 bile çalışır), train/eval ayırımına gerek bırakmaz. Öğrenilebilir γ ve β , normalizasyonun ağı fazla kısıtlamasını önler: ağ gerekirse ölççeği/kaymayı geri kazanabilir. Tek modül, tek forward — Howard’ın vurgusu “the whole thing is this code”.

26.15 14. BatchNorm: Çalışan İstatistikler

BatchNorm her kanalı, mini-batch’in ortalama/varyansı ile normalize eder. Kritik incelik: eğitimde batch istatistiği kullanılır ve bir **çalışan ortalama** (running mean/variance) lerp_ (üssel hareketli ortalama) ile biriktirilir; çıkarımda (eval) bu biriken istatistik kullanılır. register_buffer ile saklanan bu değerler parametre değildir (gradyan almaz) ama modelle kaydedilir.

Normalizasyon: $\hat{x} = (x - \mu) / \sqrt{\sigma^2 + \varepsilon}$, ardından $y = \gamma \cdot \hat{x} + \beta$.

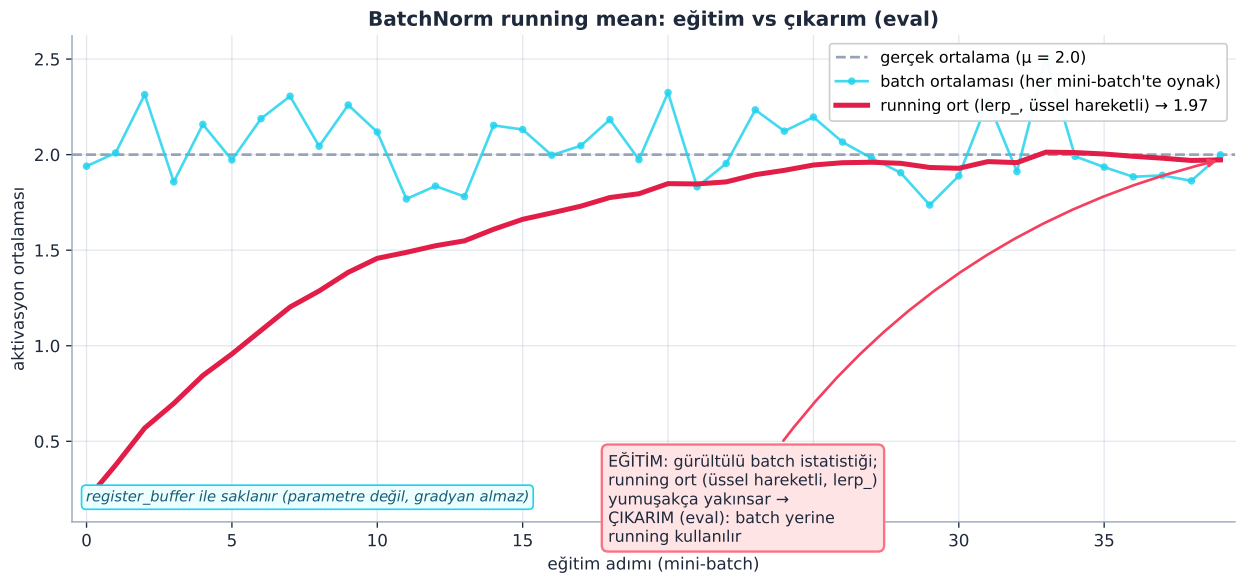

```

class BatchNorm(nn.Module):
    def __init__(self, nf, mom=0.1, eps=1e-5):
        super().__init__()
        self.mom, self.eps = mom, eps
        self.mults = nn.Parameter(torch.ones(nf, 1, 1))
        self.adds = nn.Parameter(torch.zeros(nf, 1, 1))
        self.register_buffer('vars', torch.ones(1, nf, 1, 1))
        self.register_buffer('means', torch.zeros(1, nf, 1, 1))
    def update_stats(self, x):
        m = x.mean((0, 2, 3), keepdim=True)
        v = x.var((0, 2, 3), keepdim=True)
        self.means.lerp_(m, self.mom)
        self.vars.lerp_(v, self.mom)
        return m, v
    def forward(self, x):
        if self.training:
            with torch.no_grad(): m, v = self.update_stats(x)
        else: m, v = self.means, self.vars
        x = (x - m) / (v + self.eps).sqrt()
        return x * self.mults + self.adds

```

“Batch Normalization was such an important paper.” — Howard, 1:04:26

Şekil 26.7 BatchNorm’un en kritik inceliğini **gerçek hesaplamayla** gösterir: eğitim boyunca batch ortalaması (cyan, oynak zigzag) her mini-batch’te zıplar; running mean (rose, kalın, lerp_ üssel hareketli) ise yumuşakça gerçek ortalamaya yakınsar. Çıkarımda (eval) batch istatistiği yerine bu biriken running değer kullanılır. `register_buffer` ile saklanan running mean, parametre değildir (gradyan almaz) ama modelle kaydedilir.



Şekil 26.7: (gerçek hesaplama: BatchNorm running mean — eğitimde batch ort oynak, running ort (lerp) yumuşakça gerçek ortalamaya yakınsar; eval’de running kullanılır)

! Builder Notu — BatchNorm running_mean = Karpathy makemore 3’ün En Öğretici Noktası (MERKEZİ)

- **Geriye (Karpathy makemore 3) — MERKEZİ:** Karpathy BatchNorm’u tam bu detaylarla (batch istatistiği vs running mean, train/eval ayrımı, “BatchNorm batch’leri bağlar” tuhaflığı) sıfırdan kurar. running_mean/running_var ayrımı iki kursta da en kafa karıştırıcı ama en öğretici noktadır. Karpathy’nin bnmean_running/bnstd_running değişkenleri = Howard’ın register_buffer’lı means/vars’ı.
- **Sezgi (varyans korunumu sentezi):** BatchNorm, init’in yaptığını *sürekli* yapar. Kaiming varyansı **bir kez** (başlangıçta) sabitler; eğitim ilerledikçe ağırlıklar değişip varyansı yine bozabilir. BatchNorm her forward’da aktivasyonu yeniden $\sigma \approx 1$ ’e zorlar — yani **varyans korunumunu** her adımda yeniler. İki aynı ilkenin iki uygulamasıdır: biri statik (init), öteki dinamik (norm).
- **Tuzak:** `lerp(m, mom)` üssel hareketli ortalama değildir; `no_grad` içinde, çünkü running istatistik gradyan akışına girmemeli (sadece izleyici). `model.eval()` çağırmaı unutmak → eval’de batch istatistiği kullanılır → tek örnekte saçma sonuç. Bu, production’da en sık yapılan BatchNorm hatasıdır.

26.16 15. BatchNorm vs LayerNorm — Gümüş Kurşun Değil

BatchNorm devrimdi (çok daha yüksek learning rate, hızlı eğitim) ama bedeli var: batch örneklerini birbirine bağlar, küçük batch’te bozulur, train/eval farkı karmaşıklık yaratır. LayerNorm bunlardan kaçınır (örnek-içi normalize). Howard’ın uyarısı: normalizasyon yardımcıdır ama her yerde deva değildir.

“they’re still very helpful, but they’re not a silver bullet, as it turns out.” — Howard, 1:15:30

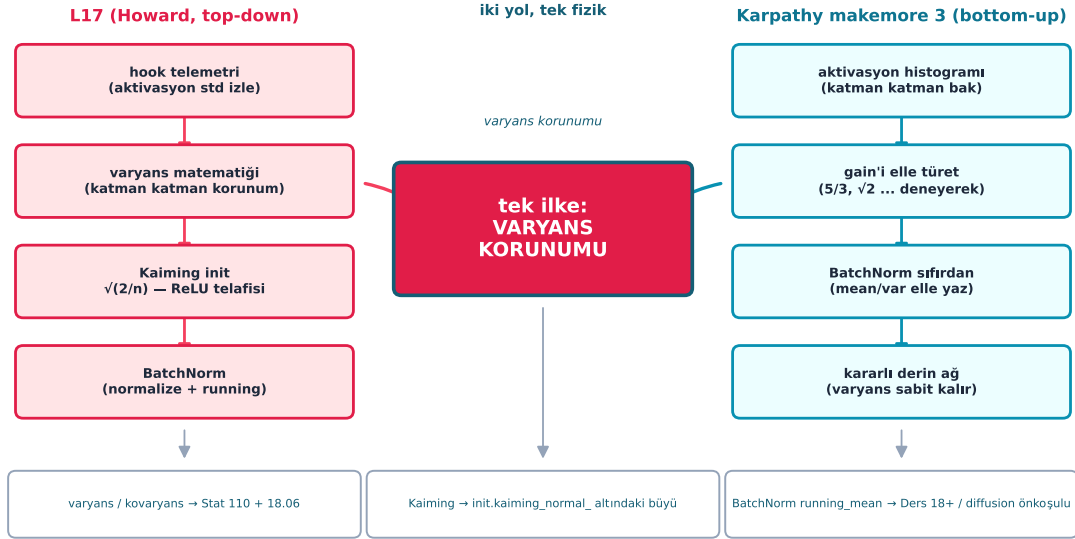
💡 Builder Notu — Norm Seçimi: Göreve Göre Karar

- **İleriye:** Modern mimariler norm seçimini (BatchNorm/LayerNorm/GroupNorm/RMSNorm) göreve göre yapar; “hangi norm” kararı production’da önemlidir. CNN’lerde genelde BatchNorm, transformer’larda LayerNorm, küçük-batch görüntüde GroupNorm.
- **Sezgi:** BatchNorm’un gücü ve zayıflığı aynı kökten: **batch’i kullanır**. Bu hızlı eğitim sağlar (batch istatistiği bedava bir regularizasyon getirir) ama örnekleri birbirine bağlar — bir örneğin çıktısı batch’teki diğerlerine bağlı hale gelir, küçük/değişken batch’te bozulur. LayerNorm bu bağı kesip her örneği yalıtır; bedeli, BatchNorm’un sağladığı batch-düzeyi regularizasyonu kaybetmektir. “Gümüş kurşun yok” — her seçimin bir bedeli var.

26.17 16. Kapanış

Ders 17, derin ağ eğitiminin en derin teknik sorununu çözdü: aktivasyonların patlaması/sönmesi. Yol: hook’larla teşhis → varyans/kovaryans matematiği → Xavier ($1/\sqrt{n}$) ve Kaiming ($\sqrt{2/n}$) init → girdi normalizasyonu + GeneralReLU → LSUV → LayerNorm ve BatchNorm sıfırdan. Hepsinin altında tek ilke var: **varyansı sabit tut**. Bu, Karpathy makemore 3’ün fast.ai karşılığı ve sonraki tüm derin mimarilerin (ResNet, diffusion) önkoşuludur.

Şekil 26.8 dersin sentezidir (**Karpathy köprüsü merkezde**): solda L17/Howard pipeline'ı (hook telemetri → varyans matematiği → Kaiming init → BatchNorm) top-down, sağda Karpathy makemore 3 pipeline'ı (aktivasyon histogramı → gain'i elle türet → BatchNorm sıfırdan → kararlı derin ağ) bottom-up. İki yol ortada tek ilkede buluşur: **varyans korunumu**. Altta üç köprü: varyans/kovaryans → Stat 110 + 18.06, Kaiming → `init.kaiming_normal_` altındaki büyü, BatchNorm `running_mean` → Ders 18+ / diffusion önkoşulu.



Şekil 26.8: (sentez: L17 = Karpathy makemore 3'ün top-down karşılığı; iki kurs 'varyans korunumu' ilkesinde birebir buluşur — kütüphane altındaki büyü = elle kurulan)

💡 Builder Notu — Kapanış: Sırada Hızlandırma ve ResNet

- **İleriye (Ders 18):** Sağlam init+norm hazır; Ders 18 hızlandırılmış optimizier'ları (momentum, Adam) ve ResNet'i (skip connection) bunun üstüne kurar. Init varyansı *bir kez* sabitler, norm *sürekli*; ResNet'in skip bağlantısı ise sinyalin derin ağda *hiç bozulmadan* akmasını sağlayan üçüncü mekanizmadır.
- **Sezgi:** Bu derste derin ağları **eğitilebilir** kıldık — patlama/sönme duvarını aştık. Sıradaki dersler eğitimi *hızlandırır* (daha iyi optimizier) ve *derinleştirir* (residual). Üç mekanizma (init + norm + skip) birlikte, modern derin öğrenmenin (ResNet'ten diffusion'a) çalışmasının temelidir.

26.18 Bu Dersin Özeti

1. **Telemetri (hooks):** Eğitim sırasında her katmanın aktivasyon ortalama/std'sini izle; sorunu görmeden çözemezsin (**telemetri**).
2. **Sorun:** Kötü init'te aktivasyonlar derin ağda ya nan'a patlar ya sıfıra söner — eğitim imkânsız (**sorun**).
3. **Varyans korunumu:** $y = \sum a_i x_i$ çıktı varyansı girdi sayısı n ile çarpılır ($\text{Var}(y) \approx n \cdot \text{Var}(a) \cdot \text{Var}(x)$); n katmanda üssel bozulma (**Xavier**).
4. **Xavier (Glorot):** Ağırlığı $\sqrt{n}$ ile böl ($\text{Var}(a) = 1/n$) → çıktı varyansı = girdi varyansı (**Xavier**).
5. **Kaiming (He):** ReLU varyansı yarılar; $\sqrt{2/n}$ ile böl → ReLU'lu derin ağların standardı (**Kaiming**).

6. **Girdi normalizasyonu + GeneralReLU:** Girdiyi 0-ortalama/1-std yap; ReLU çıktısını 0-ortalamaya kaydır ([girdi norm](#), [GeneralReLU](#)).
7. **LayerNorm:** Her örneği kendi istatistiğiyle normalize eder; tek modül, batch'ten bağımsız ([LayerNorm](#)).
8. **BatchNorm:** Batch istatistiğiyle normalize; `running_mean/running_var` (`register_buffer`, `lerp_`) çıkarımda kullanılır; `train/eval` farkı kritik. Yardımcı ama gümüş kurşun değil ([BatchNorm](#)).

! Tek Bir Cümle

Derin ağ eğitiminin sırrı aktivasyon varyansını katmandan katmana sabit tutmaktır; bunu başlatma (Xavier $1/\sqrt{n}$, ReLU için Kaiming $\sqrt{2/n}$) baştan, normalizasyon (LayerNorm/BatchNorm) eğitim boyunca sağlar — ve bu, Karpathy'nin makemore 3'te elle keşfettiği şeyin tam fast.ai karşılığıdır.

26.19 Kontrol Soruları

i Soru 1: Aktivasyonların patlaması/sönmesi nedir ve neden olur?

Cevap:

Derin bir ağda her katman çıktısının varyansı, girdi sayısı n ile orantılı değişir: $\text{Var}(y) \approx n \cdot \text{Var}(a) \cdot \text{Var}(x)$. Eğer ağırlıklar çok büyükse her katman varyansı büyütür ve birçok katman sonra aktivasyonlar **nan'a patlar**; çok küçükse her katman küçültür ve aktivasyonlar sıfıra **söner**. İkisinde de gradyanlar bozulur (patlayan/kaybolan gradyan) ve eğitim imkânsız hâle gelir. Howard bunu 50 kez matris çarpma deneyiyle gösterir: standart normal ağırlık $\rightarrow \text{nan}$, $\times 0.01 \rightarrow 0$. Çözüm, ağırlıkları varyans sabit kalacak şekilde ölçeklemek (Xavier/Kaiming init). (Şekil 26.3 üç eğriyle patlama/sönme/stabili gösterir.)

i Soru 2: Xavier ile Kaiming init arasındaki fark nedir? Neden ReLU için Kaiming?

Cevap:

Xavier (Glorot) init, ağırlıkları $\sqrt{n}$ ile böler ($\text{Var}(a) = 1/n$), böylece doğrusal bir katmanın çıktı varyansı girdi varyansına eşit kalır. Ama ReLU'yu hesaba katmaz. **ReLU** negatif değerleri sıfırlar, yani aktivasyonların yaklaşık yarısını siler ve varyansı yarıya indirir. Bunu telafi etmek için **Kaiming (He)** init varyansı iki katına ölçekler: $\text{Var}(a) = 2/n$, yani $\sqrt{2/n}$ ile böler. Sonuç: ReLU'lu derin ağlarda aktivasyon varyansı katmandan katmana korunur. Modern ReLU tabanlı ağların standardı Kaiming'dir; PyTorch'ta `init.kaiming_normal_` ile uygulanır. (Şekil 26.5 ReLU'nun $E[x^2]$ 'yi yarılادığını, Şekil 26.6 Xavier'in söndüğünü/Kaiming'in sabit kaldığını gösterir.)

i Soru 3: BatchNorm eğitim ve çıkarım (eval) modunda neden farklı davranır? `running_mean` nedir?

Cevap:

Eğitimde BatchNorm her mini-batch'in kendi ortalama/varyansını kullanarak normalize eder — ama bu istatistik batch'ten batch'e oynar. **Çıkarımda** tek bir örnek olabilir (batch istatistiği anlamsız) ve sonuç deterministik olmalı. Çözüm: eğitim sırasında `running_mean/running_var` adlı bir **çalışan (üssel hareketli) ortalama** `lerp_` ile biriktirilir (`register_buffer` ile saklanır — parametre değildir, gradyan almaz ama modelle kaydedilir). Eval modunda batch istatistiği yerine bu biriken `running` istatistik kullanılır. Bu `train/eval` ayrımı BatchNorm'un en kritik ve en sık hata yapılan noktasıdır

(`model.eval()` çağırmaı unutmak yanlış sonuç verir). (Şekil 26.7 batch ortalamasının oynak, running ortalamasının yumuşak yakınsadığını gösterir.)

i Soru 4: Bu ders Karpathy’nin makemore 3’üyle nasıl ilişkilidir? (builder bağlantısı)

Cevap:

İkisi de aynı problemi, aynı yöntemle, zıt yönlerden çözer. **Karpathy makemore 3 (bottom-up)**: aktivasyon histogramlarını elle çizer, derin ağda istatistiğin bozulduğunu görür, Kaiming gain’i ($\sqrt{2/n}$) elle türetilip ağırlıkları ölçekler, sonra BatchNorm’u sıfırdan (batch istatistiği vs `running_mean`, `train/eval`) kurar. **Howard L17 (top-down)**: önce miniai hook’larıyla `ActivationStats` telemetrisini gösterir, sorunu grafik üzerinde tespit eder, varyans/kovaryans matematiğini kurup Xavier→Kaiming’i türetilir, LayerNorm/BatchNorm’u yazar. Sonuç aynı tek ilkedir: **varyans korunumu**. Builder için bu, “kütüphanenin (`init.kaiming_normal_`, `nn.BatchNorm2d`) altındaki büyü = Karpathy’nin elle kurduğu şey” eşitliğinin en net örneğidir. (Şekil 26.8 iki pipeline’ı tek ilkede buluşturur.)

26.20 Egzersizler

Egzersiz 1 (Direkt uygulama). `torch.randn(200, 100)`’ü 50 kez rastgele matrisle çarp; standart normal, $\times 0.01$ ve $\times \sqrt{2/100}$ ağırlıkla sonucu karşılaştır (patlama/sönme/sağ kalma — Şekil 26.3’in üç eğrisini kendi sayımınla doğrula — §4).

Egzersiz 2 (İki-aşamalı). Bir modele `ActivationStats` hook callback’i ekle, kötü init (varsayılan) ve Kaiming init ile eđit; `color_dim` grafiklerini karşılaştır (§3, §8).

Egzersiz 3 (Edge case). BatchNorm’u kendin yaz; aynı modeli `model.train()` ve `model.eval()` modunda çalıştır, `running_mean` kullanımının çıktıyı nasıl deđiştirdiđini gözle (Şekil 26.7’in mantığını kendi kodunla üret — §14).

Egzersiz 4 (Kavramsal). Neden `running_mean/running_var` parametre deđil de `register_buffer` ile tanımlanır? Gradyan alsaydı ne olurdu? (§14)

Egzersiz 5 (Sonraki dersin habercisi — Ders 18). Kaiming init aktivasyon varyansını sabitler; bir sonraki ders optimizier’ı (momentum/Adam) ele alır. Momentum’un gradyanları neden “yumuşattığını” düşün (§16).

26.21 Sonraki: Ders 18 İin Hazırlık

Ders 18: Hızlandırılmış SGD ve ResNet (Accelerated SGD & ResNet)

Ders 17 sağlam init + normalizasyon kurdu. Ders 18, eğitimi *hızlandırır* iki şeyi ele alır: gelişmiş optimizier’lar (momentum, RMSProp, Adam — sıfırdan, Excel’de başlayarak) ve **ResNet** (skip connection / residual bağlantı) ile çok daha derin ağları eğitebilme.

Ana konular (Ders 18):

- SGD’den momentum’a, oradan Adam’a (sıfırdan)
- Learning rate scheduling (OneCycle)
- ResNet ve residual (skip) bağlantılar

- Daha derin ağırları sağlamca eğitmek

⚠ Ders 18 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 2 ve 3 — telemetri + BatchNorm train/eval).
- Kaiming init + BatchNorm’lu bir CNN’i Fashion-MNIST’te eğit, ActivationStats ile izle.
- Ana cümleyi tekrar oku: “Sır, varyansı katmandan katmana sabit tutmaktır.”

26.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard’da
Hook	Katmana iliştirilen, çalışınca tetiklenen fonksiyon (telemetri)	2:51
ActivationStats	Aktivasyon ort/std/histogram izleyen callback	5:32
color_dim / dead_chart	Renkli boyut histogramı / ölü ünite grafiği	5:42
Varyans	$\text{ort}((x - \mu)^2)$; yayılım ölçüsü	21:21
Kovaryans	$\text{ort}(XY) - \mu_x \cdot \mu_y$; birlikte değişim	26:01
Xavier / Glorot init	$\sqrt{n}$ ile böl ($\text{Var} = 1/n$); doğrusal katman için	29:33
Kaiming / He init	$\sqrt{2/n}$ ile böl; ReLU için (varyans korur)	35:27
init.kaiming_normal_	PyTorch hazır Kaiming başlatıcısı	36:52
GeneralReLU	leak + sub ile 0-ortalımalı ReLU	50:58
LayerNorm	Örnek-içi normalize; batch’ten bağımsız	1:06:39
BatchNorm	Batch istatistiğiyle normalize; running_mean eval’de	1:04:26
register_buffer	Parametre olmayan ama modelle kaydedilen durum	1:04:26

26.23 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: Varyans Korunumu + Karpathy makemore 3

Bu ders derin ağ eğitiminin en derin sorununu (patlama/sönme) çözer ve **varyans korunumu** ilkesiyle Karpathy makemore 3’e birebir köprü atar; köprülerin özeti:

1. **Varyans korunumu** → Karpathy makemore 3’ün omurgası; init+norm’un tek ilkesi ([Kaiming](#), [BatchNorm](#)).
2. **Kaiming init** → ReLU ağların standardı; `init.kaiming_normal_` altındaki türetme ([Kaiming](#)).
3. **Hooks/telemetri** → model içini gözlemlleme; her framework’te (PyTorch/TF) standart ([hook](#)).

4. **Varyans/kovaryans** → Stat 110 olasılık + 18.06 lineer cebir köprüsü ([varyans](#), [kovaryans](#)).
5. **BatchNorm running_mean** → train/eval ayrımı; production'da en sık hata kaynağı ([BatchNorm](#)).
6. **LayerNorm** → transformer'ların (Ders 24) standart normalizasyonu ([LayerNorm](#)).

! Bu dersten tek bir şey alıp gideceksen

Derin ağ eğitiminin sırrı **varyansı katmandan katmana sabit tutmaktır**. Aktivasyonlar patlar veya söner çünkü her katman varyansı n ile ölçekler; çözüm ağırlıkları doğru başlatmak (ReLU için Kaiming $\sqrt{2/n}$) ve aktivasyonları sürekli normalize etmektir (BatchNorm/LayerNorm). Bunu hook'larla *gördük*, matematikle *türettik*, sıfırdan *kurduk* — ve bu, Karpathy'nin makemore 3'te elle keşfettiği şeyin tam karşılığıdır. Artık `init.kaiming_normal_` ve `nn.BatchNorm2d`'nin altında ne olduğunu biliyoruz.

27 Ders 18 — Hızlandırılmış SGD ve ResNet (Accelerated SGD & ResNet)

Temeller B'nin ve 'from the foundations' omurgasının finali: eğitimi hızlandıran iki büyük fikir. Önce optimizer'lar — Howard Excel'de gradient descent'ten başlayıp SGD → momentum → RMSProp → Adam'ı sıfırdan kurar. Sonra learning rate scheduling (cosine annealing, 1-cycle super-convergence). Son olarak ResNet — skip connection ($\text{out} = F(x) + x$) ile çok daha derin ağları eğitebilme. Tek cümleyle: Adam, gradyanın ve kare-gradyanın üssel ortalamalarını birleştiren adaptif optimizer'dır; ResNet ise katmanın çıktısına girdisini ekleyerek ($\text{out} = F(x) + x$) derin ağları eğitilebilir kılan mimari fikirdir — ve bu skip connection, Karpathy'nin nanoGPT'sindeki her transformer bloğunun ($x = x + \text{attn}(x)$) residual belkemiğidir.

Bölüm bilgisi

- **Ders sayfası (video):** course.fast.ai — Lesson 18: Accelerated SGD & ResNet (~126 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 18
- **Playlist:** Part 2 — Foundations to Stable Diffusion (2022)
- **Notebook:** [course22p2](#) — [nbs/12_accel_sgd](#) + [nbs/13_resnet](#)
- **Okuma süresi:** ~40 dk
-  **Temeller B'nin finali (ETAP 5):** Ders 17'de sağlam **init** + **normalizasyon** kurduk (varyans korunumu). Ders 18 onun üstüne eğitimi *hızlandıran* iki büyük fikri koyar: gelişmiş **optimizer**'lar (momentum → Adam, sıfırdan) ve **ResNet** (skip connection). Bununla “from the foundations” omurgası tamamlanır: matmul → backprop → Learner/callback → init/norm → optimizer → ResNet.

27.1 Bu Derste Ne Var?

Temeller B'nin finali ve ETAP'ın kapanışı: eğitimi *hızlandıran* iki büyük fikir. Önce **optimizer'lar** — Howard Excel'de gradient descent'ten başlayıp SGD → momentum → RMSProp → **Adam**'ı sıfırdan kurar. Sonra **learning rate scheduling** (cosine annealing, 1-cycle). Son olarak **ResNet** — skip connection ile çok daha derin ağları eğitebilme. Part 2'nin “from the foundations” omurgası burada tamamlanır.

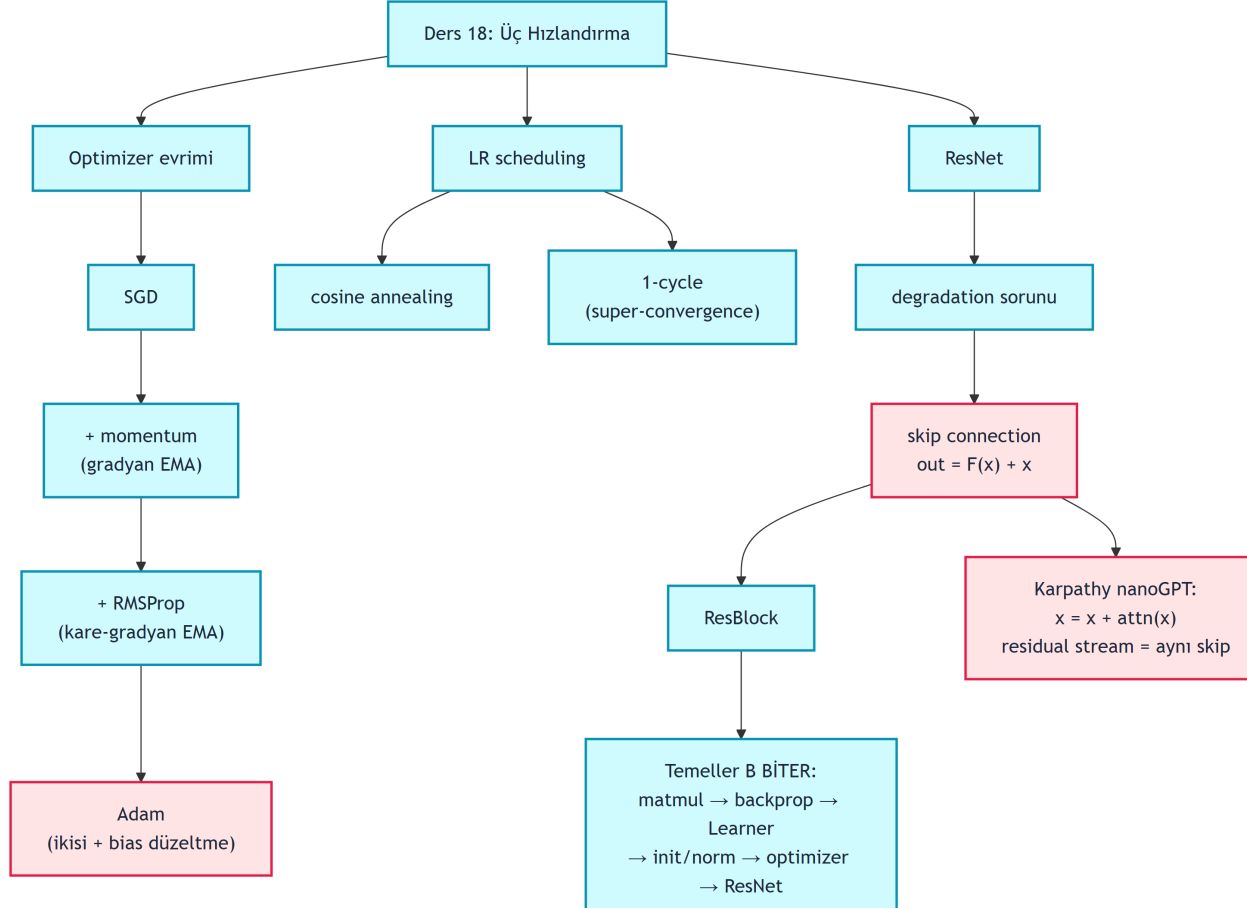
Üç temel fikir bu dersin omurgasını kurar:

1. **Optimizer evrimi** — SGD'ye momentum (gradyan ortalaması) eklenir, ona kare-gradyan ölçeklemesi (RMSProp) eklenir; ikisi birleşince **Adam** çıkar (SGD → momentum → RMSProp → Adam).
2. **LR scheduling** — learning rate'i eğitim boyunca değiştirmek (warmup + azaltma); 1-cycle ile hızlı, sağlam eğitim (cosine → 1-cycle).

3. **ResNet (skip connection)** — katmanın çıktısına girdisini ekle ($\text{out} = F(x) + x$); derin ağırları eğitilebilir kılar (**degradation** → **skip** → **ResBlock**).

“We’re going to start today in Microsoft Excel.” — Howard, 0:00

Şekil 34.1 bu üç hızlandırma eksenini tek bir yol haritasında birleştirir: optimizasyon evrimi (SGD→momentum→RMSProp→Adam), öğrenme oranı çizelgeleme (cosine + 1-cycle) ve ResNet skip connection. **Adam**, **skip connection** ve **Karpathy nanoGPT köprüsü** rose ile işaretlidir: skip connection $\text{out} = F(x) + x$, Karpathy’nin residual stream’iyle ($x = x + \text{attn}(x)$) birebir aynıdır.



Şekil 27.1: Ders 18’in üç hızlandırma eksenini: optimizasyon evrimi (SGD→momentum→RMSProp→Adam), öğrenme oranı çizelgeleme (cosine + 1-cycle) ve ResNet skip connection. Skip connection $\text{out}=F(x)+x$, Karpathy nanoGPT’nin residual stream’iyle ($x=x+\text{attn}(x)$) birebir aynıdır; bu ders Temeller B yolculuğunu kapatır. (şematik)

💡 Builder Notu — Üç Hızlandırma: Optimizer, Scheduling, Mimari

- **Geriye (Ders 17):** Sağlam init + normalizasyon (Ders 17) hazır; bu ders onun üstüne optimizasyon ve mimari hızlandırması koyar. Ders 17 derin ağırları *eğitilebilir* kıldı; Ders 18 onları *hızlı* ve *derin* yapar.
- **İleriye (Karpathy nanoGPT) — MERKEZİ:** ResNet’in + x skip connection’ı, Karpathy’nin nanoGPT’inde her transformer bloğunun belkemiğidir (residual stream). Ders 24’te (attention)

tekrar karşımıza çıkar.

- **Tek cümle:** Adam, gradyan ve kare-gradyan ortalamalarını birleştiren adaptif optimizier'dır; ResNet ise skip connection ($out = F(x) + x$) ile derin ağları eğitilebilir kılan mimari fikirdir.

27.2 1. Excel'de Gradient Descent

Howard optimizier'lara en somut yerden başlar: bir **Excel** tablosunda lineer regresyonu gradient descent ile çözmek. Her satır bir adım; her hücre bir gradyan hesabı. Soyut formül yerine, ağırlıkların adım adım nasıl güncellendiğini gözle görürsün — sonra aynı mantığı koda taşırız. Excel, optimizier evrimini (SGD → momentum → RMSProp → Adam) yan yana hücrelerde görselleştirmek için ideal bir laboratuvarıdır: her varyasyonun ham gradyanı nasıl başka türlü işlediğini satır satır izlersin.

💡 Builder Notu — Excel: Optimizier'ı En Somut Hâliyle Görmek

- **Geriye (Ders 3/5):** Gradient descent'i Ders 3-5'te kurmuştuk; burada optimizier varyasyonlarını görselleştirmek için Excel kullanılıyor — kod yerine hücreler, her güncellenmenin etkisini çıplak gösterir.
- **Sezgi:** Bir algoritmayı anlamamanın en hızlı yolu, onu kodlamadan önce gözle izlenebilir bir tabloya dökmektir. Excel'de momentum hücresi “önceki ortalamanın bir kısmı + yeni gradyanın bir kısmı” olarak görünür; soyut EMA formülü birden somutlaşır.

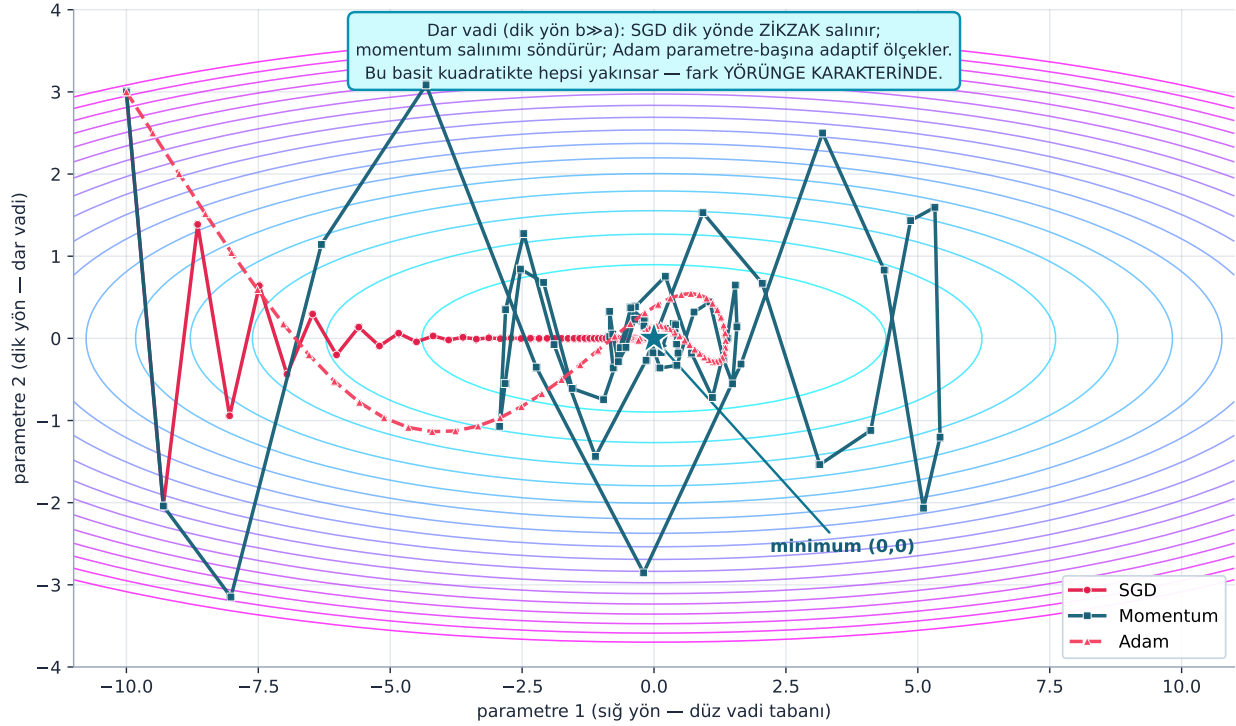
27.3 2. Basit SGD (Sıfırdan)

En temel optimizier: her parametreyi gradyanı yönünde küçük bir adım (lr) kadar güncelle. Howard'ın SGD sınıfı bunu iki parçaya ayırır: `opt_step` (gradyan adımı) ve `reg_step` (weight decay). Bu ayırım, alt sınıfların yalnızca `opt_step`'i değiştirmesine izin verir — momentum, RMSProp ve Adam hep bu sınıftan türeyecek.

```
class SGD:
    def __init__(self, params, lr, wd=0.):
        params = list(params); fc.store_attr(); self.i = 0
    def step(self):
        with torch.no_grad():
            for p in self.params:
                self.reg_step(p); self.opt_step(p)
        self.i += 1
    def opt_step(self, p): p -= p.grad * self.lr
    def reg_step(self, p):
        if self.wd != 0: p *= 1 - self.lr*self.wd
    def zero_grad(self):
        for p in self.params: p.grad.data.zero_()
```

Şekil 27.2 üç optimizier'ın (SGD, Momentum, Adam) kötü-köşüllü bir vadideki yörüngelerini **gerçek hesaplamayla** çizer (**FLAGSHIP**): dar vadide (dik yön sık yön) SGD dik yönde **zıkkak** salınır, momentum

bu salınımı söndürür, Adam ise parametre-başına adaptif ölçekler. Bu basit kuadratikte hepsi yakınsar — fark *yörünge karakterindedir*, bir “kazanan” iddia edilmez. Bu figür §2-5 boyunca kuracağımız optimizerv evriminin görsel özetidir.



Şekil 27.2: (gerçek hesaplama: SGD/Momentum/Adam yörüngeleri kötü-kışullu vadide — SGD zikzak salınır, momentum söndürür, Adam adaptif; hepsi yakınsar, fark yörünge karakterinde)

💡 Builder Notu — SGD: opt_step / reg_step Ayrımı

- **Geriye (Ders 16):** Ders 16’da `opt.step()/opt.zero_grad()` çağırmiştık; işte o optimizerv’ın sıfırdan hâli. Artık kütüphanenin `optim.SGD`’sinin içinde ne olduğunu satır satır biliyoruz.
- **Sezgi:** `opt_step` (gradyan adımı) ile `reg_step` (weight decay) ayrımı bir tasarım kararıdır: alt sınıflar yalnızca *nasıl adım atılacağını* (`opt_step`) değiştirir, weight decay mantığını dokunmadan miras alır. Momentum/RMSProp/Adam bu yüzden üç-dört satırda yazılabilir.

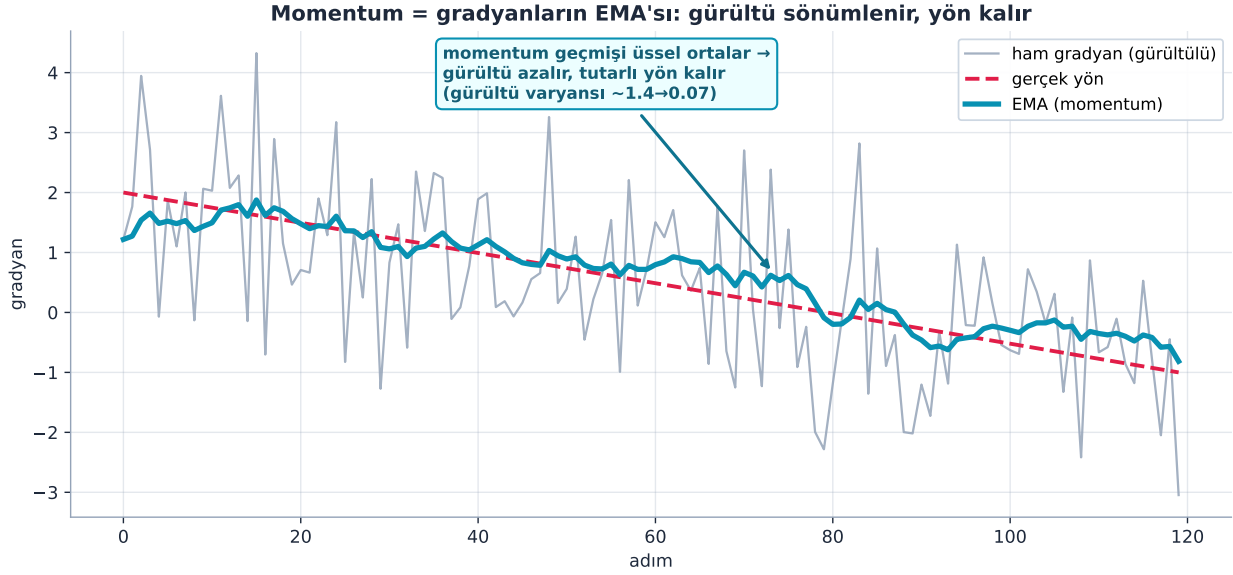
27.4 3. Momentum

Saf SGD gürültülü gradyanlarda zikzak çizer. **Momentum** çözüm: ham gradyan yerine gradyanların **üssel hareketli ortalamasını** (EMA) kullan: `grad_avg ← mom·grad_avg + (1-mom)·grad`. Tutarlı yönde hızlanır, gürültüyü yumuşatır. Momentum, SGD’den türer ve yalnızca `opt_step`’i geçersiz kılar.

```
class Momentum(SGD):
    def __init__(self, params, lr, wd=0., mom=0.9):
        super().__init__(params, lr=lr, wd=wd); self.mom=mom
```

```
def opt_step(self, p):
    if not hasattr(p, 'grad_avg'): p.grad_avg = torch.zeros_like(p.grad)
    p.grad_avg = p.grad_avg*self.mom + p.grad*(1-self.mom)
    p -= self.lr * p.grad_avg
```

Şekil 27.3 momentum'un özünü **gerçek hesaplamayla** gösterir: ham gradyan (soluk, gürültülü zigzag) etrafında EMA (cyan, kalın) yumuşakça gerçek yönü izler. Geçmiş üssel ortalayarak gürültü varyansı ~ 1.4 'ten ~ 0.07 'ye iner; tutarlı yön korunur. Momentum, “geçmiş üssel unutarak ortalama al” desenedir.



Şekil 27.3: (gerçek hesaplama: momentum = gradyanların EMA'sı; gürültülü gradyanı yumuşatır (gürültü varyansı $\sim 1.4 \rightarrow 0.07$), tutarlı yönü korur)

💡 Builder Notu — Momentum: EMA Deseni Her Yerde

- **İleriye:** EMA deseni her yerde — momentum, BatchNorm running_mean (Ders 17), Adam. “Geçmiş üssel unutarak ortalama al” temel araçtır; aynı lerp_/EMA fikrini Ders 17’de BatchNorm’un running_mean’inde de gördük.
- **Sezgi:** grad_avg, son birkaç adımın gradyanlarının ağırlıklı ortalamasıdır — eski adımlar üssel olarak unutulur (mom=0.9 ise her adımda eski ortalamasının %90’ı kalır). Gürültü rastgele yönde olduğu için ortalamada birbirini götürür; gerçek (tutarlı) yön ise üst üste binerek güçlenir. Sonuç: daha düz, daha hızlı iniş.

27.5 4. RMSProp

Farklı parametrelerin farklı ölçekte gradyanları olabilir. **RMSProp** her parametreye **adaptif** bir adım verir: kare-gradyanların EMA’sını (sqr_avg) tut, gradyanı $\sqrt{\text{sqr\_avg}}$ ile böl. Büyük gradyanlı parametre küçük adım, küçük gradyanlı büyük adım atar. ε bölmede sıfıra bölünmeyi önler.

```
class RMSProp(SGD):
    def __init__(self, params, lr, wd=0., sqr_mom=0.99, eps=1e-5):
        super().__init__(params, lr=lr, wd=wd); self.sqr_mom, self.eps = sqr_mom, eps
    def opt_step(self, p):
        if not hasattr(p, 'sqr_avg'): p.sqr_avg = p.grad**2
        p.sqr_avg = p.sqr_avg*self.sqr_mom + p.grad**2*(1-self.sqr_mom)
        p -= self.lr * p.grad/(p.sqr_avg.sqrt() + self.eps)
```

💡 Builder Notu — RMSProp: Adam’ın Diğer Yarıısı

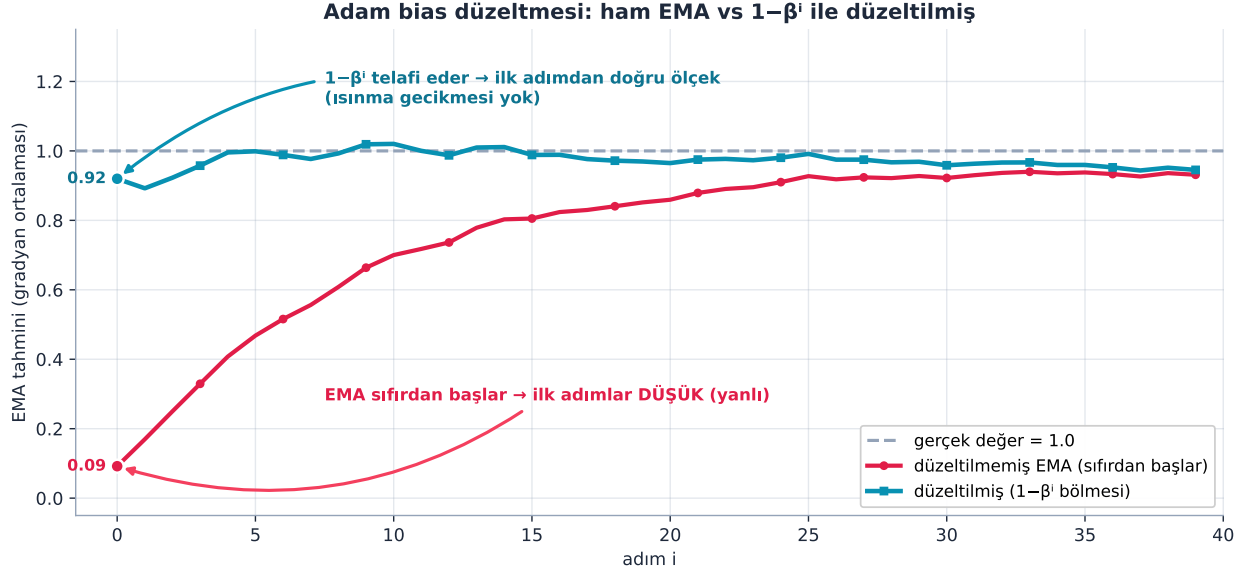
- **İleriye:** “Parametre-başına adaptif learning rate” fikri Adam’ın yarıısıdır; ϵ bölmede sıfıra bölünmeyi önler. [Adam](#) bu adaptif ölçeklemeyi momentum’la birleştirecek.
- **Sezgi:** Momentum gradyanın *yönünü* (birinci moment) ortalar; RMSProp gradyanın *büyükliğini* (ikinci moment, kare) ortalar. Bir parametre sürekli büyük gradyan alıyorsa `sqr_avg` büyür, ona bölünce adım küçülür — böylece tek bir global learning rate, her parametrenin kendi ölçeğine otomatik uyarlanır. Bu, farklı ölçekli parametreleri tek lr’yle eğitmeyi mümkün kılar.

27.6 5. Adam: Momentum + RMSProp

Adam, momentum (gradyan EMA’sı) ile RMSProp’u (kare-gradyan EMA’sı) birleştirir, üstüne **bias düzeltmesi** ekler (başlangıçta sıfırdan başlayan EMA’lar düşük tahmin verir; ilk adımlarda $1-\beta^i$ ile düzeltilir). Bugün en yaygın optimizier’dir.

```
class Adam(SGD):
    def __init__(self, params, lr, wd=0., beta1=0.9, beta2=0.99, eps=1e-5):
        super().__init__(params, lr=lr, wd=wd)
        self.beta1, self.beta2, self.eps = beta1, beta2, eps
    def opt_step(self, p):
        if not hasattr(p, 'avg'): p.avg = torch.zeros_like(p.grad.data)
        if not hasattr(p, 'sqr_avg'): p.sqr_avg = torch.zeros_like(p.grad.data)
        p.avg = self.beta1*p.avg + (1-self.beta1)*p.grad
        unbiased_avg = p.avg / (1 - (self.beta1**(self.i+1)))
        p.sqr_avg = self.beta2*p.sqr_avg + (1-self.beta2)*(p.grad**2)
        unbiased_sqr_avg = p.sqr_avg / (1 - (self.beta2**(self.i+1)))
        p -= self.lr * unbiased_avg / (unbiased_sqr_avg + self.eps).sqrt()
```

Şekil 27.4 bias düzeltmesini **gerçek hesaplamayla** gösterir: düzeltilmemiş EMA sıfırdan başlar ve ilk adımda gerçek değerin çok altında kalır (~ 0.09 , yanlış); $1 - \beta^i$ ile bölünen düzeltilmiş EMA ise ilk adımdan itibaren doğru ölçektedir (~ 0.92). Bias düzeltmesi, EMA’nın sıfır-başlangıç yanlışlığını telafi eder ve ısınma gecikmesini kaldırır.



Şekil 27.4: (gerçek hesaplama: Adam bias düzeltmesi — düzeltilmemiş EMA sıfırdan başlar (ilk adım ~0.09 düşük), $1-\beta^i$ ile düzeltilen ilk adımdan doğru (~0.92))

💡 Builder Notu — Adam: Optimizer Özü Köprüsü

- **Geriye (Karpathy nanoGPT/Zero-to-Hero):** Karpathy eğitim döngülerini AdamW ile sürer — yani optimizer’ın özü “gradyanın ortalaması (momentum) + ölçeklemesi (RMSProp)” iki kursta da aynıdır. Howard’ın sıfırdan yazdığı Adam sınıfı aynı iki momenti (birinci: avg, ikinci: sqr_avg) birleştirir; PyTorch’un AdamW’sinin içini açar.
- **Sezgi:** Adam = momentum + RMSProp + bias düzeltmesi. avg momentum’u verir (yönü ortalır), sqr_avg RMSProp’u verir (büyüklüğü ortalır), unbiased_* bölmeleri ise sıfırdan başlamanın getirdiği ilk-adım yanlılığını siler. $\beta^i \rightarrow 0$ olunca düzeltme kaybolur — yani sadece başta etki eder.

27.7 6. Learning Rate Annealing

Sabit learning rate optimal değildir: başta büyük adım (hızlı ilerleme), sonda küçük adım (ince ayar) gerekir. **Annealing** = learning rate’i eğitim boyunca kademeli azaltmak. Howard önce Excel’de, sonra PyTorch’ta gösterir. Bu, optimizer’ın *ne kadar* adım atacağını zaman içinde uyarlar — optimizer’ın *nasıl* adım attığına (momentum/Adam) ek, ortogonal bir hızlandırma kaldıracıdır.

💡 Builder Notu — Annealing: Zaman İçinde Adım Boyu

- **Geriye (Ders 3):** Ders 3’te “learning rate çok büyük/küçük” sorununu görmüştük; annealing bunu zamana yayar — başta yüksek (hızlı kaba arama), sonda düşük (hassas yerleşme).
- **Sezgi:** Optimizer’ın *yön* seçimi (momentum, adaptif ölçek) ile *adım boyu* çizelgesi (annealing) iki ayrı eksenidir; ikisi bağımsız ayarlanabilir. İyi bir reçete genellikle ikisini birlikte kullanır: Adam (yön) + cosine/1-cycle (adım boyu).

27.8 7. PyTorch Optimizer ve Scheduler Nasıl Çalışır?

Howard PyTorch'un kendi optimizer ve scheduler API'sini açar: optimizer param_groups üzerinde çalışır, scheduler her adımda optimizer'ın learning rate'ini günceller. Sıfırdan kurduğumuz mantığın PyTorch karşılığını görmek, kütüphaneyi demistifiye eder.

```
def sched_lrs(sched, steps):
    lrs = [sched.get_last_lr()]
    for i in range(steps):
        sched.optimizer.step(); sched.step()
        lrs.append(sched.get_last_lr())
    plt.plot(lrs)
```

💡 Builder Notu — PyTorch API: Yeniden Kur, Sonra Kullan

- **Geriye (Ders 10):** “Yeniden kur, sonra kullan” — artık PyTorch optimizer/scheduler'ı kara kutu değil, içini biliyoruz. `optim.Adam` ve `lr_scheduler.OneCycleLR`, §2-5'te elle yazdığımız sınıfların üretim sürümüdür.
- **Sezgi:** Scheduler, optimizer'ın param_groups'undaki lr değerini her `.step()`'te dışarıdan değiştiren ayrı bir nesnedir — optimizer'ı *sarmalamaz*, ona *müdahale eder*. Bu gevşek bağlama sayesinde herhangi bir optimizer'ı herhangi bir scheduler'la eşleştirebilirsin.

27.9 8. Scheduler Callback

Learning rate scheduling de bir **callback**'tir (Ders 16). `BaseSchedCB` scheduler'ı kurar; `BatchSchedCB` her batch'te, `EpochSchedCB` her epoch'ta adım atar. Eğitim döngüsü değişmez; scheduling ayrı bir eklenti.

```
class BaseSchedCB(Callback):
    def __init__(self, sched): self.sched = sched
    def before_fit(self, learn): self.schedo = self.sched(learn.opt)
    def _step(self, learn):
        if learn.training: self.schedo.step()

class BatchSchedCB(BaseSchedCB):
    def after_batch(self, learn): self._step(learn)
```

💡 Builder Notu — Scheduler Callback: Ders 16 Altyapısı İş Başında

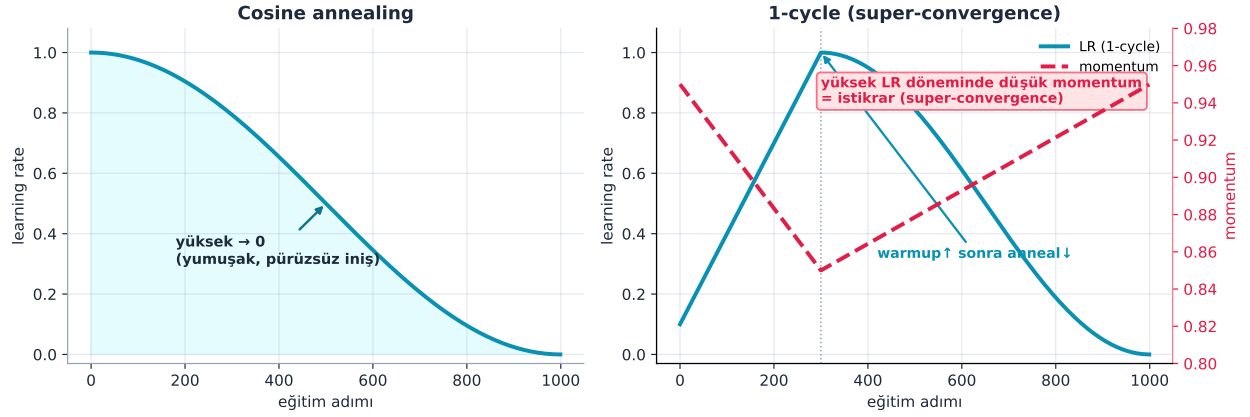
- **Geriye (Ders 16) — köprü:** Ders 16'nın callback sistemi tam burada işe yarar; scheduling çekirdek döngüye dokunmadan eklenir. `BatchSchedCB`, `after_batch` kancasına takılan üç satırlık bir eklentidir.
- **Sezgi:** Eğitim döngüsünü hiç değiştirmeden yeni bir davranış (her batch'te LR güncelle) eklemek, callback mimarisinin asıl ödülüdür. Aynı kanca noktasına `ActivationStats` (Ders 17), kayıt tutma ve scheduling birlikte takılabilir — birbirinden habersiz, birbirini bozmadan.

27.10 9. Cosine Annealing

Yaygın bir program: learning rate’i bir **kosinüs eğrisi** boyunca yüksekten sıfıra düşür. Yumuşak, kademeli azalma eğitimin sonunda ince ayara izin verir. PyTorch’ta CosineAnnealingLR.

```
tmax = epochs * len(dls.train)
sched = partial(lr_scheduler.CosineAnnealingLR, T_max=tmax)
```

Şekil 27.5 iki programı **gerçek hesaplamayla** yan yana çizer: solda **cosine annealing** (LR yüksekten 0’a yumuşak, pürüzsüz iniş), sağda **1-cycle** (LR warmup ile yükselir sonra düşer; momentum tersi yönde hareket eder). Bu figür §9-10’un görsel karşılaştırmasıdır; cosine pürüzsüz tek-yönlü iniş, 1-cycle ise warmup’lı çift-fazlı bir dögüdür.



Ders 17 sağlam init + BatchNorm, 1-cycle’in yüksek LR’sini kaldırır

Şekil 27.5: (gerçek hesaplama: cosine annealing (yüksek→0) ve 1-cycle (LR warmup+anneal, momentum tersi); 1-cycle super-convergence sağlar)

💡 Builder Notu — Cosine: Modern Standardın Programı

- **İleriye:** Cosine annealing modern eğitimin (transformer’lar dahil) standart programıdır; warmup’la birlikte kullanılır. Karpathy nanoGPT eğitiminde de warmup + cosine decay tipik reçetedir.
- **Sezgi:** Kosinüs eğrisi başta yavaş, ortada hızlı, sonda yine yavaş düşer — yani yüksek LR’de yeterince zaman geçirir (kaba arama), sonra giderek nazikleşerek (ince ayar) sıfıra yaklaşır. Doğrusal azalmaya kıyasla bu “yumuşak iniş” pratikte daha sağlam yakınsama verir.

27.11 10. 1-Cycle Learning Rate

Howard’ın favorisi: **1-cycle**. Learning rate düşük başlar, yükselir (warmup), sonra düşer; momentum ise tam tersi (yüksek → düşük → yüksek). Yüksek LR döneminde düşük momentum istikrar sağlar. Sonuç: çok hızlı ve sağlam eğitim (“super-convergence”).

```
sched = partial(lr_scheduler.OneCycleLR, max_lr=lr, total_steps=tmax)
xtra = [BatchSchedCB(sched), rec]
```

“the learning rate is starting very low and going up to high and then down again. But the momentum is starting high...” — Howard, 43:46

💡 Builder Notu — 1-cycle: Ders 17 Olmadan Çökerdi

- **Geriye (Ders 17):** 1-cycle yüksek LR’yi sağlam init + BatchNorm sayesinde kaldırabilir; Ders 17 olmadan bu hızlanma çökerdi. Yüksek LR döneminde aktivasyon varyansı kontrol altında tutulmazsa eğitim patlar — init + norm o güvenliği verir.
- **Sezgi:** LR ve momentum’un ters hareketi kasıtlıdır: yüksek LR döneminde (büyük adımlar) momentum düşürülür ki sistem aşırı ivmelenip savrulmasın; düşük LR döneminde (küçük adımlar) momentum yükseltilir ki ilerleme yavaşlasın. Bu denge, çok kısa sürede yüksek doğruluk getiren “super-convergence”in sırrıdır.

27.12 11. ResNet Sorunu: Derin Ağ Neden Kötüleşir?

“Deep Residual Learning” makalesinin gözlemi: 56 katmanlı ağ, 20 katmanlının **üst-kümesi** olmalı (fazla katmanlar identity olsa hiçbir şey bozulmaz). Yani derin ağ en az sığ kadar iyi olmalı — ama pratikte daha *kötü* eğitildi. Sorun ifade gücü değil, **eğitilebilirlik**.

“This is in a paper called ‘Deep Residual Learning for Image Recognition’ that introduced ResNets.” — Howard, 1:01:34

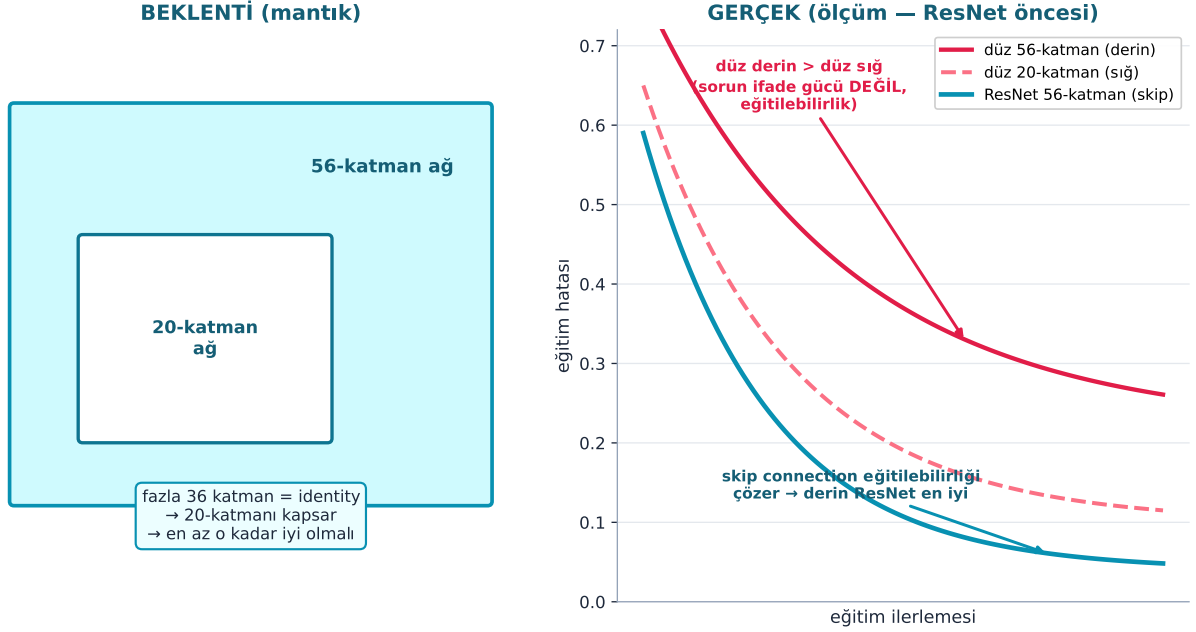
Şekil 27.6 degradation’ı iki panelde gösterir: solda **beklenti** (56-katman ağ, 20-katmanın üst-kümesidir — fazla 36 katman identity olursa kötüleşmez), sağda **gerçek** (ResNet öncesi: düz 56-katman, düz 20-katmandan *daha kötü* eğitilir). Skip connection eklenince derin ResNet en düşük hataya iner — sorunun ifade gücü değil eğitilebilirlik olduğunu kanıtlar.

💡 Builder Notu — Degradation: Init/Norm Sorununun Mimari Boyutu

- **Geriye (Ders 17):** Derin ağın eğitilememesi = init/norm sorununun mimari boyutu; ResNet bunu farklı bir açıdan çözer. Ders 17 sinyali *katman içinde* (varyans korunumu) korur; ResNet sinyali *katmanlar arasında* (skip yolu) korur.
- **Sezgi:** Paradoks şu: derin ağ teoride sığ ağı *kapsar* (fazla katmanları identity yaparak), dolayısıyla asla daha kötü olamamalı. Ama optimizasyon o identity çözümünü *bulamıyor* — gradyan derin yığında bozuluyor. Sorun modelin *yapabileceği* değil, eğitimin onu *bulabilmesi*. ResNet identity’yi varsayılan hâle getirerek bu aramayı kolaylaştırır.

27.13 12. Skip Connection: $out = F(x) + x$

ResNet’in fikri tek satır: katmanın çıktısına **girdisini geri ekle**: $out = F(x) + x$. Böylece katmanın yapması gereken, tüm dönüşüm değil, sadece **artık** (residual) — girdiden sapma. Eğer hiçbir şey öğrenmesi gerekmiyorsa $F(x) = 0$ yapması (identity) kolaydır. Bu, gradyanın derin ağda doğrudan akmasını da sağlar.



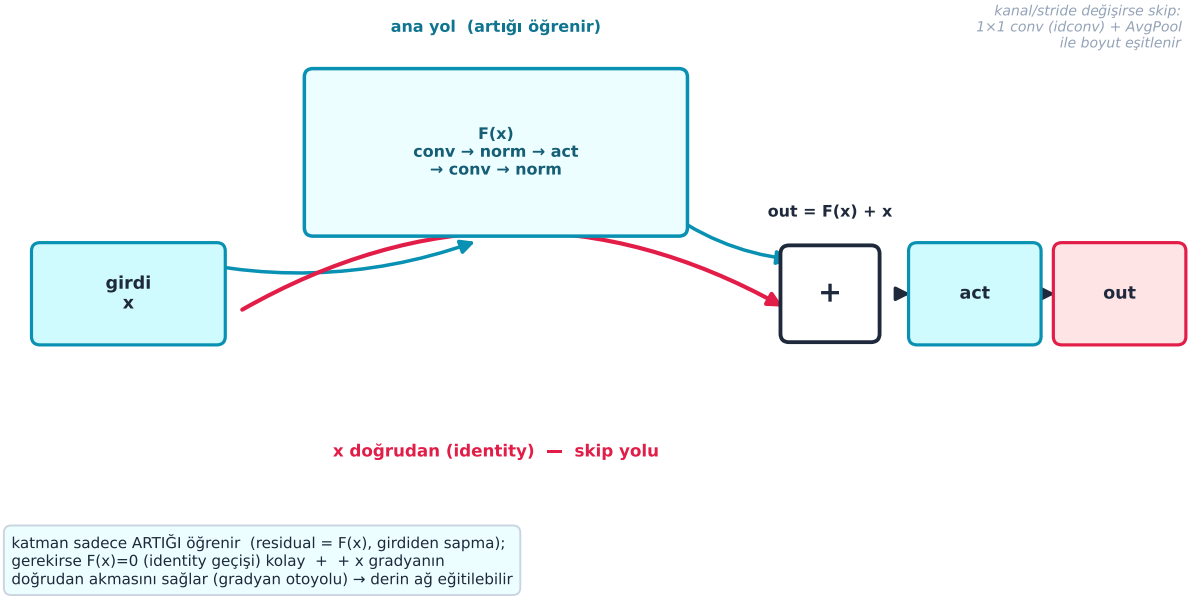
Şekil 27.6: (şematik: degradation — düz derin ağ sığdan kötü eğitilir (ifade gücü değil eğitilebilirlik sorunu); ResNet skip connection bunu çözer)

“let’s add what’s called a skip connection where instead of just being out equals... conv1... conv2.” — Howard, 1:04:36

Şekil 27.7 skip connection’ı şema olarak gösterir (**FLAGSHIP**): girdi x iki yola ayrılır — **ana yol** $F(x)$ (conv → norm → act → conv → norm, artığı öğrenir) ve **skip yolu** (x doğrudan, identity); ikisi + düğümünde toplanır ($out = F(x) + x$), sonra aktivasyondan geçer. Kanal/stride değişirse skip yolu 1×1 conv (idconv) + AvgPool ile boyut eşitlenir. + x terimi gradyanın doğrudan akmasını sağlar (gradyan otoyolu).

! Builder Notu — Skip Connection = Karpathy nanoGPT’nin Belkemiği (MERKEZİ)

- **İleriye (Karpathy nanoGPT) — MERKEZİ:** Bu $+ x$, Karpathy’nin nanoGPT’inde her transformer bloğunun belkemiğidir: $x = x + \text{attention}(x)$, $x = x + \text{mlp}(x)$. “Residual stream” derin transformer’ları eğitilebilir kılan tam bu fikirdir. ResNet (2015) → Transformer (2017) → GPT doğrudan miras zinciridir.
- **İki satır, tek fizik:** Ders 18’de sıfırdan yazdığımız `ResBlock.forward`’daki `self.convs(x) + ...` ifadesi ile Karpathy’nin GPT bloğundaki `x + self.attn(x)` aynı matematiksel fikirdir: bilgi blok boyunca değiştirilmeden akar (residual stream), her blok yalnızca bir *artık* katkı ekler. 96+ katmanlı GPT’lerin eğitilebilmesinin nedeni budur.
- **Sezgi:** $+ x$ teriminin türevi 1’dir; yani geri yayılımda gradyan, F ne yaparsa yapsın, skip yolu üzerinden derin ağda *hiç sönmeden* doğrudan geriye akar. Bu “gradyan otoyolu”, katlanan çarpımların gradyanı yok etmesini engeller — Ders 17’nin varyans korunumuna mimari bir ek.



Şekil 27.7: (şematik: ResNet skip connection $out=F(x)+x$ — katman sadece artığı öğrenir, gradyan doğrudan akar; derin ağı eğitilebilir kılar)

27.14 13. ResBlock Sıfırdan

ResBlock: iki conv'lu bir blok (convs) + skip yolu. Kanal sayısı değişirse skip yolu 1×1 conv (idconv) ile eşitlenir; stride 2 ise girdi AvgPool2d ile küçültülür. forward tek satır: iki conv'un çıktısı + (ölçeklenmiş) girdi, sonra aktivasyon.

```
def _conv_block(ni, nf, stride, act=act_gr, norm=None, ks=3):
    return nn.Sequential(conv(ni, nf, stride=1, act=act, norm=norm, ks=ks),
                          conv(nf, nf, stride=stride, act=None, norm=norm, ks=ks))

class ResBlock(nn.Module):
    def __init__(self, ni, nf, stride=1, ks=3, act=act_gr, norm=None):
        super().__init__()
        self.convs = _conv_block(ni, nf, stride, act=act, ks=ks, norm=norm)
        self.idconv = fc.noop if ni==nf else conv(ni, nf, ks=1, stride=1, act=None)
        self.pool = fc.noop if stride==1 else nn.AvgPool2d(2, ceil_mode=True)
        self.act = act()
    def forward(self, x): return self.act(self.convs(x) + self.idconv(self.pool(x)))
```

💡 Builder Notu — ResBlock: Önceki Derslerin Birleşimi

- **Geriye (Ders 15/17):** conv, act_gr (GeneralReLU), norm (BatchNorm) — hepsi önceki derslerden; ResBlock onları birleştirir. Ders 17'nin BatchNorm'u ve GeneralReLU'su burada ana yolun ($F(x)$) yapıtaşları olur.

- **Sezgi:** forward satırının zarafeti şurada: `self.convs(x)` artığı ($F(x)$) üretir, `self.idconv(self.pool(x))` skip yolunu girdiyle aynı kanal/boyuta getirir, ikisi toplanır, sonra act uygulanır. `ni==nf` ve `stride==1` durumlarında `idconv/pool` birer noop'tur — yani değişim yokken skip yolu *skip identity*'dir; tam da §12'nin istediği davranış.

27.15 14. ResNet Modeli ve Sonuç

ResBlock'ları üst üste dizerek tam ResNet kurulur (her blokta kanal artar, stride ile boyut düşer). Sonuçta aynı eğitim bütçesiyle, düz CNN'den belirgin daha yüksek doğruluk — skip connection'ın eğitilebilirlik kazancı.

```
def get_model(act=nn.ReLU, nfs=(8,16,32,64,128,256), norm=nn.BatchNorm2d):
    layers = [ResBlock(1, 8, stride=1, act=act, norm=norm)]
    layers += [ResBlock(nfs[i], nfs[i+1], act=act, norm=norm, stride=2) for i in range(len(nfs)-1)]
    layers += [nn.Flatten(), nn.Linear(nfs[-1], 10, bias=False), nn.BatchNorm1d(10)]
    return nn.Sequential(*layers).to(def_device)
```

💡 Builder Notu — `get_model`: Mimari Sonraki Tüm Derslere Taşınır

- **İleriye (Ders 19-25):** ResNet blokları, diffusion'ın U-Net'inde de temel yapıtaşdır; bu mimari sonraki tüm derslere taşınır. Ders 19+'da DDPM eğitiminde gördüğümüz U-Net, ResBlock'lardan inşa edilecek.
- **Sezgi:** `get_model` bir *desentir*: girişte tek bir stride-1 blok, sonra her adımda kanalı iki katına çıkarıp (8→16→32...) stride-2 ile uzamsal boyutu yarıya indiren bloklar, sonda flatten + lineer + BatchNorm1d. Bu “kanalı büyüt, boyutu küçült” piramidi neredeyse tüm CNN sınıflandırıcıların ortak iskeletidir.

27.16 15. ETAP Kapanışı: Temeller Tamamlandı

Ders 18, “from the foundations” yolculuğunun teknik finalidir. Adam (optimizer), 1-cycle (scheduling) ve ResNet (mimari) ile artık modern bir derin ağı sıfırdan, sağlamca eğitebilecek tüm parçalara sahibiz: matmul → backprop → Learner/callback → init/norm → optimizer → ResNet. Bundan sonrası (Ders 19+) bu temel üstünde sıfırdan diffusion.

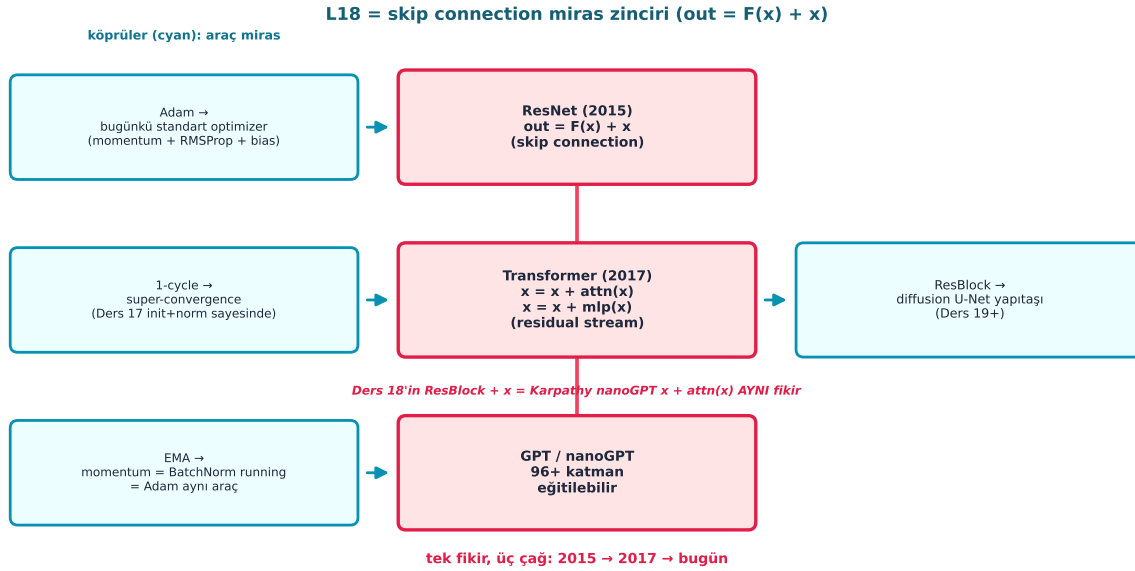
💡 Builder Notu — ETAP Kapanışı: miniai Tam Donanımlı

- **İleriye (Ders 19):** miniai artık tam donanımlı; Ders 19 bu altyapıyla DDPM (denoising diffusion) eğitmeye başlar — Part 2'nin ikinci diffusion bloğu. Optimizer + scheduling + ResNet üçlüsü, diffusion eğitiminin altyapısıdır.
- **Sezgi:** “From the foundations” sözünün anlamı tam burada netleşir: hiçbir parça (matmul, backward, Learner, init, norm, optimizer, ResBlock) kara kutu değil — her birini sıfırdan yazdık ve birleştirdik. Artık `nn`, `optim`, `lr_scheduler` çağırdığında altında ne döndüğünü biliyorsun; bu, sonraki derslerin (diffusion) güveninin temelidir.

27.17 16. Kapanış

Ders 18 eğitimi hızlandıran üç fikri kurdu: **optimizer evrimi** (SGD → momentum → RMSProp → Adam, hepsi gradyan/kare-gradyan EMA'sının katmanlanması), **LR scheduling** (cosine annealing, 1-cycle super-convergence) ve **ResNet** (skip connection $\text{out} = F(x) + x$ ile derin ağ eğitilebilirliği). Bu, Temeller B'nin ve “from the foundations” omurgasının finalidir.

Şekil 27.8 dersin sentezidir (**Karpathy köprüsü merkezde**): ortada rose omurga, skip connection'ın miras zincirini gösterir — **ResNet (2015)** $\text{out} = F(x) + x \rightarrow$ **Transformer (2017)** $x = x + \text{attn}(x)$, $x = x + \text{mlp}(x)$ (residual stream) → **GPT / nanoGPT** (96+ katman, eğitilebilir). Solda cyan köprüler (Adam → standart optimizer, 1-cycle → super-convergence, EMA → momentum/BatchNorm/Adam aynı araç), sağda ResBlock → diffusion U-Net. Tek fikir, üç çağ: 2015 → 2017 → bugün.



Şekil 27.8: (sentez: skip connection $\text{out}=F(x)+x$ — ResNet(2015)→Transformer(2017)→GPT miras zinciri; Karpathy nanoGPT residual stream birebir aynı; ResBlock→diffusion U-Net)



Builder Notu — Kapanış: ResNet'ten Transformer'a Tek Belkemek

- **İleriye:** Adam + OneCycle bugün de standart eğitim reçetesidir; ResNet skip connection ise ResNet'ten transformer'a (Karpathy nanoGPT) tüm derin mimarinin temelidir. Ders 24 (attention) bu + x 'i transformer bloğunda tekrar kuracak.
- **Sezgi:** Bu dersin iki büyük fikri (Adam, ResNet) bağımsız görünür ama ikisi de *sinyali koru* ilkesinin farklı yüzleridir: Adam gradyanın bilgisini (yön + ölçek) EMA'larla korur; ResNet sinyali (ileri) ve gradyanı (geri) skip yoluyla korur. Ders 17'nin varyans korunumuyla birlikte, üçü “derin ağda sinyali bozmadan akıt” ilkesini tamamlar.

27.18 Bu Dersin Özeti

1. **SGD:** Her parametreyi gradyanı yönünde 1r kadar güncelle; $\text{opt\_step} + \text{reg\_step}$ (weight decay) ayrımı ([SGD](#)).
2. **Momentum:** Ham gradyan yerine gradyanların EMA'sı ($\text{grad\_avg} \leftarrow \text{mom} \cdot \text{grad\_avg} + (1 - \text{mom}) \cdot \text{grad}$); zikzakı yumuşatır ([momentum](#)).
3. **RMSProp:** Kare-gradyan EMA'sıyla parametre-başına adaptif adım; gradyanı $\sqrt{\text{sqr\_avg}}$ ile böl ([RMSProp](#)).
4. **Adam:** Momentum + RMSProp + bias düzeltmesi ($1 - \beta^i$); en yaygın optimizier ([Adam](#)).
5. **LR scheduling:** Learning rate'i zamanla değiştir (cosine annealing); callback olarak eklenir ([cosine](#)).
6. **1-cycle:** LR düşük $\rightarrow$ yüksek $\rightarrow$ düşük, momentum tersi; super-convergence (hızlı, sağlam eğitim) ([1-cycle](#)).
7. **ResNet sorunu:** Derin ağ sığ ağın üst-kümesi olmalı ama eğitilemiyordu — sorun eğitilebilirlik ([degradation](#)).
8. **Skip connection:** $\text{out} = F(x) + x$; katman sadece artığı (residual) öğrenir, gradyan doğrudan akar — ResNet'ten transformer'a tüm derin mimarinin temeli ([skip](#)).

! Tek Bir Cümle

Adam, gradyanın ve kare-gradyanın üssel ortalamalarını birleştirip parametre-başına adaptif adım atan optimizier'dır; ResNet ise katmanın çıktısına girdisini ekleyerek ($\text{out} = F(x) + x$) derin ağları eğitilebilir kılan mimari fikirdir — ve bu skip connection, Karpathy'nin nanoGPT'sindeki her transformer bloğunun residual belkemiğidir.

27.19 Kontrol Soruları

i Soru 1: Momentum, RMSProp ve Adam arasındaki ilişki nedir?

Cevap:

Üçü aynı desenin (üssel hareketli ortalama, EMA) katmanlanmasıdır. **Momentum**, ham gradyan yerine gradyanların EMA'sını kullanır ($\text{grad\_avg} \leftarrow \text{mom} \cdot \text{grad\_avg} + (1 - \text{mom}) \cdot \text{grad}$) — tutarlı yönde hızlanır, gürültüyü yumuşatır. **RMSProp**, kare-gradyanların EMA'sını (sqr_avg) tutar ve gradyanı $\sqrt{\text{sqr\_avg}}$ 'a bölerek her parametreye adaptif (kendi ölçeğine uygun) bir adım verir. **Adam**, bu ikisini birleştirir: hem gradyan EMA'sı (momentum gibi) hem kare-gradyan EMA'sı (RMSProp gibi) kullanır, üstüne bir **bias düzeltmesi** ekler. Yani Adam = momentum + RMSProp + bias correction; bugün en yaygın optimizier budur. (Şekil 27.2 üç optimizier'ın yörünge karakterini gösterir.)

i Soru 2: Adam'daki 'bias düzeltmesi' ($1 - \beta^i$) ne işe yarar?

Cevap:

Adam'ın EMA'ları (avg, sqr_avg) sıfırdan başlatılır. İlk adımlarda bu ortalamalar gerçek değer çok altında kalır (sıfıra doğru "yanlı"/biased). Örneğin ilk adımda $\text{avg} = (1 - \beta_1) \cdot \text{grad} \approx 0.1 \cdot \text{grad}$ — gerçek gradyanın onda biri. **Bias düzeltmesi** bunu telafi eder: $\text{avg}'ı (1 - \beta_1^{i+1})$ ile böler. Başlangıçta bu bölen küçüktür (örn. 0.1), bu yüzden tahmini büyütür; adım sayısı i arttıkça $\beta^i \rightarrow 0$ olur ve bölen 1'e yaklaşır

(düzeltme kaybolur). Sonuç: optimizör ilk adımlardan itibaren doğru ölçekte güncelleme yapar, ısınma gecikmesi olmaz. (Şekil 27.4 düzeltilmemiş EMA'nın ilk adımda düşük, düzeltilmişin doğru olduğunu gösterir.)

i Soru 3: ResNet'in çözdüğü sorun nedir ve skip connection nasıl çözer?

Cevap:

Sorun **degradation**: teoride 56 katmanlı bir ağ, 20 katmanlının üst-kümesidir (fazla 36 katman identity yapsa hiçbir şey bozulmaz), dolayısıyla en az o kadar iyi olmalıdır. Ama pratikte daha derin ağ daha *kötü* eğitiliyordu — sorun ifade gücü değil, **eğitilebilirlik**ti (gradyan derin ağda bozuluyor). **Skip connection** çözümü tek satır: $out = F(x) + x$. Katmanın öğrenmesi gereken artık tüm dönüşüm değil, sadece **residual** (girdiden sapma); hiçbir şey gerekmiyorsa $F(x)=0$ yapmak (yani identity) kolaydır. Ayrıca $+ x$ sayesinde gradyan, ara katmanları atlayarak doğrudan geriye akabilir (gradyan otoyolu). Böylece çok daha derin ağlar sağlamca eğitilir. (Şekil 27.6 beklenti vs gerçek panellerini, Şekil 27.7 akış şemasını gösterir.)

i Soru 4: Skip connection ile Karpathy'nin nanoGPT'si arasındaki bağ nedir? (builder bağlantısı)

Cevap:

ResNet'in $out = F(x) + x$ skip connection'ı, modern transformer'ların — ve Karpathy'nin nanoGPT'sinin — **belkemiğidir**. Her transformer bloğunda iki residual bağlantı vardır: $x = x + attention(x)$ ve $x = x + mlp(x)$. Buna “residual stream” denir: bilgi blok boyunca değiştirilmeden akar, her blok sadece bir *artık* katkı ekler. Bu, ResNet'te (2015) keşfedilen tam fikrin transformer'a (2017) taşınmış hâlidir; çok derin (96+ katman) GPT'lerin eğitilebilmesinin nedeni budur. Builder için bağ net: Ders 18'de sıfırdan yazdığımız `ResBlock.forward`'daki `self.conv5(x) + ...` ifadesiyle, Karpathy'nin GPT bloğundaki $x + self.attn(x)$ aynı matematiksel fikirdir. ResNet → Transformer → GPT doğrudan miras zinciridir. (Şekil 27.8 bu miras zincirini gösterir.)

27.20 Egzersizler

Egzersiz 1 (Direkt uygulama). SGD, Momentum ve Adam'ı kendin yaz; aynı modeli üçüyle eğit, kayıp eğrilerini karşılaştır (Şekil 27.2'nin yörünge karakterini kendi sayımınla doğrula — §2, §5).

Egzersiz 2 (İki-aşamalı). Bir `BatchSchedCB` ile `OneCycleLR` kullanarak modeli eğit; learning rate ve momentum'un eğitim boyunca nasıl değiştiğini çiz (`sched_lrs` — §8, §10).

Egzersiz 3 (Edge case). `ResBlock`'u yaz; `ni != nf` (kanal değişimi) ve `stride=2` durumlarında `idconv` ve `pool`'un boyutları nasıl eşitlediğini doğrula (§13).

Egzersiz 4 (Kavramsal). Skip connection ($out = F(x) + x$) gradyanın derin ağda neden daha iyi aktığını açıkla; $+ x$ teriminin türevi nedir? (§12)

Egzersiz 5 (Sonraki dersin habercisi — Ders 19). ResNet blokları diffusion'ın U-Net'inde kullanılır. Bir görüntüye kademeli gürültü ekleyip sonra bir ağa onu geri tahmin ettirmenin (denoising) nasıl bir eğitim hedefi olduğunu düşün (§15).

27.21 Sonraki: Ders 19 İçin Hazırlık

Ders 19: DDPM ve Dropout (DDPM and Dropout)

Ders 18 “from the foundations” temellerini tamamladı (optimizer + scheduling + ResNet). Ders 19, ikinci diffusion bloğunu başlatır: **DDPM** (Denoising Diffusion Probabilistic Models) — bir görüntüye kademeli gürültü ekleyip (forward process, Ders 9B matematiği) bir ağa gürültüyü tahmin ettirerek sıfırdan diffusion model eğitmek. Artık miniai tam donanımlı.

Ana konular (Ders 19):

- DDPM forward/reverse process (sıfırdan)
- Gürültü tahmini eğitim hedefi
- U-Net mimarisi (ResNet bloklarıyla)
- Dropout düzenlileştirme

⚠ Ders 19 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 3 — optimizer + ResBlock).
- Adam + OneCycle + ResNet’li bir modeli Fashion-MNIST’te eğit, en iyi doğruluğu hedefle.
- Ders 9B’nin (diffusion matematiği) forward process formülünü tekrar oku.

27.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard’da
SGD	Gradyan yönünde 1r kadar adım; en temel optimizer	1:43
Momentum	Gradyanların EMA’sı; zikzakı yumuşatır	11:01
RMSProp	Kare-gradyan EMA’sıyla parametre-başına adaptif adım	15:37
Adam	Momentum + RMSProp + bias düzeltmesi	16:41
$\text{grad\_avg} / \text{sqr\_avg}$	Gradyan ve kare-gradyan hareketli ortalamaları	11:01
Bias düzeltmesi	EMA’nın sıfır-başlangıç yanlılığını $(1-\beta^i)$ ile telafi	16:41
LR annealing	Learning rate’i zamanla azaltma	23:02
Cosine annealing	LR’yi kosinüs eğrisiyle düşürme	40:15
1-cycle	LR düşük→yüksek→düşük, momentum tersi (super-convergence)	42:24
ResNet / degradation	Derin ağ sığın üst-kümesi ama eğitilemiyordu	1:01:34
Skip connection	$\text{out} = F(x) + x$; sadece artığı öğren, gradyan akar	1:04:36

Kavram	Tanım	Howard'da
ResBlock / idconv	İki conv + skip; 1×1 conv kanal eşitler	1:07:00

27.23 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: Hızlandırma + Karpathy nanoGPT Residual

Bu ders eğitimi hızlandıran iki büyük fikri (Adam, ResNet) kurar ve **skip connection** ile Karpathy nanoGPT residual stream'ine birebir köprü atar; köprülerin özeti:

1. **Adam** → momentum + RMSProp birleşimi; bugünkü standart optimizer ([Adam](#)).
2. **EMA deseni** → momentum, BatchNorm running_mean (Ders 17), Adam — hepsi aynı araç ([momentum](#)).
3. **1-cycle** → super-convergence; sağlam init+norm (Ders 17) sayesinde mümkün ([1-cycle](#)).
4. **Skip connection** → Karpathy nanoGPT residual stream ($x = x + \text{attn}(x)$); ResNet→Transformer mirası ([skip](#)).
5. **ResBlock** → diffusion U-Net'inin (Ders 19+) temel yapıtaşı ([ResBlock](#)).
6. **Excel'de SGD** → optimizer'ı en somut hâliyle görselleştirme (Ders 3 gradient descent köprüsü) ([Excel](#)).

❗ Bu dersten tek bir şey alıp gideceksen

İki büyük hızlandırma fikri. **Adam**, gradyanın ve kare-gradyanın hareketli ortalamalarını birleştirerek her parametreye akıllı, adaptif bir adım verir — SGD→momentum→RMSProp→Adam, hepsi aynı EMA desenin katmanlanmasıdır. **ResNet**, katmanın çıktısına girdisini ekleyerek ($\text{out} = F(x) + x$) derin ağları eğitilebilir kılar; katman tüm dönüşümü değil sadece *artığı* öğrenir, gradyan doğrudan akar. Bu $+ x$, ResNet'ten transformer'a (Karpathy nanoGPT $x = x + \text{attn}(x)$) tüm modern derin mimarinin belkemiğidir. Temeller tamamlandı — sıradaki sıfırdan diffusion.

28 Ders 19 — DDPM ve Dropout (DDPM and Dropout)

ETAP 6'nın (Sıfırdan Diffusion A) açılışı ve Part 2'nin matematik zirvesi: DDPM'i (Denoising Diffusion Probabilistic Models) sıfırdan kurmak. Ders 9B'de Waseem ve Tanishq'in türettiği forward process matematiği, burada Howard'ın elinde çalışan koda dönüşür. Kısa bir Dropout ara konusundan sonra DDPM'in kalbi: bir görüntüye kapalı formülle gürültü ekle (forward, kolay), bir U-Net'e o gürültüyü tahmin ettir (reverse, zor), kayıp sadece MSE. Tek cümleyle: DDPM, bir görüntüye kapalı formülle ($x_t = \sqrt{\alpha_t} \cdot x_0 + \sqrt{1 - \alpha_t} \cdot \epsilon$) gürültü ekleyip bir U-Net'e o gürültüyü MSE ile tahmin ettirerek eğitilen üretken modeldir; üretim, saf gürültüden başlayıp ağı tahmin ettiği gürültüyü adım adım kaldırarak imkânsız dönüşümü binlerce kolay adıma bölmektir — ve bu, 9B'de türetilen matematiğin miniai üstünde koda dökülmüş hâlidir.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 19: DDPM and Dropout](https://course.fast.ai/lesson/19) (~109 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 19
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/15_DDPM](#)
- **Okuma süresi:** ~42 dk
- **Hocalar:** Howard + Tanishq Abraham + Jonathan Whitaker (Johnno)
- **🔗 ETAP 6'nın başı (Sıfırdan Diffusion A):** Ders 18'de “from the foundations” omurgası tamamlandı (matmul → backprop → Learner/callback → init/norm → optimizer → ResNet). miniai artık tam donanımlı. Ders 19 bu altyapıyla **DDPM**'i sıfırdan kurar — Ders 9B'nin diffusion matematiğini (forward process) Howard'ın elinde çalışan koda döker. Bu, Part 2'nin teori-pratik köprüsünün doruğudur.

28.1 Bu Derste Ne Var?

ETAP'ın ve Part 2'nin matematik zirvesi: **DDPM**'i (Denoising Diffusion Probabilistic Models) sıfırdan kurmak. Ders 9B'de Waseem ve Tanishq'in türettiği diffusion matematiği (forward process), burada Howard'ın elinde çalışan koda dönüşür. Önce kısa bir **Dropout** ara konusu, sonra DDPM'in kalbi: bir görüntüye kademeli gürültü ekle (forward), bir ağa o gürültüyü tahmin ettir (reverse), kayıp sadece **MSE**. miniai (L16-18) artık tam donanımlı — DDPM onu kullanır.

Üç temel fikir bu dersin omurgasını kurar:

1. **Forward process (kolay)** — bir görüntüye kapalı formülle gürültü ekle: $x_t = \sqrt{\alpha_t} \cdot x_0 + \sqrt{1 - \alpha_t} \cdot \epsilon$. Tek adımda istenen gürültü seviyesi, ara adımları gezmeden ([forward](#) → [noisify](#)).

2. **Reverse process (zor)** — gürültülü görüntüden gürültüyü kaldırmak; bir sinir ağı (U-Net) tarafından öğrenilir (**reverse** → **sampling**).
3. **Anahtar içgörü** — ağ sadece *eklenen gürültüyü* (ϵ) tahmin eder; kayıp basit bir **MSE**'dir. Diffusion'ın tüm karmaşıklığı buna iner (**anahtar içgörü**).

“the forward direction is driven... really easily to make it something more noisy. And the reverse direction is incredibly difficult.” — Howard, 26:00

Şekil 34.1 bu üç ekseni tek bir yol haritasında birleştirir: kolay forward süreç (kapalı form gürültüleme), zor reverse süreç (U-Net gürültü tahmini) ve her şeyi tek bir MSE kaybına indirgeyen anahtar içgörü. **forward formülü**, **MSE içgörüsü** ve **9B → noisify köprüsü** rose ile işaretlidir: bu dersin kalbi, Ders 9B'de türetilen matematiğin ($x_t = \sqrt{\alpha} \cdot x_0 + \sqrt{(1-\alpha)} \cdot \epsilon$) miniai üstünde `before_batch` koduna dökülmesidir.

💡 Builder Notu — Üç Eksen: Kolay Forward, Zor Reverse, Tek MSE

- **Geriye (Ders 9B):** 9B'de Waseem/Tanishq forward process $q(x_t|x_{t-1})$ ve reparameterization'ı türetti; bu ders onu Howard'ın `before_batch`'inde üç satır koda döküyor. Teori→kod köprüsünün en saf örneği.
- **Geriye (Ders 16-18):** miniai Learner+callback (L16), Kaiming init (L17), ResBlock/U-Net (L18) — DDPM bu altyapıyı kullanır; DDPMCB, miniai TrainCB'den türer.
- **İleriye (Ders 20-25):** DDPM temeldir; mixed precision (L20), DDIM (L21), Karras (L22), latent diffusion (L25) hep bunun üstüne kurulur.
- **Tek cümle:** DDPM, bir görüntüye kapalı formülle gürültü ekleyip bir ağa o gürültüyü MSE ile tahmin ettiren üretken modeldir; üretim, saf gürültüden başlayıp adım adım gürültüyü kaldırmaktır.

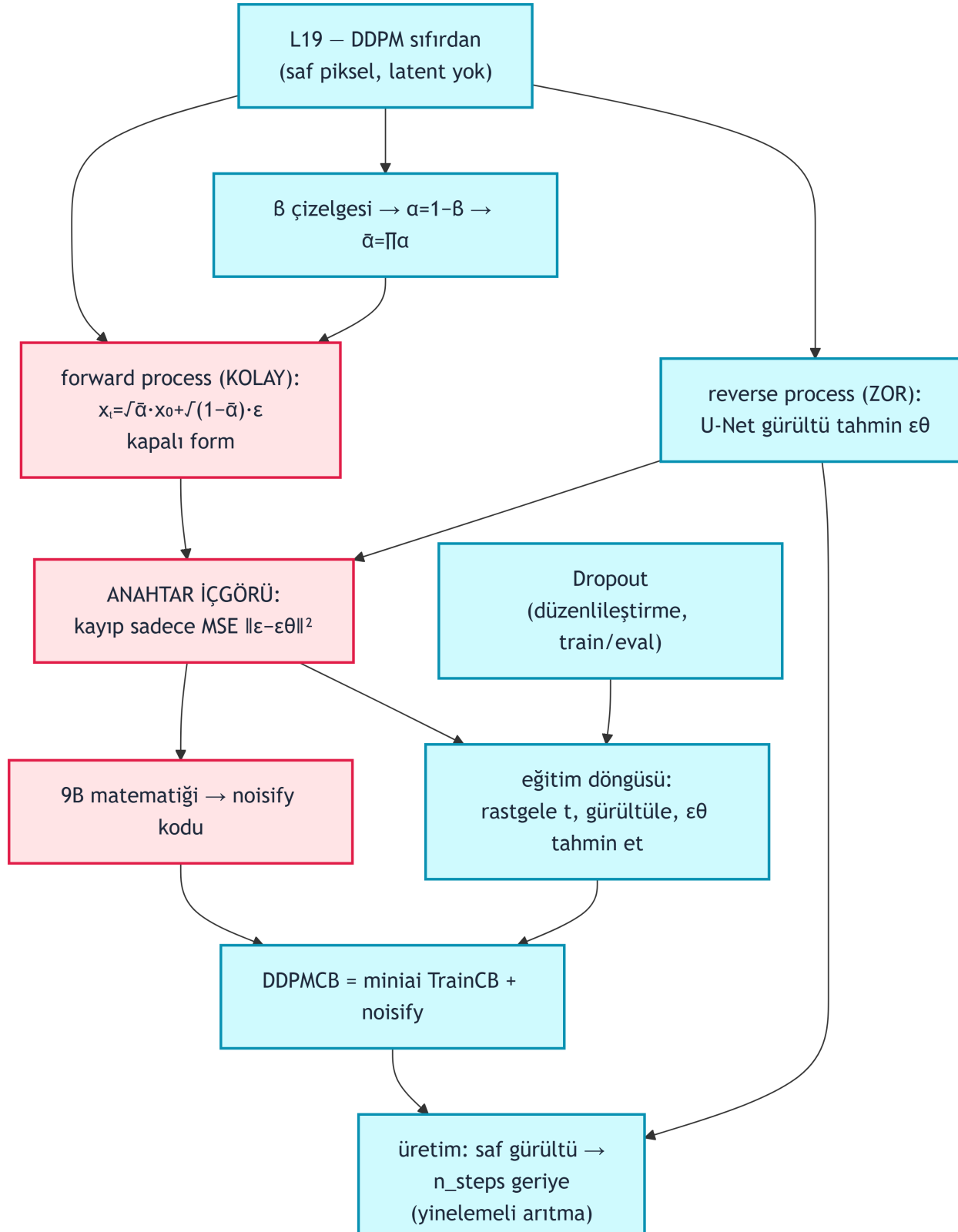
28.2 1. Dropout

Ders, kısa bir düzenleme tekniğiyle açılır: **Dropout**. Eğitim sırasında her ileri geçişte aktivasyonların rastgele bir kısmını (örn. %10) sıfırlar; ağ tek bir nörona aşırı bağımlı olamaz, daha sağlam temsiller öğrenir. Çıkarımda kapatılır. Bir `nn.Module` olarak miniai modeline eklenir.

```
class Dropout(nn.Module):
    def __init__(self, p): super().__init__(); self.p = p
    def forward(self, x):
        if not self.training: return x
        dist = distributions.binomial.Binomial(probs=1-self.p)
        return x * dist.sample(x.size()).to(x.device) * 1/(1-self.p)
```

💡 Builder Notu — Dropout: Train/Eval Ayrımı BatchNorm Gibi

- **Geriye (§4.D):** Dropout genel bir düzenleme tekniğidir; burada miniai'ye sıfırdan eklenir. Eğitim/çıkarma ayrımı (`self.training`) BatchNorm'daki (L17) ile aynı desendir: davranış eğitimde ve çıkarımda farklıdır.
- **Sezgi:** Çıkarımda hiçbir aktivasyon düşürülmediği için, eğitimde hayatta kalan aktivasyonlar $1/(1-p)$ ile ölçeklenir — böylece beklenen toplam çıktı eğitim ile çıkarma arasında tutarlı kalır. Bu “inverted dropout” hilesi, çıkarımda fazladan iş yapmayı önler.



Şekil 28.1: L19 kavram haritası: DDPM’i saf piksel uzayında sıfırdan kurmak — kolay forward süreç (kapalı form gürültüleme), zor reverse süreç (U-Net gürültü tahmini) ve her şeyi tek bir MSE kaybına indirgeyen anahtar içgörü. (şematik)

28.3 2. DDPM Nedir? (Saf Piksel Diffusion)

DDPM, en temel diffusion modelidir: **latent VAE yok**, doğrudan piksel uzayında çalışır (Stable Diffusion’ın aksine). Fikir 9B’den tanıdık: bir görüntüye kademeli gürültü ekleyen bir **forward** süreç ve onu tersine çeviren öğrenilmiş bir **reverse** süreç. Bu ders ikisini de sıfırdan kurar — “model hariç her şey”.

“DDPM doesn’t have the latent VAE thing and we’re not going to do [that here].” — Howard, 12:21

💡 Builder Notu — DDPM: En Sade Hâlimden Başla

- **Geriye (Ders 9):** Ders 9’da Stable Diffusion’ı (latent + U-Net + CLIP) *kullandık*; burada en sade hâlini (saf piksel, koşulsuz) sıfırdan kuruyoruz — “yeniden kur, sonra anla” kuralının diffusion’a uygulanışı.
- **Sezgi:** Latent uzayı (VAE), CLIP koşullamasını, guidance’ı bir kenara bırakıp çıplak çekirdeği — forward + reverse + MSE — izole etmek, diffusion’ın gerçekte ne olduğunu görmenin en temiz yoludur. Tüm o ek katmanlar (Ders 25’te dönecek) bu çekirdeği *hızlandırır* veya *koşullar*, ama özünü değiştirmez.

28.4 3. Forward Process: Gürültü Eklemek (Kolay Yön)

Forward süreç bir görüntüye adım adım gürültü ekler. Kritik kolaylık: t adımıdaki gürültülü görüntüyü, tüm ara adımları gezmeden **kapalı formülle** doğrudan üretebilirsin:

$$x_t = \sqrt{\alpha_t} \cdot x_0 + \sqrt{1 - \alpha_t} \cdot \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, I)$$

Burada x_0 orijinal görüntü, ε standart normal gürültü, t ise t ’ye kadarki gürültü çizelgesinin birikimli çarpımı. t büyükken görüntü temiz, küçükken neredeyse saf gürültü.

“the forward process, which is mostly just used for training.” — Howard, 23:58

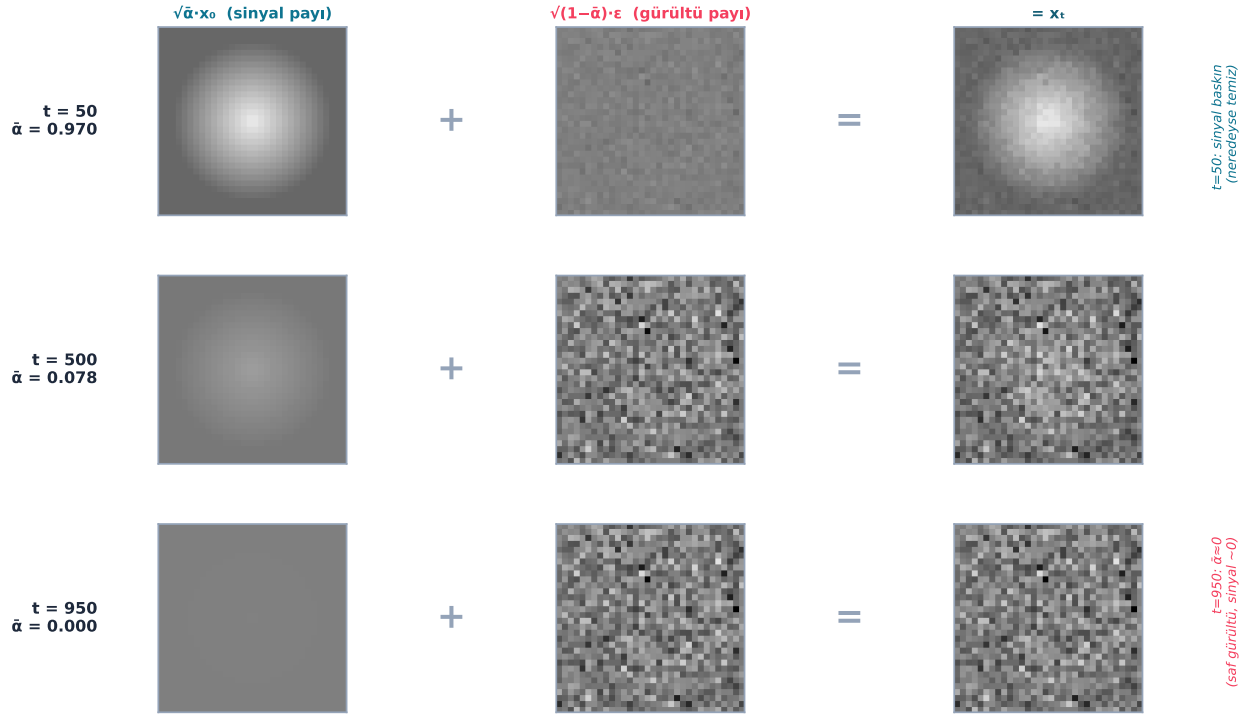
Şekil 28.2 forward gürültülemeyi **gerçek hesaplamayla** üç gürültü seviyesinde ($t=50/500/950$) ayrıştırır (**FLAGSHIP**): her satır $x_t = \sqrt{\alpha_t} \cdot x_0 + \sqrt{1 - \alpha_t} \cdot \varepsilon$ formülünü sinyal payı ($\sqrt{\alpha_t} \cdot x_0$) + gürültü payı ($\sqrt{1 - \alpha_t} \cdot \varepsilon$) = x_t olarak gösterir. t arttıkça $\sqrt{\alpha_t}$ azalır (sinyal solar), $\sqrt{1 - \alpha_t}$ artar (gürültü baskınlaşır); $t=950$ ’de $\alpha_t \approx 0$, görüntü saf gürültüye iner. Tüm bunlar **tek formülle, tek adımda** — ara adımları gezmeden.

💡 Builder Notu — Forward: 9B’nin Kapalı Formu Burada

- **Geriye (Ders 9B):** Bu tam olarak 9B’deki forward process kapalı formülüdür (reparameterization ile türetilmişti); orada matematik, burada görsel ve birazdan (§7) kod. Tek satırlık formül, tüm forward sürecin özüdür.
- **Sezgi:** Forward sürecin “kolay” olmasının sebebi, t ’ye kadarki gürültü birikiminin tek bir kapalı çarpıma ($\cdot$) sıkıştırılabilmesidir. Adım adım $q(x_t|x_{t-1})$ uygulamak yerine, doğrudan $q(x_t|x_0)$ ’a atlarsın — eğitimde rastgele t seçip o gürültü seviyesini tek hamlede üretmek bu yüzden mümkündür.

Forward noising — sinyal/gürültü ayrışımı

$x_t = \sqrt{\alpha} \cdot x_0 + \sqrt{1-\alpha} \cdot \epsilon$: $t \uparrow \rightarrow$ sinyal $\sqrt{\alpha}$ azalır, gürültü $\sqrt{1-\alpha}$ artar; kapalı formülle TEK adımda



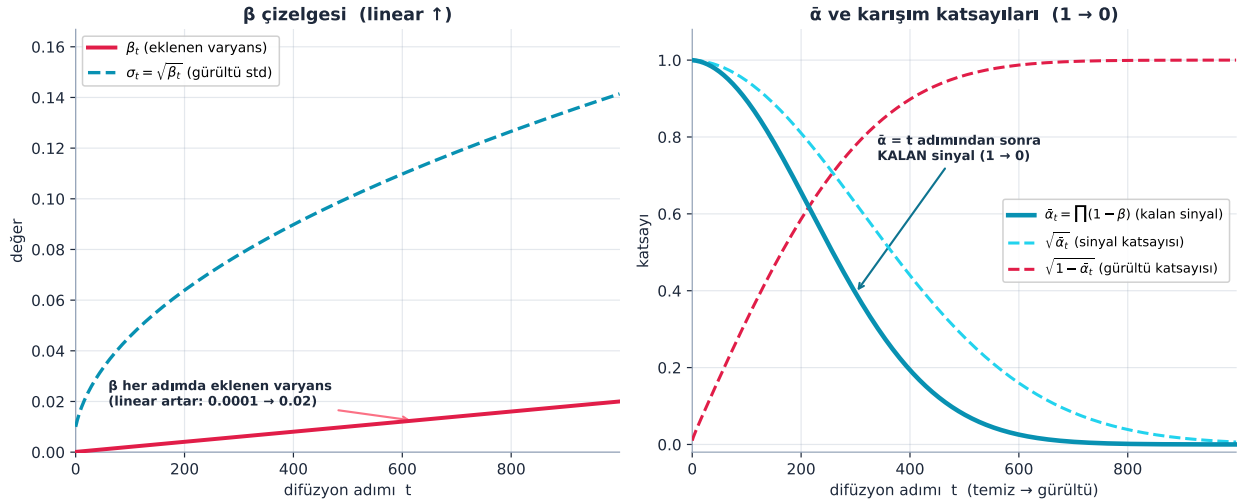
Şekil 28.2: (gerçek hesaplama: DDPM forward noisify $x_t = \sqrt{\alpha} \cdot x_0 + \sqrt{1-\alpha} \cdot \epsilon$; $t=50/500/950$ — sinyal payı azalır, gürültü artar, kapalı formülle tek adımda)

28.5 4. β Çizelgesi ve

Gürültü miktarı bir **β çizelgesi** ile belirlenir: her adımda eklenen varyans, $\beta_{\min}$ 'den $\beta_{\max}$ 'a doğrusal artar. Türetilen büyüklükler: $\alpha_t = 1 - \beta_t$, ve $\alpha_t = \prod_{s=1}^t \alpha_s$ (birikimli çarpım — t adımından sonra orijinal sinyalin ne kadar kaldığı). $\sigma_t = \sqrt{\beta_t}$.

```
self.β = torch.linspace(self.βmin, self.βmax, self.n_steps)
self.α = 1. - self.β
self.ā = torch.cumprod(self.α, dim=0)
self.σ = self.β.sqrt()
```

Şekil 28.3 çizelgeyi **gerçek hesaplamayla** iki panelde çizer: solda β_t (her adımda eklenen varyans, linear artar) ve $\sigma_t = \sqrt{\beta_t}$; sağda α_t (birikimli çarpım — kalan sinyal, 1'den 0'a düşer) ile birlikte forward formülünün iki katsayısı $\sqrt{\alpha_t}$ (sinyal katsayısı) ve $\sqrt{1 - \alpha_t}$ (gürültü katsayısı). Bu iki katsayının t boyunca makaslaması — biri inerken diğeri çıkar — forward gürültülemenin tüm dinamiğidir.



Şekil 28.3: (gerçek hesaplama: DDPM çizelgesi — β linear artar, $\alpha = 1 - \beta$, $\alpha_t = \alpha$ (kalan sinyal 1 → 0), $\sigma = \sqrt{\beta}$)

💡 Builder Notu — Çizelge: En Kritik Hiperparametre

- **İleriye (Ders 22):** $\beta/$ çizelgesi diffusion'ın en kritik hiperparametresidir; çizelgenin şekli (linear vs cosine) örnek kalitesini doğrudan etkiler. Ders 22 (Karras, cosine schedule) bunu yeniden tasarlar.
- **Sezgi:** `torch.cumprod` tek satırda tüm sihri yapar: $\alpha_t = 1 - \beta_t$ değerlerinin birikimli çarpımı, α_t 'yi verir — yani “ilk t adımda sinyalin ne kadarı hayatta kaldı?” sorusunun cevabı. β linear artarken düzgün biçimde 1'den 0'a iner; bu sayede her t için forward formülünün katsayıları önceden hesaplanıp hazır tutulur (`self.ā[t]` ile anında erişim).

28.6 5. Reverse Process: Gürültüyü Kaldırmak (Zor Yön)

Reverse süreç tersini yapar: gürültülü x_t 'den biraz daha az gürültülü x_{t-1} 'e geçer. Bu yön **çok zordur** — saf gürültüden anlamlı bir görüntüye gitmek. Çözüm: her adımı tahmin eden bir **sinir ağı** eğit. Üretim, saf gürültüden başlayıp bu adımı n_steps kez tekrarlamaktır.

“the reverse direction is incredibly difficult... Neural network. And we learn it during the training of the model.” — Howard, 26:00 / 34:24

💡 Builder Notu — Reverse: Öğrenilen Yön

- **İleriye (Ders 21):** Reverse süreç n_steps (örn. 1000) adım sürer — yavaş, her adım bir U-Net çağırısı. DDIM (Ders 21) aynı kaliteyi çok daha az adımda (~50) verir.
- **Sezgi:** Forward'ı kapalı formülle “atlayabilmemize” rağmen reverse'ü atlayamayız: tek bir gürültülü x_t 'ye karşılık sonsuz olası temiz x_0 vardır, dolayısıyla ters geçiş belirsizdir. Ağ bu belirsizliği, “bu seviyede tipik olarak hangi gürültü eklenmiş?” sorusunu öğrenerek küçük adımlarla çözer — her adım az gürültü kaldırdığı için belirsizlik yönetilebilir kalır.

28.7 6. Anahtar İçgörü: Kayıp Sadece MSE

DDPM'in tüm karmaşık matematiği (ELBO, KL ıraksamaları) pratikte tek bir şeye iner: ağ, eklenen **gürültüyü tahmin eder** ve kayıp, tahmin edilen gürültü ile gerçek gürültü arasındaki **MSE**'dir:

$$L = \|\varepsilon - \varepsilon_\theta(x_t, t)\|^2$$

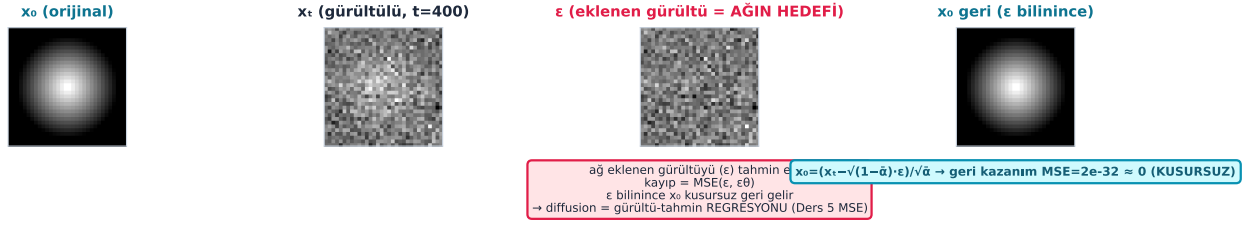
Bu içgörü dersin kalbidir: diffusion'ı eğitmek, “şu görüntüye eklediğim gürültü neydi?” sorusunu MSE ile yanıtlamaktan ibarettir.

“This is just an MSE loss... what this neural network is doing is trying to predict that noise.” — Howard, 35:59 / 37:38

Şekil 28.4 anahtar içgörüyü **gerçek hesaplamayla** dört panelde gösterir (**FLAGSHIP**): x_0 (orijinal) $\rightarrow x_t$ (gürültülü) $\rightarrow \varepsilon$ (eklenen gürültü, *ağın tahmin hedefi*) $\rightarrow x_0$ geri (ε bilinince). Kritik nokta sağ panelde: ε bilindiğinde $x_0 = (x_t - \sqrt{(1 - \beta_t)} \cdot \varepsilon) / \sqrt{\beta_t}$ formülüyle **kusursuz** geri kazanılır (geri kazanım MSE ≈ 0). Yani diffusion, “eklenen gürültüyü tahmin et” regresyon problemine — Ders 5'in basit MSE'sine — indirgenir.

💡 Builder Notu — MSE: Görkemli Modelin Basit Kalbi

- **Geriye (Ders 5/13):** MSE'yi Ders 5 ve 13'te kurmuştuk; diffusion gibi görkemli bir şeyin kalbinde aynı basit kayıp var. “Tuning is physics, not magic” ruhuyla, diffusion da sihir değildir — kalbinde sıradan bir regresyon kaybı yatar.
- **Sezgi:** ε 'yu tahmin etmekle x_0 'ı tahmin etmek matematiksel olarak denktir, çünkü forward formülü ikisini lineer bağlar: biri bilinince diğeri tek satırda çıkar (figürde gösterilen geri kazanım). Pratikte ε -prediction tercih edilir — gürültü her t 'de aynı ölçekte (standart normal) olduğu için kayıp daha kararlı davranır.

DDPM ϵ -prediction: diffusion = gürültü-tahmin regresyonu (ϵ bilinince x_0 KUSURSUZ)

Şekil 28.4: (gerçek hesaplama: DDPM ϵ -prediction — ağ eklenen gürültüyü tahmin eder (kayıp MSE), ϵ bilinince x_0 kusursuz geri kazanılır (MSE $\approx$ 0); diffusion = gürültü regresyonu)

28.8 7. noisify: Forward'ı Koda Dökmek

Forward process, eğitimde her batch'te uygulanır (before_batch): rastgele bir t adımı seç, kapalı formülle görüntüyü o seviyeye gürültüyle, ağın hedefi eklenen ϵ olsun. Üç satır matematik, üç satır kod.

```
def before_batch(self, learn):
    device = learn.batch[0].device
     $\epsilon$  = torch.randn(learn.batch[0].shape, device=device) # gürültü
    x0 = learn.batch[0] # orijinal görüntü
    n = x0.shape[0]
    t = torch.randint(0, self.n_steps, (n,), device=device, dtype=torch.long) # rastgele adım
     $\bar{\alpha}_t$  = self. $\bar{\alpha}$ [t].reshape(-1, 1, 1, 1).to(device)
    xt =  $\bar{\alpha}_t$ .sqrt()*x0 + (1- $\bar{\alpha}_t$ ).sqrt()* $\epsilon$  # forward kapalı form
    learn.batch = ((xt, t),  $\epsilon$ ) # girdi: (gürültülü, t); hedef:  $\epsilon$ 
```

! Builder Notu — noisify: Teori→Kod Köprüsünün Zirvesi (MERKEZÎ)

- **Geriye (Ders 9B) — köprü doruğu:** $x_t = \bar{\alpha}_t \cdot x_0 + \sqrt{1 - \bar{\alpha}_t} \cdot \epsilon$ satırı, 9B'nin forward process formülünün (§3) birebir koda dökülmüş hâlidir. Block math forward formülü = bu tek satır. Bu, Part 2'nin teori-pratik köprüsünün doruğudur (§12 bu köprüyü açar).
- **İki satır, tek fizik:** §3'teki $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon$ formülü ile koddaki $\bar{\alpha}_t \cdot x_0 + (1 - \bar{\alpha}_t) \cdot \epsilon$ aynı matematiksel ifadedir — sadece notasyon değişti ($\sqrt{\cdot} \rightarrow \bar{\alpha}_t \cdot \text{sqrt}(\cdot)$, $\epsilon \rightarrow \epsilon$). Notebook'un Unicode değişkenleri (β , α , ϵ) doğrudan 9B notasyonunu kodda korur.
- **Sezgi:** before_batch'in zarafeti: eğitim verisini *yeniden tanımlar*. Girdi artık temiz görüntü değil, (gürültülü görüntü, t adımı) ikilisidir; hedef de etiket değil, *eklenen gürültü ϵ* 'dir. Böylece standart miniai eğitim döngüsü, hiç değişmeden, bir diffusion modeli eğitir.

28.9 8. U-Net: Gürültü Tahmincisi

Gürültüyü tahmin eden ağ bir **U-Net**'tir: girdisi gürültülü görüntü + zaman adımı t, çıktısı aynı şekilde bir görüntü (tahmin edilen gürültü). Bu derste hazır `diffusers UNet2DModel` kullanılır; mimarisi L18'in

ResBlock'larından kuruludur (encoder-decoder + skip connections).

```
from diffusers import UNet2DModel
model = UNet2DModel(in_channels=1, out_channels=1, block_out_channels=(32, 64, 128, 128))
```

“image and predict the noise. So the shapes of the input and the output are the same.” — Howard, 41:07

💡 Builder Notu — U-Net: Foundations Bloklarından Kurulu

- **Geriye (Ders 18):** U-Net'in iç yapıtaşı ResBlock'tur (L18); diffusion'ın ağı, foundations'ta sıfırdan kurduğumuz blokların birleşimidir. L18'in $out = F(x) + x$ skip connection'ı burada hem ResBlock içinde hem U-Net'in encoder-decoder skip'lerinde iki kez karşımıza çıkar.
- **İleriye (Ders 26):** Ders 26'da U-Net'i diffusers'tan hazır almak yerine, tıpkı ResBlock'u kurduğumuz gibi *sıfırdan* kuracağız — “yeniden kur, sonra kullan” zincirinin bir halkası daha.
- **Sezgi:** Girdi ve çıktının aynı şekilde (örn. $1 \times 28 \times 28$) olması bir tesadüf değil, görevin doğasıdır: ağ “her piksele eklenen gürültü neydi?” sorusunu yanıtlar, dolayısıyla çıktı da girdiyle aynı uzamsal boyutta bir gürültü haritasıdır. U-Net'in encoder-decoder + skip yapısı tam bu “aynı boyutta gir, aynı boyutta çık ama dönüştür” görevi için idealdir.

28.10 9. Eğitim Döngüsü: DDPMCB

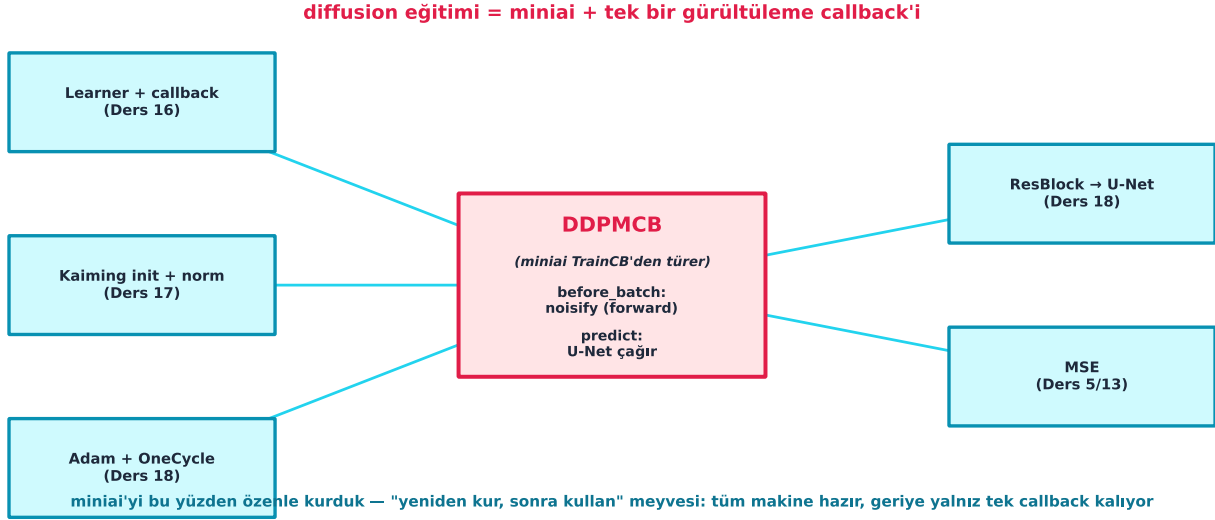
Tüm DDPM mantığı bir **callback**'te toplanır: DDPMCB, miniai'nin TrainCB'sinden türer. `before_batch` gürültüleme yapar, `predict` U-Net'i çağırır. Eğitim, standart bir miniai Learner ile: MSE kayıp, Adam, OneCycle scheduler.

```
ddpm_cb = DDPMCB(n_steps=1000, beta_min=0.0001, beta_max=0.02)
cbs = [ddpm_cb, DeviceCB(), ProgressCB(plot=True), MetricsCB(), BatchSchedCB(sched)]
learn = Learner(model, dls, nn.MSELoss(), lr=lr, cbs=cbs, opt_func=optim.Adam)
learn.fit(epochs)
```

Şekil 28.5 eğitimin yapısını şematik gösterir: ortada DDPMCB (miniai TrainCB'den türer — `before_batch` gürültüler, `predict` U-Net'i çağırır), çevresinde foundations'tan gelen tüm parçalar — Learner+callback (L16), Kaiming init+norm (L17), Adam+OneCycle (L18), ResBlock→U-Net (L18), MSE (L5/13). Mesaj net: **diffusion eğitimi = miniai + tek bir gürültüleme callback'i.**

❗ Builder Notu — DDPMCB: Foundations'ın Tümü Burada Birleşir (MERKEZİ)

- **Geriye (Ders 16-18) — MERKEZİ:** DDPM, baştan kurduğumuz her şeyi birleştirir: Learner+callback (L16), Adam+OneCycle (L18), MSE. Diffusion eğitimi = miniai + bir callback. miniai'nin tam donanımlı olmasının sebebi tam buydu — diffusion gibi karmaşık bir modeli sağlam bir altyapı üstünde birkaç parçayla eğitebilmek.
- **Sezgi:** DDPMCB'nin TrainCB'den türemesi tüm yükü taşır: eğitim döngüsünün geri kalanı (forward/backward/step) miras alınır, yalnızca iki kanca özelleşir — `before_batch` (gürültüleme) ve `predict` (U-Net çağırısı). Bu, L16'da callback mimarisini bu kadar özenle kurmanın asıl ödülüdür: yeni bir model ailesini (diffusion) çekirdek döngüye hiç dokunmadan eklersin.



Şekil 28.5: (şematik: DDPM = miniai + bir callback — DDPMCB noisify+predict; Learner/init/Adam/ResBlock-U-Net hepsi foundations'tan (L16-18))

28.11 10. Sampling: Reverse'ü Koda Dökmek

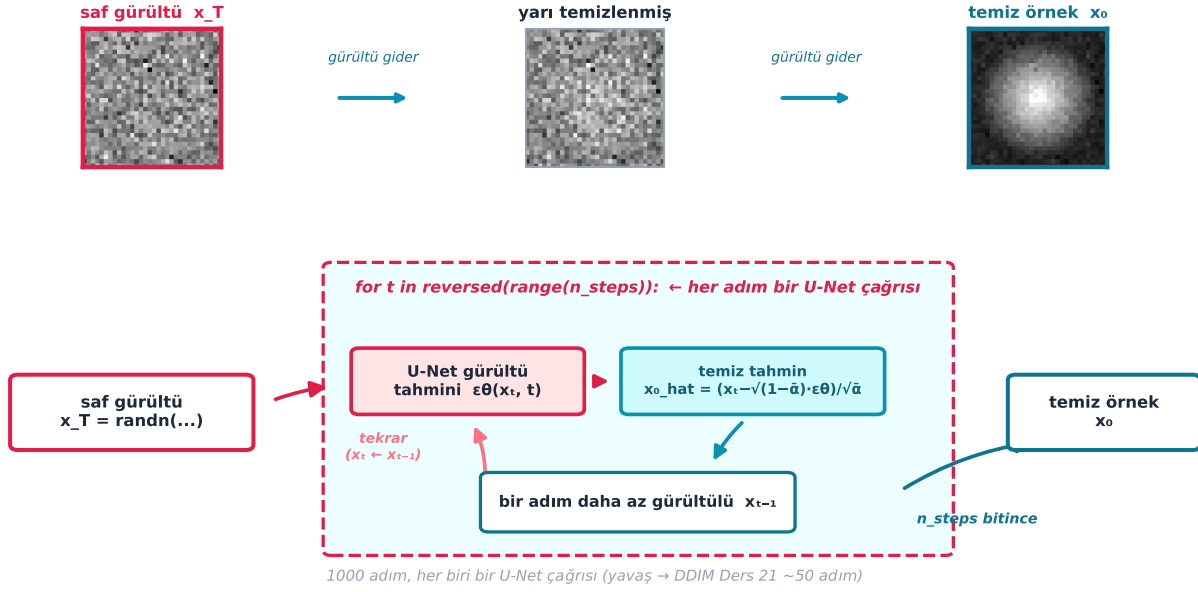
Üretim (sample): saf gürültüyle başla ($x_t = \text{randn}$), n_steps adım geriye git. Her adımda U-Net gürültüyü tahmin eder, ondan temiz görüntü tahminini (x_{0_hat}) çıkar, bir adım daha az gürültülü duruma geç. Son adımda temiz örnek elde edilir.

```

@torch.no_grad()
def sample(self, model, sz):
    x_t = torch.randn(sz, device=device)
    preds = []
    for t in reversed(range(self.n_steps)):
        z = torch.randn(x_t.shape) if t > 0 else torch.zeros(x_t.shape)
        noise_pred = learn.model(x_t, t_batch).sample
        x_0_hat = ((x_t - (1-self.α[t]).sqrt()*noise_pred)/self.α[t].sqrt()).clamp(-1,1)
        x_t = x_0_hat*x0_coeff + x_t*xt_coeff + self.σ[t]*z
        preds.append(x_t.cpu())
    return preds
  
```

"by predicting the noise and subtracting it out, we are going back to the distribution." — Howard, 39:34

Şekil 28.6 sampling döngüsünü şematik gösterir: üstte kademeli gürültü→görüntü şeridi (saf gürültü $x_T \rightarrow$ yarı temizlenmiş $\rightarrow$ temiz örnek x_0), altta akış şeması — saf gürültüyle başla, `for t in reversed(range(n_steps))` döngüsünde her adım U-Net gürültü tahmini ($\epsilon\theta$) $\rightarrow$ temiz tahmin (x_{0_hat}) $\rightarrow$ bir adım az gürültülü (x_{t-1}), ve geri besleme ile tekrar. n_steps bitince temiz örnek çıkar. Alt not, döngünün yavaşlığını (1000 adım) ve **DDIM çözümü**nü (Ders 21, ~50 adım) hatırlatır.



Şekil 28.6: (şematik: DDPM sampling — saf gürültüden başla, her adım U-Net gürültü tahmini + kademeli çıkar; n_steps geriye giderek temiz görüntü)

💡 Builder Notu — Sampling: Yavaş ama Doğru

- **İleriye (Ders 21):** Bu döngü 1000 adım sürer — bir görüntü için 1000 U-Net çağırısı, yavaş. DDIM (Ders 21) aynı kaliteyi ~50 adımda verir; aradaki fark üretim hızında devrim yaratır.
- **Sezgi:** Son adımda ($t=0$) $z = \text{zeros}$ olması bilinçlidir: ara adımlarda küçük rastgele gürültü ($\sigma_t \cdot z$) eklemek üretimi çeşitlendirir, ama en son adımda gürültü eklemek nihai temiz örneği yeniden kirletirdi. x_0_hat her adımda U-Net'in “şu anki en iyi temiz tahmini”dir; tam ona atlamak yerine bir adım kadar yaklaşmak (kısmi güncelleme), reverse sürecin küçük ve güvenli adımlarla ilerlemesini sağlar.

28.12 11. DDPM'in Üç Adımlı Eğitim Reçetesi

Howard her şeyi tek cümlede özetler: rastgele bir zaman adımı seç, görüntüye o kadar gürültü ekle, ağa gürültülü görüntü + adımı ver, eklenen gürültüyü MSE ile tahmin ettir. Bu kadar.

“we select a random timestep and then we add noise to our image... we pass the noisy image to a model as well as the timestep. And we are trying to predict the amount of noise... with the MSE loss.” — Howard, 43:55

💡 Builder Notu — Reçete: Önce Çalıştır, Sonra Anla

- **Geriye (Ders 3):** “Önce çalıştır, sonra anla” — karmaşık ELBO matematiğini bilmesen de bu üç adımlı reçeteyle diffusion eğitebilirsin; anlama sonra gelir. Howard kasıtlı olarak olasılıksal türevi atlar ve doğrudan çalışan koda gider.

- **Sezgi:** Reçetenin üç adımı doğrudan üç kod parçasına eşlenir: “rastgele t + gürültü ekle” → `before_batch` (§7); “gürültülü + t ’yi ağa ver” → `U-Net predict` (§8); “gürültüyü MSE ile tahmin et” → `nn.MSELoss()` (§6). Üç cümle, üç satır, bir model.

28.13 12. 9B Matematiğinin Koda Dökülmesi

Bu ders, Part 2’nin teori-pratik köprüsünün doruğudur. Ders 9B’de Waseem ve Tanishq forward process $q(x_t|x_{t-1})$, Gaussian transitions, reparameterization ve forward’ın kapalı formunu *türetti*. Burada o matematik, `before_batch`’teki tek satıra ($x_t = \bar{\alpha}_t.\text{sqrt}()*x_0 + (1-\bar{\alpha}_t).\text{sqrt}()*\epsilon$) iner. Notebook’un Unicode değişkenleri ($\beta, \alpha, \epsilon, \sigma$) doğrudan 9B notasyonunu yansıtır.

Köprüyü açık biçimde yan yana koyalım:

9B’de (matematik)	L19’da (kod)
$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon$	<code>x_t = $\bar{\alpha}_t.\text{sqrt}()*x_0 + (1-\bar{\alpha}_t).\text{sqrt}()*\epsilon$</code>
$\bar{\alpha}_t = \prod_{s=1}^t (1 - \beta_s)$	<code>self.$\bar{\alpha}$ = torch.cumprod(self.α, dim=0)</code>
$L = \ \epsilon - \epsilon_\theta(x_t, t)\ ^2$	<code>nn.MSELoss()</code>
$\epsilon \sim \mathcal{N}(0, I)$	<code>ϵ = torch.randn(...)</code>

! Builder Notu — Köprü: Matematik Bilen Kodu, Kod Bilen Matematiği Anlar (MERKEZÎ)

- **Geriye (Ders 9B) — köprü:** 9B’nin block math forward process formülü = bu dersin noisify kodu. Matematik bilen builder kodu, kod bilen builder matematiği anlar. Tablo bu eşlemeyi satır satır gösterir: her matematiksel ifade tam bir kod satırına karşılık gelir.
- **Sezgi:** Bu eşlemenin değeri çift yönlüdür. 9B’yi izleyip türevi anlayan biri, koddaki `$\bar{\alpha}_t.\text{sqrt}()*x_0$` ’ın *neden* doğru olduğunu bilir (reparameterization). Tersine, kodu önce gören biri (Howard’ın “önce çalıştır” yaklaşımı) `cumprod` ve `randn`’ün ne yaptığını görüp matematiği geriye doğru sökebilir. İkisi de aynı tek gerçeğe — forward kapalı formuna — varır.

28.14 13. NYU Köprüsü: Üretken Paradigmalar

DDPM’i diğer üretken modellerle konumlandırmak (NYU Deep Learning, LeCun): **VAE** (NYU H8, auto-encoder/EBM) verili bir latent’ten tek adımda üretir; **GAN** (NYU H9, contrastive EBM) bir ayırıcıya karşı üretir; **diffusion** ise üretimi *yinelemeli arıtmaya* (iterative refinement) böler — gürültüyü kademeli kaldırarak. DDPM’in noise-prediction hedefi, GAN’ın adversarial hedefine bir alternatiftir; daha kararlı eğitim verir.

💡 Builder Notu — NYU: Diffusion Üçüncü Büyük Alternatif

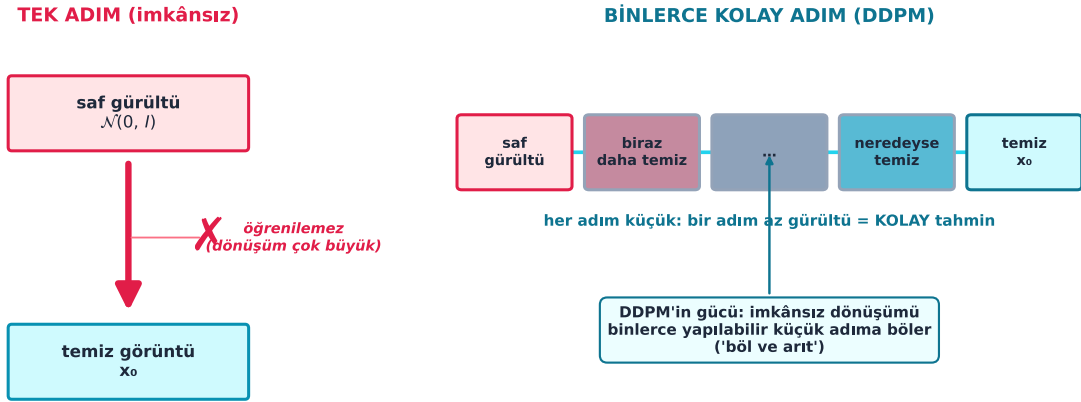
- **Geriye (NYU H8-9):** NYU’da EBM/VAE/GAN paradigmalarını gördük; diffusion bunların üçüncü büyük alternatiftir. NOT: BYOL/VICReg/MAE/JEPA gibi post-2020 yöntemler NYU DLSP20’de yoktu — diffusion da o kursta yalnızca kavramsaldı; burada sıfırdan kuruluyor.
- **Sezgi:** Üç paradigmayı ayıran şey, üretim *kaç adımda* olur: VAE tek decode adımında, GAN

tek generator geçişinde, diffusion ise binlerce küçük denoising adımında üretir. Bu “çok adıma bölme”, diffusion’ın eğitimini GAN’a kıyasla daha kararlı kılar (adversarial denge yok, sadece bir MSE regresyonu) — pratikteki popüleritesinin temel sebeplerinden biri.

28.15 14. Neden DDPM Çalışır? (Yinelemeli Arıtma)

DDPM’in gücü, zor bir problemi (saf gürültüden görüntü) çok sayıda kolay probleme (bir adım daha az gürültü) bölmektedir. Her adımda ağ küçük, yapılabilir bir tahmin yapar; binlerce küçük adım birikince tek seferde imkânsız olan dönüşüm gerçekleşir. Bu “böl ve arıt” stratejisi, modern üretken modellemenin temel fikridir.

Şekil 28.7 bu içgörüyü iki panelde karşılaştırır: solda **tek adım (imkânsız)** — saf gürültüden temiz görüntüye tek dev sıçrama, öğrenilemez (dönüşüm çok büyük); sağda **binlerce kolay adım (DDPM)** — saf gürültü → biraz daha temiz → ... → temiz, her adım küçük ve yapılabilir bir tahmin. DDPM’in gücü tam burada: imkânsız dönüşümü binlerce yapılabilir küçük adıma bölmek.



yinelemeli arıtma — modern üretken modellemenin temel fikri

Şekil 28.7: (şematik: yinelemeli arıtma — diffusion zor problemi (gürültü→görüntü) binlerce kolay adıma (bir adım az gürültü) böler)

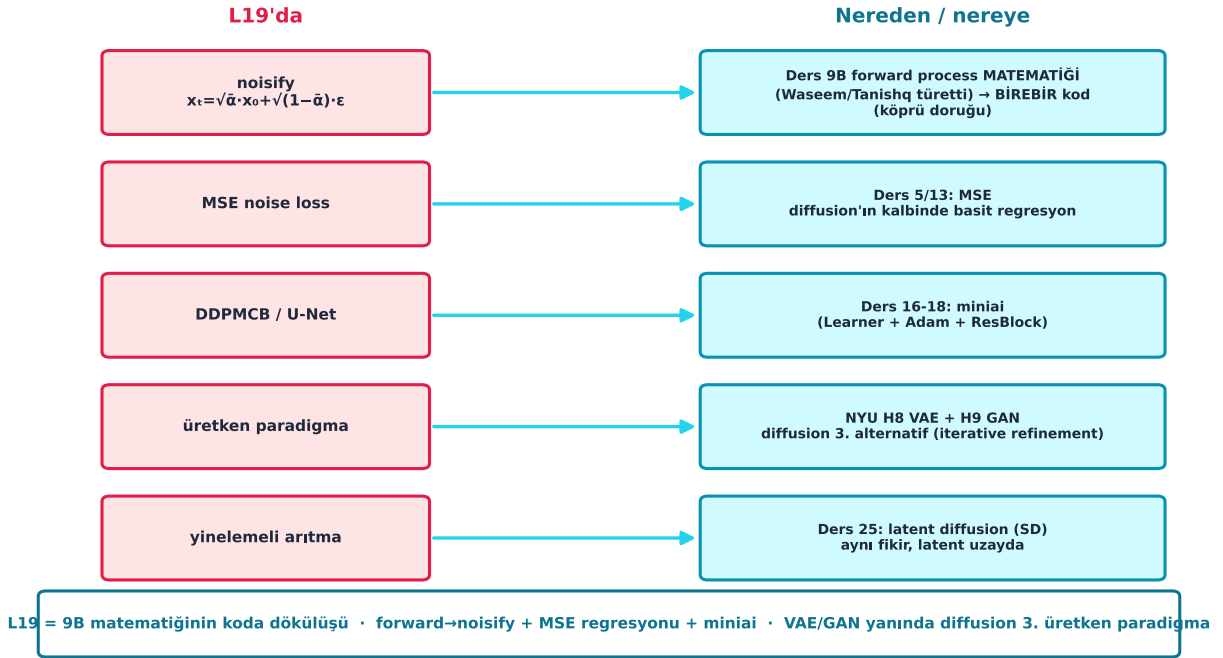
💡 Builder Notu — Yinelemeli Arıtma: Böl ve Arıt

- **İleriye (Ders 25):** Aynı yinelemeli arıtma, latent uzayda yapılır (Ders 25) Stable Diffusion olur — daha hızlı, daha yüksek çözünürlük. Fikir değişmez, sadece piksel uzayı yerine sıkıştırılmış latent uzayda çalışır.
- **Sezgi:** Bir adımda gürültüden görüntüye gitmek imkânsızdır çünkü dağılımlar arası mesafe çok büyüktür — ağırlık öğrenmesi gereken eşleme aşırı karmaşık olurdu. Her adımı küçük tutarak ($x_t \rightarrow x_{t-1}$, az gürültü farkı) her tahmin yerel ve kolay kalır; zorluk binlerce adıma yayılır. Bu, “büyük bir problemi yönetilebilir küçük parçalara böl” mühendislik ilkesinin üretken modellemedeki tezahürüdür.

28.16 15. Kapanış

Ders 19, Part 2'nin teori-pratik köprüsünü tamamladı: DDPM'i sıfırdan kurduk. Forward process (kapalı formülle gürültü ekle) + reverse process (U-Net ile gürültüyü tahmin et) + MSE kaybı + miniai eğitim döngüsü. 9B'nin matematiği koda döküldü; foundations'ta (L15-18) kurduğumuz her şey (Learner, callback, Adam, ResBlock/U-Net) burada birleşti. Diffusion artık kara kutu değil.

Şekil 28.8 dersin sentezidir: solda L19'da kurduğumuz parçalar (rose), sağda nereden geldikleri / nereye gittikleri (cyan). En üstte köprü doruğu — $\text{noisify}(x_t = \sqrt{\bar{\alpha}} \cdot x_0 + \sqrt{1 - \bar{\alpha}} \cdot \epsilon)$ = Ders 9B forward process matematiğinin birebir kodu; altında MSE → Ders 5/13 regresyonu, DDPMCB/U-Net → Ders 16-18 miniai, üretken paradigma → NYU H8 VAE + H9 GAN yanında diffusion üçüncü alternatif, yinelemeli arıtma → Ders 25 latent diffusion. Sentez: **L19 = 9B matematiğinin koda dökülüşü + MSE regresyonu + miniai**.



Şekil 28.8: (sentez: L19 = 9B matematiğinin koda dökülüşü (forward→noisify) + MSE regresyonu + miniai; NYU VAE/GAN yanında diffusion 3. üretken paradigma)

💡 Builder Notu — Kapanış: Diffusion Artık Kara Kutu Değil

- **İleriye (Ders 20-22):** DDPM temeldir; Ders 20 (mixed precision + style transfer), 21 (FID/KID metrikleri + DDIM hızlandırma), 22 (cosine schedule + Karras) hep bunun üstüne kurulur. ETAP 6'nın bu açılış dersi, gelecek tüm diffusion derslerinin çekirdeğini verir.
- **Sezgi:** Bu dersten sonra `diffusers` kütüphanesini çağırdığında altında ne döndüğünü biliyorsun: forward formülü, ϵ -prediction, MSE, reverse sampling döngüsü. Tıpkı L18 sonrası `optim.Adam`'ın içini bilmek gibi — diffusion da artık demistifiye edilmiş bir araç, sihirli bir kara kutu değil. “From the foundations” yolculuğunun meyvesi tam budur.

28.17 Bu Dersin Özeti

1. **Dropout:** Eğitimde aktivasyonların rastgele kısmını sıfırlayan düzenleme; çıkarımda kapalı (Batch-Norm gibi train/eval ayrımı) ([Dropout](#)).
2. **DDPM:** Saf piksel diffusion (latent yok); forward (gürültü ekle) + reverse (gürültü kaldır) süreçleri ([DDPM nedir](#)).
3. **Forward (kolay):** $x_t = \sqrt{\alpha_t} \cdot x_0 + \sqrt{1 - \alpha_t} \cdot \epsilon$; kapalı formülle tek adımda istenen gürültü seviyesi ([forward](#)).
4. **β çizelgesi:** β_t doğrusal artar; $\alpha_t = 1 - \beta_t$, $\alpha_t = \alpha_s$ (birikimli çarpım, kalan sinyal) ([çizelge](#)).
5. **Reverse (zor):** Gürültüden görüntüye; bir U-Net her adımı öğrenir, n_steps kez tekrarlanır ([reverse](#)).
6. **Anahtar içgörü:** Ağ eklenen gürültüyü (ϵ) tahmin eder; kayıp = $MSE(\epsilon, \epsilon\theta)$. Diffusion'ın tüm karmaşıklığı buna iner ([MSE](#)).
7. **DDPMCB:** Tüm mantık miniai TrainCB'den türeyen bir callback'te; eğitim = standart Learner + MSE + Adam + OneCycle ([DDPMCB](#)).
8. **Sampling:** Saf gürültüden başla, U-Net'le gürültüyü kademeli kaldır; zor problemi binlerce kolay adıma böl ([sampling](#), [yinelemeli arttırma](#)).

! Tek Bir Cümle

DDPM, bir görüntüye kapalı formülle ($x_t = \sqrt{\alpha_t} \cdot x_0 + \sqrt{1 - \alpha_t} \cdot \epsilon$) gürültü ekleyip bir U-Net'e o gürültüyü MSE ile tahmin ettirerek eğitilen üretken modeldir; üretim, saf gürültüden başlayıp ağın tahmin ettiği gürültüyü adım adım kaldırarak imkânsız dönüşümü binlerce kolay adıma bölmektir — ve bu, 9B'de türetilen matematiğin miniai üstünde koda dökülmüş hâlidir.

28.18 Kontrol Soruları

i Soru 1: Forward process neden 'kolay', reverse process neden 'zor'dur?

Cevap:

Forward process bir görüntüye gürültü eklemektir — bu önemsizdir, sadece rastgele gürültü ekliyorsun. Üstelik kapalı formül sayesinde tüm ara adımları gezmeden, t adımıdaki gürültülü görüntüyü tek hamlede üretebilirsin: $x_t = \sqrt{\alpha_t} \cdot x_0 + \sqrt{1 - \alpha_t} \cdot \epsilon$. **Reverse process** tersidir: saf gürültüden (veya gürültülü bir görüntüden) anlamlı, daha temiz bir görüntüye gitmek. Bu son derece zordur çünkü sonsuz olası “temiz görüntü” vardır ve hangi gürültünün eklendiğini bilmek gerekir. Çözüm: bu zor ters adımı bir **sinir ağına** (U-Net) öğretmek. Ağ “bu gürültülü görüntüye hangi gürültü eklenmiş?” sorusunu yanıtlar; o gürültüyü çıkararak bir adım geriye gideriz. (Şekil 28.2 forward'ın kolaylığını, Şekil 28.6 reverse'ün adım adım zorluğunu gösterir.)

i Soru 2: DDPM'in kaybı neden 'sadece MSE' diye özetlenir?

Cevap:

DDPM'in tam teorik türevi olasılıksaldır (ELBO — evidence lower bound, KL ıraksamaları, Gaussian geçişler). Ama bu matematik sadeleştirildiğinde pratik eğitim hedefi tek bir şeye iner: ağ, görüntüye **eklenen gürültüyü tahmin eder** ($\epsilon\theta$), ve kayıp, tahmin edilen gürültü ile gerçek eklenen gürültü

arasındaki ortalama kare hatadır: $L = \|\epsilon - \epsilon\theta(x_t, t)\|^2$. Yani diffusion eğitmek, “şu görüntüye eklediğim ϵ neydi?” regresyon sorusunu MSE ile çözmektir. Howard’ın vurgusu tam buradadır: görkemli diffusion modellerinin kalbinde, Ders 5’te gördüğümüz basit MSE vardır — sihir değil. (Şekil 28.4 ϵ bilinince x_0 ’ın kusursuz geri kazanıldığını gösterir.)

i Soru 3: α_t (alpha bar) nedir ve forward formülünde ne işe yarar?

Cevap:

α_t , gürültü çizelgesinin **birikimli çarpımıdır**: $\alpha_t = \alpha_{t-1} \alpha_s$, burada $\alpha_s = 1 - \beta_s$. Sezgisi: t adımından sonra orijinal sinyalin (görüntünün) ne kadarının “hayatta kaldığı”. Forward formülünde $x_t = \sqrt{\alpha_t} \cdot x_0 + \sqrt{1 - \alpha_t} \cdot \epsilon$ iki katsayıyı belirler: $\sqrt{\alpha_t}$ orijinal görüntünün ağırlığı, $\sqrt{1 - \alpha_t}$ gürültünün ağırlığı. t küçükken $\alpha_t \approx 1$ (görüntü neredeyse temiz); t büyükken $\alpha_t \approx 0$ (neredeyse saf gürültü). Birikimli çarpım sayesinde herhangi bir t için gürültülü görüntüyü tek adımda hesaplayabiliriz — ara adımları tek tek uygulamaya gerek yok. Kod: `self.̑ = torch.cumprod(self.̑, dim=0)`. (Şekil 28.3 ’nın 1’den 0’a düşüşünü ve iki katsayının makaslamasını gösterir.)

i Soru 4: DDPM, foundations derslerini (L16-18) nasıl kullanır? (builder bağlantısı)

Cevap:

DDPM, “from the foundations” yolculuğunda kurduğumuz her parçayı birleştirir. **Ders 16 (Learner+callback)**: Tüm DDPM mantığı (DDPMCB) miniai’nin TrainCB’sinden türeyen bir callback’tir; eğitim döngüsü standart miniai Learner’dır. **Ders 17 (init/norm)**: U-Net’in katmanları Kaiming init + normalizasyon kullanır, yoksa eğitilemezdi. **Ders 18 (Adam/OneCycle/ResBlock)**: Eğitim Adam + OneCycle ile yapılır; U-Net’in iç yapıtaşı L18’in ResBlock’udur. Yani DDPM eğitimi = miniai + bir gürültüleme callback’i. Howard’ın L16-18’de miniai’yi bu kadar özenle kurmasının sebebi tam buydu: diffusion gibi karmaşık bir modeli, sağlam bir altyapı üstünde birkaç parçayla eğitebilmek. Bu, “yeniden kur, sonra kullan” kuralının meyvesidir. (Şekil 28.5 foundations parçalarının DDPMCB’de birleşimini gösterir.)

28.19 Egzersizler

Egzersiz 1 (Direkt uygulama). β çizelgesini (`torch.linspace`) kur, α_t ve β_t ’yi hesapla; β_t ’nin t boyunca 1’den 0’a nasıl düştüğünü çiz (Şekil 28.3’nin eğrisini kendi sayımla doğrula — §4).

Egzersiz 2 (İki-aşamalı). `before_batch` gürültüleme kodunu bir görüntüye uygula; farklı t değerlerinde (örn. $t=50, 500, 950$) gürültülü görüntüyü görselleştir (Şekil 28.2’in sinyal/gürültü ayrışımını üret — §7).

Egzersiz 3 (Edge case). `sample` döngüsünü yaz; $t=0$ ’da $z=0$ (son adımda gürültü eklenmez) olmasının neden gerekli olduğunu açıkla (§10).

Egzersiz 4 (Kavramsal). Ağ neden gürültüyü (ϵ) tahmin eder de doğrudan temiz görüntüyü (x_0) tahmin etmez? İkiisi matematiksel olarak nasıl ilişkili? (§6)

Egzersiz 5 (Sonraki dersin habercisi — Ders 20). DDPM eğitimi yavaş ve bellek-yoğun. Mixed precision (fp16) eğitimi nasıl hızlandırıp belleği nasıl azaltabilir, düşün (§15).

28.20 Sonraki: Ders 20 İçin Hazırlık

Ders 20: Mixed Precision ve Style Transfer

Ders 19 DDPM'i sıfırdan kurdu. Ders 20 iki konu işler: **mixed precision** (fp16/bf16 ile daha hızlı, daha az bellekli eğitim — HuggingFace Accelerate) ve **style transfer** (bir görüntüyü optimize ederek, VGG16 özellikleri + Gram matrisi style loss ile bir resmi başka bir stilde yeniden çizmek). Bu, optimizasyonu *ağırlıklar yerine girdi görüntüsü* üzerinde yapmanın güzel bir örneğidir.

Ana konular (Ders 20):

- MixedPrecision callback + HuggingFace Accelerate
- Görüntüyü optimize etme (ağırlık değil, piksel)
- VGG16 ile özellik çıkarma (hooks)
- Content loss + Style loss (Gram matrisi)

⚠ Ders 20 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 2 — β çizelgesi + gürültüleme görselleştirme).
- DDPM'i Fashion-MNIST'te eğit, sample ile örnek üret.
- Ders 17'nin hook bölümünü tekrar oku (style transfer'da VGG16 özellikleri hook'la çıkarılır).

28.21 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
DDPM	Saf piksel denoising diffusion; latent yok	12:13
Dropout	Aktivasyonların rastgele kısmını sıfırlama (düzenleştirme)	2:08
Forward process	$x_t = \sqrt{\alpha_t} x_0 + \sqrt{1 - \alpha_t} \epsilon$; gürültü ekleme (kolay)	23:58
Reverse process	Gürültüden görüntüye; U-Net öğrenir (zor)	33:51
β çizelgesi	$\beta_{\min} \rightarrow \beta_{\max}$ doğrusal artan varyans	40:26
(alpha bar)	$\bar{\alpha}_t = \alpha_s$; birikimli çarpım (kalan sinyal)	47:06
ϵ -prediction	Ağ eklenen gürültüyü tahmin eder ($\epsilon\theta$)	37:38
MSE loss	$\ \epsilon - \epsilon\theta\ ^2$; diffusion'ın tüm kaybı budur	35:59
noisify / before_batch	Forward'ı eğitimde uygulama	40:26
U-Net	Gürültü tahmincisi; girdi=çıktı şekli (ResBlock'lardan)	41:16
DDPMC	miniai TrainCB'den türeyen DDPM callback'i	56:07

Kavram	Tanım	Howard'da
Sampling	Saf gürültüden n_steps geriye giderek üretim	39:17

28.22 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: 9B Matematiği → Kod + miniai Sentezi

Bu ders, Ders 9B'de türetilen forward process matematiğini miniai üstünde koda döker ve diffusion'ı diğer üretken paradigmalar arasında konumlandırır; köprülerin özeti:

1. **Forward process** → Ders 9B matematiği (Waseem/Tanishq); orada türev, burada `noisify` kodu — köprü doruğu ([noisify](#), [9B köprü](#)).
2. **MSE noise loss** → Ders 5/13 MSE; diffusion'ın kalbinde basit regresyon ([MSE](#)).
3. **DDPMCB** → Ders 16-18 miniai (Learner+callback+Adam+ResBlock); diffusion = miniai + bir callback ([DDPMCB](#)).
4. **U-Net** → Ders 18 ResBlock'ları; diffusion ağı foundations bloklarından kurulu ([U-Net](#)).
5. **Üretken paradigmalar** → NYU H8 VAE/EBM + H9 GAN; diffusion üçüncü büyük alternatif (iterative refinement) ([NYU köprü](#)).
6. **Yinelemeli arıtma** → Ders 25 latent diffusion (Stable Diffusion); aynı fikir latent uzayda ([yinelemeli arıtma](#)).

! Bu dersten tek bir şey alıp gideceksen

Diffusion sihir değildir. Forward process bir görüntüye kapalı formülle gürültü ekler (kolay); reverse process bir U-Net'in tahmin ettiği gürültüyü kademeli kaldırır (zor, öğrenilmiş). Tüm karmaşık olasılık matematiği pratikte tek bir şeye iner — ağ eklenen gürültüyü tahmin eder, kayıp basit bir MSE'dir. 9B'de türetilen matematiği burada miniai üstünde koda döktük; foundations'ta kurduğumuz her parça (Learner, callback, Adam, ResBlock/U-Net) burada birleşti. Üretim, imkânsız bir dönüşümü binlerce kolay adıma bölmektir.

29 Ders 20 — Mixed Precision ve Style Transfer (Mixed Precision and Style Transfer)

ETAP 6'nın (Sıfırdan Diffusion A) iki pratik konusu, tek dersin altında. Önce Howard'ın elinde mixed precision: eğitimi fp16 (yarı hassasiyet) ile hızlandırıp belleği azaltmak — hem sıfırdan bir MixedPrecision callback'ile (autocast + GradScaler) hem HuggingFace Accelerate ile. Sonra Johno ve Tanishq'le style transfer: bir görüntüyü, ağırlıkları değil görüntünün kendisini optimize ederek başka bir resmin stiline dönüştürmek. İkinci konunun builder dersi büyük: gradient descent yalnızca model ağırlıkları için değildir — kaybı tanımla, hedefi parametre yap, optimize et. Tek cümleyle: mixed precision fp16 ile eğitimi hızlandırıp belleği azaltır (autocast + GradScaler, bir callback); style transfer ise gradient descent'i ağırlıklar yerine bir görüntü üzerinde çalıştırır — VGG16 ara katman özellikleri içeriği, Gram matrisi (kanal korelasyonları) stili ölçer, ikisinin MSE kaybı pikselleri yeni bir resme dönüştürür.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 20: Mixed Precision & Style Transfer](https://course.fast.ai/lesson/20) (~106 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 20
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — miniai/accel.py + nbs/16A_StyleTransfer](#)
- **Okuma süresi:** ~38 dk
- **Hocalar:** Howard (mixed precision) + Jonathan Whitaker / Johno + Tanishq Abraham (style transfer)
- **🔗 ETAP 6 (Sıfırdan Diffusion A):** Ders 19'da DDPM'i sıfırdan kurduk; diffusion artık kara kutu değil. Ders 20 iki pratik aracı ekler. Birincisi **mixed precision** — ağır diffusion eğitimlerini (L21+) hızlandırmanın standart yolu, miniai'ye bir callback olarak girer. İkincisi **style transfer** — Ders 9'da “piksel üzerinde gradient descent” diye tanıdığımız sezginin saf, doğrudan hâli: bu kez ağırlık değil, *görüntünün kendisi* optimize edilir.

29.1 Bu Derste Ne Var?

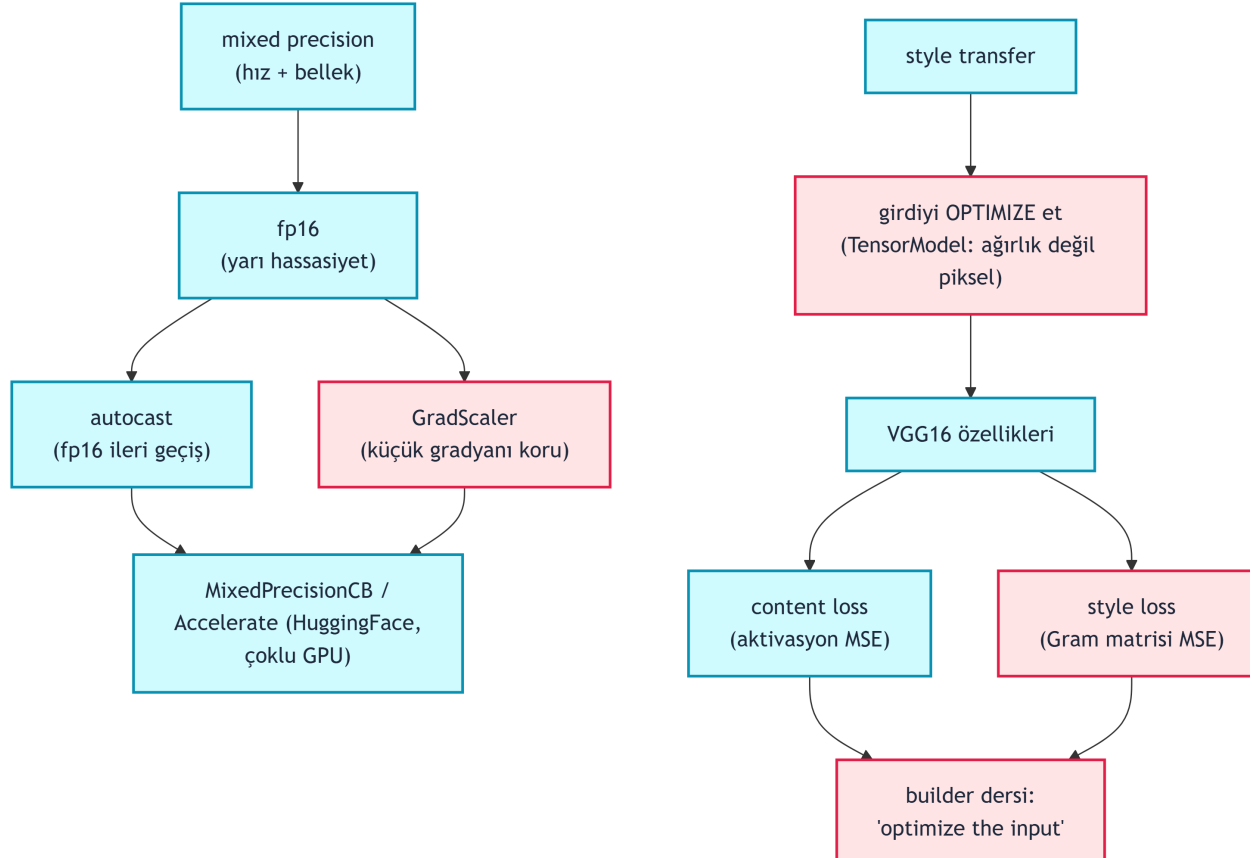
İki ayrı pratik konu, tek dersin altında. Önce **mixed precision** — eğitimi fp16 (yarı hassasiyet) ile hızlandırıp belleği azaltmak; hem sıfırdan bir callback ile hem HuggingFace **Accelerate** ile. Sonra **style transfer** — bir görüntüyü, *ağırlıkları değil görüntünün kendisini* optimize ederek başka bir resmin stiline dönüştürmek. İkinci konu, gradient descent'in ağırlıklar dışında da kullanılabileceğini gösteren zarif bir örnektir.

Üç temel fikir bu dersin omurgasını kurar:

1. **Mixed precision** — fp16 hesaplama daha hızlı ve daha az bellekli; bir MixedPrecision/AccelerateCB callback'ıyla eklenir (autocast + GradScaler) ([mixed precision](#) → [callback](#)).
2. **Görüntüyü optimize etme** — style transfer'da model bir TensorModel'dir; öğrenilen ağırlık değil, görüntünün pikselleridir ([optimize the input](#)).
3. **VGG16 özellikleri + Gram matrisi** — içerik, ara katman aktivasyonlarıyla; stil ise bu özelliklerin Gram matrisi (korelasyonları) ile ölçülür ([VGG16](#) → [content loss](#) → [Gram](#)).

“if you Google for PyTorch mixed precision... with autocast device equals ‘cuda’ type equals float16, get your predictions and call your loss.” — Howard, 3:01

Şekil 34.1 bu iki konuyu tek bir kavram haritasında birleştirir: üstte mixed precision (fp16 ile hız + bellek, autocast ileri geçişi yarı-hassasiyette çalıştırır, GradScaler küçük gradyanları korur, MixedPrecisionCB / Accelerate çoklu GPU'ya kadar uzanır); altta style transfer (ağırlığı değil **girdiyi** — pikselleri — optimize et, VGG16 özelliklerinden content + style/Gram kaybı türet). **GradScaler**, **girdiyi optimize et** ve **‘optimize the input’ builder dersi** rose ile işaretlidir: dersin asıl çekirdeği, optimizasyon hedefinin ağırlık olmak zorunda olmadığıdır.



Şekil 29.1: L20 kavram haritası: iki pratik konu. ÜST — mixed precision (fp16 ile hız+bellek, autocast ileri geçişi yarı-hassasiyette, GradScaler küçük gradyanları korur, MixedPrecisionCB / Accelerate çoklu GPU). ALT — style transfer: ağırlığı değil GİRDİYİ (pikselleri) optimize et, VGG16 özelliklerinden content + style (Gram matrisi) kaybı türet; ‘optimize the input’ builder dersi. (şematik)

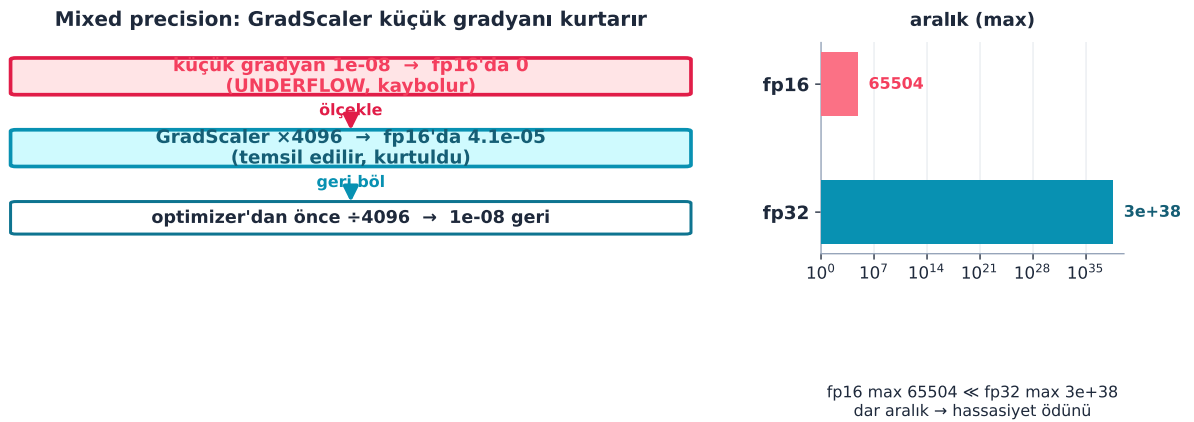
💡 Builder Notu — İki Konu, Bir Çatı: Hızlandırma ve Optimizasyon Genelliği

- **Geriye (Ders 16-18):** Mixed precision yine bir callback (miniai); Ders 16 callback sistemi sayesinde eğitime tek satırla eklenir. Style transfer ise standart bir Learner'ın farklı bir kullanımıdır — verisiz bir optimizasyon problemine uyarlanmış.
- **Geriye (Ders 17):** Style transfer'da VGG16 özellikleri hook fikriyle çıkarılır — Ders 17'nin hook altyapısı burada işe yarar.
- **İleriye (Ders 21-25):** Mixed precision, ağır diffusion eğitimlerini (L21+) hızlandırmak için standart; style transfer ise “optimize the input” fikrinin (Ders 9 pixel gradient) güzel bir örneği.
- **Tek cümle:** Mixed precision fp16 ile eğitimi hızlandırır; style transfer ise gradient descent'i ağırlıklar yerine bir görüntü üzerinde çalıştırarak, VGG16 özellikleri (içerik) ve Gram matrisi (stil) kayıplarıyla bir resmi yeniden çizer.

29.2 1. Mixed Precision Nedir?

Varsayılan eğitim fp32 (32-bit) sayılarla yapılır. **Mixed precision**, hesabın çoğunu fp16 (16-bit) ile yapar: daha hızlı (modern GPU'lar fp16'da çok daha verimli) ve daha az bellek. “Mixed” çünkü bazı hassas işlemler (kayıp birikimi, ağırlık güncellemeleri) fp32'de kalır. Sonuç: aynı GPU'da daha büyük model/batch ve daha hızlı eğitim.

Şekil 29.2 sorunun ve çözümün özünü **gerçek hesaplama**yla gösterir: fp16'nın aralığı dardır ($\max \approx 65504$), fp32'ye kıyasla küçüktür. Bu yüzden çok küçük bir gradyan (örn. $1e-8$) fp16'da 0'a yuvarlanır — **underflow**, kaybolur. GradScaler bunu kurtarır: gradyanı büyük bir çarpanla ($\times 4096$) ölçekler, böylece fp16 aralığına girer; optimizier adımından önce aynı çarpana geri böler. Sağ paneldeki log-ölçekli çubuklar, fp16'nın dar aralığını (hassasiyet ödünü) fp32 yanında görselleştirir.



Şekil 29.2: (gerçek hesaplama: fp16 küçük gradyanı ($1e-8$) 0'a yuvarlar (underflow); GradScaler $\times 4096$ ölçekleyip kurtarır, sonra geri böler)

💡 Builder Notu — Mixed Precision: Production’da Neredeyse Her Zaman Açık

- **İleriye (Ders 21):** Diffusion eğitimleri büyük ve yavaş; mixed precision onları kaldırılabilir kılar — production’da neredeyse her zaman açıktır.
- **Sezgi:** “Mixed” kelimesi bir dengeyi anlatır: hesabın ağır kısmı (matris çarpımları, evrişimler) fp16’da hızlanır, ama hassasiyet kritik birkaç adım (kayıp birikimi, ağırlık güncellemesi) fp32’de tutulur. Hız fp16’dan, kararlılık fp32’den gelir — ikisinin en iyisi.

29.3 2. MixedPrecision Callback (Sıfırdan)

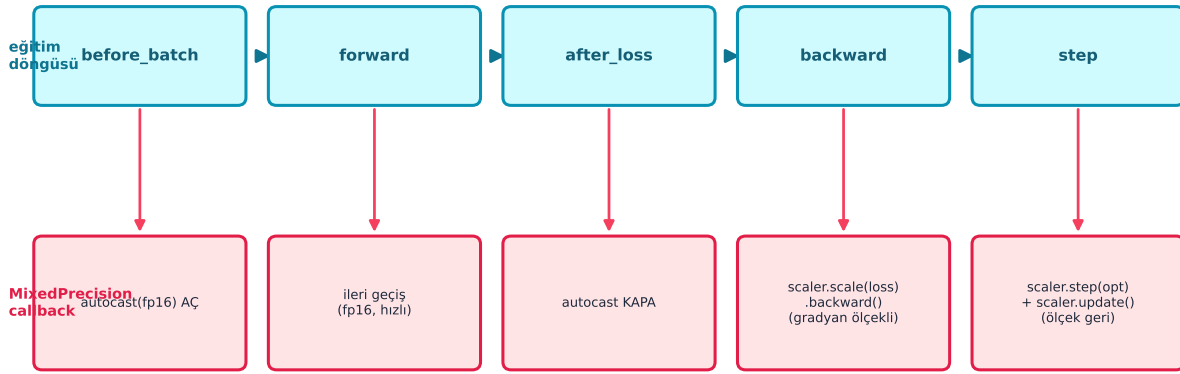
Mixed precision’in PyTorch karşılığı iki parçadır: autocast (ileri geçişi fp16’da çalıştırır) ve GradScaler (fp16’da kaybolan küçük gradyanları ölçekleyerek korur). miniai’de bunlar bir TrainCB alt sınıfında toplanır — her eğitim adımı kancasına (before_batch, after_loss, backward, step) tek bir callback müdahale eder.

```
class MixedPrecision(TrainCB):
    order = DeviceCB.order+10
    def before_fit(self, learn): self.scaler = torch.cuda.amp.GradScaler()
    def before_batch(self, learn):
        self.autocast = torch.autocast("cuda", dtype=torch.float16)
        self.autocast.__enter__()
    def after_loss(self, learn): self.autocast.__exit__(None, None, None)
    def backward(self, learn): self.scaler.scale(learn.loss).backward()
    def step(self, learn):
        self.scaler.step(learn.opt)
        self.scaler.update()
```

Şekil 29.3 callback’in eğitim döngüsüne nasıl oturduğunu şematik gösterir: üstte miniai eğitim döngüsünün kancaları (before_batch → forward → after_loss → backward → step), altta her kancada MixedPrecision’in eylemi — autocast aç (fp16), ileri geçiş, autocast kapa, scaler.scale(loss).backward() (gradyan ölçekli), scaler.step + update (ölçek geri). Mesaj net: fp16 hesap + GradScaler **tek bir callback’te** toplanır (Ders 16 sistemi), Accelerate aynısını çoklu GPU/TPU ile yapar.

💡 Builder Notu — Callback: Eğitim Adımının Her Parçasına Müdahale

- **Geriye (Ders 16):** Eğitim adımının her parçasına (before_batch, after_loss, backward, step) callback ile müdahale — Ders 16’nın callback sisteminin gücü tam burada. Çekirdek döngüye hiç dokunmadan, fp16 mantığı dışarıdan takılır.
- **Sezgi:** Dört kanca dört rol üstlenir: before_batch autocast’i açar (ileri geçiş fp16 olur), after_loss autocast’i kapatır (kayıp birikimi fp32’ye döner), backward gradyanı ölçekler (underflow’u önler), step ölçeği geri alıp adımı atar. Her biri tek satır — karmaşıklık callback sınırında kapsüllenmiştir.



fp16 hesap + GradScaler tek bir callback'te (Ders 16 sistemi); Accelerate aynı + çoklu GPU/TPU

Şekil 29.3: (şematik: MixedPrecision callback — autocast (fp16 ileri) + GradScaler (backward/step); miniai callback olarak eğitime tek satırla eklenir)

29.4 3. HuggingFace Accelerate

Aynı işi (ve dahasını — çoklu GPU, TPU) HuggingFace **Accelerate** kütüphanesi yapar. Tek bir Accelerator nesnesi modeli, optimizier'ı ve veri yükleyicileri sarar; `acc.prepare` ile hazırlanır, `acc.backward` ile geri yayılır. miniai'de yine bir callback (AccelerateCB).

```

from accelerate import Accelerator
class AccelerateCB(TrainCB):
    order = DeviceCB.order+10
    def __init__(self, n_inp=1, mixed_precision="fp16"):
        super().__init__(n_inp=n_inp)
        self.acc = Accelerator(mixed_precision=mixed_precision)
    def before_fit(self, learn):
        learn.model, learn.opt, learn.dls.train, learn.dls.valid = self.acc.prepare(
            learn.model, learn.opt, learn.dls.train, learn.dls.valid)
    def backward(self, learn): self.acc.backward(learn.loss)
  
```

“Accelerate is a library that provides this single Accelerator that does things to accelerate your training loops. And one of the things it does is mixed precision training.” — Howard, 7:50



Builder Notu — Accelerate: Production Standardı, Hâlâ Bir Callback

- **İleriye (production):** Accelerate çoklu GPU/TPU eğitimini tek API'ye indirir; production diffusion eğitiminin standart aracı. miniai'ye bir callback olarak girer — kuplaj yok.
- **Sezgi:** Accelerate'in zarafeti, mixed precision'ı tek bir detaydan ibaret kılmasıdır: `Accelerator(mixed_precision="fp16")` der, gerisini (autocast, GradScaler, çoklu cihaz dağıtımı) kütüphane halleder. miniai açısından bu yine yalnızca iki kanca özelleşmiş bir `TrainCB`'dir — `before_fit` (prepare) ve `backward` (`acc.backward`). Sıfırdan yazdığımız

MixedPrecision ile production-grade Accelerate, miniai'ye aynı kapıdan girer.

29.5 4. Sneaky Trick: GPU'yu Meşgul Tutmak

Howard küçük bir hile gösterir: küçük modellerde GPU çoğu zaman boş bekler (CPU veri hazırlarken). Bir DataLoader'ı tekrarlayan bir sarmalayıcıyla (MultiDL) batch'leri üst üste besleyerek GPU'yu meşgul tutmak, küçük işlerde belirgin hızlanma sağlar.

💡 Builder Notu — GPU Doygunluğu: Sık Göz Ardı Edilen Darboğaz

- **İleriye (production):** “GPU'yu aç tutma” (data pipeline darboğazını giderme) production performansının sık göz ardı edilen tarafıdır; profil çıkarmak değerlidir.
- **Sezgi:** Mixed precision hesabı hızlandırır, ama hesap GPU'ya hiç ulaşmıyorsa fayda etmez. Küçük modellerde darboğaz GPU değil, veri hazırlama (CPU) olabilir; GPU her batch'in gelmesini bekler. Önce nerede beklenildiğini ölçmek (profil), sonra doğru yeri optimize etmek mühendisliğin temel disiplini.

29.6 5. Style Transfer'a Giriş

İkinci konu (John): **style transfer** — klasik bir AI sanat tekniği. Bir “içerik” görüntüsünü (örn. bir fotoğraf) alıp, başka bir görüntünün (örn. bir tablo) **stiline** dönüştürmek. Mona Lisa'yı farklı sanatçıların fırça darbeleriyle yeniden çizmek gibi.

“we're going to try to change the style so that the style looks like the style of some other picture. And the way we're going to be doing that is by doing an optimization.” — John, 26:59

💡 Builder Notu — Style Transfer: Ders 9 Sezgisinin Saf Hâli

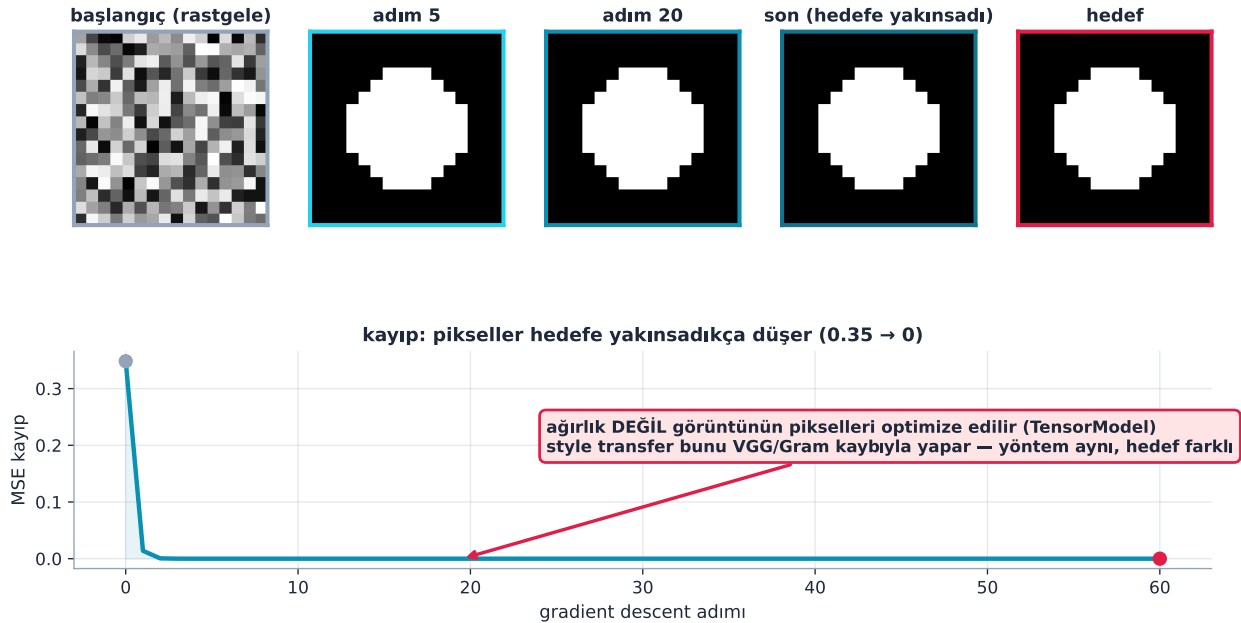
- **Geriye (Ders 9):** Ders 9'da diffusion'ı “piksel üzerinde gradient descent” sezgisiyle anlamıştık; style transfer aynı fikrin doğrudan, saf hâli. Diffusion'da o sezgi U-Net, scheduler, latent uzay katmanlarının altında gizliydi; burada hiçbir ara katman olmadan, doğrudan piksellere uygulanır.
- **Sezgi:** “Görüntünün stilini değiştir” gibi öznel bir hedefin matematiksel bir kayba çevrilebilmesi, dersin asıl numarasıdır — bu kayıp tanımlandıktan sonra gerisi tanıdık: gradient descent.

29.7 6. Görüntüyü Optimize Etmek (Ağırlık Değil)

Anahtar fikir: burada **ağırlıkları değil, görüntünün kendisini** optimize ederiz. Görüntü, öğrenilebilir bir parametre olan bir TensorModel'dir. Gradient descent her adımda pikselleri, kaybı azaltacak yönde günceller — yani çıktı doğrudan optimize edilen görüntüdür.

```
class TensorModel(nn.Module):
    def __init__(self, t):
        super().__init__()
        self.t = nn.Parameter(t.clone())
    def forward(self, x=0): return self.t
```

Şekil 29.4 bu çekirdeği **gerçek hesaplamayla** kanıtlar (**FLAGSHIP**): üst satırda pikseller rastgele gürültüden başlar (adım 0), gradient descent adımlarıyla (5, 20, 60) hedefe yakınsar, son kare hedefle neredeyse ayırt edilemez; altta MSE kaybı 0.35’ten 0’a düşer. Vurgu: optimize edilen şey **ağırlık değil, görüntünün pikselleridir**. Style transfer aynı mekanizmayı VGG/Gram kaybıyla yapar — yöntem aynı, hedef farklı.



Şekil 29.4: (gerçek hesaplamayla) optimize the input pikseller (ağırlık değil). Gradient descent ile hedefe yakınsar (loss 0.35 → 0); style transfer’ın TensorModel çekirdeği)

- **Geriye (Ders 3/5):** Gradient descent’i ağırlık öğrenmek için kullanmıştık; burada aynı mekanizma bir görüntüyü “öğreniyor”. Optimizasyon hedefi değişti, yöntem aynı. `nn.Parameter` neyi sararsa onu optimize edersin — bu satır dersin tüm derinliğinin kaynağıdır.
- **Sezgi:** `TensorModel`’in `forward`’ı girdisini yok sayıp parametresini geri döndürür — yani “model” aslında optimize edilen görüntünün ta kendisidir. Geri yayılım kaybın gradyanını bu piksellere taşır, optimizasyon onları günceller. Normal eğitimde veri sabit/ağırlık değişkendi; burada ağırlık (VGG16) sabit/görüntü değişken. Mekanizma birebir aynı, roller takas edildi.

29.8 7. Optimizasyon Döngüsü: Dummy Learner

Görüntüyü optimize etmek için miniai Learner kullanılır — ama veri yok (optimize edilen tek şey görüntü). Bu yüzden uzunluğu sabit bir “kukla” DataLoader (LengthDataset) kurulur; her “batch” aslında aynı görüntü üzerinde bir optimizasyon adımdır. İlerlemeyi ImageLogCB görselleştirir.

```
class LengthDataset():
    def __init__(self, length=1): self.length=length
    def __len__(self): return self.length
    def __getitem__(self, idx): return 0,0
```

💡 Builder Notu — Dummy Learner: miniai Verisiz Bile Çalışır

- **Geriye (Ders 16):** miniai Learner o kadar esnek ki, “verisiz” bir optimizasyon problemine bile uyarlanabiliyor — callback sistemi ve gevşek bağlılık sayesinde. Eğitim döngüsünün hiçbir parçasını değiştirmeden, sadece veriyi “boş” kılarırsın.
- **Sezgi:** LengthDataset her `__getitem__` çağrısında `(0, 0)` döndürür — gerçek bir veri yok, sadece döngünün dönmesi için bir sayaç. miniai açısından bu sıradan bir eğitim; ama her “batch” aynı görüntü üzerinde bir gradient descent adımdır. Çekirdek sade tutulduğu için, “verisiz optimizasyon” bile özel bir kod gerektirmez.

29.9 8. VGG16 ile Özellik Çıkarma

İçeriği ve stili ölçmek için ham pikseller yetmez; **anlamsal özellikler** gerekir. Önceden eğitilmiş bir **VGG16** ağının ara katman aktivasyonları kullanılır: alt katmanlar kenar/doku, üst katmanlar nesne/yapı yakalar. `calc_features` istenen katmanların çıktısını toplar.

```
vgg16 = timm.create_model('vgg16', pretrained=True).to(def_device).features
def calc_features(imgs, target_layers=(18, 25)):
    x = normalize(imgs)
    feats = []
    for i, layer in enumerate(vgg16[:max(target_layers)+1]):
        x = layer(x)
        if i in target_layers: feats.append(x.clone())
    return feats
```

💡 Builder Notu — VGG16 Özellikleri: Ders 17 Hook Fikri

- **Geriye (Ders 17):** Ara katman aktivasyonlarını yakalamak = Ders 17’nin hook fikri; burada doğrudan katmanları gezerek aynı şeyi yaparız. Hook’la ağa “dışarıdan dinleme cihazı” takmak yerine, katmanları tek tek dolaşıp istediklerimizi toplayan basit bir döngü kullanılır — aynı amaç, daha açık kod.
- **Sezgi:** VGG16’nın katman derinliği bir soyutlama merdivenidir: alt katmanlar piksel-yakın (kenar, doku, renk geçişi), üst katmanlar anlam-yakın (nesne parçası, yapı). Bu yüzden içerik (yapı) üst katmanlarda, stil (doku) hem alt hem üst katmanlarda ölçülür — `target_layers` seçimi neyi

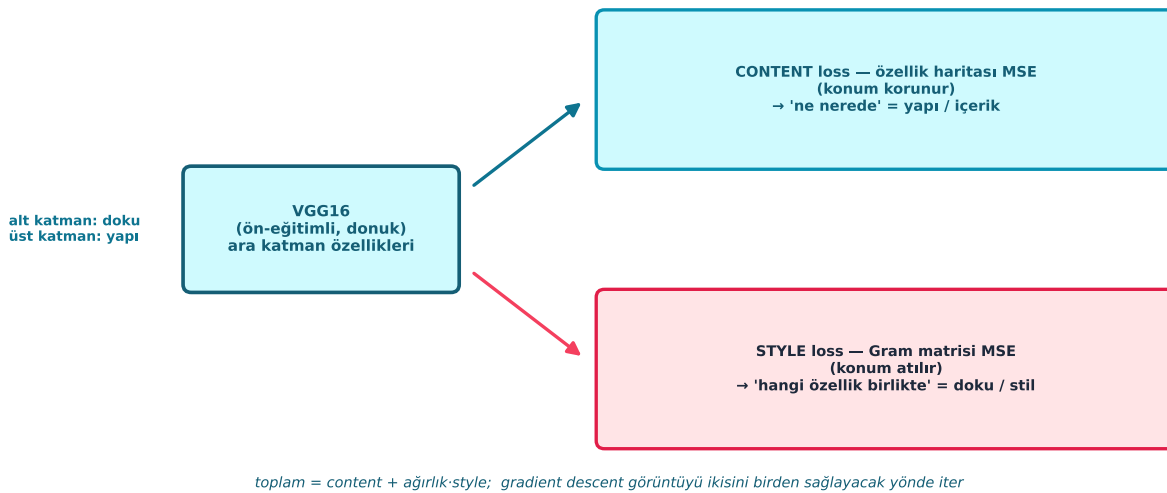
ölçtüğünü belirler.

29.10 9. Content Loss: İçeriği Korumak

İçerik kaybı, optimize edilen görüntünün VGG16 özelliklerinin, içerik görüntüsünün özelliklerine yakın olmasını sağlar — basit bir **MSE**: ara katman aktivasyon haritaları arasındaki kare fark. Böylece çıktı, içerik görüntüsünün yapısını (ne olduğunu) korur.

```
class ContentLossToTarget():
    def __init__(self, target_im, target_layers=(18, 25)):
        with torch.no_grad():
            self.target_features = calc_features(target_im, target_layers)
    def __call__(self, input_im):
        return sum((f1-f2).pow(2).mean() for f1, f2 in
            zip(calc_features(input_im, self.target_layers), self.target_features))
```

Şekil 29.5 content ve style kaybının farkını şematik gösterir: ortada ön-eğitilmiş (donuk) VGG16 ara katman özellikleri, iki dala ayrılır. Üst dal **content loss** — özellik haritası MSE, konum korunur, “ne nerede” = yapı/içerik. Alt dal **style loss** — Gram matrisi MSE, konum atılır, “hangi özellik birlikte” = doku/stil. Toplam = content + ağırlık-style; gradient descent görüntüyü ikisini birden sağlayacak yönde iter.



Şekil 29.5: (şematik: content loss = özellik MSE (yapı korunur), style loss = Gram MSE (doku, konumsuz); ikisinin dengesi görüntüyü yeniden çizer)

💡 Builder Notu — Content Loss: Yine MSE, Ama Özellik Uzayında

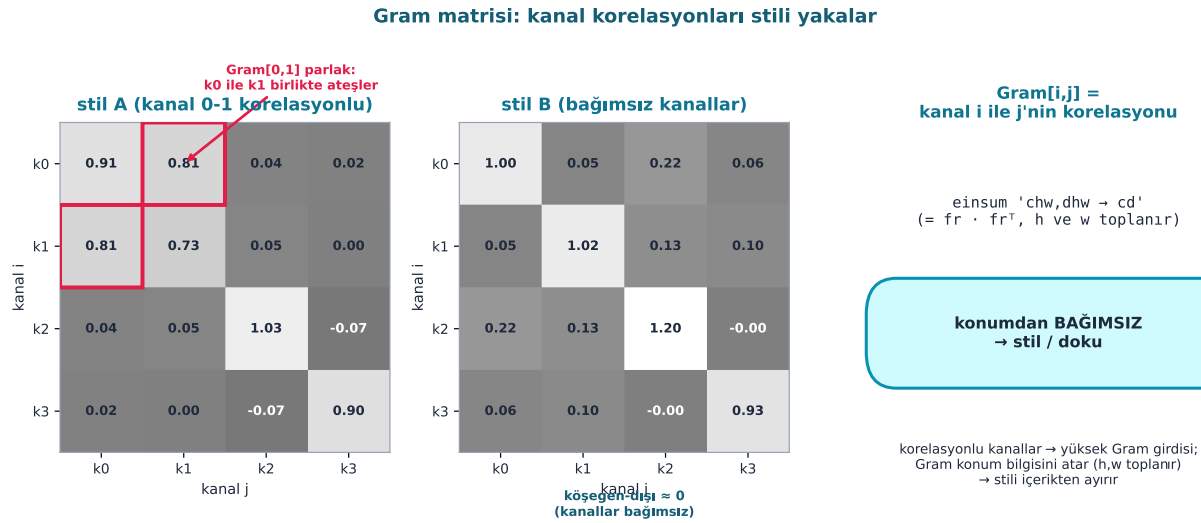
- **Geriye (Ders 5):** Yine MSE — ama piksellerde değil, anlamsal özelliklerde. “Ne” karşılaştırması özellik uzayında yapılır. Piksel MSE’si iki resmi “aynı yerde aynı renk mi?” diye karşılaştırırdı; özellik MSE’si “aynı nesneler, aynı yapı var mı?” diye karşılaştırır.
- **Sezgi:** İçeriği piksel düzeyinde zorlamak çıktıyı içerik görüntüsünün birebir kopyasına hapsederdi — stil değişmezdi. Özellik uzayında ölçmek esneklik tanır: “köpek burada, gökyüzü orada” korunur (üst katman aktivasyonları benzer), ama her pikselin tam rengi serbest kalır — böylece stil, içeriği bozmadan değişebilir.

29.11 10. Gram Matrisi: Stili Ölçmek

Stil, “nerede ne var” değil, “hangi özellikler birlikte görülüyor” sorusudur. Bunu **Gram matrisi** yakalar: bir katmanın özellik kanallarının birbirleriyle korelasyonu (her kanal çiftinin iç çarpımı). Konumdan bağımsız olduğu için stili (doku, renk, fırça darbesi) içerikten ayırır.

```
def calc_grams(img, target_layers=(1, 6, 11, 18, 25)):
    return L(torch.einsum('chw, dhw -> cd', x, x) / (x.shape[-2]*x.shape[-1]))
    for x in calc_features(img, target_layers))
```

Şekil 29.6 Gram matrisini **gerçek hesaplamayla** gösterir: solda korelasyonlu kanalları olan bir stilin Gram matrisi — kanal 0 ile kanal 1 birlikte ateşlediği için $\text{Gram}[0,1]$ parlak; ortada bağımsız kanalların matrisi — köşegen-dışı ≈ 0 ; sağda açıklama: $\text{Gram}[i,j]$ = kanal i ile j’nin korelasyonu (einsum 'chw,dhw $\rightarrow$ cd', h ve w toplanır). Gram konum bilgisini atar (h, w üzerinden toplanır), bu yüzden stili içerikten ayırır.



Şekil 29.6: (gerçek hesaplama: Gram matrisi = kanal korelasyonu (einsum chw,dhw $\rightarrow$ cd); korelasyonlu kanallar parlak, bağımsız köşegen — konumdan bağımsız stil ölçüsü)

 Builder Notu — Gram Matrisi: Ders 11-12 einsum’un Zarif Uygulaması

- **Geriye (Ders 12/11):** `einsum('chw,dhw->cd')` = kanallar arası iç çarpım (özellik korelasyon matrisi); Ders 11-12’de kurduğumuz `einsum/matmul`’un zarif bir uygulaması. Aynı `einsum` gramerini orada matris çarpımı için kullanmıştık; burada kanal korelasyonu için.
- **Sezgi:** Gram matrisinin konumdan bağımsız olmasının sırrı `einsum`’un indekslerinde gizli: `'chw,dhw->cd'`’de `h` ve `w` (uzamsal boyutlar) çıktıda yok — yani üzerlerinden toplanıp atılırlar. Geriye sadece kanal-kanal (`cd`) ilişkisi kalır. Bu yüzden bir fırça darbesi resmin neresinde olursa olsun, aynı kanal korelasyonunu üretir — stil “nerede”den bağımsızdır.

29.12 11. Style Loss ve Birleşim

Stil kaybı, optimize edilen görüntünün Gram matrislerinin, stil görüntüsünün Gram matrislerine yakın olmasıdır (yine MSE). Toplam kayıp = içerik kaybı + stil kaybı (ağırlıklı). Gradient descent pikselleri ikisini birden azaltacak yönde iter: içeriği koru, stili eşleştir.

```
class StyleLossToTarget():
    def __init__(self, target_im, target_layers=(1, 6, 11, 18, 25)):
        with torch.no_grad(): self.target_grams = calc_grams(target_im, target_layers)
    def __call__(self, input_im):
        return sum((f1-f2).pow(2).mean() for f1, f2 in
                    zip(calc_grams(input_im, self.target_layers), self.target_grams))
```

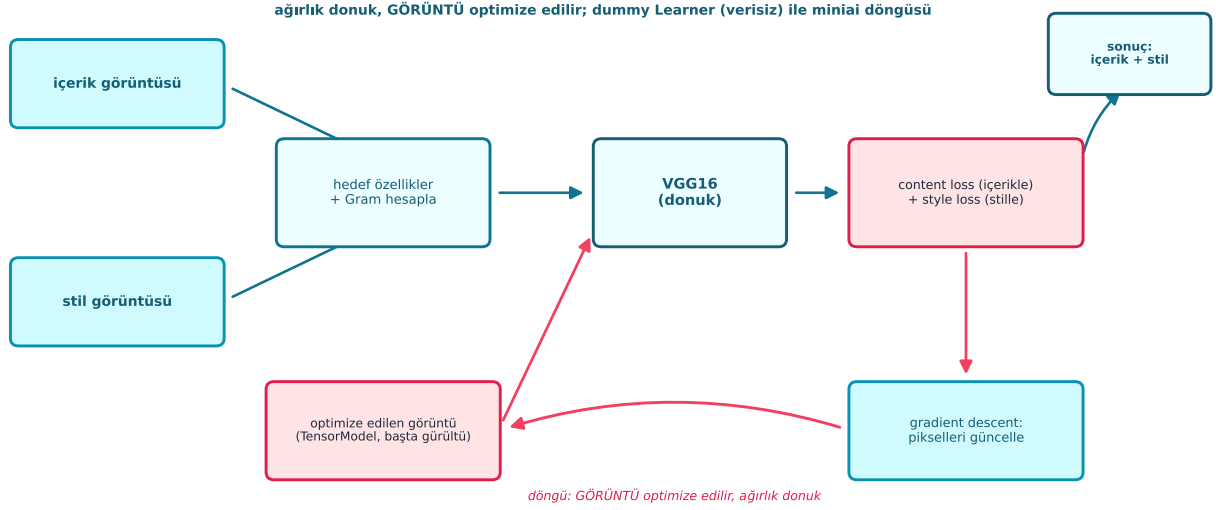
Şekil 29.7 tüm style transfer hattını şematik gösterir: solda içerik ve stil görüntüsünden hedef özellikler + Gram hesaplanır; ortada optimize edilen görüntü (TensorModel, başta gürültü) donuk VGG16’dan geçer; sağda content + style loss üretilir; gradient descent pikselleri günceller ve döngü geri besler. Anahtar mesaj: **ağırlık donuk, görüntü optimize edilir**; tüm bu hat, verisiz bir dummy Learner ile miniai döngüsü üstünde çalışır.

 Builder Notu — Birleşim: Loss Tasarımı Üretken Modellemenin Sanatı

- **İleriye:** İçerik + stil kaybının dengesi (ağırlıklar) sonucu belirler; “loss tasarımı” üretken modellemenin sanatıdır (diffusion guidance’ta tekrar görülür). Stil ağırlığı çok yüksekse içerik kaybolur (sadece doku), çok düşükse stil görünmez (sadece içerik).
- **Sezgi:** İki kaybın aynı pikseller üzerinde çekişmesi, sonucu belirler: content loss pikselleri “yapıyı koru” yönüne, style loss “dokuyu eşleştir” yönüne iter. Optimizer ikisinin ağırlıklı toplamını azaltır — yani dengede, içeriği tanınabilir ama stili yeni bir görüntü ortaya çıkar. Bu denge bir hiperparametredir; tasarımcının elindedir.

29.13 12. Builder İçgörüsü: Optimizasyon Hedefi Değişebilir

Bu dersin en derin builder dersi: gradient descent yalnızca model ağırlıkları için değildir. Aynı mekanizmayla bir **girdiyi** (görüntüyü), bir **gömüyü** (embedding), hatta bir **prompt**’u optimize edebilirsin. Style transfer,



Şekil 29.7: (şematik: style transfer hattı — TensorModel görüntüyü optimize eder, VGG16 content+style loss verir, gradient descent pikselleri içerik+stil birleşimine çeker)

“kayıbı tanımla, optimize edilecek şeyi parametre yap, gerisini gradient descent halletsin” esnekliğinin saf göstereimidir.

! Builder Notu — Optimize the Input: Dersin Kalbi (MERKEZİ)

- **İleriye (Ders 9A guidance):** Diffusion’da “classifier-free guidance” ve custom guidance da aynı fikir — örnekleme sırasında bir kaybı optimize ederek üretimi yönlendirmek. Style transfer’da görüntüyü, guidance’ta diffusion ara durumunu optimize edersin; mekanizma birebir aynıdır.
- **Çekirdek ders:** Normal eğitim “veriyi sabit tut, ağırlığı öğren” derdi. Bu ders tersini gösterdi: “ağırlığı sabit tut (VGG16 donuk), girdiyi öğren”. Genelleme şudur — nn . Parameter neyi sararsa, gradient descent onu optimize eder. Hedef ağırlık, görüntü, embedding, prompt ya da bir diffusion ara durumu olabilir. Yöntem değişmez.
- **Sezgi:** Bu içgörünün gücü, modern üretken yaratıcılığın çoğunu tek bir reçeteye indirmesidir: textual inversion bir embedding’i, DreamBooth birkaç ağırlığı, custom guidance bir ara görüntüyü optimize eder — hepsi “kayıbı tanımla → hedefi parametre yap → optimize et”. Bir kez bu çerçeveyi gördüğünde, yeni teknikleri “neyi optimize ediyorlar?” sorusuyla anında çözebilirsin.

29.14 13. miniai Köprüsü

Her iki konu da miniai üstünde çalışır. Mixed precision bir callback (MixedPrecision/AccelerateCB); style transfer standart bir Learner + özel kayıp + TensorModel. Foundations’ta kurduğumuz esnek altyapı, birbirinden çok farklı iki problemi (eğitim hızlandırma, görüntü optimizasyonu) aynı çatı altında çözüyor.

**Builder Notu — miniai: Çekirdeği Sade Tut, Davranışı Callback ile Genişlet**

- **Geriye (Ders 16):** “Çekirdeği sade tut, davranışı callback ile genişlet” ilkesinin meyvesi: mixed precision = bir callback, style transfer = Learner’ın farklı bir kullanımı. İki tamamen farklı problem, aynı sade çekirdeğe bağlanır.
- **Sezgi:** Bu derste miniai’ye eklenen iki yetenek de çekirdek döngüye tek satır dokunmaz: mixed precision bir TrainCB alt sınıfıyla (callback), style transfer ise verinin “boş”, modelin TensorModel, kaybın özel olduğu sıradan bir fit çağrısıyla girer. L16’da callback mimarisini bu kadar özenle kurmanın ödülü tam burada görünür.

29.15 14. NYU Köprüsü: Özellik Uzayında Çalışmak

Style transfer’ın özünde, önceden eğitilmiş bir ağıın **özellik temsillerini** kullanmak var — NYU Deep Learning’in (LeCun) merkezî temalarından biri. İçerik (özellik aktivasyonları) ve stil (Gram korelasyonları), bir ağıın öğrendiği temsilin ne kadar zengin olduğunu gösterir. Bu, transfer learning ve self-supervised temsil öğrenmeyle (NYU) aynı sezgiye dayanır.

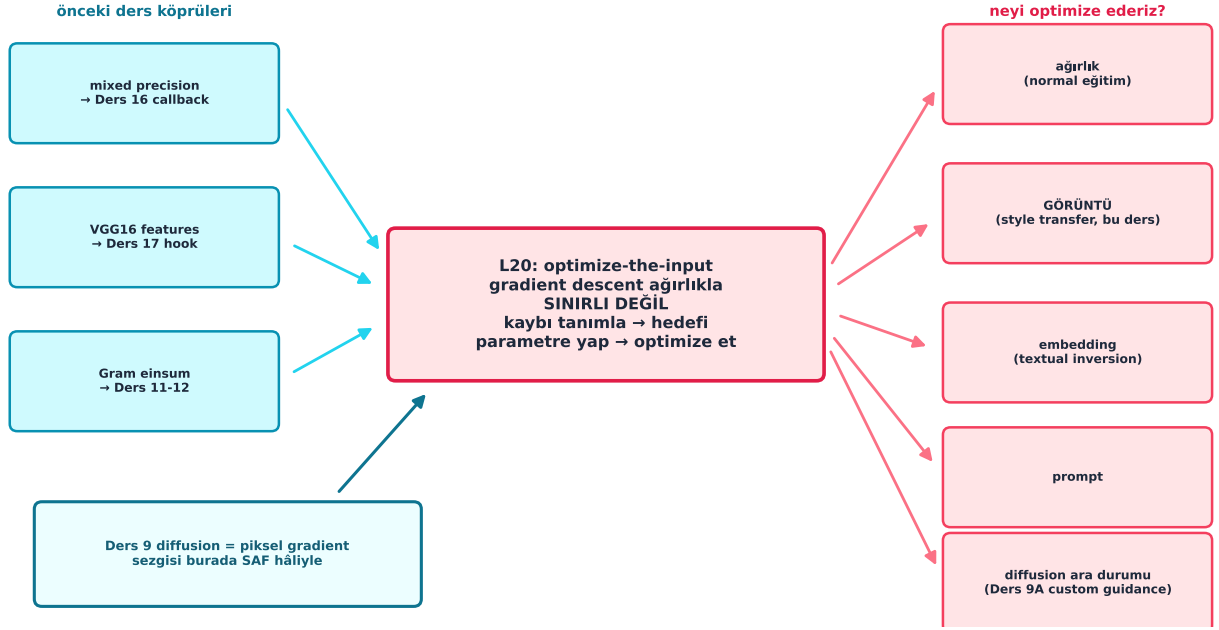
**Builder Notu — NYU: Önceden Eğitilmiş Özelliklerin Yaratıcı Kullanımı**

- **Geriye (NYU):** Önceden eğitilmiş özelliklerin yeniden kullanımı; style transfer bunun yaratıcı bir uygulamasıdır. NYU’da bir ağıın iç temsillerinin transfer learning ve SSL için ne kadar değerli olduğunu görmüştük.
- **Sezgi:** VGG16 sınıflandırma için eğitilmişti, style transfer’ı hiç bilmez. Ama öğrendiği ara temsiller o kadar genel ki, hiç amaçlanmamış bir görevde (içerik/stil ayrımı) doğrudan işe yarar. Bu, “iyi bir temsil görevden bağımsız zengindir” NYU sezgisinin somut kanıtıdır — aynı özellikler hem sınıflandırma hem sanat için kullanılabilir.

29.16 15. Kapanış

Ders 20 iki pratik araç verdi: **mixed precision** (fp16 ile hızlı, az bellekli eğitim — bir callback) ve **style transfer** (görüntüyü optimize etme, VGG16 özellikleri + Gram matrisi). İkincisinin builder dersi büyük: gradient descent ağırlıklarla sınırlı değil; kaybı tanımla, hedefi parametre yap, optimize et. miniai her iki problemi de zarifçe kaldırdı.

Şekil 29.8 dersin sentezidir: merkezde çekirdek fikir — **optimize-the-input**, gradient descent ağırlıkla sınırlı değil (kayıp tanımla → hedefi parametre yap → optimize et); sağda “neyi optimize ederiz?” örnekleri (ağırlık, görüntü/style transfer, embedding/textual inversion, prompt, diffusion ara durumu/9A custom guidance); solda önceki ders köprüleri (mixed precision → L16 callback, VGG16 → L17 hook, Gram einsum → L11-12); alt-solda Ders 9 diffusion = aynı fikrin saf hâli. Sentez: **L20 = optimize-the-input genelliği + mixed precision hızlandırma, ikisi de miniai üstünde.**



Şekil 29.8: (sentez: L20 = optimize-the-input genelliği — gradient descent ağırlık/görüntü/embedding/prompt’a uygulanır; Ders 9 diffusion sezgisinin saf hâli + Ders 11/16/17 köprüleri)



Builder Notu — Kapanış: Gradient Descent Bir Araç, Neye Uygularsan O

- **İleriye (Ders 21):** Mixed precision artık elimizde; Ders 21 ağır diffusion eğitimlerini W&B ile izler, FID/KID ile ölçer ve DDIM ile hızlandırır.
- **Sezgi:** Bu dersten sonra “gradient descent neyi öğrenir?” sorusunun cevabı netleşti: ne `nn.Parameter` ile sararsan onu. Ağırlıklar yalnızca en sık karşılaşılan hedef; görüntü, embedding, prompt da aynı mekanizmayla optimize edilir. Bu genellik, diffusion guidance’tan textual inversion’a modern üretken yaratıcılığın temelidir.

29.17 Bu Dersin Özeti

1. **Mixed precision:** Hesabı fp16’da yap (hız + bellek); hassas kısımlar fp32’de — “mixed” ([mixed precision](#)).
2. **MixedPrecision callback:** autocast (fp16 ileri geçiş) + GradScaler (küçük gradyanları koru); miniai TrainCB ([callback](#)).
3. **Accelerate:** HuggingFace tek API’si — çoklu GPU/TPU + mixed precision; `acc.prepare` + `acc.backward` ([Accelerate](#)).
4. **Style transfer:** Bir görüntüyü, başka bir resmin stiline dönüştürme; optimizasyonla ([giriş](#)).
5. **Görüntüyü optimize etme:** Model bir `TensorModel` (öğrenilen ağırlık değil, piksellerdir); gradient descent pikselleri günceller ([optimize the input](#)).
6. **VGG16 özellikleri:** İçerik, ara katman aktivasyonlarıyla; alt katman doku, üst katman yapısı ([VGG16](#)).
7. **Content loss / Style loss:** İçerik = özellik MSE; stil = Gram matrisi (kanal korelasyonları) MSE ([content](#), [Gram](#)).

8. **Builder dersi:** Gradient descent ağırlıklarla sınırlı değil — girdi, gömü, prompt da optimize edilebilir (**builder içgörüsü**).

! Tek Bir Cümle

Mixed precision fp16 ile eğitimi hızlandırıp belleği azaltır (autocast + GradScaler, bir callback); style transfer ise gradient descent'i ağırlıklar yerine bir görüntü üzerinde çalıştırır — VGG16 ara katman özellikleri içeriği, Gram matrisi (kanal korelasyonları) ise stili ölçer, ikisinin MSE kaybı pikselleri yeni bir resme dönüştürür.

29.18 Kontrol Soruları

i Soru 1: Mixed precision nedir ve neden GradScaler gerekir?

Cevap:

Mixed precision, eğitim hesabının çoğunu fp16 (16-bit kayan nokta) ile yapmaktır — modern GPU'larda çok daha hızlı ve daha az bellek kullanır. "Mixed" denmesinin sebebi, hassasiyet gerektiren bazı işlemlerin (örn. kayıp ve ağırlık güncellemeleri) fp32'de tutulmasıdır. Sorun: fp16'nın dar aralığında, çok küçük gradyanlar sıfıra yuvarlanıp (underflow) kaybolabilir. **GradScaler** bunu çözer: geri yayılımdan önce kaybı büyük bir çarpanla ölçekler (böylece gradyanlar fp16 aralığında kalır), optimizer adımından önce gradyanları geri ölçekler. Yani `scaler.scale(loss).backward() + scaler.step(opt) + scaler.update()`. autocast ileri geçişi fp16'da çalıştırır; GradScaler gradyanların hayatta kalmasını sağlar. (Şekil 29.2 küçük gradyanın underflow'unu ve GradScaler'in kurtarışını gösterir.)

i Soru 2: Style transfer'da ne optimize edilir? Bu, normal eğitimden nasıl farklıdır?

Cevap:

Normal eğitimde **model ağırlıkları** optimize edilir (veri sabit, ağırlık değişir). Style transfer'da tam tersi: **görüntünün pikselleri** optimize edilir, ağırlıklar (VGG16) donuktur. Model bir `TensorModel`'dir — tek parametresi optimize edilen görüntüdür (`nn.Parameter`). Her gradient descent adımı, kaybı (içerik + stil) azaltacak yönde pikselleri günceller. Yani çıktı, eğitilmiş bir ağ değil, doğrudan optimize edilmiş görüntünün kendisidir. Bu, gradient descent'in genelliğini gösterir: "neyi parametre yaparsan onu öğrenir". Aynı mekanizma (geri yayılım + optimizer) hedefi değiştirir — ağırlık yerine görüntü. (Şekil 29.4 piksellerin görüntüden hedefe yakınsamasını gerçek hesaplamayla gösterir.)

i Soru 3: İçerik ve stil neden farklı şekilde (özellik vs Gram matrisi) ölçülür?

Cevap:

İçerik "ne nerede" sorusudur — görüntünün yapısı, nesneleri, kompozisyonu. Bu, VGG16'nın ara katman **aktivasyon haritalarıyla** doğrudan ölçülür (konum bilgisi korunur): iki görüntünün özellik haritaları benzerse içerikleri benzerdir. **Stil** ise "ne nerede"den bağımsızdır — doku, renk paleti, fırça darbesi her yere yayılmıştır. Bunu **Gram matrisi** yakalar: bir katmanın özellik kanallarının birbirleriyle korelasyonu (`einsum('chw,dhw->cd')`). Gram matrisi konumu yok sayar (h, w boyutları üzerinden toplanır), sadece "hangi özellikler birlikte aktive oluyor"u tutar. Bu yüzden stili içerikten ayırabilir:

içerik kaybı yapıyı korur, stil kaybı (Gram) dokuyu eşleştirir. (Şekil 29.6 korelasyonlu kanalların parlak Gram girdisi ürettiğini, Şekil 29.5 iki kaybın farkını gösterir.)

i Soru 4: Bu ders gradient descent hakkında hangi derin builder dersini verir?

Cevap:

Gradient descent **ağırlık öğrenmekle sınırlı değildir** — bir kayıp fonksiyonu tanımlayıp optimize edilecek şeyi bir `nn.Parameter` yaptığın sürece, *herhangi bir şeyi* optimize edebilirsin. Style transfer’da bu bir görüntüdür (pikseller). Ama aynı fikirle bir gömü (embedding), bir prompt, ya da diffusion örneklemede bir ara durum optimize edilebilir. Builder içgörüsü: “kayıp tanımla → hedefi parametre yap → gerisini gradient descent halletsin”. Bu genellik, Ders 9’daki diffusion sezgisinden (piksel üzerinde gradient descent), Ders 9A’daki custom guidance’a, modern textual inversion/DreamBooth gibi tekniklere kadar uzanır. Gradient descent bir araçtır; onu neye uyguladığın sana kalmış. (Şekil 29.8 bu genelliği — ağırlık/görüntü/embedding/prompt — sentezler.)

29.19 Egzersizler

Egzersiz 1 (Direkt uygulama). Bir modeli `MixedPrecision` (veya `AccelerateCB`) callback’iyle eğit; eğitim süresini ve GPU bellek kullanımını `fp32` ile karşılaştır (§2, §3).

Egzersiz 2 (İki-aşamalı). `TensorModel + ContentLossToTarget` ile sadece içerik kaybını optimize et; çıktının içerik görüntüsüne yakınsadığını gözle (Şekil 29.4’in yakınsamasını üret — §6, §9).

Egzersiz 3 (Edge case). Style loss ağırlığını çok yüksek/çok düşük yap; içerik-stil dengesinin sonucu nasıl değiştirdiğini gör (§11).

Egzersiz 4 (Kavramsal). Gram matrisi neden konumdan bağımsızdır? `einsum('chw,dhw->cd')`’de `h` ve `w` boyutlarına ne olur? (§10)

Egzersiz 5 (Sonraki dersin habercisi — Ders 21). DDPM örnekleri ürettik ama “iyi mi” nasıl ölçeriz? Üretilen görüntülerin kalitesini sayısal olarak ölçmenin (gerçek dağılıma yakınlık) nasıl mümkün olduğunu düşün (§15).

29.20 Sonraki: Ders 21 İçin Hazırlık

Ders 21: Deney Takibi, FID/KID ve DDIM

Ders 20 mixed precision ve style transfer verdi. Ders 21 üç konu işler: **deney takibi** (Weights & Biases callback), üretilen görüntü kalitesini ölçen **FID/KID** metrikleri (Fréchet/Kernel Inception Distance — Inception özellikleri + kovaryans + matris karekökü) ve **DDIM** (DDPM’i çok daha az adımda, deterministik örnekleyen hızlandırma).

Ana konular (Ders 21):

- W&B ile deney takibi (callback)
- FID/KID: üretilen görüntü kalitesi metriği
- Inception ağı + kovaryans + Fréchet mesafesi

- DDIM: hızlı, deterministik örnekleme



Ders 21 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 2 — mixed precision + content loss).
- DDPM modelini mixed precision ile eğit, hızlanmayı gözle.
- Ders 19 sampling döngüsünü tekrar oku (DDIM onu hızlandıracak).

29.21 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Mixed precision	fp16 hesap (hız + bellek); hassas kısım fp32	0:06
autocast	İleri geçişi fp16'da çalıştıran context manager	3:17
GradScaler	fp16'da kaybolan küçük gradyanları ölçekleyip korur	3:23
Accelerate	HuggingFace çoklu GPU/TPU + mixed precision API'si	7:34
AccelerateCB	acc.prepare + acc.backward; miniai callback	11:05
Style transfer	Görüntüyü başka resmin stiline dönüştürme	20:38
TensorModel	Optimize edilen görüntü (ağırlık değil, piksel)	24:25
VGG16 features	Ön-eğitilmiş ağırlık ara katman aktivasyonları	35:04
Content loss	Özellik haritası MSE; içeriği (yapıyı) korur	47:24
Gram matrisi	Kanal korelasyonları (einsum chw,dhw→cd); stili ölçer	56:08
Style loss	Gram matrisi MSE; dokuyu/stili eşleştirir	56:08
Optimize the input	Gradient descent ağırlık yerine girdiye uygulanır	26:59

29.22 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: Hızlandırma + Optimizasyon Genelliği

Bu ders iki pratik aracı (mixed precision, style transfer) önceki derslerin altyapısına bağlar ve “optimize the input” genelliğini ortaya koyar; köprülerin özeti:

1. **Mixed precision** → Ders 16 callback; eğitime tek satırla eklenir (miniai) ([callback](#)).
2. **VGG16 features** → Ders 17 hook fikri; ara katman aktivasyonları yakalama ([VGG16](#)).
3. **Gram matrisi (einsum)** → Ders 11-12 einsum/matmul; kanal korelasyon matrisi ([Gram](#)).
4. **Content/style loss** → Ders 5 MSE; özellik uzayında karşılaştırma ([content](#), [style](#)).
5. **Optimize the input** → Ders 9 diffusion (piksel gradient) + 9A custom guidance; gradient descent’in genelliği ([builder içgörüsü](#)).
6. **Ön-eğitilmiş özellikler** → NYU temsil öğrenme; transfer learning’in yaratıcı kullanımı ([NYU köprü](#)).

❗ Bu dersten tek bir şey alıp gideceksen

Bu dersten iki şey alıp gideceksin. Birincisi pratik: **mixed precision** (fp16 + autocast + GradScaler, bir callback) eğitimi hızlandırır ve belleği azaltır — ağır diffusion işleri için vazgeçilmez. İkincisi kavramsal ve daha derin: **gradient descent ağırlıklarla sınırlı değil**. Style transfer’da bir görüntünün piksellerini optimize ettik; VGG16 özellikleri içeriği, Gram matrisi stili ölçtü. “Kaybı tanımla, hedefi parametre yap, optimize et” — bu esneklik, diffusion guidance’tan textual inversion’a tüm modern üretken yaratıcılığın temelidir.

30 Ders 21 — Deney Takibi, FID/KID ve DDIM (Experiment Tracking, FID/KID and DDIM)

ETAP 6'nın (Sıfırdan Diffusion A) üç pratik aracı, tek dersin altında — Howard, Johno ve Tanishq'le. Önce deney takibi: büyük modelleri eğitirken neyin işe yaradığını sistematik izlemek, miniai'ye bir Weights & Biases callback'i (WandBCB) olarak. Sonra FID ve KID: üretilen görüntülerin gerçek görüntülere ne kadar benzediğini gözle değil sayısal ölçmek — bir Inception ağının özellik istatistikleriyle (ortalama ve kovaryans, Fréchet mesafesi; ya da polinom çekirdekli MMD). Son olarak DDIM: DDPM örneklemesini 1000 adımdan elli civarına indiren, eta ile deterministik olabilen hızlandırma. Dersin builder dersi şu: üretkeni ölçmek demek, üretilen dağılımı gerçek dağılımla anlamsal özellik uzayında karşılaştırmak demektir — sınıflandırmadaki doğruluk gibi tek bir doğru cevap yoktur, dağılım yakınlığı vardır. Tek cümleyle: deney takibi neyin işe yaradığını izler, FID ve KID üretilen görüntü kalitesini gerçek dağılıma yakınlıkla ölçer, DDIM ise diffusion örneklemesini adım atlayarak ve eta ile deterministik yaparak hızlandırır.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 21: Experiment tracking, FID & DDIM](https://course.fast.ai/lesson-21-experiment-tracking-fid-ddim) (~116 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 21
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/21_cifar10_and_wandb](#) + [nbs/18_fid](#) + [nbs/20_DDIM](#)
- **Okuma süresi:** ~42 dk
- **Hocalar:** Howard + Jonathan Whitaker (Johno) + Tanishq Abraham
- **🔗 ETAP 6 (Sıfırdan Diffusion A):** Ders 19'da DDPM'i sıfırdan kurduk; diffusion artık kara kutu değil — ama iki açık sorun kaldı. Birincisi: örnekleme yavaş (1000 adım, her biri bir U-Net çağırısı). İkincisi: kaliteyi gözle değerlendirdik (öznel, ölçeklenmez). Ders 21 ikisini de çözer. **FID/KID** kaliteyi sayısal ölçer, **DDIM** örneklemeyi hızlandırır; **deney takibi** ise ağır eğitimlerde neyin işe yaradığını izler.

30.1 Bu Derste Ne Var?

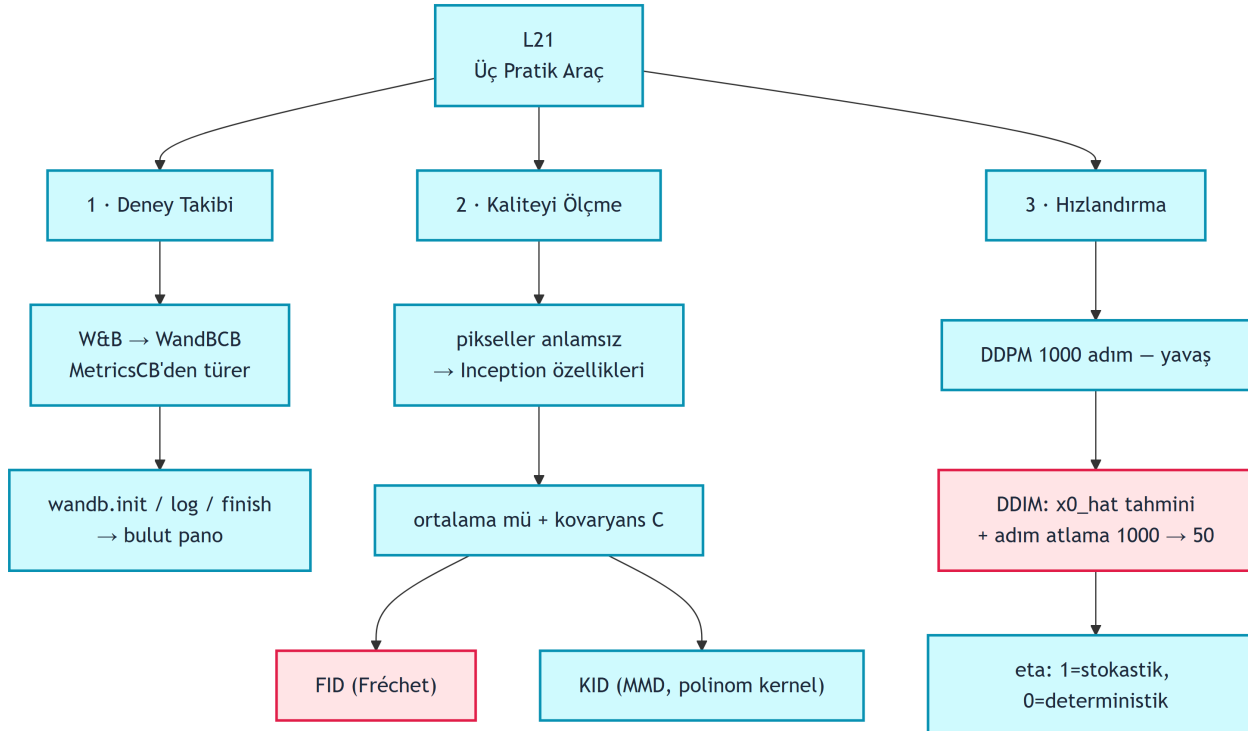
Üretken modelciliğin üç pratik aracı, tek dersin altında. Önce **deney takibi** (Weights & Biases) — büyük modelleri eğitirken neyin işe yaradığını sistematik izlemek. Sonra **FID/KID** — üretilen görüntülerin “ne kadar iyi” olduğunu *sayısal* ölçmek (gözle bakmak yetmez). Son olarak **DDIM** — DDPM örneklemesini 1000 adımdan ~50 adıma indiren, deterministik olabilen hızlandırma.

Üç temel fikir bu dersin omurgasını kurar:

1. **Deney takibi (W&B)** — eğitim metriklerini, hiperparametreleri ve örnekleri otomatik kaydetmek; bir callback ile (deney takibi → WandBCB).
2. **FID/KID** — üretilen görüntü kümesinin gerçek görüntülere yakınlığını, bir Inception ağının özellik istatistikleriyle (ortalama + kovaryans) ölçmek (FID, KID).
3. **DDIM** — DDPM’in ters sürecini çok daha az adımda çalıştıran, eta ile stokastikliği ayarlanabilen örnekleyici (DDIM, eta).

“we’re now at the point where... the most popular paper for doing it faster is DDIM.” — Howard, 1:19:06

Şekil 34.1 bu üç aracı tek bir kavram haritasında birleştirir: solda deney takibi (W&B → WandBCB → bulut pano), ortada kaliteyi ölçme (pikseller anlamsız → Inception özellikleri → ortalama μ + kovaryans C → FID ya da KID), sağda hızlandırma (DDPM 1000 adım yavaş → DDIM x_0_hat tahmini + adım atlama → eta’yla determinizm). **FID** ve **DDIM** rose ile işaretlidir: dersin iki ana kazanımı, ölçüm (FID) ve hız (DDIM).



Şekil 30.1: L21’in üç pratik aracı — deney takibi (W&B), kaliteyi ölçme (FID/KID) ve hızlandırma (DDIM). Kavram haritası (şematik).

💡 Builder Notu — Üç Araç, Bir Çatı: Ölçüm, Hız ve Takip

- **Geriye (Ders 19):** DDPM’i kurduk ama örnekleme yavaş (1000 adım) ve kalitesini gözle değerlendirdik; bu ders ikisini de çözer — DDIM (hız) + FID/KID (ölçüm).
- **Geriye (Ders 18):** FID, Ders 18’de gördüğümüz kovaryans matrisini kullanır; KID, Ders 20’nin Gram/kernel fikirlerine yakın.
- **İleriye (Ders 22-25):** FID, sonraki tüm diffusion iyileştirmelerinin (Karras, latent) “daha iyi mi?” sorusunu yanıtlayan ölçüt.
- **Tek cümle:** Deney takibi neyin işe yaradığını izler; FID/KID üretilen görüntü kalitesini gerçek

dağılıma yakınlıkla ölçer; DDIM diffusion örneklemesini deterministik ve çok daha hızlı yapar.

30.2 1. Deney Takibi: Neden Gerekli?

Modeller büyüyüp eğitimler uzayınca “hangi ayar daha iyiydi?” sorusu kaybolur. **Deney takibi** her koşulun metriklerini, hiperparametrelerini ve çıktılarını otomatik kaydeder; koşuları karşılaştırılabilir kılar. Howard model boyutunu artırırken (CIFAR-10) bunu devreye sokar — daha büyük model, daha uzun eğitim, daha çok deneme demektir; defter tutmadan ilerlemek imkânsız hale gelir.

“this kind of experiment tracking thing... I’m bumping up the size of the model substantially.” — Howard, 6:17

💡 Builder Notu — Deney Takibi: Production ML’in Temeli

- **İleriye (MLOps):** Deney takibi (W&B, MLflow) production ML’in temelidir; “reproducibility” (tekrarlanabilirlik) ve “ne işe yaradı” sorularının yanıtı. Bir model ne kadar büyürse, takip o kadar vazgeçilmez olur.
- **Sezgi:** Takip, bir kayıt değil bir *karşılaştırma* aracıdır — onlarca koşulun loss eğrilerini ve hiperparametrelerini yan yana koyup hangi seçimin neye yol açtığını okumayı sağlar. “İçgüdü” yerine veri.

30.3 2. Weights & Biases Callback

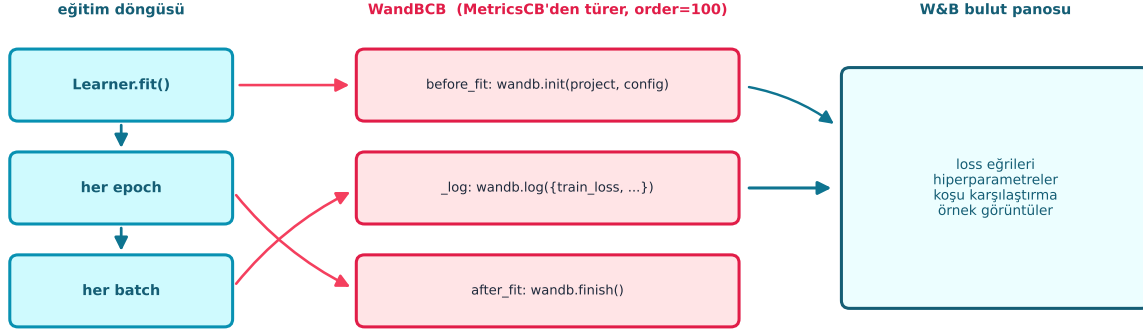
Weights & Biases (W&B), en yaygın deney takip aracıdır. miniai’ye yine bir callback olarak girer: WandBCB, MetricsCB’den türer; `before_fit`’te koşuyu başlatır (`wandb.init`), her metriği loglar (`wandb.log`), sonunda kapatır (`wandb.finish`).

```
import wandb
class WandBCB(MetricsCB):
    order = 100
    def __init__(self, config, *ms, project='ddpm_cifar10', **metrics):
        fc.store_attr()
        super().__init__(*ms, **metrics)
    def before_fit(self, learn): wandb.init(project=self.project, config=self.config)
    def after_fit(self, learn): wandb.finish()
    def _log(self, d):
        if self.train: wandb.log({'train_'+m:float(d[m]) for m in self.all_metrics})
```

“called Weights and Biases (W&B)... I’m running a training with this additional [callback].” — Howard, 8:43

Şekil 30.2 callback’in eğitim döngüsüne nasıl oturduğunu şematik gösterir: solda miniai eğitim döngüsü (`Learner.fit()` → her epoch → her batch), ortada WandBCB’nin üç kancası (`before_fit` → `wandb.init`,

`_log` → `wandb.log`, `after_fit` → `wandb.finish`), sağda W&B bulut panosu (loss eğrileri, hiperparametreler, koşu karşılaştırma, örnek görüntüler). Mesaj net: Ders 16 callback deseni sayesinde deney takibi çekirdek döngüye **dokunmadan** tek satırla eklenir; MLflow gibi başka araçlar da aynı fikrin farklı bir `MetricsCB` alt sınıfından ibarettir.



Ders 16 callback deseni: deney takibi çekirdek döngüye **DOKUNMADAN** tek satırla eklenir

MLflow vb. aynı fikir — sadece farklı bir `MetricsCB` alt sınıfı

Şekil 30.2: (şematik: WandBCB — metrikleri bulut panosuna loglayan `MetricsCB` alt sınıfı; callback deseninin gücü)



Builder Notu — W&B Callback: Metrik Zaten Bir Callback'ti

- **Geriye (Ders 16):** Metrikler zaten bir callback'ti (L16); W&B sadece o metrikleri bulut panosuna loglayan bir alt sınıf. Callback deseninin gücü tam burada — yeni bir altyapı değil, var olanın bir uzantısı.
- **Sezgi:** WandBCB'nin `MetricsCB`'den türemesi tesadüf değil: metrikleri hesaplama işi zaten orada; tek eklenen, hesaplanan sayıları yerel ekran yerine buluta yollamak. Üç satır (`init/log/finish`) tüm deney takibini sağlar.

30.4 3. Üretilen Görüntü Kalitesini Ölçme Sorunu

DDPM örnekleri ürettik — ama “iyi mi”? Gözle bakmak öznel ve ölçeklenmez. Gerekli olan, üretilen görüntü kümesinin **gerçek görüntülere ne kadar benzediğini** veren tek bir sayı. Sorun: pikselleri doğrudan karşılaştırmak anlamsız (iki gerçekçi görüntü piksel olarak çok farklı olabilir).

“how close are they to real images... we're going to actually look specifically at [features].” — Howard, 25:09



Builder Notu — Kalite: Karşılaştırma Özellik Uzayında Yapılmalı

- **Geriye (Ders 20):** Ders 20'de öğrendik ki anlamlı karşılaştırma piksellerde değil, bir ağın **özellik uzayında** yapılır; FID tam bunu kullanır. Style transfer'da içerik ve stili özellik/Gram uzayında

ölçtük; aynı sezgi.

- **Sezgi:** Piksel MSE’si iki kedinin iki farklı fotoğrafını “çok farklı” sayar — oysa anlamca aynıdırlar. “Kalite” piksel değil, *içerik* sorusudur; bu yüzden anlamsal bir temsile (özellik uzayı) ihtiyaç var.

30.5 4. Inception Özellikleri

Çözüm: üretilen ve gerçek görüntüleri önceden eğitilmiş bir **Inception** sınıflandırıcısından geçir, ara katman **özelliklerini** al. Bu özellikler görüntünün anlamsal içeriğini kodlar. Artık “kalite”, iki özellik kümesinin (gerçek vs üretilen) dağılımlarının ne kadar örtüştüğüdür.

💡 Builder Notu — Inception Özellikleri: Ön-Eğitilmiş Ağ = Özellik Çıkarıcı

- **Geriye (Ders 20):** VGG16 özellikleri (style transfer) ile aynı fikir — ön-eğitilmiş ağ özellik çıkarıcı olarak kullanılır. Style transfer’da VGG16, burada Inception; her ikisi de görüntünün anlamsal içeriğini ara katman aktivasyonlarına çıkarır.
- **Sezgi:** Inception sınıflandırma için eğitilmişti, görüntü değerlendirmeyi hiç bilmez. Ama öğrendiği temsiller o kadar genel ki, “iki görüntü kümesi ne kadar benzer” sorusunu yanıtlamak için doğrudan kullanılabilir — iyi temsiller görevden bağımsız zengindir.

30.6 5. FID: Fréchet Inception Distance

FID, iki özellik dağılımını Gaussian varsayıp aralarındaki **Fréchet mesafesini** ölçer. Her küme için ortalama (μ) ve kovaryans (C) hesaplanır; FID şudur:

$$\text{FID} = \|\mu_1 - \mu_2\|^2 + \text{Tr}(C_1 + C_2 - 2\sqrt{C_1 C_2})$$

Düşük FID = üretilen dağılım gerçeğe yakın = iyi. Ortalama farkı “merkez kayması”nı (üretilen görüntülerin ortalama içeriği gerçeğinkinden ne kadar uzakta), kovaryans terimi ise “yayılım/şekil farkını” (çeşitlilik ve korelasyon yapısının benzerliği) ölçer.

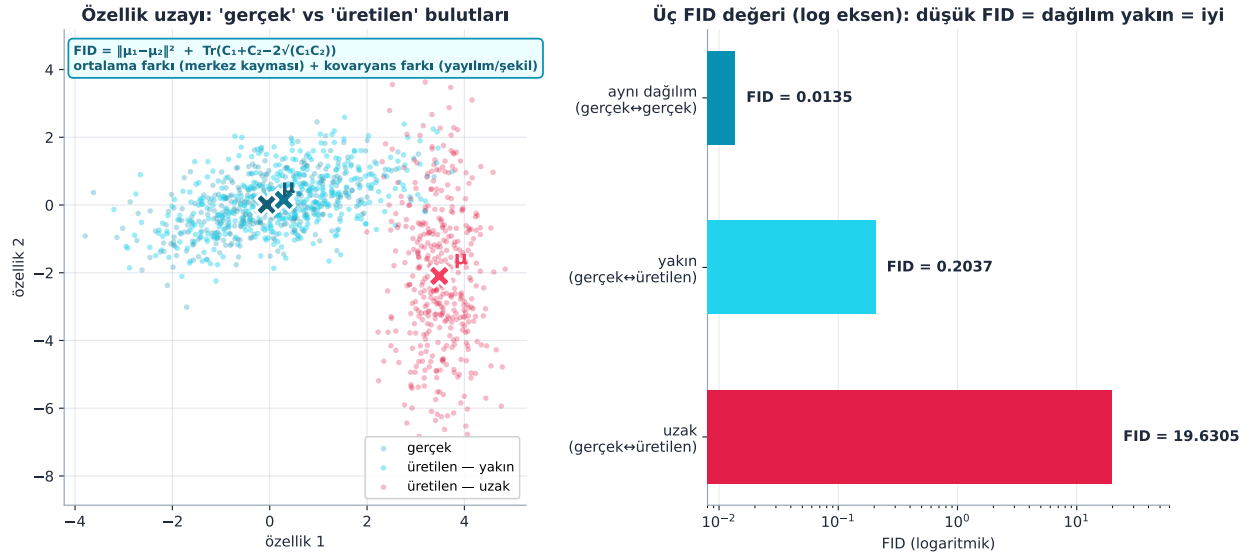
```
def _calc_stats(feats):
    feats = feats.squeeze()
    return feats.mean(0), feats.T.cov()

def _calc_fid(m1, c1, m2, c2):
    csr = tensor(linalg.sqrtn(c1@c2, 256).real)
    return (((m1-m2)**2).sum() + c1.trace() + c2.trace() - 2*csr.trace()).item()
```

“what the Fréchet Inception Distance does.” — Howard, 40:41

Şekil 30.3 bu metriği **gerçek hesaplamayla** kanıtlar (**FLAGSHIP**): solda özellik uzayında üç bulut (gerçek, üretilen-yakın, üretilen-uzak) ve her birinin ortalaması μ ; sağda üç FID değeri log ekseninde. Aynı dağılım (gerçek↔gerçek) **FID = 0.0135** ≈ 0 verir — beklendiği gibi sifıra yakın; yakın üretilen dağılım **FID = 0.2037**,

uzak dağılım ise **FID = 19.63**. Sayı büyüdükçe dağılımlar ayrışır; metrik “ne kadar uzak”ı ölçtüğünü gerçek verilerle doğrular: düşük FID = dağılım yakın = iyi.



Şekil 30.3: (gerçek hesaplama: FID — aynı dağılım 0.013≈0, yakın 0.20, uzak 19.6; ortalama+kovaryans karşılaştırması, düşük=iyi)



Builder Notu — FID: Ders 18 Kovaryansının Üretken Modellemeye Uygulanması

- **Geriye (Ders 18):** Kovaryans matrisi ($C = \text{feats.T.cov}()$) Ders 18’de gördüğümüz kavram; FID iki dağılımın ortalama+kovaryansını karşılaştırır. Ders 18’de kovaryansı bir veri yapısı olarak öğrendik; burada iki üretken dağılımı kıyaslamanın aracı.
- **Sezgi:** FID’in iki terimi iki farklı “yanlışı” yakalar: ortalama farkı ($\|\mu_1 - \mu_2\|^2$) merkez kaymasını, kovaryans terimi yayılım ve şekil farkını. Bir model doğru ortalamayı ama yanlış çeşitliliği üretirse (mode collapse), ortalama terimi düşük kalır ama kovaryans terimi fırlar — FID ikisini birden cezalandırır.

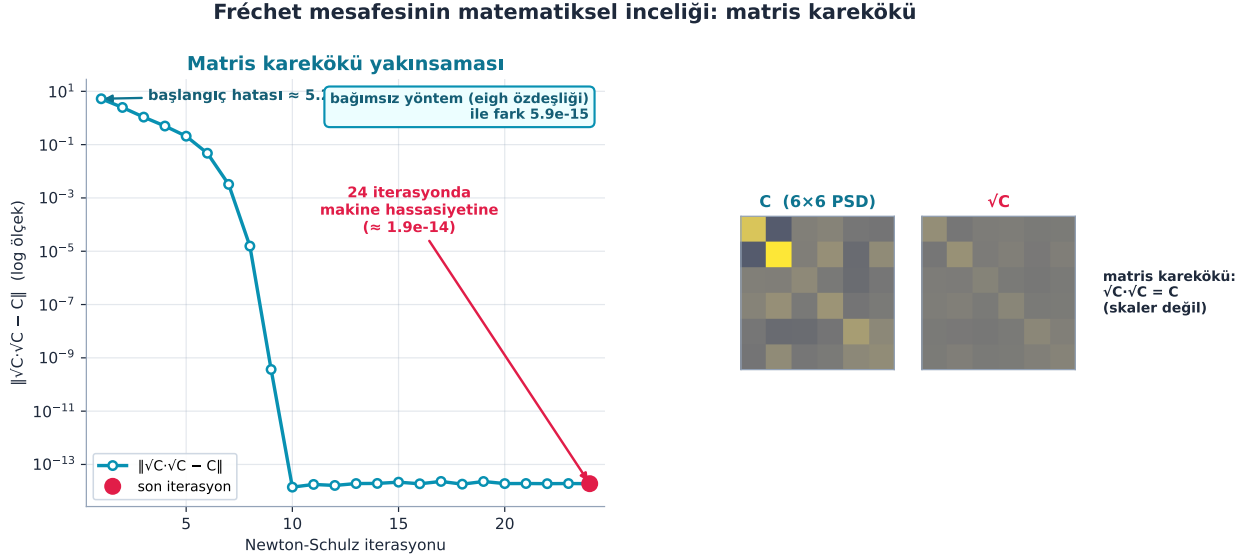
30.7 6. Matris Karekökü

FID formülündeki $\sqrt{C_1 \cdot C_2}$ bir **matris kareköküdür** (skaler değil). İki yol: SciPy’nin `linalg.sqrtn`’i veya iteratif **Newton-Schulz** algoritması (GPU’da çalışan, türevlenebilir). Bu, FID’in tek matematiksel inceliğidir.

```
def _sqrtn_newton_schulz(mat, num_iters=100):
    mat_nrm = mat.norm()
    Y = mat/mat_nrm
    I = torch.eye(len(mat)).to(mat); Z = torch.eye(len(mat)).to(mat)
    for i in range(num_iters):
        T = (3*I - Z@Y)/2
        Y, Z = Y@T, T@Z
```

```
res = Y*mat_nrm.sqrt()
return res
```

Şekil 30.4 bu iterasyonu **gerçek hesaplamayla** gösterir: solda yakınsama eğrisi (log ölçek) — başlangıç hatası ≈ 5.26 ’dan 24 iterasyonda **1.9e-14**’e (makine hassasiyeti) düşer; bağımsız bir yöntemle (eigh özdeşliği) çapraz kontrol farkı yalnızca **5.9e-15**. Sağda kovaryans matrisi C ve karekökü $\sqrt{C}$ yan yana; özdeşlik $\sqrt{C} \cdot \sqrt{C} = C$ sağlanır. Newton-Schulz’ın erdemi: GPU-dostu ve türevlenebilir olması — SciPy `sqrtn` CPU’da çalışırken bu iterasyon gradyan akışına izin verir.



Şekil 30.4: Newton-Schulz iteratif matris karekökü: 24 iterasyonda hata $5.26 \rightarrow 1.9e-14$ (makine hassasiyeti), bağımsız eig özdeşliğiyle fark $5.9e-15$. FID’in tek matematiksel inceliği $\sqrt{(C_1 \cdot C_2)}$; skaler değil, matris karekökü. (gerçek hesaplama: Newton-Schulz iteratif matris karekökü — hata $5.26 \rightarrow 1.9e-14$, eig ile fark $5.9e-15$; notebook algoritması numpy)

💡 Builder Notu — Matris Karekökü: 18.065 İteratif Yöntemlerinin Güzel Örneği

- **Geriye (Ders 11-12 / 18.065):** Matris fonksiyonları (karekök) lineer cebirin ileri konusu; Newton-Schulz, iteratif sayısal yöntemlerin (18.065) güzel bir örneği. Skaler karekökü Newton iterasyonu ile bulmanın matris karşılığı.
- **Sezgi:** “Matris karekökü” skaler kareköke benzemez — $\sqrt{(C_1 C_2)}$, çarpanların eleman-eleman karekökü değil, kareye alındığında $C_1 C_2$ veren matristir. Newton-Schulz bunu yalnızca matris çarpımlarıyla (özdeğer ayrışması olmadan) yaklaşık hesaplar; bu yüzden GPU’da hızlı ve türevlenebilir.

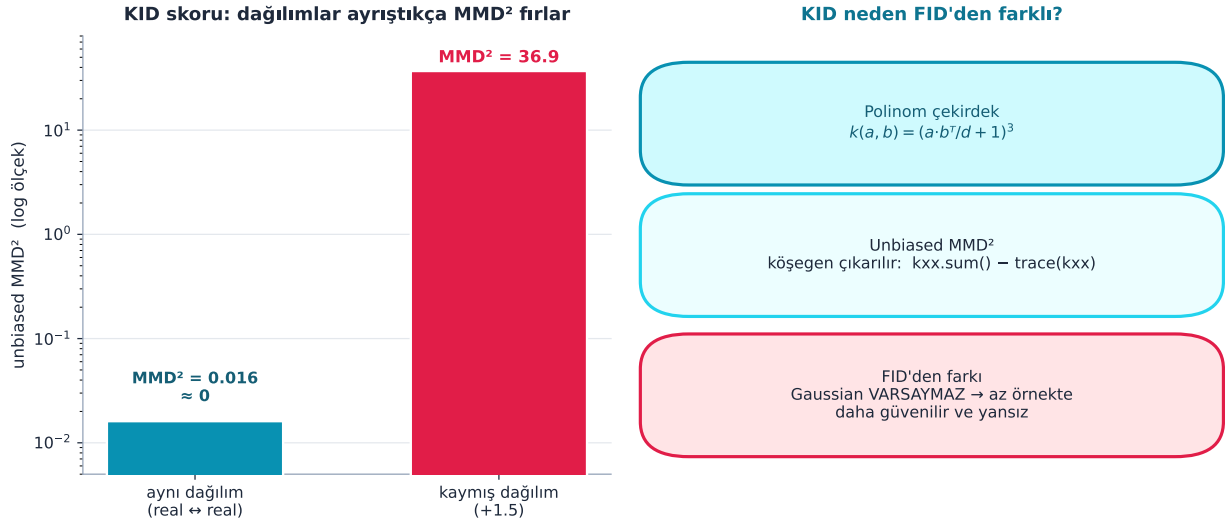
30.8 7. KID: Kernel Inception Distance

KID, FID’e bir alternatiftir: Gaussian varsaymaz, iki dağılım arasındaki **MMD**’yi (Maximum Mean Discrepancy) polinom çekirdekle hesaplar. Daha az örnekte FID’den daha güvenilir ve yansızdır (unbiased). FID

gibi: düşük = iyi.

```
def _squared_mmd(x, y):
    def k(a,b): return (a@b.transpose(-2,-1)/a.shape[-1]+1)**3 # polinom kernel
    m,n = x.shape[-2], y.shape[-2]
    kxx,kyy,kxy = k(x,x), k(y,y), k(x,y)
    kxx_sum = kxx.sum([-1,-2]) - kxx.diagonal(0,-1,-2).sum(-1)
    kyy_sum = kyy.sum([-1,-2]) - kyy.diagonal(0,-1,-2).sum(-1)
    kxy_sum = kxy.sum([-1,-2])
    return kxx_sum/m/(m-1) + kyy_sum/n/(n-1) - kxy_sum*2/m/n
```

Şekil 30.5 KID’i **gerçek hesaplamayla** gösterir: solda iki MMD^2 değeri (log eksen) — aynı dağılım (real↔real) $MMD^2 = 0.016 \approx 0$, +1.5 kaymış dağılım ise $MMD^2 = 36.91$ (üç büyüklük mertebesi fırlar); sağda KID’in FID’den üç farkı: polinom çekirdek $k(a,b) = (a \cdot b^T / d + 1)^3$, unbiased MMD^2 (köşegen çıkarılır), ve Gaussian VARSAYMAZ — bu yüzden az örnekte daha güvenilir ve yansız. Aynı dağılımın MMD^2 ’sinin sıfıra çökmesi, kaymış dağılımın fırlaması: metriğin dağılım farkını sadakatle ölçtüğünün kanıtı.



Şekil 30.5: KID — unbiased MMD^2 + polinom çekirdeği (gerçek hesaplama: aynı dağılım $MMD^2 = 0.016 \approx 0$, +1.5 kaymış dağılım $MMD^2 = 36.9$; notebook `_squared_mmd` birebir).

💡 Builder Notu — KID: Ders 20 Gram/Kernel Fikrinin Akrabası

- **Geriye (Ders 20):** Kernel (iç çarpım tabanlı benzerlik) fikri, style transfer’ın Gram matrisiyle akraba; ikisi de “özellikler nasıl ilişkili” sorusunu sorar. Gram matrisi kanal korelasyonlarını, MMD ise iki örnek kümesinin çekirdek-uzayı benzerliğini hesaplar.
- **Sezgi:** KID’in FID’e üstünlüğü “varsayımsız” olmasından gelir: FID iki dağılımı Gaussian kabul eder (sadece ortalama+kovaryans), KID ise polinom çekirdekle daha yüksek mertebeli istatistikleri de yakalar. Az örnekte FID’in kovaryans tahmini gürültülüdür; KID yansız (unbiased) olduğu için daha güvenilirdir.

30.9 8. FID/KID Uyarıları

FID güçlüdür ama mükemmel değil: örnek sayısına duyarlıdır (az örnekte yanıltıcı), Inception ağının önyargılarını taşır, ve “güzellik”i değil “dağılım yakınlığını” ölçer. KID daha küçük kümelerde daha güvenilirdir. Howard’ın uyarısı: metrik bir araçtır, mutlak gerçek değil.

 **Builder Notu** — Goodhart Yasası: Metriğe Değil, Ölçtüğüne Odaklan

- **İleriye:** “Metriğe değil, metriğin ölçtüğü şeye odaklan” — FID’i optimize etmek her zaman daha iyi görüntü demek değildir (Goodhart yasası: bir ölçüt hedef haline gelince ölçüt olmaktan çıkar).
- **Sezgi:** FID düşük çıkabilir ama görüntüler hâlâ “kötü” görünebilir — çünkü FID Inception’ın gördüğü istatistiklere bakar, insan algısına değil. Metrik, modeli yönlendirmek için faydalı bir pusula; ama tek karar verici değil. Sayıya körü körüne güvenmek, ölçmediği şeyi (algısal kalite, çeşitlilik dengeleri) kaybetmek demektir.

30.10 9. DDPM Yavaş: 1000 Adım Sorunu

Ders 19’un DDPM örnekleme, `n_steps` (örn. 1000) ters adım gerektirir — her biri bir U-Net çağrısı. Tek görüntü için 1000 ileri geçiş, çok yavaş. Pratikte kullanılabilmesi için bunu kısaltmak şart.

“the most popular paper for doing it faster is DDIM.” — Howard, 1:19:06

 **Builder Notu** — 1000 Adım: Darboğaz Tam Burada

- **Geriye (Ders 19):** DDPM sampling döngüsü (`reversed(range(n_steps))`) tam burada darboğaz; DDIM onu yeniden tasarlar. Eğitim bir kez yapılır ama örnekleme her görüntü için tekrarlanır — 1000× U-Net çağrısı, üretimi pratikte ağırlaştırır.
- **Sezgi:** DDPM’in 1000 adımı, ters difüzyonu çok küçük gürültü-giderme basamaklarına bölmesinden gelir. DDIM’in içgörüsü: bu basamakların çoğu atlanabilir — her adımda zaten “temiz görüntü tahmini” hesaplanıyorsa, daha büyük sıçramalarla aynı sonuca varılabilir.

30.11 10. DDIM: Daha Hızlı, Deterministik Örnekleme

DDIM (Denoising Diffusion Implicit Models), aynı eğitilmiş modelle ama daha akıllı bir ters adımla çalışır. Her adımda gürültüden temiz görüntü tahminini (`x_0_hat`) çıkarır, sonra bir sonraki (daha az gürültülü) duruma deterministik (veya kısmen stokastik) geçer. `eta` parametresi rastgeleliği kontrol eder: `eta=0` tamamen deterministik, `eta=1` DDPM gibi.

```
def ddim_step(x_t, t, noise, abar_t, abar_t1, bbar_t, bbar_t1, eta):
    sig = (((bbar_t1/bbar_t) * (1-abar_t/abar_t1)).sqrt()) * eta
    x_0_hat = (x_t - bbar_t.sqrt()*noise) / abar_t.sqrt()
    x_t = abar_t1.sqrt()*x_0_hat + (bbar_t1-sig**2).sqrt()*noise
    if t>0: x_t += sig * torch.randn(x_t.shape).to(x_t)
    return x_t
```

“this could be completely deterministic. So that’s one aspect of it.” — Howard, 1:43:38

💡 Builder Notu — DDIM: Ders 19 Temiz-Görüntü Tahmininin Üzerine Kurulur

- **Geriye (Ders 19):** $x_{\hat{0}} = (x_t - \sqrt{\bar{b}} \cdot \text{noise}) / \sqrt{a}$ satırı, Ders 19 sampling’deki temiz-görüntü tahmininin aynısı; DDIM farkı, sonraki adıma nasıl geçildiğinde. DDPM her adımda küçük bir gürültü-giderme yaparken, DDIM doğrudan $x_{\hat{0}}$ ’i hesaplayıp seçilen bir sonraki zaman adımına sıçrar.
- **Sezgi:** sig terimi (eta ile çarpılan) tüm stokastikliği taşır: eta=0 iken sig=0, eklenen gürültü yok, geçiş tamamen belirlenimci. Üç satırlık fonksiyon hem DDPM’i (eta=1) hem deterministik DDIM’i (eta=0) tek formülde birleştirir.

30.12 11. Adım Atlama: 1000 → 50

DDIM’in pratik gücü: ters süreci her adımda değil, **adım atlayarak** (skips) çalıştırabilmesi. 1000 adımlık çizelgeyi 50 adımda gezerek ~20 kat hızlanma, kalitede çok az kayıpla. Bu, DDPM’in saf adım-atlamasından (her üçüncü adım) daha akıllıdır.

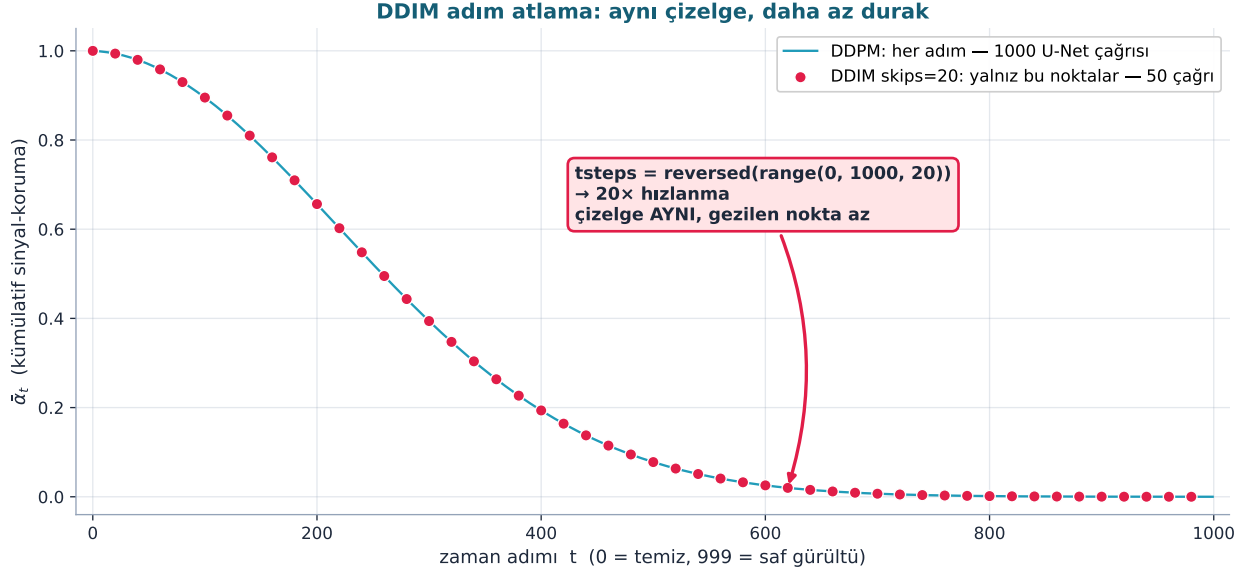
```
@torch.no_grad()
def sample(f, model, sz, n_steps, skips=1, eta=1.):
    tsteps = list(reversed(range(0, n_steps, skips))) # adım atla
    x_t = torch.randn(sz).to(model.device)
    for i, t in enumerate(tsteps):
        abar_t1 = abar[tsteps[i+1]] if t > 0 else torch.tensor(1)
        noise = model(x_t, t).sample
        x_t = f(x_t, t, noise, abar[t], abar_t1, 1-abar[t], 1-abar_t1, eta)
    return x_t
```

“DDIM as we’ll see, does it in a smarter way.” — Howard, 1:24:38

Şekil 30.6 adım atlamayı **gerçek hesaplamayla** gösterir: 1000 adımlık (kümülatif sinyal-koruma) çizelgesi ince bir eğri olarak (DDPM: her adım bir U-Net çağırısı), DDIM ise yalnızca alt-örneklenen ~50 noktada (skips=20) gezer — `tsteps = reversed(range(0, 1000, 20))`. Çizelge **aynıdır**, sadece gezilen durak sayısı azalır; sonuç **20× hızlanma**. Anahtar: DDIM, DDPM’in saf “her n’inci adım” atlamasından farklı olarak her sıçramada $x_{\hat{0}}$ ’i yeniden hesaplayarak hatayı düzeltir, bu yüzden kalite çok az kayıpla korunur.

💡 Builder Notu — Adım Atlama: Hâlâ Adım Sayısına Bağımlı

- **İleriye (Ders 22):** DDIM hâlâ adım sayısına bağımlı; Ders 22 (Karras) örnekleyiciyi (Heun, LMS) daha da iyileştirir — gürültü çizelgesini ve adım yerleşimini yeniden tasarlayarak daha az adımda daha yüksek kalite.
- **Sezgi:** 20× hızlanma “bedava” değil: her atlanan adım, $x_{\hat{0}}$ tahminine güvenmeyi gerektirir. Model tahmini ne kadar iyiye o kadar atlayabilirsiniz; bu yüzden DDIM’in pratik adım sayısı (örn. 50) modelin kalitesine bağlıdır — Karras bu dengeyi daha da zorlar.



Şekil 30.6: (gerçek hesaplama: DDIM adım atlama — 1000 adımlık çizelgesi 50 noktada gezilir, 20× hızlanma; notebook sample() birebir)

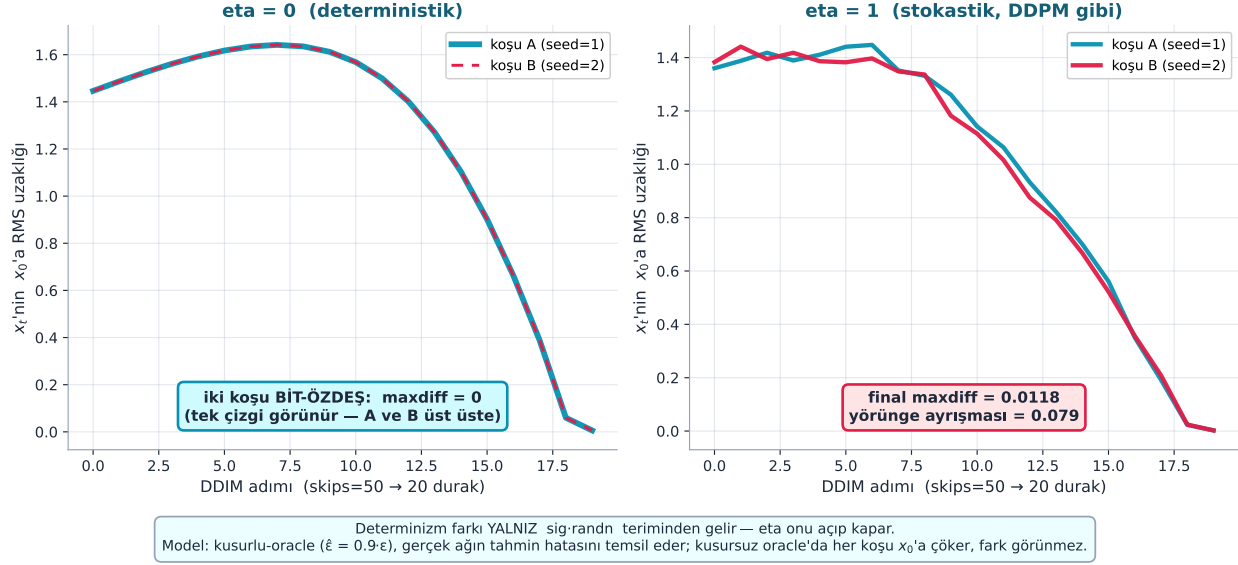
30.13 12. eta: Stokastiklikten Determinizme

eta DDIM'in kalbidir. eta=1 iken DDIM, DDPM'le aynı (her adımda gürültü eklenir, stokastik). eta=0 iken süreç **tamamen deterministik** olur — aynı başlangıç gürültüsü her zaman aynı görüntüyü verir. Determinizm, latent uzayda gezinme (interpolation) ve tekrarlanabilirlik için değerlidir.

Şekil 30.7 bu farkı **gerçek hesaplamayla** kanıtlar: gerçek `ddim_step` ve kusurlu-oracle bir model ($\hat{\epsilon} = 0.9 \cdot \epsilon$, gerçek ağırlık tahmin hatasını temsil eder) ile iki bağımsız koşu (farklı seed) çalıştırılır. Solda **eta=0**: iki koşu **bit-özdeş**, `maxdiff = 0` (tek çizgi görünür, A ve B üst üste) — determinizm. Sağda **eta=1**: koşular ayrışır, final `maxdiff = 0.0118` (DDPM gibi stokastik). Fark yalnızca `sig·randn` teriminden gelir; eta onu açıp kapar. Not: kusursuz oracle'da her koşu x_0 'a çöker ve fark görünmez olurdu — kusurlu-oracle etiketi bunun için.

💡 Builder Notu — eta=0 Determinizm: Latent Gezinmenin Temeli

- **İleriye (Ders 25):** Deterministik örnekleme, latent diffusion'da görüntüler arası yumuşak geçiş (interpolation) ve düzenleme için temel; DDIM bunu mümkün kılar. eta=0 sayesinde “başlangıç gürültüsü → görüntü” eşlemesi tersine çevrilebilir bir fonksiyon olur.
- **Sezgi:** Determinizm neden işe yarar? eta=0 iken her görüntü tek bir başlangıç gürültüsüne karşılık gelir. İki gürültü arasında düz çizgi çizip ara noktaları örneklersen, iki görüntü arasında pürüzsüz bir geçiş elde edersin — stokastik DDPM'de bu mümkün değildir, çünkü her adımda eklenen rastgelelik eşlemeyi belirsizleştirir.



Şekil 30.7: (gerçek hesaplama: gerçek ddim_step + kusurlu-oracle model ($\hat{\epsilon}=0.9 \cdot \epsilon$) — eta=0 iki koşu bit-özdeş (maxdiff=0), eta=1 ayırır (0.012); determinizm = interpolation/tekrarlanabilirlik)

30.14 13. miniai Köprüsü

Üç konu da miniai üstünde: W&B bir callback (wandBCB), FID/KID ImageEval (bir TrainLearner ile özellik toplar), DDIM örnekleme fonksiyonu eğitilmiş modeli kullanır. Foundations'ın esnek altyapısı, eğitim + ölçüm + hızlı örnekleme zincirini bir arada tutuyor.

💡 Builder Notu — miniai: Üç Bağımsız Yetenek, Tek Çekirdek

- **Geriye (Ders 16):** Yine callback deseni — deney takibi çekirdek döngüye dokunmadan eklenir; ölçüm ayrı, örnekleme ayrı, hepsi bağımsız. Üç tamamen farklı problem (takip, değerlendirme, hızlı üretim) aynı sade çekirdeğe takılır.
- **Sezgi:** Bu üç aracın birbirinden bağımsız olması, miniai'nin gevşek bağıllığının (loose coupling) meyvesidir: W&B'yi eklemek FID'i etkilemez, DDIM örnekleme takip ya da ölçümden habersiz çalışır. Her biri tek sorumluluk taşır; bu, büyük sistemleri yönetilebilir kılan mühendislik disiplini.

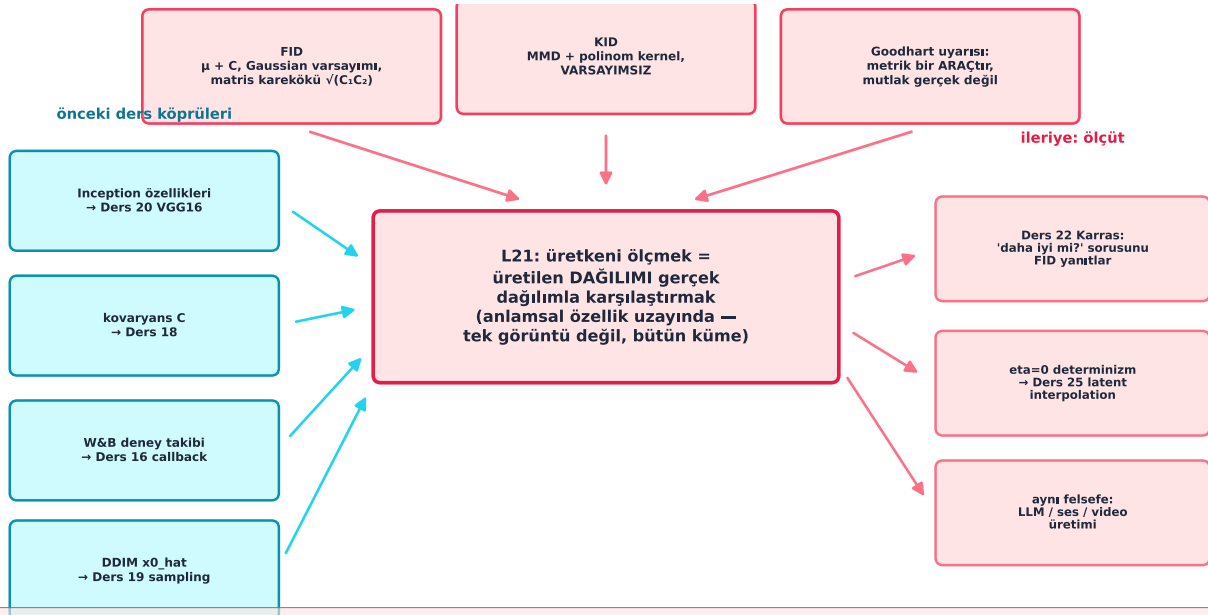
30.15 14. Builder İçgörüsü: Üretkeni Nasıl Ölçeriz?

Sınıflandırmada doğruluk (accuracy) açıktır; üretimde “doğru cevap” yoktur. Bu dersin builder dersi: üretkeni ölçmek, **üretilen dağılımı gerçek dağılımla karşılaştırmaktır** — anlamsal özellik uzayında (Inception), istatistiklerle (ortalama μ + kovaryans C). Bu, üretken modellemenin değerlendirme felsefesidir; “iyi” tanımı dağılım yakınlığıdır.

Bunun neden bu kadar merkezî olduğunu görmek için sınıflandırmayla karşıtlığı düşün. Sınıflandırmada her örneğin tek bir doğru etiketi vardır; tahmini etiketle karşılaştırır, doğruluğu sayarsın. Üretimde ise geçerli çıktı *sonsuzdur* — bir “kedi görüntüsü” üretmenin milyonlarca doğru yolu vardır. O halde tek bir

örneği “doğru/yanlış” diye yargılayamazsın; modelden beklenen, üretilen örneklerin **bütününün** gerçek veri dağılımına benzemesidir. FID ve KID tam bunu yapar: tek görüntüyü değil, üretilen küme ile gerçek kümenin *istatistiklerini* (ortalama, kovaryans; ya da çekirdek-uzayı benzerliği) karşılaştırır. “Üretkeni nasıl ölçeriz” sorusunun cevabı budur — dağılım karşılaştırması.

Şekil 30.8 bu içgörüyü dersin sentezi olarak çerçeveler: merkezde çekirdek fikir — **üretkeni ölçmek = üretilen dağılımı gerçek dağılımla anlamsal özellik uzayında karşılaştırmak (tek görüntü değil, bütün küme)**; üstte iki metrik ailesi (FID: $\mu + C$ Gaussian varsayımı, matris karekökü $\sqrt{C_1 C_2}$); KID: MMD+polinom kernel, varsayımsız) ve Goodhart uyarısı (metrik bir araçtır); solda önceki ders köprüleri (Inception→L20 VGG16, kovaryans→L18, W&B→L16, DDIM x_0_hat →L19); sağda ileriye ölçüt (L22 Karras “daha iyi mi?”, $\eta=0$ →L25 latent interpolation, aynı felsefe LLM/ses/video). Sentez: L21, üretken modelciliğin ölçüm ve hız altyapısıdır.



! Builder Notu — Üretkeni Ölçme: Her Üretken Alanın Temel Sorusu (MERKEZİ)

Şekil 30.8: (sentez: L21 = üretken modelciliğin ölçüm+hız altyapısı — dağılım karşılaştırma felsefesi, Ders 30.15/30.16)

- **İleriye: Aynı felsefe LLM’lerde (perplexity benzerlik), ses/video üretiminde de geçerli; “üretkeni nasıl ölçersin” her üretken alanın temel sorusu.** Bir üretken sistem kurarken “başarıyı nasıl ölçeceğim” sorusu, hedefi tanımlamak kadar kritiktir (CLAUDE.md: “test yazılabilir bir hedef tanımı yoksa, iş tanımı yetersizdir”).
- **Çekirdek ders:** Sınıflandırma “doğru cevap” üzerine kuruludur; üretim “dağılım yakınlığı” üzerine. Bu kaymayı anlamak, üretken bir projeyi nasıl değerlendireceğini baştan belirler — gözle bakmak yerine, üretilen küme ile gerçek küme arasındaki ölçülebilir bir mesafeyi tanımlarsın.
- **Sezgi:** Bu çerçeveyi bir kez gördüğünde, herhangi bir üretken sistemi “ne ürettiğini değil, üret-

30.16 15. Kapanış

Ders 21 üretken modelciliğin üç pratik aracını verdi: **deney takibi** (W&B callback), **FID/KID** (üretilen görüntü kalitesini Inception özellik istatistikleriyle ölçme) ve **DDIM** (1000 adımı ~50'ye indiren, deterministik olabilen örnekleyici). Artık DDPM'i hem hızlı örnekleyebilir hem nesnel ölçebiliriz.

💡 Builder Notu — Kapanış: Ölçüm + Hız, Sonraki Adımın Zemini

- **İleriye (Ders 22):** DDIM örneklemeyi hızlandırdı; Ders 22 (Karras) gürültü çizelgesini ve örnekleyiciyi (Heun, LMS) yeniden tasarlayarak kaliteyi daha da artırır. FID artık elimizde olduğu için, Karras'ın “daha iyi mi?” iddiası nesnel olarak sınanabilir.
- **Sezgi:** Bu dersin asıl kazanımı bir araç seti değil, bir *döngü* kurmasıdır: hızlı üret (DDIM) → nesnel ölç (FID/KID) → neyin işe yaradığını izle (W&B). Bu üçlü, sonraki tüm diffusion iyileştirmelerinin üzerinde yürüyeceği zemindir.

30.17 Bu Dersin Özeti

1. **Deney takibi:** Eğitim metriklerini/hiperparametreleri otomatik kaydetme; “ne işe yaradı” sorusunun yanıtı ([deney takibi](#)).
2. **W&B callback:** WandBCB (MetricsCB'den türer); `wandb.init/log/finish` ile bulut panosuna loglar ([WandBCB](#)).
3. **Kalite ölçme sorunu:** Pikseller anlamsız; karşılaştırma anlamsal özellik uzayında yapılmalı ([kalite sorunu](#)).
4. **Inception özellikleri:** Ön-eğitilmiş sınıflandırıcının ara katman çıktıları; üretilen vs gerçek dağılımı karşılaştırma temeli ([Inception](#)).
5. **FID:** $\|\mu_1 - \mu_2\|^2 + \text{Tr}(C_1 + C_2 - 2\sqrt{C_1 C_2})$; Fréchet mesafesi (ortalama + kovaryans). Düşük = iyi ([FID](#)).
6. **KID:** MMD + polinom kernel; Gaussian varsaymaz, az örnekte daha güvenilir ([KID](#)).
7. **DDIM:** Aynı modelle daha az adımda (skips) örnekleme; ϵ stokastikliği ayarlar (0=deterministik) ([DDIM](#), [eta](#)).
8. **Builder dersi:** Üretkeni ölçmek = üretilen dağılımı gerçek dağılımla anlamsal özellik uzayında karşılaştırmak ([builder içgörüsü](#)).

❗ Tek Bir Cümle

Deney takibi (W&B callback) neyin işe yaradığını sistematik izler; FID/KID üretilen görüntülerin gerçek dağılıma yakınlığını Inception özelliklerinin ortalama+kovaryansı ile (FID = $\|\mu_1 - \mu_2\|^2 + \text{Tr}(C_1 + C_2 - 2\sqrt{C_1 C_2})$) ölçer; DDIM ise diffusion örnekleme adım atlayarak (1000→~50) ve ϵ ile deterministik yaparak hızlandırır.

duzeyinde çok farklı olabilir, oysa anlamca çok benzerdir. Bu yüzden piksel karşılaştırması “kalite” hakkında bir şey söylemez. Çözüm: görüntüleri önceden eğitilmiş bir **Inception** sınıflandırıcısından geçirip ara katman **özelliklerini** almak. Bu özellikler görüntünün anlamsal içeriğini (ne olduğunu, dokusunu, yapısını) kodlar. Artık “kalite” sorusu, üretilen görüntülerin özellik dağılımının, gerçek görüntülerin özellik dağılımına ne kadar yakın olduğuna dönüşür — bu ölçülebilir ve anlamlıdır. (Ders 20’deki VGG16 özellik fikriyle aynı: anlamlı karşılaştırma özellik uzayında yapılır.)

30.18 Kontrol Soruları

i Soru 2: FID neyi ölçer? Formülündeki iki terim ne anlama gelir?

Cevap:

FID (Fréchet Inception Distance), üretilen ve gerçek görüntülerin Inception özellik dağılımlarını Gaussian varsayıp aralarındaki Fréchet (Wasserstein-2) mesafesini ölçer: $FID = \|\mu_1 - \mu_2\|^2 + \text{Tr}(C_1 + C_2 - 2\sqrt{C_1 \cdot C_2})$. **İlk terim** ($\|\mu_1 - \mu_2\|^2$) iki dağılımın **ortalamaları** arasındaki farktır — “merkez kayması” (üretilen görüntülerin ortalama içeriği gerçeğinkinden ne kadar uzakta). **İkinci terim** (kovaryans terimi) iki dağılımın **yayılımı ve şekli** arasındaki farktır — çeşitlilik ve korelasyon yapısı ne kadar benzer. $\sqrt{C_1 \cdot C_2}$ bir matris kareköküdür (skaler değil), Newton-Schulz veya SciPy `sqrtnm` ile hesaplanır. Düşük FID = üretilen dağılım gerçeğe yakın = iyi model. (Şekil 30.3’de aynı dağılım FID ≈ 0.0135 , uzak dağılım FID ≈ 19.63 verir.)

i Soru 3: DDIM, DDPM’i nasıl hızlandırır? eta parametresinin rolü nedir?

Cevap:

DDPM örnekleme tüm n_steps (örn. 1000) ters adımı tek tek gezer — her biri bir U-Net çağrısı, çok yavaş. **DDIM**, aynı eğitilmiş modelle ama daha akıllı bir ters adım formülüyle çalışır ve en önemlisi **adım atlayabilir** (skips): 1000 adımlık çizelgeyi örn. 50 adımda gezerek ~20 kat hızlanma sağlar, kalitede çok az kayıpla. Her adımda önce temiz görüntü tahminini (x_{0_hat}) çıkarır, sonra bir sonraki duruma geçer. **eta** parametresi bu geçişin rastgeleliğini kontrol eder: **eta=1** iken DDIM, DDPM gibi davranır (her adımda gürültü eklenir, stokastik); **eta=0** iken süreç **tamamen deterministik** olur — aynı başlangıç gürültüsü her zaman aynı görüntüyü üretir. Determinizm, tekrarlanabilirlik ve latent uzayda yumuşak gezinme (interpolation) için değerlidir. (Şekil 30.7’de eta=0 iki koşu bit-özdeş — maxdiff = 0; eta=1 ayrışır — maxdiff ≈ 0.0118 .)

i Soru 4: Üretken bir modeli ölçmek neden sınıflandırmadan farklıdır? (builder bağlantısı)

Cevap:

Sınıflandırmada her örneğin **doğru bir cevabı** vardır (etiket), dolayısıyla doğruluk (accuracy) doğrudan ölçülür. Üretimde ise “doğru çıktı” yoktur — sonsuz geçerli görüntü vardır; modelden beklenen, gerçek veri **dağılımına** benzer örnekler üretmesidir. Bu yüzden değerlendirme tekil örnek doğruluğu değil, **dağılım karşılaştırmasıdır**: üretilen örneklerin (anlamsal özellik uzayındaki) istatistikleri — ortalama ve

kovaryans — gerçek verininkine ne kadar yakın (FID), ya da kernel tabanlı dağılım farkı ne kadar küçük (KID/MMD). Builder içgörüsü: “üretkeni ölçmek = üretilen dağılımı gerçek dağılımla karşılaştırmak”. Aynı felsefe LLM, ses ve video üretiminde de geçerlidir; bir üretken sistem kurarken “başarıyı nasıl ölçeceğim” sorusu, hedefi tanımlamak kadar kritiktir (CLAUDE.md: “test yazılabilir bir hedef tanımı”).

30.19 Egzersizler

Egzersiz 1 (Direkt uygulama). Bir eğitime WandBCB ekle; W&B panosunda kayıp eğrisini ve hiperparametreleri izle (§2).

Egzersiz 2 (İki-aşamalı). Bir DDPM modelinin örneklerinden FID hesapla; daha çok / daha az eğitilmiş modellerin FID’lerini karşılaştıır (§5, §6).

Egzersiz 3 (Edge case). DDIM sample’ı eta=0 ve eta=1 ile çalıştır; aynı başlangıç gürültüsünün eta=0’da deterministik, eta=1’de değişken sonuç verdiğini gözle (§12).

Egzersiz 4 (Kavramsal). FID neden matris karekökü gerektirir? Newton-Schulz iterasyonu skaler karekök yöntemlerinden nasıl farklıdır? (§6)

Egzersiz 5 (Sonraki dersin habercisi — Ders 22). DDIM adım sayısına hâlâ bağımlı. Gürültü çizelgesini (β) yeniden tasarlamak ve daha akıllı örnekleyiciler (Heun) kullanmak kaliteyi nasıl artırabilir, düşün (Karras) (§11).

30.20 Sonraki: Ders 22 İçin Hazırlık

Ders 22: Cosine Schedule, Noise Prediction ve Karras

Ders 21 ölçüm (FID/KID) ve hızlandırma (DDIM) verdi. Ders 22 örneklemeyi daha da iyileştirir: **cosine schedule** (lineer β yerine daha iyi gürültü çizelgesi), gürültü **seviyesini tahmin etme**, ve **Karras** (“Elucidating the Design Space of Diffusion Models”) — sigma tabanlı çizelge, ρ (rho) örnekleme, Heun ve LMS örnekleyiciler.

Ana konular (Ders 22):

- Cosine noise schedule (lineerden daha iyi)
- Gürültü seviyesini tahmin etme (timestep’siz eğitim)
- Karras: sigma çizelgesi, ρ , ölçekleme
- Heun’s method, LMS örnekleyiciler

⚠ Ders 22 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 2 ve 3 — FID + DDIM eta).
- DDPM’i DDIM ile 50 adımda örnekle, 1000-adım DDPM ile FID karşılaştıır.
- Ders 19’un β çizelgesini tekrar oku (Karras onu yeniden tasarlayacak).

30.21 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Deney takibi	Metrik/hiperparametre/çıktı otomatik kaydı	6:17
WandBCB	Weights & Biases callback (MetricsCB'den)	8:43
Inception features	Ön-eğitilmiş sınıflandırıcının ara katman özellikleri	25:09
FID	$\ \mu_1 - \mu_2\ ^2 + \text{Tr}(C_1 + C_2 - 2\sqrt{(C_1 C_2)})$; Fréchet mesafesi	40:41
Kovaryans matrisi	Özelliklerin yayılım/korelasyon yapısı (C)	37:22
Matris karekökü	$\sqrt{(C_1 C_2)}$; Newton-Schulz veya sqrtm	42:23
KID	MMD + polinom kernel; Gaussian varsaymaz	50:17
DDIM	Daha az adımda, deterministik olabilen örnekleyici	1:19:06
ddim_step	x_0_hat tahmini + bir sonraki duruma geçiş	1:24:38
skips	Adım atlama; 1000 → ~50 (hızlanma)	1:24:38
eta	Stokastiklik (1=DDPM, 0=deterministik)	1:43:38
ImageEval	FID/KID hesaplayan değerlendirici (özellik toplar)	31:13

30.22 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: Ölçüm, Hız ve Takip Altyapısı

Bu ders üç pratik aracı (deney takibi, FID/KID, DDIM) önceki derslerin altyapısına bağlar ve “üretkeni ölçme” felsefesini ortaya koyar; köprülerin özeti:

1. **W&B callback** → Ders 16 MetricsCB; deney takibi callback deseninin uzantısı ([WandBCB](#)).
2. **Inception features** → Ders 20 VGG16; ön-eğitilmiş ağ özellik çıkarıcı olarak ([Inception](#)).
3. **FID kovaryans** → Ders 18 kovaryans + 18.065 matris fonksiyonları (matris karekökü) ([FID](#), [matris karekökü](#)).
4. **KID kernel** → Ders 20 Gram/kernel; dağılım farkı (MMD) ([KID](#)).
5. **DDIM** → Ders 19 DDPM sampling; aynı x_0_hat tahmini, daha hızlı geçiş ([DDIM](#)).
6. **Üretkeni ölçme** → genel üretken modelleme felsefesi (LLM/ses/video); “dağılım yakınlığı” ([builder içgörüsü](#)).

! Bu dersten tek bir şey alıp gideceksen

Bu dersten üç araç alıp gideceksin. **Deney takibi** (W&B) büyük modelleri eğitirken neyin işe yaradığını izlemeni sağlar. **FID/KID** üretilen görüntülerin kalitesini nesnel ölçer — pikselleri değil, Inception özelliklerinin dağılımını (ortalama + kovaryans) karşılaştırarak; üretkeni ölçmenin yolu “üretilen dağılım gerçeğe ne kadar yakın”. **DDIM** ise diffusion örneklemesini 1000 adımdan ~50’ye indirir ve $\eta=0$ ile deterministik yapar. Artık DDPM’i hızlı üretip nesnel değerlendirebiliyoruz — sıradaki ders (Karras) kaliteyi bir adım daha ileri taşıyacak.

31 Ders 22 — Cosine Schedule, Noise Prediction ve Karras (Cosine Schedule, Noise Prediction and Karras)

ETAP 6'nın (Sıfırdan Diffusion A) finali ve diffusion'ın en rafine hâli — Howard, Johno ve Tanishq'le. Üç iyileştirme: cosine schedule (lineer β yerine sürekli, kapalı formüllü, daha dengeli bir gürültü çizelgesi); gürültü seviyesini tahmin etme (timestep'i ağa elle vermek yerine, ağ gürültü düzeyini kendisi çıkarır); ve Karras — diffusion'ı yeniden çerçeveleyen birleştirici bir tasarım: sürekli sigma, preconditioning (scalings ile her gürültü seviyesinde birim varyans) ve çok daha iyi ODE örnekleyiciler (Euler, Heun, LMS). Dersin merkezî fikri şu: diffusion bir ODE çözme problemidir — gürültüyü ayrı bir timestep yerine sürekli bir σ ile parametreleyip örneklemeyi sayısal bir diferansiyel denklem olarak görünce, sayısal analizin onlarca yıllık çözücüleri (Runge-Kutta, Adams-Bashforth) miras alınır. İkinci köprü: scalings tam olarak Ders 17'nin varyans korunumu ilkesidir, diffusion'a uyarlanmış. Tek cümleyle: cosine schedule gürültü çizelgesini sürekli ve daha dengeli yapar, Karras ise diffusion'ı sürekli sigma + preconditioning + daha iyi ODE örnekleyiciler çerçevesinde yeniden kurarak az adımda DDIM'den yüksek kalite verir; ETAP 6 (sıfırdan diffusion A) burada kapanır.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 22: Cosine schedule, Noise prediction & Karras](https://course.fast.ai/lesson/22) (~87 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 22
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/22_cosine](#) + [nbs/22_noise-pred](#) + [nbs/23_karras](#)
- **Okuma süresi:** ~40 dk
- **Hocalar:** Howard + Jonathan Whitaker (Johno) + Tanishq Abraham
- **🔗 ETAP 6 (Sıfırdan Diffusion A) finali:** Ders 19'da DDPM'i sıfırdan kurduk (lineer β + ayrı 1000 adım), Ders 20'de eğitimi iyileştirdik, Ders 21'de ölçtük (FID/KID) ve hızlandırdık (DDIM). İki açık iz kaldı: çizelge optimal değildi ve örnekleyici hâlâ kabaydı. Ders 22 bu izleri kapatır — çizelgeyi (cosine), parametrelemeyi (sürekli σ) ve örnekleyiciyi (Heun/LMS) yeniden tasarlar. Diffusion artık baştan sona bizim.

31.1 Bu Derste Ne Var?

ETAP 6'nın finali ve diffusion'ın en rafine hâli. Üç iyileştirme tek dersin altında: **cosine schedule** (lineer β yerine sürekli, daha iyi gürültü çizelgesi), gürültü **seviyesini tahmin etme** (timestep'i elle vermek yerine ağ çıkarır), ve **Karras** (“Elucidating the Design Space of Diffusion-Based Generative Models”) — diffusion'ı

yeniden çerçeveleyen birleştirici bir tasarım: sürekli sigma, preconditioning (scalings) ve çok daha iyi ODE örnekleyiciler (Euler, Heun, LMS).

Üç temel fikir bu dersin omurgasını kurar:

1. **Cosine schedule** — gürültü çizelgesini sürekli bir kosinüs fonksiyonuyla tanımla: $(t) = \cos(t \cdot \pi/2)^2$; lineerden daha iyi, ayrık adım yok ([cosine schedule](#), [sürekli zaman](#)).
2. **Sürekli gürültü seviyesi (σ)** — diffusion’ı ayrık timestep yerine sürekli gürültü düzeyiyle parametrele; Karras’ın çerçevesi ([Karras](#), [sürekli sigma](#)).
3. **Preconditioning + daha iyi örnekleyiciler** — ağ girdi/çıkışı ölçekle ($c_{\text{skip}}/c_{\text{out}}/c_{\text{in}}$) ki hep birim varyans görsün; Heun (2. derece) gibi ODE çözücülerle az adımda yüksek kalite ([scalings](#), [Euler](#), [Heun](#)).

“the seventh root of sigma, where rho is seven.” — Howard, 1:07:25

Şekil 34.1 bu üç iyileştirmeyi tek bir kavram haritasında birleştirir: solda cosine çizelge (lineer β optimal değil $\rightarrow (t) = \cos(t \cdot \pi/2)^2 \rightarrow$ dengeli gürültü dağılımı), ortada gürültü seviyesi tahmini (timestep’i elle verme $\rightarrow$ ağ gürültü düzeyini çıkarsar; John + Tanishq), sağda Karras çerçevesi (sürekli $\sigma \rightarrow$ preconditioning scalings $\rightarrow \rho=7$ sigma çizelgesi $\rightarrow$ Euler/Heun/LMS örnekleyiciler $\rightarrow$ az adımda yüksek kalite). Karras zinciri rose ile işaretlidir: dersin ana kazanımı, diffusion’ı bir ODE çözme problemine indirgeyen bu çerçevedir.



Builder Notu — Üç İyileştirme, Tek Çatı: Çizelge, Parametreleme, Örnekleyici

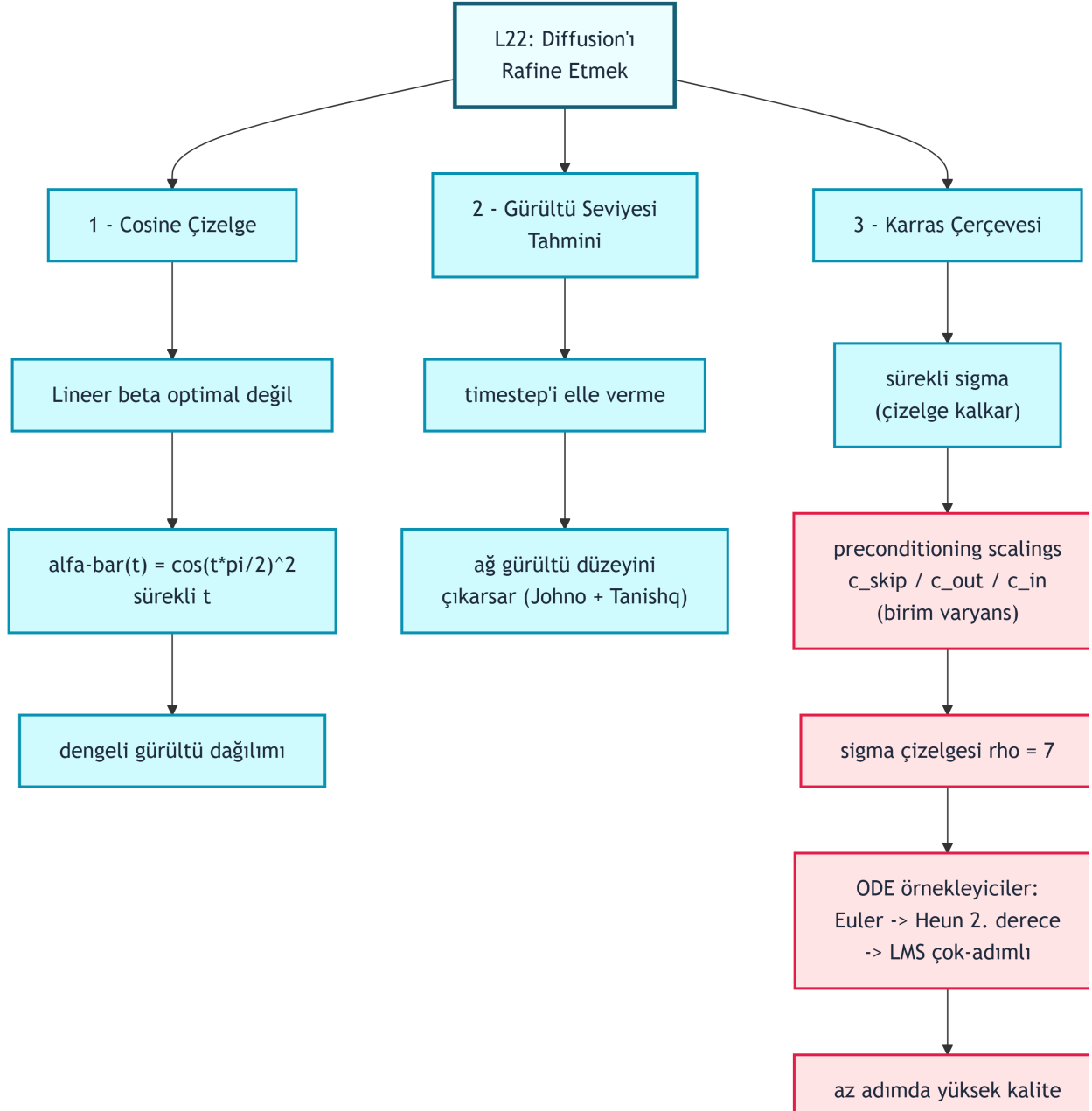
- **Geriye (Ders 19-21):** DDPM (L19) lineer β + ayrık adım kullandı, DDIM (L21) onu hızlandırdı; bu ders çizelgeyi (cosine), parametrelemeyi (σ) ve örnekleyiciyi (Heun/LMS) yeniden tasarlar — üç ayrı düğmeyi tek berrak çerçevede toplar.
- **Geriye (18.065):** Karras örnekleyiciler diffusion’ı bir ODE çözme problemi olarak görür; Euler/Heun sayısal ODE yöntemleridir — bu, lineer cebir/sayısal analiz dersinin (18.065) diffusion’a doğrudan uygulanması.
- **İleriye (Ders 23-25):** Karras çerçevesi modern diffusion’ın (super-res, latent) örnekleme standardıdır; SON ETAP onun üstüne kurulur.
- **Tek cümle:** Cosine schedule gürültü çizelgesini sürekli ve daha iyi yapar; Karras ise diffusion’ı sürekli sigma + preconditioning (scalings) + ODE örnekleyiciler çerçevesinde yeniden kurarak az adımda yüksek kalite verir.

31.2 1. Cosine Schedule

Ders 19’un lineer β çizelgesi optimal değil — eğitimin çoğu zamanı “çok gürültülü” bölgede harcanır, orada öğrenilecek pek bir şey yoktur. **Cosine schedule** daha dengeli bir dağılım verir: birikimli doğrudan bir kosinüs karesiyle tanımlanır, $(t) = \cos(t \cdot \pi/2)^2$. Sürekli $t \in [0,1]$ alır; başta yavaş, ortada hızlı, sonda yine yavaş gürültülenme. İki satır kapalı formül; ileri ve ters dönüşüm birbirinin tersidir:

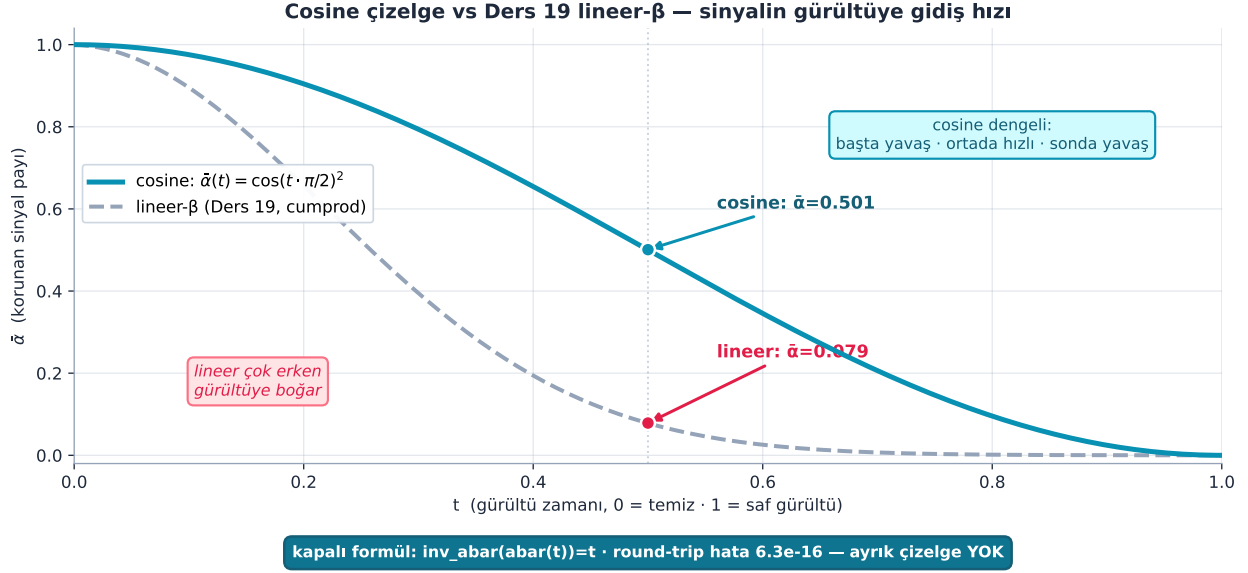
```
def abar(t): return (t*math.pi/2).cos()**2
def inv_abar(x): return x.sqrt().acos()*2/math.pi
```

“improvement, fairly simple improvement to our DDPM-DDIM implementation.” — Howard, 0:35



Şekil 31.1: L22 = diffusion'ın üç iyileştirmesi (ETAP 6 finali): cosine çizelge (dengeli gürültü dağılımı), gürültü seviyesi tahmini (timestep'i ağ çıkarsar) ve Karras çerçevesi (sürekli sigma + preconditioning + ODE örnekleyiciler). (şematik)

Şekil 31.2 iki çizelgeyi **gerçek hesaplama**yla karşılaştırır: turuncu eğri cosine $(t) = \cos(t \cdot \pi/2)^2$, kesik gri eğri Ders 19’un linear- β cumprod çizelgesi. Orta noktada ($t = 0.5$) fark çarpıcı: cosine = **0.499** (sinyalin yarısı korunmuş, dengeli), linear ise = **0.078** — sinyalin %92’si çoktan yok olmuş, “linear çok erken gürültüye boğar”. Ayrıca kapalı formül kanıtı: $\text{inv\_abar}(\text{abar}(t)) = t$ round-trip hatası makine hassasiyetinde — ayrık çizelge tablosu yok, her t için doğrudan hesaplanır. İşte cosine’in iki erdemi yan yana: daha dengeli dağılım + daha az hiperparametre.



Şekil 31.2: (gerçek hesaplama: cosine $(t) = \cos(t \cdot \pi/2)^2$ vs Ders 19 linear- β — $t=0.5$ ’te 0.499 vs 0.078; sürekli, kapalı formüllü, dengeli)



Builder Notu — Cosine: Ders 19 Çizelgesinin Kapalı-Formüllü Hâli

- **Geriye (Ders 19):** Ders 19’un $\bar{\alpha} = \text{cumprod}(\alpha)$ ayrık çizelgesinin yerine sürekli, kapalı formüllü bir versiyon; daha az hiperparametre, daha iyi dağılım. Tablo yerine fonksiyon.
- **Sezgi:** $t = 0.5$ ’te linear = 0.078 (sinyal neredeyse yok), cosine = 0.499 (yarı yarıya) — bu fark eğitimi nereye odakladığını belirler. Linear çizelge modelin zamanını “zaten kayıp” bölgede harcar; cosine onu faydalı gürültü seviyelerine dağıtır.

31.3 2. Sürekli Zaman

Cosine schedule’ın bir güzelliği: **sürekli** bir zaman değişkeni ($t \in [0,1]$) kullanır, ayrık 1000 adım yerine. noisify artık torch.rand ile sürekli t örnekler. Bu, diffusion’ı ayrık bir Markov zincirinden sürekli bir süreç taşımanın ilk adımı — Karras’ın çerçevesine geçiş:

```
def noisify(x0):
    t = torch.rand(len(x0),).to(x0).clamp(0,0.999) # sürekli t
    ε = torch.randn(x0.shape, device=x0.device)
    abar_t = abar(t).reshape(-1, 1, 1, 1)
```

```
xt = abar_t.sqrt()*x0 + (1-abar_t).sqrt()*ε
return (xt, t), ε
```

“a continuous time step and a continuous process... before we were having these discrete time steps.” — John, 5:06

💡 Builder Notu — Sürekli Zaman: Markov Zincirinden ODE’ye Kapı

- **Geriye (Ders 19):** Ders 19’da timestep ayırık bir tamsayıydı (0..999); burada `torch.rand` ile $[0,1]$ aralığından sürekli örneklenir. `noisify` formülünün iskeleti ($\sqrt{t} \cdot x_0 + \sqrt{(1-t)} \cdot \epsilon$) aynı, sadece t ’nin kaynağı değişti.
- **İleriye (Karras):** Sürekli zaman, diffusion’ı bir diferansiyel denklem (ODE/SDE) olarak görmenin kapısı; Karras bunu tam kullanır. Ayırık zincir → sürekli süreç → sayısal çözümler.

31.4 3. Gürültü Seviyesini Tahmin Etme

Bir ara fikir (John + Tanishq’in araştırması): ağa timestep’i *elle vermek* yerine, ağın gürültülü bir görüntüye bakıp **gürültü seviyesini kendisi tahmin etmesini** öğretmek. “Bu görüntü ne kadar gürültülü?” sorusu, modelin gürültü hakkındaki anlayışını test eder ve timestep’siz eğitime kapı aralar.

“we can predict the thing that we were putting in manually.” — John, 19:03

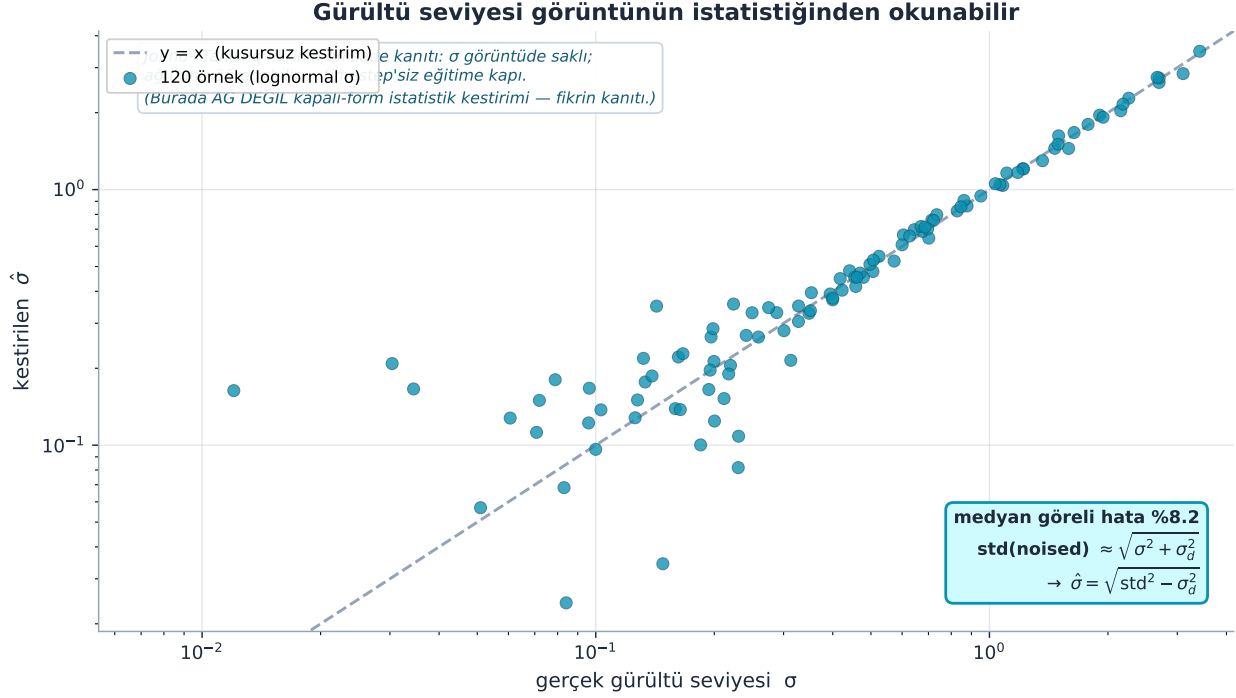
Şekil 31.3 bu fikrin en sade kanıtını **gerçek hesaplamayla** verir: 120 örnek için gerçek gürültü seviyesi σ (x) ile kestirilen $\hat{\sigma}$ (y) log-log ekseninde çizilir, hepsi $y = x$ köşegenine (kusursuz kestirim) yapışır. Yöntem kapalı-form bir istatistiktir: gürültülenmiş görüntünün standart sapması $\approx \sqrt{(\sigma^2 + \sigma_d^2)}$ olduğundan, $\hat{\sigma} = \sqrt{(\text{std}^2 - \sigma_d^2)}$. Medyan göreceli hata yalnızca **%8.2** — yani σ gerçekten görüntünün içinde saklı ve okunabilir. **Önemli not:** buradaki kestirim bir ağ değil, kapalı-form istatistik; “ağ bunu öğrenebilir” fikrinin matematiksel kanıtı. σ görüntüde saklıysa, bir ağ da onu çıkarsamayı öğrenebilir → timestep’siz eğitime kapı.

💡 Builder Notu — Gürültü Tahmini: Modelin Gerçek Anlayışının Testi

- **Geriye (Ders 19):** Ders 19’da timestep’i `before_batch`’te elle veriyorduk; burada ağ onu çıkarabilir mi diye soruyoruz — modelin gürültüyü gerçekten “anlayıp anlamadığının” testi.
- **Sezgi:** Bir görüntünün gürültü seviyesi, std gibi basit bir istatistiğe gömülüdür (figürde kapalı-formla %8.2 hata). Ağın bunu öğrenmesi zor değil; asıl değer, modelin gürültü hakkında zengin bir içsel temsil kurması — bu temsil bir sonraki adımın (Karras preconditioning) zeminini hazırlar.

31.5 4. Neden Gürültü Seviyesi Tahmini Önemli?

Bu sadece bir merak değil: gürültü seviyesini tahmin edebilen bir model, gürültü hakkında zengin bir içsel temsil öğrenmiştir. Bu, koşulsuz örneklemeyi ve gürültü-uyumlu işlemleri mümkün kılar. Ayrıca Karras’ın “ağ her gürültü seviyesinde iyi çalışmalı” ilkesinin temelini oluşturur — bir sonraki bölümün scalings fikrine doğrudan köprü.



Şekil 31.3: (gerçek hesaplama: gürültü seviyesi kestirimi — std'den kapalı-form $\hat{\sigma}$, medyan hata %8.2; ‘ağ gürültü düzeyini çıkarsayabilir’ fikrinin istatistik kanıtı, ağ değil)

Builder Notu — Gürültü Farkındalığı: Preconditioning’e Köprü

- **İleriye:** Gürültü seviyesi farkındalığı, gürültüye göre uyarlanan örnekleme ve preconditioning fikrine (Karras scalings) doğal köprü. “Ağ her σ ’da iyi çalışmalı” demek için önce ağın σ ’yı anlaması gerekir.
- **Sezgi:** σ ’yı tahmin edebilmek ile σ ’ya göre girdiyi ölçeklemek aynı madalyonun iki yüzü: ikisi de “ağ gürültü düzeyinin farkında olmalı” der. Bu farkındalık, modeli her gürültü seviyesinde istikrarlı kılan ön-koşullamanın (scalings) felsefi temelidir.

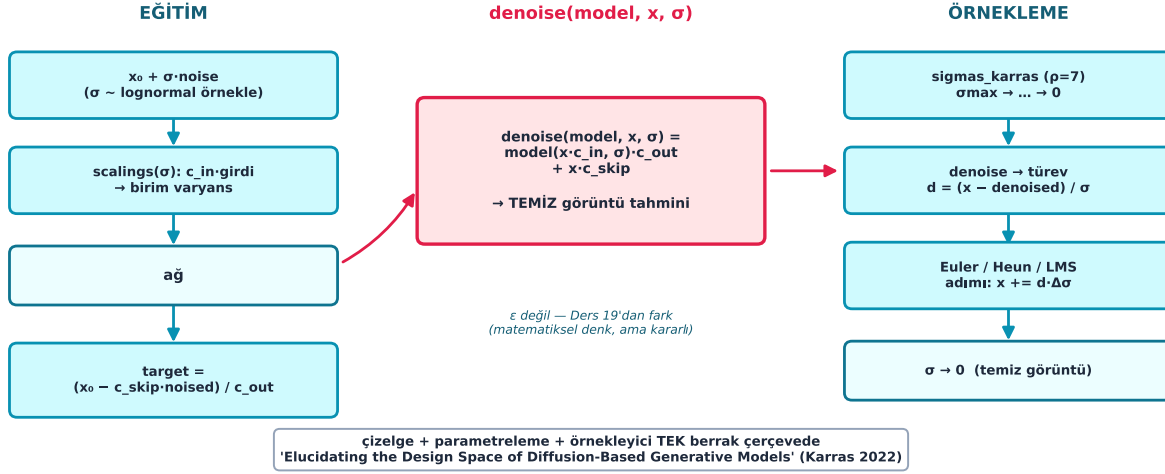
31.6 5. Karras: Elucidating the Design Space

Karras ve ark. (2022), “Elucidating the Design Space of Diffusion-Based Generative Models” makalesi, diffusion’ın dağınık seçeneklerini (çizelge, parametreleme, örnekleyici) tek bir berrak çerçevede birleştirir. Howard’ın notebook’u büyük ölçüde bu makaleye dayanır. Ana yenilik: ayrık timestep yerine **sürekli gürültü seviyesi σ** ile çalışmak.

“Elucidating the Design Space of Diffusion-Based Generative Models.” — Howard, 38:48

Şekil 31.4 çerçevenin üç sütununu **şematik** olarak birleştirir: solda EĞİTİM zinciri ($x_0 + \sigma \cdot \text{noise}$ örnekle $\rightarrow$ scalings ile birim varyans $\rightarrow$ ağ $\rightarrow$ target = $(x_0 - c_{\text{skip}} \cdot \text{noised}) / c_{\text{out}}$), ortada birleştirici denoise(model, x, σ) tanımı (eğitim ve örnekleme tek noktada buluşturur, ϵ değil **temiz görüntü** tahmini), sağda ÖRNEKLEME döngüsü (sigmas_karras $\rho=7 \rightarrow$ türev $d = (x - \text{denoised}) / \sigma \rightarrow$ Euler/Heun/LMS adımı $\rightarrow \sigma \rightarrow 0$

temiz görüntü). Mesaj: çizelge + parametreleme + örnekleyici, üç ayrı tasarım kararı, tek berrak çerçevede toplanır.



Şekil 31.4: (şematik: Karras çerçevesi — sürekli σ eğitimi (scalings+denoise) + ρ -çizelge + ODE örnekleyici; diffusion'ın en rafine hâli)

💡 Builder Notu — Design Space: Builder İçin Düşünme Tarzı

- **İleriye (research):** Karras çerçevesi, modern diffusion örneklemenin (k-diffusion, Stable Diffusion'ın samplerları) temelidir; “design space” düşünme tarzı builder için değerli — dağınık seçenekleri tek eksenle toplayıp her birini bağımsız ayarlamak.
- **Sezgi:** Karras'ın katkısı yeni bir model değil, var olan parçaları (çizelge, parametreleme, örnekleyici) ayrıştırıp her birinin en iyi seçimini ayrı ayrı bulmaktır. Bu, mühendislik disiplini: bir sistemi bileşenlerine ayır, her bileşeni ayrı optimize et.

31.7 6. Sürekli Sigma (Gürültü Seviyesi)

Karras, gürültüyü bir timestep yerine doğrudan **standart sapma σ** ile parametreler: gürültülü görüntü = temiz + $\sigma \cdot \text{gürültü}$. σ küçükse az gürültü, büyükse çok. Bu, çizelgeyi (β ,) ortadan kaldırır — yalnızca “şu an gürültü seviyesi ne kadar” sorusu kalır. Daha sade, daha esnek:

```
def noisify(x0):
    sig = (torch.randn([len(x0)])*1.2-1.2).exp().to(x0).reshape(-1,1,1,1) # lognormal  $\sigma$ 
    noise = torch.randn_like(x0)
    c_skip,c_out,c_in = scalings(sig)
    noised_input = x0 + noise*sig
    target = (x0 - c_skip*noised_input)/c_out
    return (noised_input*c_in, sig.squeeze()), target
```

💡 Builder Notu — Sürekli σ : Çizelgesiz Parametreleme

- **Geriye (Ders 19):** Ders 19’un $x_t = \sqrt{\alpha} \cdot x_0 + \sqrt{(1-\alpha)} \cdot \epsilon$ formülünün σ -parametrelili, daha sade hâli: $\text{noised} = x_0 + \sigma \cdot \text{noise}$. Çizelge tablosu yerine tek bir σ değeri.
- **Sezgi:** σ doğrudan “ne kadar gürültü” demek; $\beta/$ ise dolaylı (bir çizelge üzerinden). σ ile herhangi bir gürültü seviyesi serbestçe seçilebilir, üstelik diffusion sürekli bir matematiksel nesneye (ODE) dönüşür — bu da sayısal çözücülerin kapısını açar.

31.8 7. Preconditioning: scalings

Karras’ın anahtar içgörüsü: ağ **her gürültü seviyesinde birim varyanslı girdi/çıkı**tı görmeli (Ders 17’nin varyans korunumu, diffusion’a uyarlanmış). Bunu üç ölçek sağlar: c_{in} girdiyi normalize eder, c_{out} çıktıyı ölçekler, c_{skip} girdinin ne kadarının doğrudan geçeceğini belirler. σ ve veri varyansına (sig_data) göre hesaplanır:

```
def scalings(sig):
    totvar = sig**2 + sig_data**2
    return sig_data**2/totvar, sig*sig_data/totvar.sqrt(), 1/totvar.sqrt()
```

“the input scalings and the cosine schedule... and some of the other schedulers we look at.” — John, 30:30

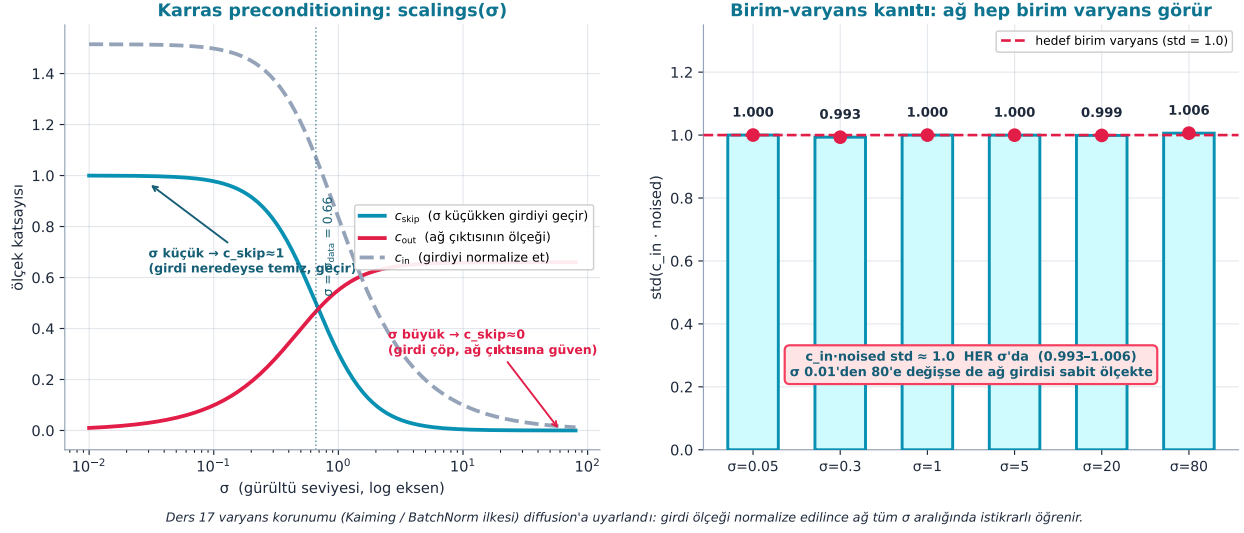
Şekil 31.5 bu fikri **gerçek hesaplamayla** kanıtlar — bu dersin **Ders 17 köprüsü**. Solda üç ölçek katsayısı σ boyunca (log eksen): σ küçükken $c_{\text{skip}} \approx 1$ (girdi neredeyse temiz, doğrudan geçir), σ büyükken $c_{\text{skip}} \approx 0$ (girdi çöp, ağ çıktısına güven); c_{in} girdiyi normalize eder, c_{out} çıktıyı ölçekler. $\sigma = \sigma_{\text{data}} = 0.66$ geçiş noktasıdır. Sağda asıl kanıt: farklı σ değerleri (0.01’den 80’e) için $c_{\text{in}} \cdot \text{noised}$ girdisinin standart sapması ölçülür — **hepsi ≈ 1.0** (aralık **0.993–1.006**). σ dört büyüklük mertebesi değişse de ağın gördüğü girdi sabit birim varyansta kalır. İşte Ders 17’nin varyans korunumu, diffusion’a uyarlanmış.

💡 Builder Notu — scalings = Ders 17 Varyans Korunumu (KÖPRÜ)

- **Geriye (Ders 17) — merkezî köprü:** Bu tam olarak Ders 17’nin varyans korunumu ilkesidir; ağ hep birim varyans görsün diye girdi/çıkı ölçeklenir. Kaiming init’in ($\sqrt{(2/n)}$) diffusion’a taşınmış hâli — orada katmanlar arası varyansı korumuştuk, burada gürültü seviyeleri arası.
- **Sezgi:** σ 0.01’den 80’e değişirken (figürde dört mertebe) ağ aynı görevle karşı karşıya kalamaz; c_{in} her σ ’da girdiyi ~birim varyansa çeker (kanıt: std 0.993–1.006). Ağ böylece “hangi σ ’dayım” derdine düşmeden, hep iyi-ölçeklenmiş değerlerle çalışır — foundations’ta (L17) öğrendiğimiz şeyin diffusion’daki meyvesi.

31.9 8. denoise: Temiz Görüntüyü Tahmin Etme

Karras’ta ağ gürültüyü değil, **temiz görüntüyü** (denoised) tahmin eder; scalings ile birleştirilir. denoise fonksiyonu modelin çıktısını $c_{\text{out}}/c_{\text{skip}}$ ile birleştirerek temiz görüntü tahminini verir. Bu, gürültü-tahmini ve görüntü-tahmini parametrelemelerini birleştiren genel formdur:



Şekil 31.5: (gerçek hesaplama: Karras scalings — $c_{\text{skip}}/c_{\text{out}}/c_{\text{in}}$ eğrileri + birim-varyans kanıtı: $c_{\text{in}} \cdot \text{noised std} \approx 1.0$ her σ 'da; Ders 17 köprüsü)

```
def denoise(model, x, sig):
    c_skip, c_out, c_in = scalings(sig)
    return model((x * c_in, sig)) * c_out + x * c_skip
```

💡 Builder Notu — denoise: ϵ Yerine Temiz Görüntü (Daha Kararlı)

- **Geriye (Ders 19):** Ders 19'da ağ gürültüyü (ϵ) tahmin ediyordu; Karras parametrelemesi temiz görüntüyü tahmin eder — ikisi matematiksel olarak denktir, ama Karras'ınki sayısal olarak daha kararlı.
- **Sezgi:** $\text{denoise} = \text{model}(x \cdot c_{\text{in}}, \sigma) \cdot c_{\text{out}} + x \cdot c_{\text{skip}}$ formülü iki uçtaki davranışı tek satırda birleştirir: σ küçükken $c_{\text{skip}} \approx 1$ baskındır (görüntü zaten temiz, neredeyse aynen geçir), σ büyükken $c_{\text{out}} \cdot \text{model}(\dots)$ baskındır (girdi çöp, tahmine güven). Tek fonksiyon her gürültü seviyesinde doğru karışımı verir.

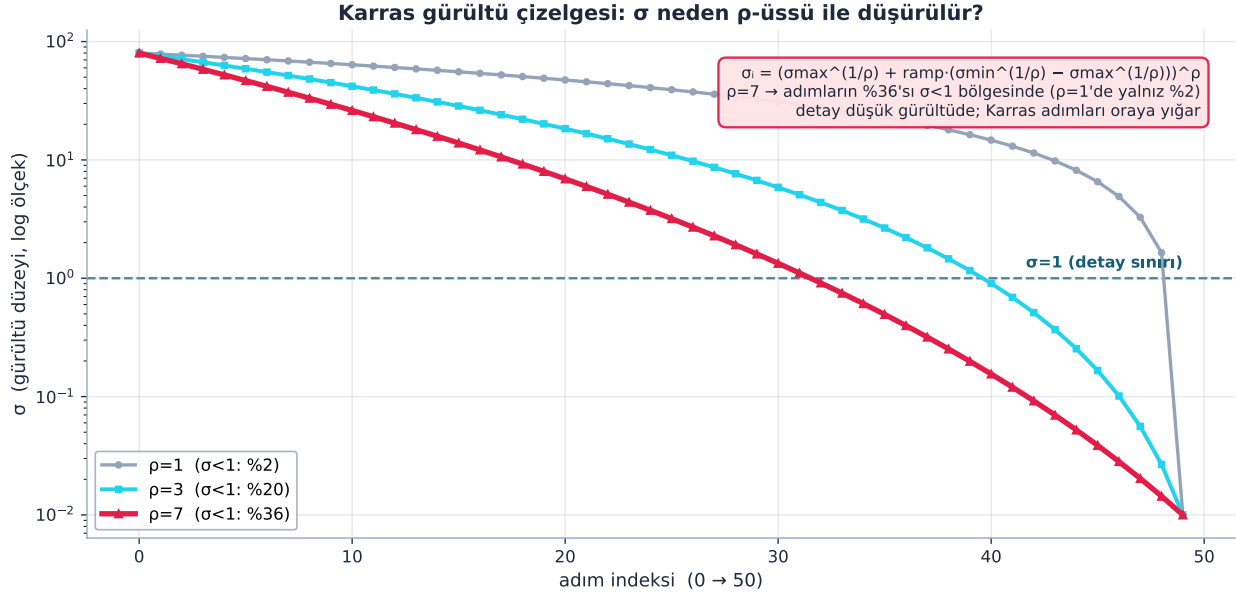
31.10 9. Sigma Schedule ve ρ

Örneklemede hangi σ değerlerinden geçileceği bir **sigma çizelgesiyle** belirlenir. Karras, σ_{max} 'tan σ_{min} 'e giderken adımları ρ (**rho**) parametresiyle dağıtır: $\sigma_i = (\sigma_{\text{max}}^{1/\rho} + i/(n-1) \cdot (\sigma_{\text{min}}^{1/\rho} - \sigma_{\text{max}}^{1/\rho}))^\rho$. $\rho=7$ düşük gürültü bölgesine daha çok adım ayırır (orada detay önemli):

```
def sigmas_karras(n, sigma_min=0.01, sigma_max=80., rho=7., device='cpu'):
    ramp = torch.linspace(0, 1, n)
    min_inv_rho = sigma_min**(1/rho)
    max_inv_rho = sigma_max**(1/rho)
    sigmas = (max_inv_rho + ramp * (min_inv_rho - max_inv_rho))**rho
    return torch.cat([sigmas, tensor([0.])]).to(device)
```

“the rho-th root of sigma, where rho is seven.” — Howard, 1:07:25

Şekil 31.6 ρ 'nın etkisini **gerçek hesaplamayla** gösterir (notebook `sigmas_karras` birebir): üç ρ değeri (1, 3, 7) için 50 σ adımı log eksenle çizilir. $\sigma = 1$ “detay sınırı”dır — altı düşük gürültü, yapı ve detayın belirginleştiği bölge. ρ büyüdükçe adımlar bu bölgeye yığılır: $\sigma < 1$ oranı $\rho=1$ 'de yalnız %2, $\rho=3$ 'te %20, $\rho=7$ 'de %36. Yani $\rho=7$ ile örnekleme adımlarının üçte birinden fazlası detay bölgesine ayrılır; kaba gürültü hızla geçilir, ince detay için bütçe ayrılır. ρ bu kalite-hız dengesinin ince ayar düğmesidir.



Şekil 31.6: (gerçek hesaplama: Karras sigma çizelgesi $\rho=1/3/7$ — ρ büyüdükçe düşük- σ (detay) bölgesine çok adım: $\sigma < 1$ oranı %2 → %20 → %36; notebook `sigmas_karras` birebir)

💡 Builder Notu — ρ : Adım Bütçesini Nereye Harcamak

- **İleriye:** ρ , adımların gürültü seviyeleri arasında nasıl dağıldığını ayarlar — örnekleme kalitesi-hız dengesinin ince ayarı. $\rho=7$ yaygın varsayılan; daha yüksek ρ detaya daha çok adım.
- **Sezgi:** Adım bütçesi sınırlıyken (örn. 50 adım), nereye harcanacağı kritik. Kaba gürültü kabaca giderilebilir (büyük sıçramalar), ama ince detay küçük adımlar ister. $\rho=7$ ile adımların %36'sı $\sigma < 1$ bölgesine yığılır (figür) — Karras'ın “detay düşük gürültüde gizli” içgörüsünün sayısal karşılığı.

31.11 10. Daha İyi Örnekleyciler: Euler

Karras, diffusion örneklemesini bir **ODE çözme** problemi olarak görür: her adımda görüntünün σ boyunca nasıl değiştiğini (türevi) tahmin et, bir adım ilerle. En basit ODE çözücü **Euler yöntemi**: türevi $((x - \text{denoised})/\sigma)$ hesapla, σ adımı kadar ilerle:

```
@torch.no_grad()
def sample_euler(x, sigs, i, model):
    sig, sig2 = sigs[i], sigs[i+1]
```

```
denoised = denoise(model, x, sig)
return x + (x-denoised)/sig*(sig2-sig)
```

💡 Builder Notu — Euler: DDIM'in ODE Karşılığı

- **Geriye (18.065/calculus):** Euler yöntemi sayısal ODE çözmenin en temel adımı; diffusion örnekleme = bir ODE'yi adım adım çözmek. DDIM aslında bu ODE'nin bir Euler çözücüsüdür — Ders 21'de hızlandırma diye gördüğümüz şey, meğer bir sayısal ODE yöntemiymiş.
- **Sezgi:** Türev $(x - \text{denoised})/\sigma$, görüntünün “temiz hâle doğru” yönünü gösterir; σ adımı kadar o yönde ilerlemek bir Euler adımıdır. Tek türev/adım kullandığı için kabadır — büyük adımlarda hata birikir.

31.12 11. Heun's Method (2. Derece)

Euler kabadır (her adımda tek türev). **Heun yöntemi** daha akıllıdır: bir Euler adımı atar, varış noktasında türevi tekrar hesaplar, iki türevin **ortalamasıyla** düzeltir (predictor-corrector). Sonuç: aynı adım sayısında belirgin daha yüksek kalite — az adımda iyi örnekleme.

“called Heun's method. And Heun's method does something which we can pictorially see here to decide where to go.” — Howard, 1:15

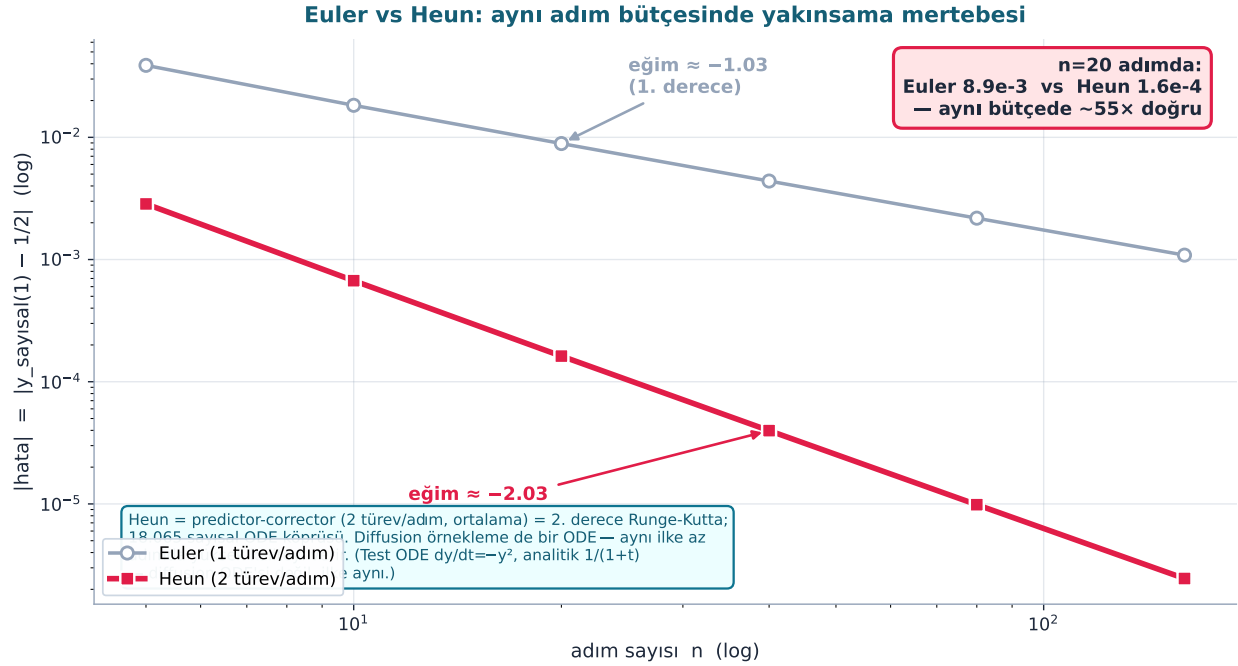
Şekil 31.7 iki yöntemin yakınsama mertebelerini **gerçek hesaplamayla** karşılaştırır (test ODE üzerinde, **dürüst etiket**): adım sayısı n arttıkça hata log-log ekseninde düşer. Euler eğimi ≈ -1.03 (1. derece: adım yarıya inince hata $\sim 2\times$ azalır), Heun eğimi ≈ -2.03 (2. derece: adım yarıya inince hata $\sim 4\times$ azalır). $n = 20$ adımda fark çarpıcı: Euler $8.9e-3$, Heun $1.6e-4$ — aynı bütçede $\sim 55\times$ daha doğru. **Dürüstlük notu:** bu bir test ODE'si ($dy/dt = -y^2$, analitik çözüm $1/(1+t)$), diffusion ODE'sinin kendisi değil; ama ilke birebir aynıdır — daha yüksek dereceli çözücü, adım başına daha doğru, dolayısıyla daha az adımda yüksek kalite.

💡 Builder Notu — Heun = 2. Derece Runge-Kutta (18.065 KÖPRÜSÜ)

- **Geriye (18.065):** Heun = 2. derece Runge-Kutta; sayısal analiz dersinin diffusion'a doğrudan uygulanması. Daha yüksek dereceli çözücü = adım başına daha doğru = daha az adım.
- **Sezgi:** Euler bir kez türev alıp “düz çizgide” ilerler; Heun varış noktasında türevi yeniden hesaplayıp iki türevin ortalamasını kullanır (predictor-corrector). Bu ekstra hesap, eğimin -1.03 'ten -2.03 'e çıkmasını sağlar (figür) — yani hata adım sayısının karesiyle azalır, lineerle değil. Bir ekstra ağ çığrısının bedeli, çok daha hızlı yakınsamayla fazlasıyla geri ödenir.

31.13 12. LMS Sampler (Lineer Çok-Adımlı)

LMS (Linear Multistep) örnekleyici, önceki birkaç adımın türevlerini saklayıp ağırlıklı birleştirir (Adams-Bashforth gibi). Geçmiş bilgiyi kullanarak daha yüksek dereceli doğruluk sağlar — Karras çerçevesindeki güçlü örnekleyicilerden biri:



Şekil 31.7: (gerçek hesaplama: Euler vs Heun yakınsama — log-log eğim -1.03 vs -2.03 (1. vs 2. derece); test ODE üzerinde, diffusion örnekleyicilere aynı ilke)

```
@torch.no_grad()
def sample_lms(model, steps=100, order=4, sigma_max=80.):
    x = torch.randn(sz).to(model.device)*sigma_max
    sigs = sigmas_karras(steps, sigma_max=sigma_max)
    ds = []
    for i in range(len(sigs)-1):
        denoised = denoise(model, x, sigs[i])
        d = (x-denoised)/sigs[i]
        ds.append(d)
        if len(ds) > order: ds.pop(0)
        # geçmiş türevleri ağırlıklı birleştir (Adams-Bashforth katsayıları)
    return x
```

💡 Builder Notu — LMS: Geçmiş Türevleri Bedava Kullan

- **Geriye (18.065):** Çok-adımlı yöntemler sayısal ODE'nin ileri konusu; geçmiş türevleri kullanmak hesap maliyetini artırmadan doğruluğu yükseltir.
- **Sezgi:** Heun bir adımda iki türev hesaplar (ekstra ağ çağrısı); LMS ise zaten hesaplanmış geçmiş türevleri saklayıp birleştirir — yeni ağ çağrısı gerektirmez. order=4 ile son dört türevin ağırlıklı toplamı (Adams-Bashforth katsayıları), yüksek dereceli doğruluğu neredeyse bedavaya getirir.

31.14 13. Sonuç: Az Adımda Yüksek Kalite

Bütün bu iyileştirmeler (cosine/sigma çizelge + preconditioning + Heun/LMS) birleşince: aynı eğitilmiş model, çok daha az adımda (örn. 100 yerine ~20-50) DDIM'den daha iyi FID verir. Karras çerçevesi, diffusion'ı pratik ve yüksek kaliteli kılan dönüm noktasıdır.

“a bit better than we had on 100 steps for our previous DDIM.” — John, 9:14

💡 Builder Notu — Az Adımda Yüksek Kalite: Üretimin Anahtarı

- **İleriye (Ders 25):** Bu örnekleyiciler, latent diffusion (Stable Diffusion) ve modern üretimde standart; az adımda yüksek kalite, gerçek-zamanlı üretimin anahtarı.
- **Sezgi:** Üç iyileştirme bağımsız ama birbirini güçlendirir: cosine/sigma daha iyi çizelge verir, scalings ağı her σ 'da istikrarlı tutar, Heun/LMS az adımda doğru entegrasyon sağlar. Ders 21'de DDIM 100 adımda iyiydi; Karras ~20-50 adımda daha iyi FID — örnekleme maliyetini düşürmek, üretimi pratik kılar.

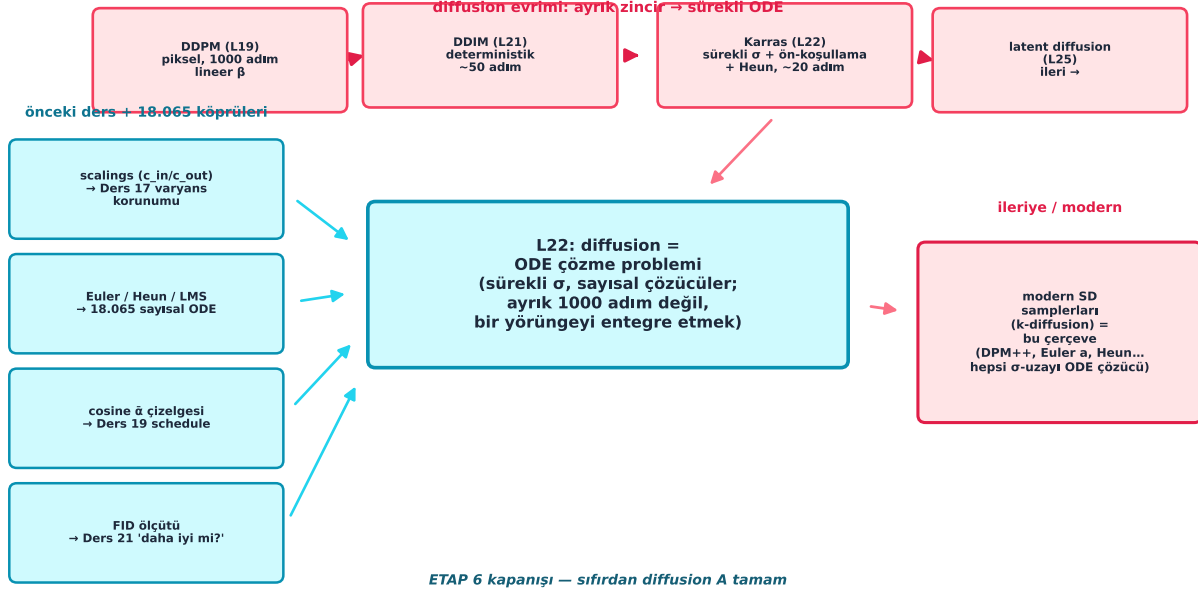
31.15 14. ETAP Kapanışı: Sıfırdan Diffusion A Tamamlandı

Ders 22, “sıfırdan diffusion A” blogunu (L19-22) tamamlar. DDPM'i kurduk (L19), hızlandırdık ve eğitimi iyileştirdik (L20), ölçtük ve daha da hızlandırdık (L21 FID/DDIM), şimdi en rafine hâline getirdik (L22 cosine/Karras). Diffusion artık baştan sona bizim — matematiğinden örnekleyicisine.

Şekil 31.8 bu kapanışı dersin sentezi olarak çerçeveler: merkezde çekirdek fikir — **L22: diffusion = ODE çözme problemi** (sürekli σ , sayısal çözücüler; ayrı 1000 adım değil, bir yörüngeyi entegre etmek). Üstte diffusion evrim zinciri: DDPM (L19, piksel, 1000 adım, lineer β) → DDIM (L21, deterministik, ~50 adım) → Karras (L22, sürekli σ + ön-koşullama + Heun, ~20 adım) → latent diffusion (L25, ileri). Solda köprüler: scalings → Ders 17 varyans korunumu, Euler/Heun/LMS → 18.065 sayısal ODE, cosine → Ders 19 schedule, FID ölçütü → Ders 21 “daha iyi mi?”. Sağda ileriye: modern SD samplerları (k-diffusion, DPM++, Euler a, Heun...) hepsi σ -uzayı ODE çözücüleridir — bu çerçevenin türevleri. Altta: ETAP 6 kapanışı, sıfırdan diffusion A tamam.

❗ Builder Notu — ETAP 6 Kapanışı: Diffusion = ODE (MERKEZİ)

- **Çekirdek ders:** Bu dersten tek bir şey alıp gideceksen: **diffusion bir ODE çözme problemidir.** Gürültüyü sürekli bir σ ile parametreleyip (ayrık timestep ve çizelgeyi kaldırarak), örneklemeyi bir yörüngeyi entegre etmek olarak görünce, sayısal analizin onlarca yıllık çözücüler (Euler, Heun, LMS, Runge-Kutta, Adams-Bashforth) doğrudan miras alınır.
- **İki büyük köprü:** (1) scalings = Ders 17 varyans korunumu (ağı her σ 'da birim varyans görür, Kaiming'in diffusion'a taşınmış hâli); (2) örnekleyiciler = 18.065 sayısal ODE (DDIM bile aslında bir Euler çözücüsü). Bir problemi tanıdık bir çerçeveye oturtmak, o çerçevenin tüm araç kutusunu kazandırır.
- **İleriye (Ders 23-25, SON ETAP):** Temel diffusion tam; ETAP 7 (super-res, attention, latent diffusion) bu temeli gerçek Stable Diffusion'a taşır. Karras örnekleyicileri elimizde, sıradaki uygulamalara hazırız.



Şekil 31.8: (sentez: L22 = diffusion'ın ODE olarak yeniden çerçevelenişi — DDPM→DDIM→Karras evrimi, Ders 17/19/21 + 18.065 köprüleri; ETAP 6 kapanır)

31.16 15. Kapanış

Ders 22 diffusion'ı en rafine hâline getirdi: **cosine schedule** (sürekli, daha iyi çizelge), gürültü **seviyesi tahmini** ve **Karras** çerçevesi (sürekli σ + preconditioning/scalings + Euler/Heun/LMS örnekleyiciler). Diffusion'ı bir ODE çözme problemi olarak gördük; daha iyi çözümler az adımda yüksek kalite verdi. “Sıfırdan diffusion A” tamamlandı.

💡 Builder Notu — Kapanış: Rafine Diffusion, SON ETAP'ın Zemini

- **İleriye (Ders 23):** Karras örnekleyicileri elimizde; SON ETAP super-resolution ile başlayıp attention ve latent diffusion'la gerçek Stable Diffusion'a ulaşır.
- **Sezgi:** ETAP 6 boyunca diffusion'ı kara kutudan baştan sona bizim olan bir sisteme çevirdik: matematiği (L19), eğitimi (L20), ölçümü ve hızı (L21), rafine çizelge/örnekleyiciyi (L22). Sıradaki ETAP artık “nasıl çalışır”ı değil, “ne yapabiliriz”i (super-res, koşullu üretim, latent) soracak.

31.17 Bu Dersin Özeti

1. **Cosine schedule:** $(t) = \cos(t \cdot \pi/2)^2$; lineer β 'dan daha iyi, sürekli gürültü dağılımı ($t=0.5$ 'te 0.499 vs lineer 0.078) (**cosine**).
2. **Sürekli zaman:** Ayrık 1000 adım yerine $t \in [0,1]$; diffusion'ı sürekli sürece taşır (**sürekli zaman**).
3. **Gürültü seviyesi tahmini:** Ağ timestep'i elle almak yerine gürültü düzeyini çıkarsayabilir; gürültü anlayışının testi (kapalı-form kanıt %8.2 hata) (**gürültü tahmini**).

4. **Karras çerçevesi:** Diffusion'ı sürekli σ (gürültü seviyesi) ile parametreler; çizelgeyi kaldırır (Karras, sürekli sigma).
5. **Preconditioning (scalings):** $c_{in}/c_{out}/c_{skip}$ ile ağ her σ 'da birim varyans görür (std 0.993–1.006); Ders 17 varyans korunumu (scalings).
6. **denoise:** Ağ temiz görüntüyü tahmin eder (scalings ile); gürültü-tahminine denk ama daha kararlı (denoise).
7. **Sigma schedule + ρ :** $\sigma_{max} \rightarrow \sigma_{min}$ adımları $\rho=7$ ile dağıtılır; düşük gürültüye daha çok adım ($\rho=7 \rightarrow \%36$ vs $\rho=1 \rightarrow \%2$) (sigma schedule).
8. **ODE örnekleyiciler:** Euler (1. derece, eğim -1.03), Heun (2. derece, eğim -2.03 , predictor-corrector), LMS (çok-adımlı); az adımda yüksek kalite (Euler, Heun, LMS).

! Tek Bir Cümle

Cosine schedule gürültü çizelgesini sürekli ve daha dengeli yapar ($(t) = \cos(t \cdot \pi/2)^2$); Karras ise diffusion'ı sürekli gürültü seviyesi σ + preconditioning (scalings, ağ her σ 'da birim varyans görsün) + daha iyi ODE örnekleyiciler (Euler/Heun/LMS) çerçevesinde yeniden kurarak, diffusion'ı bir ODE çözme problemi olarak görüp az adımda DDIM'den yüksek kalite verir.

31.18 Kontrol Soruları

i Soru 1: Cosine schedule lineer β çizelgesinden neden daha iyidir?

Cevap:

Lineer β çizelgesinde (Ders 19) gürültü, zaman boyunca sabit hızla artar; bu, eğitim adımlarının çoğunun “neredeyse tamamen gürültülü” bölgede harcanmasına yol açar — orada öğrenilecek pek bir şey yoktur. **Cosine schedule** birikimli 'yı doğrudan bir kosinüs karesiyle tanımlar: $(t) = \cos(t \cdot \pi/2)^2$. Bu, gürültülenmeyi başta yavaş, ortada hızlı, sonda yine yavaş yapar — modelin zamanını faydalı gürültü seviyelerine daha dengeli dağıtır. Somut fark Şekil 31.2'de: $t = 0.5$ 'te cosine = 0.499 (sinyalin yarısı korunmuş), lineer = 0.078 (sinyalin %92'si çoktan yok). Ayrıca cosine sürekli bir $t \in [0,1]$ alır (ayrık adım sayısına bağlı değil) ve kapalı formüllüdür (daha az hiperparametre). Sonuç: daha iyi eğitilmiş model, daha yüksek örnek kalitesi.

i Soru 2: Karras'ın 'sürekli sigma' parametrelemesi DDPM'in timestep'inden nasıl farklıdır?

Cevap:

DDPM (Ders 19) gürültüyü **ayrık bir timestep** ($t = 0..999$) ve buna bağlı bir $\beta/$ çizelgesiyle tanımlar — ağ “hangi adımdayız” bilgisini alır. **Karras** bunun yerine gürültüyü doğrudan **sürekli bir standart sapma σ** ile parametreler: gürültülü görüntü = temiz + σ ·gürültü. Artık çizelge ($\beta,$) yoktur; tek soru “şu anki gürültü seviyesi (σ) ne kadar”. Bu daha sadedir (bir hiperparametre kümesi kalkar), daha esnektir (herhangi bir σ kullanılabilir) ve diffusion'ı sürekli bir matematiksel nesneye — bir diferansiyel denkleme (ODE) — dönüştürür. Bu ODE bakışı, Euler/Heun gibi sayısal çözücülerle örneklemeyi mümkün kılar (Euler, Heun).

i Soru 3: Karras'taki 'scalings' (preconditioning) ne işe yarar ve hangi önceki derse bağlanır?

Cevap:

scalings (c_{in} , c_{out} , c_{skip}), ağın **her gürültü seviyesinde birim varyanslı girdi ve çıktı** görmesini sağlar. Sorun: σ çok değişken (0.01'den 80'e); ağ böyle farklı ölçeklerdeki girdilerle iyi çalışamaz. Çözüm: c_{in} girdiyi normalize eder (ağ hep ~birim varyans görür), c_{out} çıktıyı uygun ölçeğe taşır, c_{skip} girdinin ne kadarının doğrudan atlanacağını (skip) belirler — hepsi σ ve veri varyansına ($\text{sig\_data} = 0.66$) göre hesaplanır. Şekil 31.5 bunu kanıtlar: $c_{in} \cdot \text{noised standart sapması}$ her σ 'da ≈ 1.0 (aralık 0.993–1.006), σ dört merteye değişse de. Bu **doğrudan Ders 17'nin varyans korunumu ilkesidir** (Kaiming init / BatchNorm): ağ hep iyi-ölçeklenmiş değerlerle çalışsın ki eğitilebilsin (**scalings**). Yani Karras preconditioning'i, "varyansı sabit tut" fikrinin diffusion'a uyarlanmış hâlidir — foundations'ta (L17) öğrendiğimiz şeyin meyvəsi.

i Soru 4: Diffusion örneklemesini bir ODE çözme problemi olarak görmek ne kazandırır? (builder bağlantısı)

Cevap:

DDPM örneklemesi (Ders 19) ayrık, olasılıksal bir süreç gibi görünüyordu — 1000 küçük rastgele adım. Karras bunu **sürekli bir adi diferansiyel denklem (ODE)** olarak yeniden çerçeveler: görüntünün σ (gürültü seviyesi) boyunca nasıl evrildiğini bir türev tanımlar, örnekleme bu ODE'yi σ_{max} 'tan 0'a çözmektir. Bu bakış güçlüdür çünkü **sayısal analizin tüm araç kutusunu** açar: en basiti **Euler** (DDIM aslında budur), daha iyisi **Heun** (2. derece, predictor-corrector — bir adım atıp varışta düzeltir), daha da iyisi **LMS** (çok-adımlı, geçmiş türevleri kullanır). Daha yüksek dereceli çözücü = adım başına daha doğru = **çok daha az adımda yüksek kalite** (100 yerine ~20-50). Şekil 31.7 bunu sayısal kanıtlar: Euler yakınsama eğimi -1.03 (1. derece), Heun -2.03 (2. derece) — $n=20$ adımda Heun $\sim 55\times$ daha doğru. Builder dersi: bir problemi tanıdık bir matematiksel çerçeveye (ODE) oturtmak, o çerçevenin onlarca yıllık çözüm yöntemini (Runge-Kutta, Adams-Bashforth) miras almanı sağlar — 18.065'in diffusion'a doğrudan uygulanması.

31.19 Egzersizler

Egzersiz 1 (Direkt uygulama). $\text{abar}(t) = \cos(t \cdot \pi/2)^2$ cosine çizelgesini çiz; lineer ile karşılaştır, gürültülenmenin nasıl farklı dağıldığını gör ($t=0.5$ 'te 0.499 vs 0.078) (§1).

Egzersiz 2 (İki-aşamalı). sigmas_karras 'ı farklı ρ değerleriyle (1, 3, 7) çiz; ρ 'nun adımları gürültü seviyeleri arasında nasıl dağıttığını gözle ($\sigma < 1$ oranı $\%2 \rightarrow \%20 \rightarrow \%36$) (§9).

Egzersiz 3 (Edge case). sample_euler ve sample_heun 'u aynı adım sayısı ile çalıştır, FID'lerini karşılaştır; Heun'un neden daha iyi olduğunu açıkla (2. derece, predictor-corrector) (§10, §11).

Egzersiz 4 (Kavramsal). scalings ($c_{in}/c_{out}/c_{skip}$) neden gereklidir? σ çok değişkenken ağ neden birim varyanslı girdi görmeli (Ders 17 bağlantısı)? (§7)

Egzersiz 5 (Sonraki dersin habercisi — Ders 23). Diffusion'ı koşullu hâle getirmek (örn. düşük çözünürlüklü görüntüden yüksek çözünürlük) nasıl mümkün olur? Super-resolution'ın bir diffusion problemi olduğunu düşün.

31.20 Sonraki: Ders 23 İçin Hazırlık

Ders 23: Super-Resolution (SON ETAP başlangıcı)

Ders 22 diffusion'ı en rafine hâline getirdi (cosine + Karras). ETAP 7 (SON ETAP, L23-25) bu temeli gerçek uygulamalara taşır. Ders 23 **super-resolution** ile başlar: düşük çözünürlüklü bir görüntüden yüksek çözünürlüklü üretmek — koşullu diffusion'ın güzel bir örneği.

Ana konular (Ders 23):

- Koşullu diffusion (düşük-çöz → yüksek-çöz)
- U-Net'i koşturma
- Perceptual/feature loss
- Üretim kalitesi değerlendirme (FID ile)

⚠ Ders 23 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 3 — cosine + Euler/Heun).
- Karras örnekleyicilerle (Heun) DDPM modelini ~30 adımda örnekle, FID ölç.
- Ders 21 FID ve Ders 22 Karras örnekleyicileri tekrar gözden geçir.

31.21 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Cosine schedule	$(t) = \cos(t \cdot \pi/2)^2$; sürekli, dengeli çizelge	0:35
Sürekli zaman	Ayrık adım yerine $t \in [0,1]$	5:06
Gürültü seviyesi tahmini	Ağ timestep'i değil gürültü düzeyini çıkarsar	19:03
Karras	"Elucidating the Design Space"; birleştirici çerçeve	38:48
Sürekli σ	Gürültü = temiz + $\sigma \cdot \text{noise}$; çizelgesiz parametre	38:48
scalings	$c_{\text{in}}/c_{\text{out}}/c_{\text{skip}}$; ağ her σ 'da birim varyans görür	30:30
denoise	Ağ temiz görüntüyü tahmin eder (scalings ile)	—
Sigma schedule (ρ)	$\sigma_{\text{max}} \rightarrow \sigma_{\text{min}}$, $\rho=7$ ile adım dağılımı	1:07:25
Euler sampler	1. derece ODE çözücü (DDIM'e denk)	—
Heun's method	2. derece ODE (predictor-corrector); az adımda iyi	1:15
LMS sampler	Çok-adımlı (geçmiş türevler); yüksek derece	—

Kavram	Tanım	Howard'da
Diffusion = ODE	Örnekleme bir ODE çözme; sayısal çözücüler uygulanır	—

31.22 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: Çizelge, Varyans ve Sayısal ODE

Bu ders üç iyileştirmeyi (cosine, gürültü tahmini, Karras) önceki derslerin altyapısına bağlar ve “diffusion = ODE” felsefesini ortaya koyar; köprülerin özeti:

1. **Cosine schedule** → Ders 19 çizelgesi; sürekli, kapalı formüllü iyileştirme ([cosine](#)).
2. **Karras scalings** → Ders 17 varyans korunumu (Kaiming/BatchNorm); ağ her σ 'da birim varyans ([scalings](#)).
3. **Diffusion = ODE** → 18.065 sayısal ODE (Euler/Heun/Runge-Kutta); örnekleme = ODE çözme ([Euler](#)).
4. **Heun/LMS** → 18.065 yüksek dereceli çözücüler; az adımda yüksek doğruluk ([Heun](#), [LMS](#)).
5. **Sürekli σ** → Ders 21 DDIM'in deterministik ODE bakışının genellemesi ([sürekli sigma](#)).
6. **denoise (temiz görüntü tahmini)** → Ders 19 ϵ -tahmini ile denk; sayısal olarak daha kararlı ([denoise](#)).

❗ Bu dersten tek bir şey alıp gideceksen

Bu dersten tek bir şey alıp gideceksen: diffusion bir **ODE çözme problemidir**. Karras, gürültüyü sürekli bir σ ile parametreleyip (ayrık timestep ve çizelgeyi kaldırarak), ağa preconditioning (scalings) ile her σ 'da birim varyans gösterip (Ders 17'nin varyans korunumu — figürde $c_{in-noised}$ std 0.993–1.006), ve örnekleme bir ODE olarak görüp sayısal analizin güçlü çözücülerini (Euler, Heun, LMS — Heun yakınsama eğimi -2.03 , Euler -1.03) uygulayarak diffusion'ı en rafine hâline getirdi. Sonuç: çok daha az adımda yüksek kalite. “Sıfırdan diffusion A” tamamlandı; sıradaki SON ETAP gerçek Stable Diffusion'a ulaşacak.

32 Ders 23 — Super-Resolution (Süper Çözünürlük)

SON ETAP'ın (ETAP 7) açılışı — Howard, Jonathan Whitaker (Johno) ve Tanishq Abraham'la. Düşük çözünürlüklü bir görüntüden yüksek çözünürlük üretmek; bu, üretimi bir girdiye koşullamanın (conditional generation) en sade örneği ve koşullu diffusion'a doğrudan köprü. Anahtar ders şu: piksel MSE keskin görüntü vermez (bulanıklaştırır), çünkü tek bir düşük çözünürlüklü girdiye karşılık birçok geçerli keskin çıktı vardır ve MSE bunların ortalamasını ödüllendirir. Çözüm, Ders 20'den tanıdığımız perceptual (feature) loss — çıktıyı ve hedefi önceden eğitilmiş bir ağıın özellik uzayında karşılaştırmak; bu, style transfer'ın content loss'unun birebir aynısıdır. Üç temel fikir: super-res = koşullu üretim (bir U-Net düşük→yüksek dönüşümünü öğrenir), pixel loss bulanıklaştırır, perceptual loss keskinliği geri kazandırır. Model sıfırdan kurulan bir U-Net (TinyUnet): L18'in ResBlock'larından + L19'un U-Net yapısından. Dersin builder içgörüsü tek cümlede: doğru loss doğru sonucu belirler — «neyi optimize ettiğin» (loss) çoğu zaman «nasıl optimize ettiğinden» (mimari) kritiktir. SON ETAP burada başlar; sıradaki attention (L24) ve text→image (L25) ile gerçek Stable Diffusion'a yaklaşıyoruz.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 23: Super-resolution](https://course.fast.ai/lesson/23) (~101 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 23
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/25_superres](#) (+ Tiny ImageNet U-Net)
- **Okuma süresi:** ~38 dk
- **Hocalar:** Howard + Jonathan Whitaker (Johno) + Tanishq Abraham
- **SON ETAP (ETAP 7) açılışı:** ETAP 6'da diffusion'ı baştan sona kurduk — DDPM (L19), iyileştirilmiş eğitim (L20), FID/DDIM (L21), cosine/Karras (L22). Artık temel diffusion bizim. SON ETAP artık “nasıl çalışır”ı değil “ne yapabiliriz”i soruyor: super-res (L23) → attention (L24) → latent diffusion (L25). Ders 23, koşullu üretimin en sade örneğiyle bu kapıyı açar.

32.1 Bu Derste Ne Var?

SON ETAP'ın açılışı: **super-resolution** — düşük çözünürlüklü bir görüntüden yüksek çözünürlüklü üretmek. Bu, üretimi bir girdiye **koşullamanın** (conditional generation) ilk örneğidir ve koşullu diffusion'a doğrudan köprüdür. Anahtar ders: piksel MSE keskin görüntü vermez (bulanıklaştırır); çözüm, Ders 20'den tanıdığımız **perceptual (feature) loss** — çıktıyı özellik uzayında karşılaştırmak.

Üç temel fikir bu dersin omurgasını kurar:

1. **Super-resolution = koşullu üretim** — çıktı, düşük çözünürlüklü girdiye koşullanır; bir U-Net düşük→yüksek dönüşümünü öğrenir (**super-res nedir, koşullu üretim**).
2. **Pixel loss bulanıklaştırır** — MSE birçok olası keskin çıktının ortalamasını alır, sonuç pürüzlü; keskinlik için yetersiz (**pixel loss**).
3. **Perceptual loss** — çıktı ve hedefi önceden eğitilmiş bir ağdan geçirip **özellik uzayında** karşılaştırır; doku ve keskinliği geri kazandırır (**perceptual loss**).

“we want to use perceptual loss. And I think it was John who taught us about perceptual loss.” — Howard, 1:15:00

Şekil 34.1 bu üç fikri tek bir kavram haritasında birleştirir: super-res (düşük çözünürlük → yüksek çözünürlük; eksik detay uydurulur = üretken) → koşullu üretim (çıktı girdiye koşullanır) → TinyUnet (ResBlock aşağı + yukarı + çapraz bağlantılar) → loss sorunu (pixel MSE = geçerli çıktılarının ortalaması = bulanık) → perceptual loss (özellik uzayında karşılaştır, cmodel) → comb_loss (pixel + özellik/10) → koşullama L24 attention ve L25 text→image kapısını açar. Rose ile işaretli düğümler (koşullu üretim, perceptual loss, attention) dersin iki çekirdek mesajını taşır: koşullama ve özellik uzayında karşılaştırma.

💡 Builder Notu — Giriş: SON ETAP, Koşullama ve Perceptual Loss

- **Geriye (Ders 20) — köprü:** Perceptual loss, Ders 20'deki style transfer'in content loss'unun birebir aynısı — özellik uzayında MSE. Aynı fikir, yeni problem.
- **Geriye (Ders 18-19):** Super-res U-Net (TinyUnet), L18'in ResBlock'larından + L19'un U-Net yapısından kurulu.
- **İleriye (Ders 24-25):** Super-res = koşullu üretim; koşullama, latent diffusion'ın (L25) text→image'ının temeli; attention (L24) bu koşullamayı güçlendirir.
- **Tek cümle:** Super-resolution, bir U-Net'i düşük çözünürlüklü girdiye koşullayarak yüksek çözünürlük üretir; keskinlik için piksel MSE değil, özellik uzayında perceptual loss kullanılır.

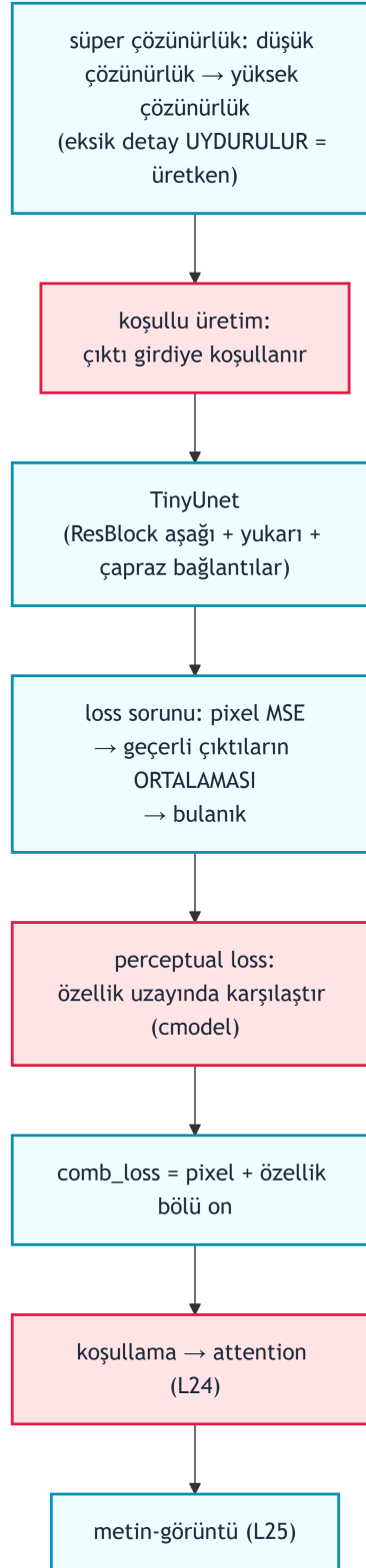
32.2 1. Super-Resolution Nedir?

Super-resolution, düşük çözünürlüklü (örn. 32×32) bir görüntüyü alıp yüksek çözünürlüklü (örn. 64×64) versiyonunu üretmektir. Eksik detayları “uydurmak” gerekir — bu yüzden bir üretken problemdir. Çıktı düşük çözünürlüklü girdiye **koşullanır**: model serbestçe üretmez, verilen görüntüyü büyütür.

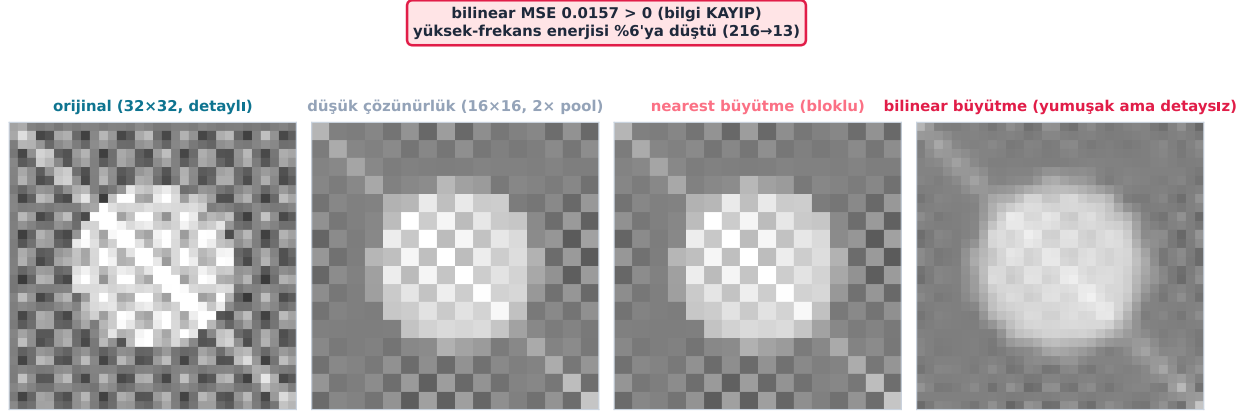
Buradaki incelik küçültme sırasında bilginin **gerçekten kaybolmasıdır**. Bir 2× pool, ince dokuları ve kenarları siler; geriye dönüş bilinear interpolasyonla yapılamaz, çünkü interpolasyon var olanı yayar, yeni detay üretmez. Şekil 32.2 bunu gerçek hesaplamayla gösterir: 2× downsample sonrası yüksek-frekans (HF) enerjisi orijinalin yalnızca %6'sına düşer (216→13), bilinear geri büyütmenin görüntüye karşı MSE'si **0'ın üstündedir** (bilgi kayıp). Nearest büyütme bloklu, bilinear yumuşak ama detaysız — ikisi de kaybolan dokuyu geri getiremez. Demek ki super-res, modelin öğrendiği görüntü dağılımından makul detay **üretmesini** ister.

💡 Builder Notu — Super-Res Bir Üretken Problemdir

- **İçgörü:** Küçültme bilgiyi yok eder (HF enerji %6'ya düşer); kaybolanı geri getirmek = makul detayı uydurmak = üretken problem. Basit büyütme (nearest/bilinear) detay üretmez, var olanı yayar.



Şekil 32.1: L23 kavram haritası: süper çözünürlük = koşullu üretken bir problem; pixel MSE bulanıklığı perceptual loss ile çözülür, koşullama L24 attention ve L25 metin-görüntü kapısını açar (şematik).



Şekil 32.2: (gerçek hesaplama: super-res problemi — 2× downsample bilgiyi yok eder (HF enerji %6'ya), bilinear geri getiremez; eksik detay üretilmeli)

- **İleriye (Ders 25):** “Çıktıyı bir girdiye koşullama” fikri, latent diffusion’da metni (text→image) koşullamanın temeli; super-res en sade koşullu üretim örneği.

32.3 2. Tiny ImageNet’e Geçiş

Fashion-MNIST super-res için fazla basit (zaten net). Howard daha zengin, gerçek görüntülere — **Tiny ImageNet**’e — geçer. Bu, perceptual loss’un farkını gerçekten görebilmek için gerekli: gerçek dokular, kenarlar, detaylar. MNIST’te bir karakterin keskin/bulanık farkı zaten zor görülür; gerçek doku üzerinde perceptual loss’un kazandırdığı keskinlik çıplak gözle ayırt edilebilir.

“we’re going to move on to Tiny Imagenet... I want to show [the difference].” — Howard, 6:06

💡 Builder Notu — Gerçek Veri: Foundations + Uygulama Birleşir

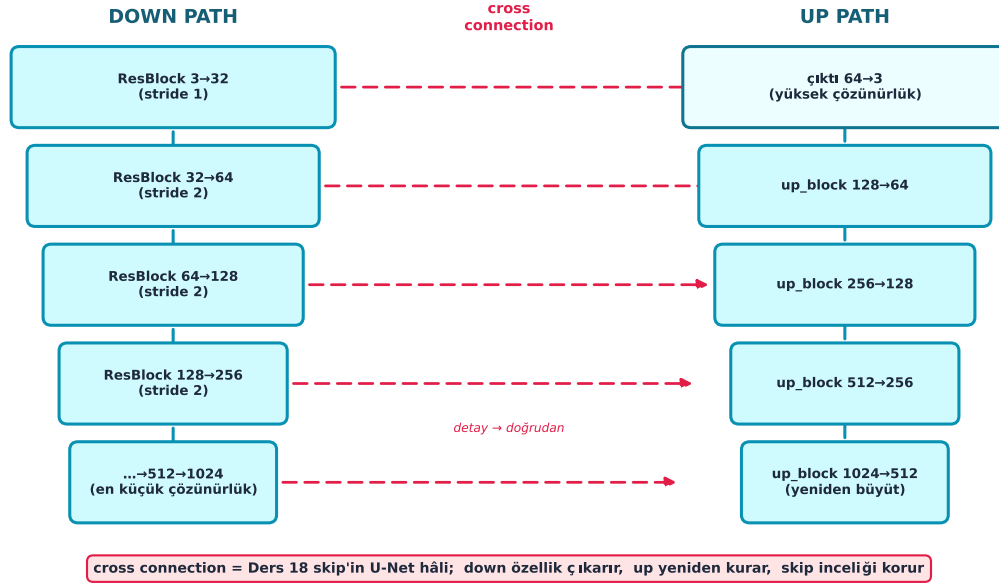
- **Geriye (Ders 1-8):** Gerçek görüntü verisiyle çalışmak Part 1’in DataBlock/DataLoaders altyapısını kullanır; foundations + uygulama birleşir.
- **Sezgi:** Bir tekniğin değerini görmek için onu zorlayacak veri gerekir. MNIST’in keskin/bulanık farkı görünmez; Tiny ImageNet’in gerçek dokusu perceptual loss’un katkısını gözle ölçülebilir kılar.

32.4 3. Super-Res U-Net (TinyUnet)

Model bir **U-Net**’tir, sıfırdan kurulur (TinyUnet): bir aşağı yol (down path, ResBlock’larla küçültürük özellik çıkarır) ve bir yukarı yol (up path, büyütürük görüntüyü yeniden kurar). L18’in ResBlock’u temel yapıtaşdır.

```
class TinyUnet(nn.Module):
    def __init__(self, act=act_gr, nfs=(32,64,128,256,512,1024), norm=nn.BatchNorm2d):
        super().__init__()
        self.start = ResBlock(3, nfs[0], stride=1, act=act, norm=norm)
        self.dn = nn.ModuleList([ResBlock(nfs[i], nfs[i+1], act=act, norm=norm, stride=2)
                                for i in range(len(nfs)-1)])
        self.up = nn.ModuleList([up_block(nfs[i], nfs[i-1]) for i in range(len(nfs)-1, 0, -1)])
```

Şekil 32.3 mimariyi şematik gösterir: solda DOWN PATH (stride-2 ResBlock'larla 3→32→64→128→256→512→1024, uzamsal olarak küçülerek özellik çıkarır), sağda UP PATH (up_block'larla 1024→512→...→3, büyütür yüksek çözünürlüğü yeniden kurar), ortada çapraz bağlantılar (cross connection). U-Net = encoder-decoder + skip; down özellik çıkarır, up yeniden kurar, skip inceliği korur.



Şekil 32.3: (şematik: TinyUnet — ResBlock down path + up path + cross connections; Ders 18 ResBlock + Ders 19 U-Net mirası)

💡 Builder Notu — TinyUnet = Ders 18 ResBlock + Ders 19 U-Net

- **Geriye (Ders 18):** Aşağı yol = stride-2 ResBlock'lar (L18); yukarı yol bunları tersine çevirir. U-Net = encoder-decoder + skip.
- **Geriye (Ders 19):** Diffusion U-Net'inin yapısı burada yeniden kullanılır; aynı encoder-decoder iskeleti, yeni görev (super-res).

32.5 4. Cross Connections (U-Net Skip)

U-Net'in gücü **cross connection**'lardadır: aşağı yoldaki her seviyenin çıktısı, yukarı yolda aynı seviyeye eklenir (skip). Böylece ince detaylar (down path'te kaybolan) up path'e doğrudan taşınır. Bu, L18'in skip

connection'ının U-Net mimarisindeki hâlidir.

Sezgi şu: down path görüntüyü sıkıştırırken yüksek-frekans detayları (kenarlar, dokular) eritir; up path bunları sıfırdan yeniden üretmek zorunda kalsa zorlanırdı. Cross connection, down path'in henüz erimemiş erken-seviye özelliklerini up path'in karşılık gelen seviyesine doğrudan köprüler — detay yeniden icat edilmek yerine taşınır.

💡 Builder Notu — Cross Connection = Skip'in U-Net Hâli

- **Geriye (Ders 18):** Cross connection = skip connection out = $F(x) + x$ 'in U-Net versiyonu; gradyan ve detay akışını korur.
- **Sezgi:** Down path'te kaybolan ince detay, cross connection'la up path'e doğrudan taşınır — model onu yeniden üretmek zorunda kalmaz; bu, super-res kalitesinin temel taşı.

32.6 5. Pixel Loss'un Yetersizliği

İlk deneme: çıktı ile hedef arasında piksel MSE. Sorun: bir düşük çözünürlüklü girdiye karşılık birçok geçerli yüksek çözünürlüklü çıktı vardır; MSE bunların **ortalamasını** alır — sonuç bulanık, keskinlikten yoksun. Piksel benzerliği “iyi görüntü” demek değildir.

Şekil 32.4 bu FLAGSHIP içgörüyü gerçek hesaplamayla kanıtlar. Aynı düşük çözünürlüklü girdiye karşılık 50 geçerli keskin çıktı kurulur (kenar konumu örnekler arası kayar). MSE-optimal tahmin matematiksel olarak bu 50 çıktının **ortalamasıdır** — çünkü ortalama, tüm olasılıklara olan kare hatayı en aza indirir. Sonuç: tek bir keskin örneğin dik kenar basamağı, ortalamada yayvan bir rampaya dönüşür. Keskinlik ölçüsü $\max|\nabla x|$ **1.000'den 0.340'a** düşer — yani **2.9× kayıp**. Matematiksel olarak optimal olan çözüm, görsel olarak yanıltır: keskin desenler birbirini siler, bulanıklık kalır.

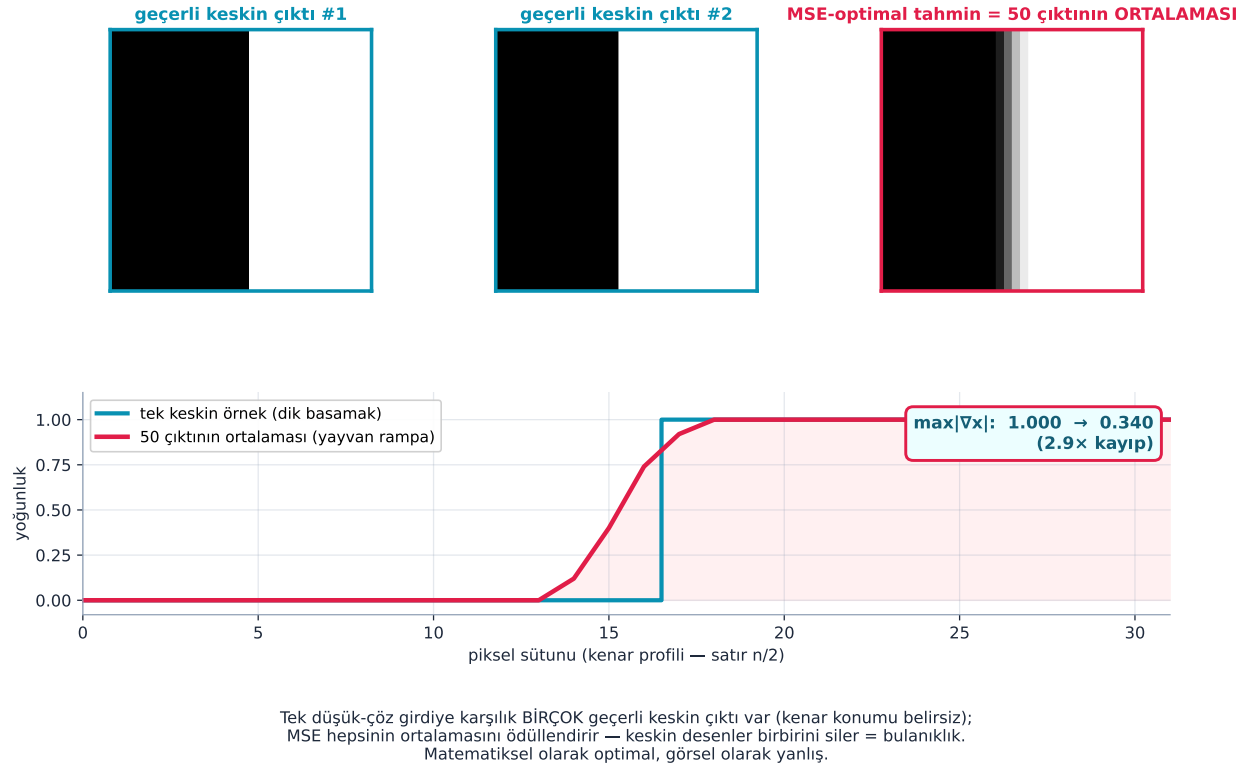
💡 Builder Notu — Pixel MSE Ortalamayı Ödüllendirir

- **İçgörü:** Tek girdiye karşılık çok geçerli keskin çıktı var; MSE hepsinin ortalamasını ödüllendirir (kare hatayı minimize eder) — keskin desenler birbirini siler, keskinlik 1.000→0.340 (2.9× kayıp).
- **Geriye (Ders 21):** Ders 21'de öğrendik — piksel karşılaştırması anlamsal kaliteyi yakalamaz; aynı sorun super-res'te bulanıklık olarak çıkar.

32.7 6. Perceptual (Feature) Loss

Çözüm **perceptual loss**: çıktıyı ve hedefi önceden eğitilmiş bir ağdan (içerik modeli) geçir, **özellik uzayında** karşılaştır. Özellikler dokuyu ve yapıyı kodlar; bu uzayda yakınlık, keskin ve gerçekçi çıktı demektir. Bu, Ders 20'deki style transfer'in content loss'unun **birebir aynısıdır**.

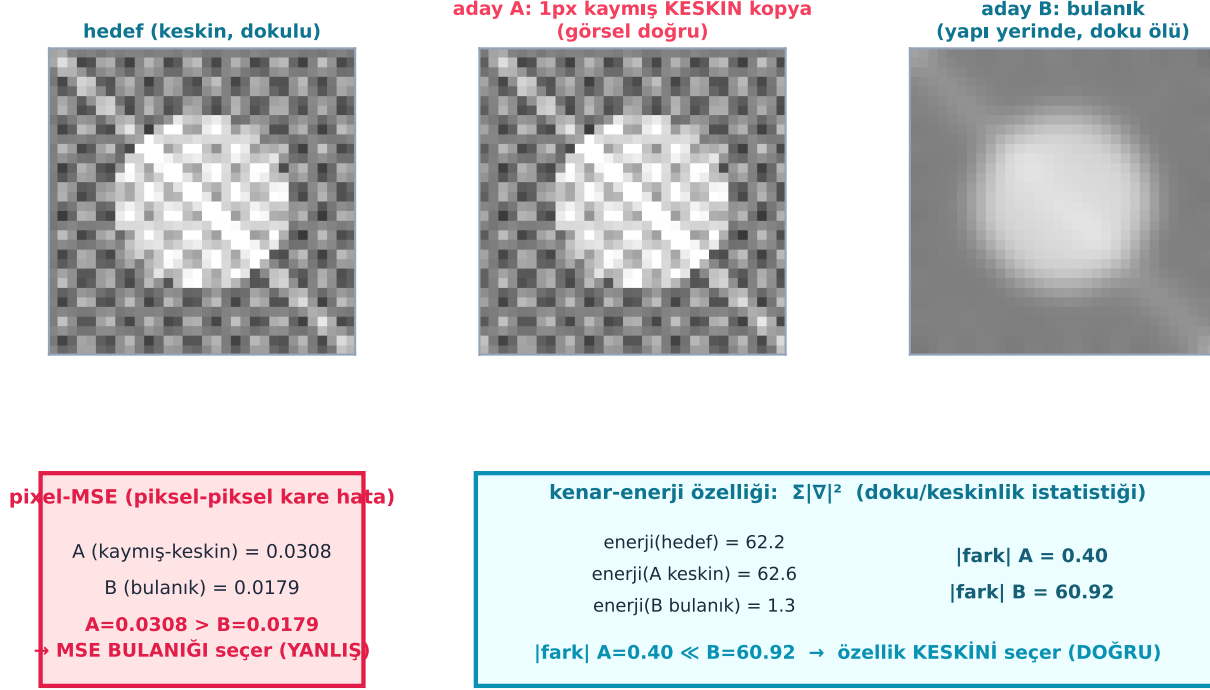
Şekil 32.5 ilkeyi gerçek hesaplamayla gösterir (özellik burada **elle** tanımlı — kenar enerjisi $\Sigma|\nabla|^2$ — ama ilke gerçek perceptual loss'la aynı). Üç görüntü: hedef (keskin, dokulu), aday A (1px kaymış keskin kopya — görsel olarak doğru), aday B (bulanık — yapı yerinde, doku ölü). Piksel MSE A'yı **0.0308**, B'yi **0.0179** ölçer; yani MSE bulanık B'yi seçer (**YANLIŞ**) — çünkü 1px kayma her piksele büyük hata yazar, bulanıklık ise hatayı yumuşatır. Kenar-enerji özelliği ise tersini yapar: hedefe özellik farkı A için **0.40**, B için **60.92** —



Şekil 32.4: Pixel-MSE neden bulanıklaştırır (gerçek hesaplama: pixel-MSE bulanıklığı — 50 geçerli keskin çıktının MSE-optimal ortalaması rampalaşır; keskinlik 1.000→0.340).

özellik uzayı keskin A'yı seçer (**DOĞRU**). İşte perceptual loss'un özü: karşılaştırmayı piksel piksel değil, özellik uzayında yap.

“we want to use perceptual loss.” — Howard, 1:15:00



Özellik burada ELLE tanımlı (kenar enerjisi $\sum |\nabla|^2$); gerçek perceptual loss öğrenilmiş ağ (cmodel) özellikleri kullanır — İLKE aynı: karşılaştırmayı özellik uzayında yap (= Ders 20 content loss, Ders 21 FID).

Şekil 32.5: (gerçek hesaplama: pixel-MSE kaymış-keskini cezalandırıp bulanığı seçer; özellik uzayı doğru sıralar — perceptual loss ilkesi, özellik elle)

💡 Builder Notu — Perceptual Loss = Ders 20 Content Loss (KÖPRÜ)

- **Geriye (Ders 20) — merkezî köprü:** Style transfer’da içeriği özellik uzayında ölçmüştük; super-res’te keskinliği aynı şekilde ölçüyoruz. **Perceptual loss = özellik uzayında MSE** — birebir aynı teknik.
- **Sezgi:** Pixel MSE 1px kaymış keskini cezalandırıp bulanığı seçer (0.0308 > 0.0179); özellik uzayı doğru sıralar (0.40 60.92). “İyi görüntü” tanımı piksellerde değil, özelliklerde yapılır.
- **Dürüstlük notu:** Figürdeki özellik **elle** (kenar enerjisi); gerçek perceptual loss öğrenilmiş ağ (cmodel) özelliklerini kullanır — ilke aynı.

32.8 7. cmodel: Özellik Modeli

Perceptual loss için bir **özellik modeli** (cmodel) gerekir — burada Tiny ImageNet’te özel eğitilmiş bir sınıflandırıcı. Çıktı ve hedef bu modelden geçirilip ara özellikleri karşılaştırılır. (Style transfer’da VGG16

kullanmıştık; burada veriye uygun özel bir model.)

```
cmodel = torch.load('models/inettiny-custom-25').cuda()
```

⚠ Güvenlik Builder Notu — torch.load (2022 kodu)

- Yukarıdaki çağrı Howard’ın **2022 hâli**; torch.load weights_only belirtmez. Eski PyTorch varsayılanında torch.load modeli **unpickle** eder — bu, dosyanın içine gömülmüş **keyfi kodu çalıştırabilir** (güvenilmeyen checkpoint = uzaktan kod yürütme riski).
- **Modern PyTorch (2.6+)** artık weights_only=True’yı varsayılan yapar; yalnızca tensör ağırlıklarını yükler, keyfi kodu çalıştırmaz. Eski sürümde bunu **elle** ver: torch.load(path, weights_only=True) (tam model nesnesi değil, state_dict yüklenir).
- **Pratik:** Güvenilmeyen kaynaktan gelen .pt/.pth dosyalarını asla weights_only=False ile açma; mümkünse state_dict + bilinen mimari kullan. (Bu not zenginleştirme katmanıdır; üstteki Notion kodu 2022 orijinali olarak korunmuştur.)

💡 Builder Notu — cmodel: Veriye Uygun Özellik Modeli

- **Geriye (Ders 20):** Style transfer VGG16 kullandı; super-res kendi veri kümesinde eğitilmiş bir model kullanır — özellikler göreve daha uygun.
- **Sezgi:** Özellikleri çıkaran ağ ne kadar veriye uygunsa, “iyi görüntü” tanımı o kadar isabetli olur; genel VGG yerine Tiny ImageNet sınıflandırıcısı bu görevin dokusunu daha iyi tanır.

32.9 8. comb_loss: Pixel + Feature

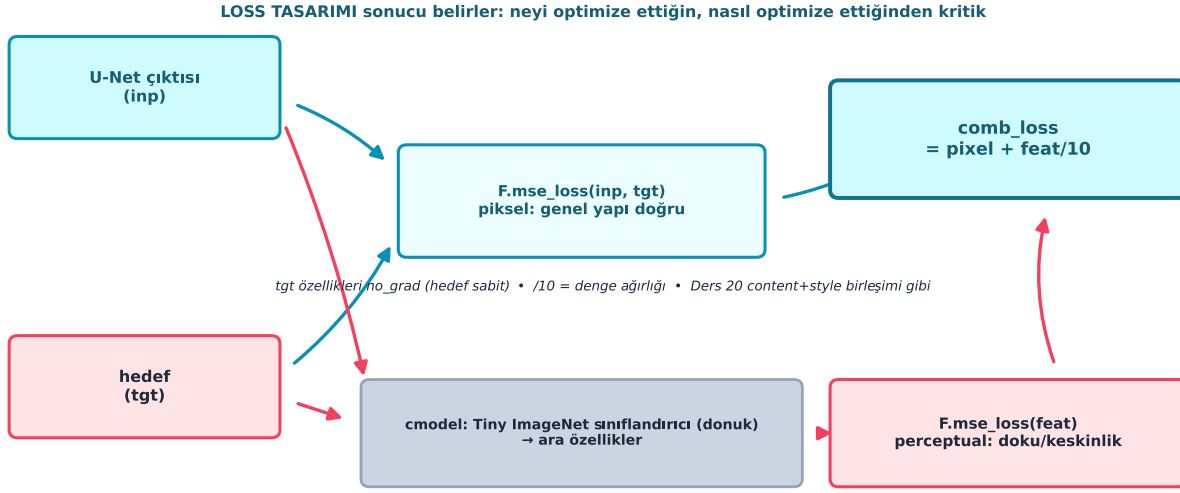
Pratik çözüm ikisini birleştirir: piksel MSE (genel yapı doğru olsun) + perceptual loss (keskin/gerçekçi detay). comb_loss çıktının ve hedefin özelliklerini karşılaştırır, piksel MSE ile toplar (ağırlıklı).

```
def comb_loss(inp, tgt):
    with torch.no_grad(): tgt_feat = cmodel(tgt).float()
    inp_feat = cmodel(inp).float()
    feat_loss = F.mse_loss(inp_feat, tgt_feat)
    return F.mse_loss(inp, tgt) + feat_loss/10
```

Şekil 32.6 bu birleşimi şematik gösterir: U-Net çıktısı (inp) ve hedef (tgt) iki yola girer — doğrudan piksel MSE (genel yapı doğru olsun) ve cmodel ara özellikleri üzerinden perceptual MSE (doku/keskinlik). İkisi comb_loss = pixel + feat/10 olarak birleşir. Hedef özellikleri no_grad altında (sabit), /10 denge ağırlığıdır — Ders 20’nin content+style birleşimi gibi. Mesaj: loss tasarımı sonucu belirler.

💡 Builder Notu — comb_loss: İki Loss’un Dengesi

- **Geriye (Ders 20):** İçerik + stil kaybını birleştirmek gibi; burada piksel + özellik kaybı. “Loss tasarımı” sonucu belirler.
- **Sezgi:** Piksel MSE genel yapıyı (renk, konum) doğru tutar, perceptual loss detayı keskinleştirir; /10 ağırlığı ikisini dengeler — yalnız perceptual artefakt üretebilir, yalnız piksel bulanıklaştırır.



Şekil 32.6: (şematik: $\text{comb\_loss} = \text{piksel MSE (yapı)} + \text{perceptual loss}/10$ (keskinlik); loss tasarımı = hedef tanımı)

32.10 9. Eğitim ve Keskinlik

comb_loss ile eğitilen super-res U-Net, saf piksel MSE’ye göre belirgin daha keskin çıktı verir. Perceptual loss, modelin “doğru piksel değeri” yerine “doğru görünen doku” üretmesini teşvik eder. Sonuç: gerçekçi, keskin yüksek çözünürlük. §5’teki bulanıklık (keskinlik 1.000→0.340) ile §6’daki doğru sıralama (özellik 0.40 60.92) birleştğinde, comb_loss ’un neden işe yaradığı netleşir: piksel terimi yapıyı, özellik terimi keskinliği tutar.

“we’re doing all this from scratch.” — Howard, 45:09

💡 Builder Notu — Keskinlik: Perceptual Loss’un Meyvesi

- **İçgörü:** comb_loss modeli “doğru görünen doku” üretmeye iter; saf piksel MSE’nin bulanıklığını aşar.
- **İleriye:** Aynı perceptual loss fikri, GAN’lar ve modern super-res/restorasyon modellerinde standarttır.

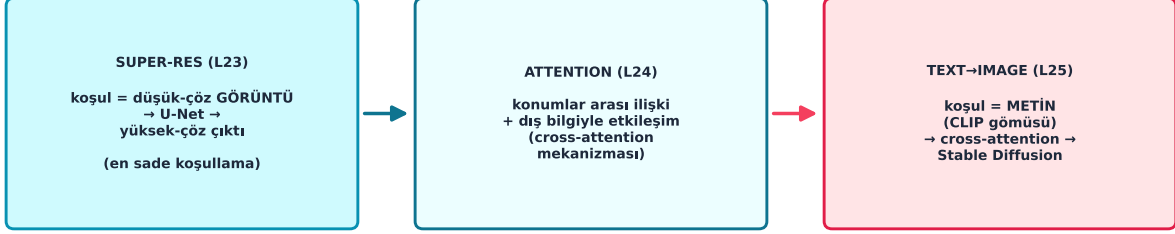
32.11 10. Super-Resolution = Koşullu Üretim

Daha derin bir bakış: super-res, çıktıyı bir girdiye (düşük çözünürlüklü görüntü) **koşullayan** bir üretken modeldir. Model serbestçe değil, verilen koşula göre üretir. Bu, koşullu üretimin (conditional generation) en sade örneğidir ve sonraki derslerin (text→image) temelidir.

Şekil 32.7 koşullamanın üç-aşamalı ilerlemesini şematik gösterir: SUPER-RES (L23, koşul = düşük-çöz görüntü → U-Net → yüksek-çöz, en sade koşullama) → ATTENTION (L24, konumlar arası ilişki + dış bilgiyle etkileşim, cross-attention) → TEXT→IMAGE (L25, koşul = metin/CLIP gömüsü → cross-attention

→ Stable Diffusion). Üstte vurgu: koşullama, üretken modeli **kontrol edilebilir** kılar (serbest üretimden işe yarar üretime). Altta: diffusion da koşullanabilir — L19 DDPM + koşul = SD upscaler / inpainting.

koşullama = üretken modeli **KONTROL EDİLEBİLİR** kılar (serbest üretimden → işe yarar üretime)



diffusion da koşullanabilir: L19 DDPM + koşul = SD upscaler / inpainting

Şekil 32.7: (şematik: koşullu üretim ilerlemesi — görüntüye koşullama (L23) → attention (L24) → metne koşullama (L25); SON ETAP'ın omurgası)

💡 Builder Notu — Koşullama: Kontrol Edilebilir Üretim

- **İleriye (Ders 24-25):** Koşullama mekanizması — bir girdiyi U-Net'e besleyip çıktıyı yönlendirmek — attention (L24) ve text conditioning (L25) ile genişler.
- **Sezgi:** Koşullama, üretken modeli “kontrol edilebilir” kılan mekanizmadır; serbest üretimden (DDPM koşulsuz örnekleme) işe yarar üretime (super-res, text→image) geçiş.

32.12 11. Diffusion'a Köprü: Koşullu Diffusion Super-Res

Super-res aynı zamanda bir **diffusion** problemi olarak da kurulabilir: düşük çözünürlüklü görüntüye koşullanmış bir diffusion modeli, gürültüden yüksek çözünürlük üretir. Bu, DDPM'i (L19) koşullu hâle getirmenin doğal yoludur ve modern super-res diffusion'ın (örn. SD upscaler) temelidir.

💡 Builder Notu — Koşullu DDPM: Super-Res, Inpainting, Text→Image

- **Geriye (Ders 19-22):** DDPM/Karras altyapısı koşullu hâle gelince super-res, inpainting, text→image hepsi mümkün; super-res en sade koşullama.
- **Sezgi:** L19'da kurduğumuz koşulsuz DDPM'e bir koşul (düşük-çöz görüntü) eklemek yeterli; aynı örnekleme döngüsü, koşula göre yönlendirilmiş üretim verir.

32.13 12. Builder İçgörüsü: Loss Tasarımı

Bu dersin builder dersi: **doğru loss, doğru sonucu belirler**. Piksel MSE “matematiksel olarak doğru” ama görsel olarak yanlış (bulanık) çıktı verir; perceptual loss insanın algıladığı kaliteyi hedefler. Üretken modellemede “neyi optimize ettiğin” (loss), “nasıl optimize ettiğinden” (mimari) çoğu zaman daha kritiktir.

Bu içgörü dersin merkezidir: aynı TinyUnet mimarisi, loss’a göre tamamen farklı çıktı verir — piksel MSE ile bulanık (keskinlik 1.000→0.340), comb_loss ile keskin. Mimari sabit, sonucu değiştiren loss tasarımı. Perceptual loss’un Ders 20 content loss’uyla birebir aynı olması da bunu pekiştirir: “neyi optimize edeceğin” sorusunun cevabı, görevden göreve taşınabilen güçlü bir araçtır.

💡 Builder Notu — Loss = Test Yazılabilir Hedef Tanımı

- **Geriye (CLAUDE.md):** “Test yazılabilir bir hedef tanımı” — loss tam olarak budur; hedefi yanlış tanımlarsan (piksel MSE) model yanlış şeyi optimize eder.
- **Merkezî ders:** “Doğru loss doğru sonucu belirler” — perceptual loss = Ders 20 content loss BİREBİR; «neyi optimize ettiğin» mimariden kritik. Aynı model + farklı loss = farklı sonuç (bulanık vs keskin).

32.14 13. miniai Köprüsü

Super-res yine miniai üstünde: TinyUnet bir nn.Module, comb_loss özel bir kayıp, eğitim standart Learner + callback’lerle. Foundations’ın esnek altyapısı, yeni bir problemi (super-res) minimum kodla çözüyor.

💡 Builder Notu — miniai: Yeni Problem, Aynı Altyapı

- **Geriye (Ders 16):** Yeni problem = yeni model + yeni loss; eğitim altyapısı (Learner, callback, Adam, OneCycle) değişmeden kalır.
- **Sezgi:** ETAP 4-5’te kurduğumuz Learner çerçevesi super-res’te de aynen çalışır — yalnız model (TinyUnet) ve loss (comb_loss) değişir; altyapının yeniden kullanılabilirliği foundations’ın amacıdır.

32.15 14. NYU/Foundations Köprüsü

Perceptual loss’un özü, önceden eğitilmiş bir ağın **temsillerini** (representations) kullanmaktır — NYU Deep Learning’in (LeCun) merkezî teması. Bir ağın öğrendiği özellikler o kadar zengindir ki, “iyi görüntü” tanımı bile o uzayda yapılabilir. Bu, transfer learning ve temsil öğrenmenin yaratıcı bir uygulamasıdır.

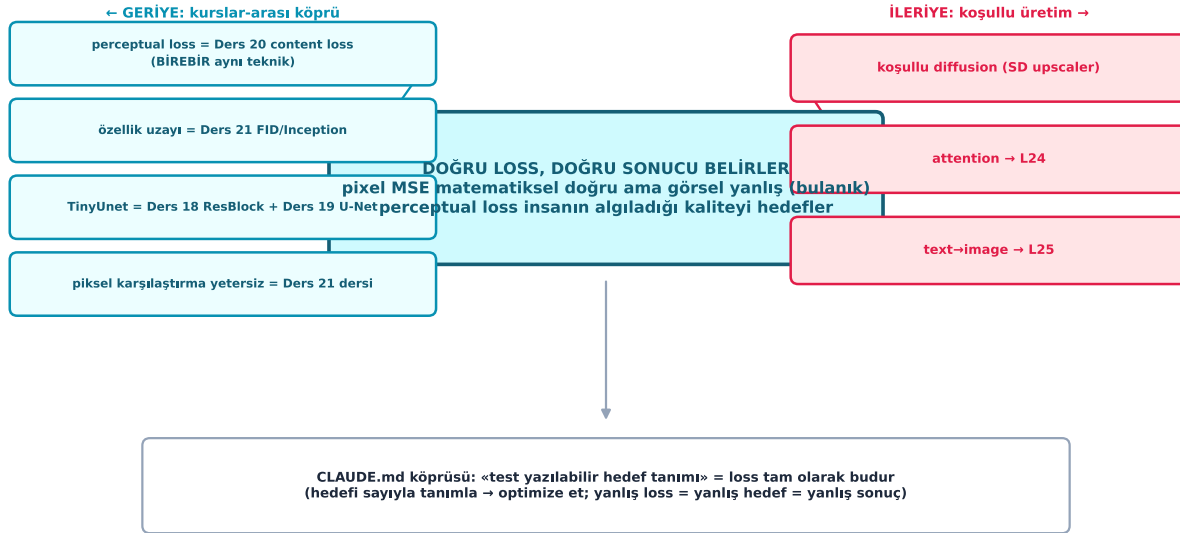
💡 Builder Notu — Temsil Öğrenme: Özellik Uzayının Gücü

- **Geriye (NYU + Ders 20-21):** Özellik uzayı (perceptual loss, FID, style transfer) tüm bu derslerin ortak aracı — temsil öğrenmenin gücü.
- **Sezgi:** Bir ağın öğrendiği temsiller o kadar zengindir ki, kalite ölçütü (perceptual loss), benzerlik ölçütü (FID), içerik ölçütü (style transfer) hepsi o uzayda tanımlanabilir.

32.16 15. Kapanış

Ders 23, SON ETAP'ı super-resolution ile açtı: bir U-Net'i (sıfırdan, ResBlock'larla) düşük çözünürlüğe koşullayarak yüksek çözünürlük üretmek. Anahtar ders, perceptual loss — piksel MSE'nin bulanıklığını, özellik uzayında karşılaştırma ile aşmak. Super-res, koşullu üretimin en sade örneği; sonraki derslerin (attention, text→image) köprüsü.

Şekil 32.8 bu kapanışı dersin sentezi olarak çerçeveler: merkezde çekirdek mesaj — **doğru loss, doğru sonucu belirler** (pixel MSE matematiksel doğru ama görsel yanlış/bulanık; perceptual loss insanın algıladığı kaliteyi hedefler). Solda geriye köprüler: perceptual loss = Ders 20 content loss (birebir), özellik uzayı = Ders 21 FID/Inception, TinyUnet = Ders 18 ResBlock + Ders 19 U-Net, piksel karşılaştırma yetersiz = Ders 21 dersi. Sağda ileriye: koşullu diffusion (SD upscaler), attention → L24, text→image → L25. Altta: CLAUDE.md köprüsü — “test yazılabilir hedef tanımı” = loss tam olarak budur.



Şekil 32.8: (sentez: L23 = loss tasarımı dersi — perceptual loss Ders 20/21 özellik-uzayı fikrinin super-res'e uygulanışı; koşullu üretim L24-25'in temeli)

! Builder Notu — Kapanış: SON ETAP Başladı

- **İleriye (Ders 24):** Koşullama elimizde; Ders 24 **attention**'ı sıfırdan kurar — U-Net'e güçlü bir koşullama ve ilişkilendirme mekanizması ekler (Karpathy nanoGPT köprüsü).
- **Çekirdek ders:** Doğru loss her şeyi belirler — super-res'i piksel MSE ile eğitirsen bulanık (keskinlik 1.000→0.340); perceptual loss (Ders 20 content loss'unun aynısı) keskinliği geri getirir. Super-res = koşullu üretimin en sade örneği, attention ve text→image'a köprü.

32.17 Bu Dersin Özeti

1. **Super-resolution:** Düşük çözünürlüklü görüntüden yüksek çözünürlük üretme; eksik detayları “uydurma” (HF enerji küçültmede %6’ya düşer) ([super-res nedir](#)).
2. **Koşullu üretim:** Çıktı düşük çözünürlüklü girdiye koşullanır; serbest üretim değil ([koşullu üretim](#)).
3. **Tiny ImageNet:** Gerçek görüntüler (MNIST yetersiz); perceptual loss’un farkını görmek için ([Tiny ImageNet](#)).
4. **TinyUnet:** Sıfırdan U-Net (ResBlock down path + up path + cross connections) ([TinyUnet](#)).
5. **Pixel loss bulanıklaştırır:** MSE birçok geçerli çıktının ortalamasını alır; keskinlik 1.000→0.340 (2.9× kayıp) ([pixel loss](#)).
6. **Perceptual loss:** Çıktı ve hedefi özellik uzayında (önceden eğitilmiş cmodel) karşılaştır; özellik keskinini seçer (0.40 60.92) ([perceptual loss](#)).
7. **comb_loss:** Piksel MSE + perceptual loss/10 (ağırlıklı); yapı + detay birlikte ([comb_loss](#)).
8. **Builder dersi:** Doğru loss doğru sonucu belirler; “neyi optimize ettiğin” mimariden kritik ([loss tasarımı](#)).

! Tek Bir Cümle

Super-resolution, bir U-Net’i (sıfırdan, ResBlock’larla kurulu) düşük çözünürlüklü girdiye koşullayarak yüksek çözünürlük üretir; piksel MSE bulanıklaştırdığı için (keskinlik 1.000→0.340) keskinlik, Ders 20’nin perceptual loss’uyla (özellik uzayında karşılaştırma, özellik 0.40 60.92) sağlanır — ve bu, koşullu üretimin en sade örneği olarak attention ve text→image’a köprüdür.

32.18 Kontrol Soruları

i Soru 1: Super-resolution neden bir üretken (generative) problemdir, basit bir büyütme değil?

Cevap:

Düşük çözünürlüklü bir görüntüde bilgi **kaybolmuştur** — küçültme sırasında ince detaylar (dokular, kenarlar) silinir (Şekil 32.2: HF enerji %6’ya düşer). Bunları geri getirmek, kaybolan bilgiyi “uydurma”dır: tek bir düşük çözünürlüklü girdiye karşılık birçok geçerli yüksek çözünürlüklü çıktı vardır. Basit interpolasyon (bilineer büyütme) bulanık sonuç verir çünkü detay üretmez, sadece var olanı yayar (figürde bilinear MSE > 0, detay geri gelmez). Gerçek super-resolution, modelin öğrendiği görüntü dağılımından makul detaylar **üretmesini** gerektirir — bu yüzden üretken bir problemdir. Çıktı düşük çözünürlüklü girdiye **koşullanır** (serbest üretim değil), ama yine de eksik detayı yaratıcı biçimde doldurur.

i Soru 2: Pixel MSE neden bulanık çıktı verir? Perceptual loss bunu nasıl çözer?

Cevap:

Bir düşük çözünürlüklü girdiye karşılık birçok eşit derecede geçerli keskin çıktı vardır (örn. bir dokunun tam piksel düzeni belirsiz). Piksel MSE, bu olasılıkların **ortalamasını** alan çıktıyı ödüllendirir — çünkü ortalama, tüm olasılıklara olan kare hatayı en aza indirir. Ama keskin detayların ortalaması **bulanıktır** (farklı keskin desenler birbirini siler — Şekil 32.4’de keskinlik 1.000→0.340, 2.9× kayıp). Sonuç: matematiksel olarak “en iyi” MSE çözümü, görsel olarak yumuşak/pürüzlü. **Perceptual loss** çözer

çünkü çıktıyı piksel piksel değil, önceden eğitilmiş bir ağın **özellik uzayında** karşılaştırır. Bu uzayda “keskin doku var mı” gibi anlamsal özellikler kodlanır (Şekil 32.5: özellik uzayı keskini seçer, fark 0.40 60.92, oysa piksel MSE bulanığı seçer $0.0179 < 0.0308$); ortalama almak yerine, çıktının gerçekçi doku/yapıya sahip olmasını teşvik eder. Sonuç keskin ve gerçekçi.

i Soru 3: Perceptual loss ile Ders 20’nin style transfer’ı arasındaki bağ nedir?

Cevap:

İkisi **aynı tekniktir**: önceden eğitilmiş bir ağın özellik uzayında MSE. Ders 20’deki style transfer’da **content loss**, optimize edilen görüntünün VGG16 özelliklerinin içerik görüntüsünün özelliklerine yakın olmasını sağlıyordu — yani içeriği özellik uzayında ölçüyordu. Super-res’teki **perceptual loss** tam olarak budur: super-res çıktısının özelliklerini, yüksek çözünürlüklü hedefin özelliklerine yaklaştırır (`F.mse_loss(cmodel(inp), cmodel(tgt))`). Tek fark: style transfer’da görüntü optimize ediliyordu (ağırlık sabit), super-res’te U-Net ağırlıkları optimize ediliyor (perceptual loss bir eğitim kaybı olarak). Builder içgörüsü: “özellik uzayında karşılaştırma” — style transfer, FID (L21), super-res perceptual loss — hepsi aynı güçlü fikrin farklı uygulamaları.

i Soru 4: Super-resolution neden ‘koşullu üretim’ olarak görülür ve sonraki derslere nasıl bağlanır? (builder bağlantısı)

Cevap:

Super-resolution’da model serbestçe üretmez (DDPM’in koşulsuz örnekleme gibi); çıktısını **düşük çözünürlüklü girdiye koşullar** — verilen görüntüyü büyütür. Bu, **koşullu üretimin** (conditional generation) en sade örneğidir: “şu girdiye göre üret”. Aynı koşullama fikri sonraki derslerin temelidir (Şekil 32.7): Ders 24’te **attention**, U-Net’in girdinin farklı bölgelerini ilişkilendirmesini ve dış bilgiyle (koşulla) etkileşmesini sağlar; Ders 25’te **text conditioning**, çıktıyı bir metin promptuna koşullar (text→image). Yani super-res (görüntüye koşullama) → attention (ilişkilendirme/cross-attention) → text→image (metne koşullama) doğal bir ilerlemedir. Builder için: koşullama, üretken modeli “kontrol edilebilir” kılan mekanizmadır — serbest üretimden işe yarar üretime geçiş.

32.19 Egzersizler

Egzersiz 1 (Direkt uygulama). Bir super-res U-Net’i sadece piksel MSE ile eğit; çıktının bulanık olduğunu gözle (keskinlik kaybını §5’teki $1.000 \rightarrow 0.340$ ile karşılaştır) (§5).

Egzersiz 2 (İki-aşamalı). Aynı modeli `comb_loss` (piksel + perceptual) ile eğit; iki çıktıyı keskinlik açısından karşılaştır (§8, §9).

Egzersiz 3 (Edge case). `comb_loss`’taki perceptual ağırlığını (`feat_loss/10`) değiştir (örn. /1, /100); keskinlik-doğruluk dengesinin nasıl kaydığını gör (§8).

Egzersiz 4 (Kavramsal). Perceptual loss neden özellik uzayında MSE kullanır da piksel MSE’yi tamamen atmaz? İkisinin birlikte rolü nedir (piksel yapısı, özellik keskinliği)? (§6, §8)

Egzersiz 5 (Sonraki dersin habercisi — Ders 24). Super-res U-Net girdinin yerel bölgelerine bakar (convolution). Görüntünün uzak bölgelerini birbiriyle ilişkilendirmek (örn. simetri) için nasıl bir mekanizma gerekir, düşün (attention) (§10).

32.20 Sonraki: Ders 24 İçin Hazırlık

Ders 24: Attention ve Transformer Blokları

Ders 23 super-res ile koşullu üretimi açtı. Ders 24 **attention**'ı sıfırdan kurar — convolution'ın yerel bakışını aşp görüntünün (veya dizinin) tüm bölgelerini birbiriyle ilişkilendiren mekanizma. Self-attention ($Q/K/V$), multi-head attention ve U-Net'e attention ekleme. Karpathy'nin nanoGPT'sinde sıfırdan kurduğu attention'ın fast.ai/miniai'da yeniden inşası.

Ana konular (Ders 24):

- Self-attention (query, key, value; $\text{softmax}(QK^T/\sqrt{d})V$)
- Multi-head attention
- Attention'ı U-Net'e ekleme
- Transformer bloğu (residual + attention)



Ders 24 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 2 — pixel vs perceptual loss).
- Ders 18'in residual (skip) bağlantısını tekrar oku (attention + residual = transformer bloğu).
- Karpathy nanoGPT'deki self-attention'ı hatırla (bu ders onu yeniden kuracak).

32.21 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Super-resolution	Düşük çözünürlükten yüksek çözünürlük üretme	6:18
Koşullu üretim	Çıktı bir girdiye (düşük-çöz) koşullanır	6:18
Tiny ImageNet	Gerçek görüntü kümesi (MNIST yetersiz)	6:06
TinyUnet	Sıfırdan U-Net (ResBlock down/up + cross)	—
Cross connection	U-Net skip; detay down→up taşır	20:06
Pixel loss bulanıklığı	MSE çıktıların ortalamasını alır → pürüzlü (1.000→0.340)	—
Perceptual loss	Özellik uzayında MSE (keskinlik geri gelir)	1:15:00
cmodel	Perceptual loss için özellik (içerik) modeli	—
comb_loss	Piksel MSE + perceptual loss/10 (ağırlıklı)	—
Loss tasarımı	Doğru loss = doğru sonuç (mimariden kritik)	—

32.22 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: Loss, U-Net ve Temsil Öğrenme

Bu ders super-res'i önceki derslerin altyapısına bağlar ve “doğru loss doğru sonucu belirler” felsefesini ortaya koyar; köprülerin özeti:

1. **Perceptual loss** → Ders 20 style transfer content loss; özellik uzayında MSE (BİREBİR aynı teknik) ([perceptual loss](#)).
2. **TinyUnet** → Ders 18 ResBlock + Ders 19 diffusion U-Net; encoder-decoder + skip ([TinyUnet](#)).
3. **Cross connection** → Ders 18 skip connection; U-Net detay akışı ([cross connections](#)).
4. **Koşullu üretim** → Ders 25 text→image; super-res en sade koşullama örneği ([koşullu üretim](#)).
5. **Özellik uzayı** → NYU temsil öğrenme + Ders 21 FID; “iyi” tanımı özelliklerde ([NYU köprüsü](#)).
6. **Loss tasarımı** → CLAUDE.md “test yazılabilir hedef”; üretkende loss = hedef tanımı ([loss tasarımı](#)).

❗ Bu dersten tek bir şey alıp gideceksen

Bu dersten tek bir şey alıp gideceksen: **doğru loss her şeyi belirler**. Super-resolution'ı piksel MSE ile eğitirsen bulanık çıktı alırsın (keskinlik 1.000→0.340, 2.9× kayıp) — çünkü MSE birçok geçerli keskin çıktının ortalamasını ödüllendirir. Çözüm, çıktıyı önceden eğitilmiş bir ağın özellik uzayında karşılaştıran **perceptual loss** (Ders 20'nin style transfer content loss'unun aynısı; figürde özellik keskini doğru seçer, fark 0.40 60.92). Super-res ayrıca koşullu üretimin en sade örneği — çıktıyı bir girdiye koşullama — ve bu, sonraki derslerin (attention ile ilişkilendirme, text→image koşullama) köprüsü. SON ETAP başladı; sıradaki attention ile gerçek Stable Diffusion'a yaklaşıyoruz.

33 Ders 24 — Attention ve Transformer Blokları (Attention and Transformer Blocks)

SON ETAP'ın teknik zirvesi — Howard, Jonathan Whitaker (Johno) ve Tanishq Abraham'la. Attention'ı sıfırdan kurmak: convolution yereldir (yalnız komşu pikseller), attention ise görüntünün ya da dizinin uzak bölgelerini uzaklıktan bağımsız biçimde birbiriyle ilişkilendirir. Self-attention ($Q/K/V$), $\text{softmax}(QK^T/\sqrt{d})V$, multi-head attention ve U-Net'e attention ekleme. Bu ders Phase 2'nin iki ana kursunun tam buluşma noktasıdır: Karpathy nanoGPT'de attention'ı metin için sıfırdan kurdu, Howard burada aynı attention'ı görüntü/diffusion için miniai'da yeniden inşa eder — aynı kod, aynı matematik. $\sqrt{d}$ ölçeklemesi Ders 17'nin varyans korunumudur; residual $x + \text{inp}$ Ders 18'in skip'idir; ikisi + attention = transformer bloğu. Dersin tek cümlesi: attention bir ağırlıklı ortalama — her konum, diğer tüm konumların value'larını query-key benzerliğinden gelen ağırlıklarla harmanlar, böylece convolution'ın yerelliğini aşar.

Bölüm bilgisi

- **Ders sayfası (video):** course.fast.ai — Lesson 24: Attention & Transformers (~116 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 24
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/27_attention](#) (+ [nbs/26_diffusion_unet](#))
- **Okuma süresi:** ~42 dk
- **Hocalar:** Howard + Jonathan Whitaker (Johno) + Tanishq Abraham
- **🔗 Phase 2'nin iki-kurs buluşma dersi:** Karpathy nanoGPT'de self-attention'ı ($Q/K/V$, scaled dot product, multi-head, residual) sıfırdan kurmuştu; bu ders aynı attention'ı [fast.ai/miniai](#)'da yeniden inşa eder. İki kurs burada tam buluşur — “yeniden kur, sonra kullan” döngüsünün doruğu.

33.1 Bu Derste Ne Var?

SON ETAP'ın teknik zirvesi: **attention**'ı sıfırdan kurmak. Convolution (Ders 15-18) yereldir — her çıktı pikseli yalnızca küçük bir komşuluğa (çekirdek boyutu) bakar. Attention ise görüntünün (veya dizinin) **uzak bölgelerini birbiriyle ilişkilendirir** — uzaklıktan bağımsız. Self-attention ($Q/K/V$), multi-head attention ve U-Net'e attention ekleme. Bu, Karpathy'nin nanoGPT'sinde sıfırdan kurduğu attention'ın [fast.ai/miniai](#)'da yeniden inşasıdır — “yeniden kur, sonra kullan” döngüsünün doruğu.

Üç temel fikir bu dersin omurgasını kurar:

1. **Attention = ağırlıklı ortalama** — her konum, diğer tüm konumların ağırlıklı ortalamasını alır; ağırlık, query-key benzerliğinden gelir ([attention fikri](#), $Q/K/V$).

2. **$\text{softmax}(\mathbf{QK}^T/\sqrt{d})\mathbf{V}$** — query ve key’lerin iç çarpımı benzerliği verir, softmax onu ağırlığa çevirir, value’lar bu ağırlıklarla harmanlanır ([formül](#)).
3. **Transformer bloğu = residual + attention** — L18’in skip connection’ı + L24’ün attention’ı, tam bir transformer bloğu yapar (Karpathy nanoGPT’siyle birebir) ([transformer bloğu](#), [Karpathy köprüsü](#)).

“[attention is] one of the two basic building blocks of Transformers. A Transformer layer is attention attached to a [MLP].” — Howard, 48:21

Şekil 34.1 bu fikirleri tek bir kavram haritasında birleştirir: convolution yerel komşuluğu işler, uzak ilişkiyi tek adımda kuramaz → attention = uzaklıktan bağımsız ağırlıklı ortalama ($\mathbf{Q/K/V}$, dot product benzerliği, $\text{softmax} + \sqrt{d}$ ölçekleme) → SelfAttention + residual ile derin ağa girer → multi-head ile çoğalır → U-Net’e eklenir → transformer bloğu Karpathy nanoGPT ile birebir buluşur. Rose ile işaretli düğümler (SelfAttention, multi-head, transformer bloğu, Karpathy buluşması) dersin builder mesajını taşır: aynı attention iki kursta da sıfırdan kuruldu.

💡 Builder Notu — Giriş: İki Kursun Buluşması, Sıfırdan Attention

- **Geriye (Karpathy nanoGPT) — MERKEZİ:** Karpathy nanoGPT’de self-attention’ı sıfırdan kurdu ($\mathbf{Q/K/V}$, scaled dot product, multi-head); bu ders aynı kodu fast.ai/miniai’da yeniden inşa eder. İki kurs burada tam buluşur.
- **Geriye (Ders 18):** L18’in residual’ı + bu dersin attention’ı = transformer bloğu; L18 olmadan bu blok eğitilemezdi.
- **İleriye (Ders 25):** Attention, latent diffusion’da metin koşullamasının (cross-attention, text→image) mekanizmasıdır.
- **Tek cümle:** Attention, her konumun diğer tüm konumların ağırlıklı ortalamasını ($\text{softmax}(\mathbf{QK}^T/\sqrt{d})\mathbf{V}$) alarak uzak ilişkileri yakalamasıdır; residual ile birleşince transformer bloğunu oluşturur — Karpathy’nin nanoGPT’sinin fast.ai’deki yeniden inşası.

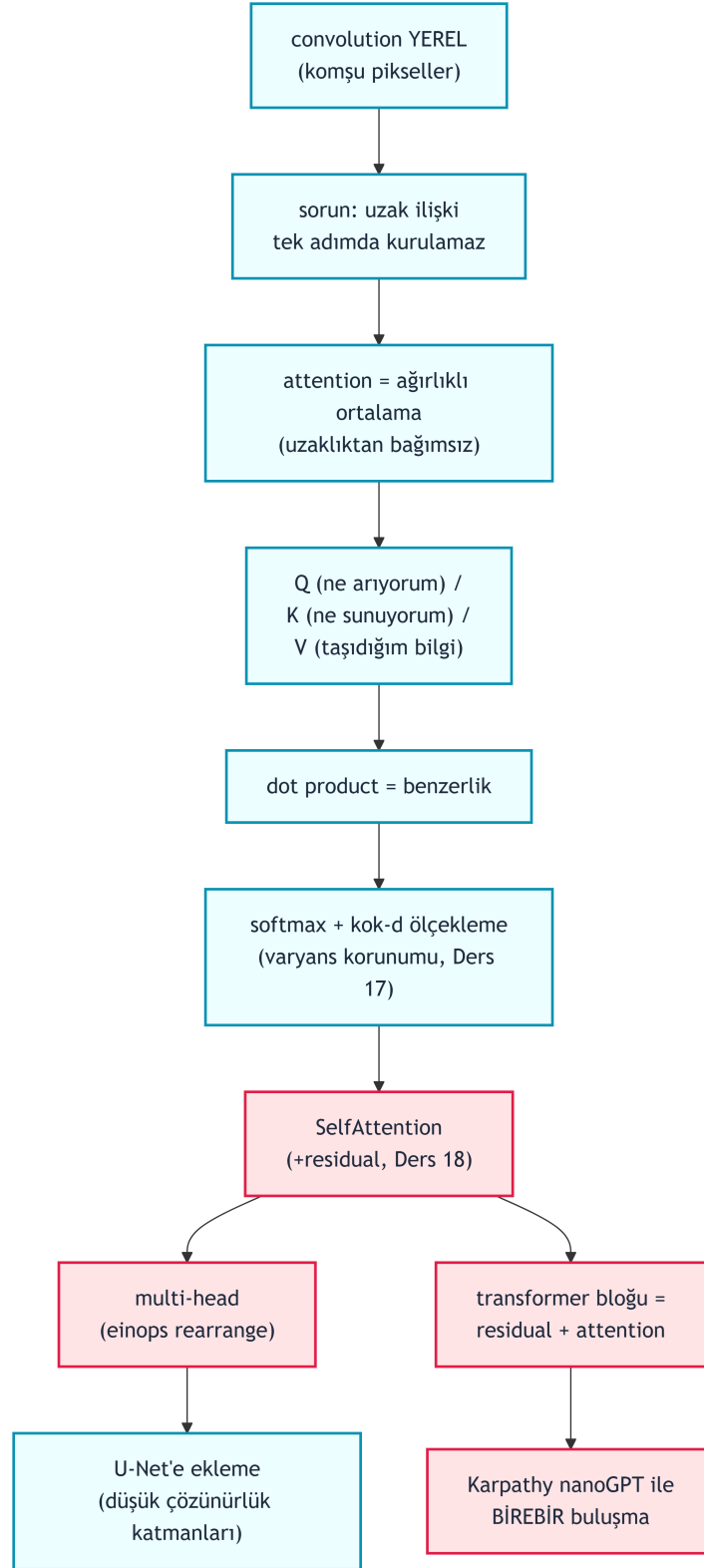
33.2 1. Convolution’ın Sınırı: Yerel Bakış

Convolution (Ders 15-18) güçlüdür ama **yereldir**: her çıktığı pikseli yalnızca küçük bir komşuluğa (çekirdek boyutu) bakar. Görüntünün uzak iki bölgesini (örn. bir yüzün iki gözü, bir desenin simetrisi) doğrudan ilişkilendiremez — bilgi ancak çok katman sonra yayılır. Attention bu sınırı aşar.

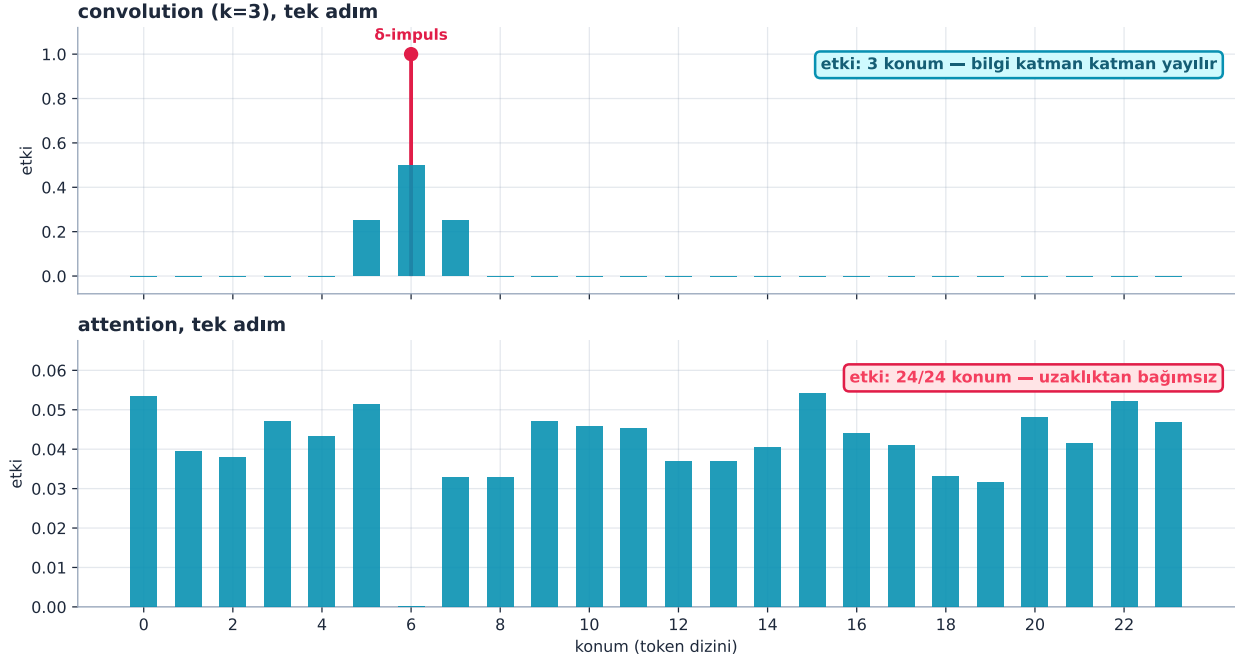
Şekil 33.2 bu farkı gerçek bir menzil testiyle gösterir: tek bir δ -impulse (6. konumda) verildiğinde, bir 3×3 convolution ($k=3$) tek adımda yalnızca **3 konumu** etkiler (impulse ve ± 1 komşusu); bilgi diziyeye ancak katman katman yayılır. Aynı impulse’a attention uygulandığında, impulse’un value’su **24/24 konumun** hepsinin ağırlıklı ortalamasına girer — yani tek adımda dizinin tamamına ulaşır. Yerel (3) ile küresel (24) arasındaki uçurum, attention’ın uzun-menzilli bağımlılık için neden gerekli olduğunu sayısal olarak ortaya koyar.

💡 Builder Notu — Convolution Yerel, Attention Tamamlayıcı

- **Geriye (Ders 8/15):** Convolution’ın parametre paylaşımı + yerelliği, görüntü için harika; ama uzak ilişki için yetersiz — attention tamamlayıcıdır.
- **Sayısal kanıt:** Aynı δ -impulse, conv’da tek adımda 3 konumu, attention’da 24/24 konumu etkiler; menzil farkı $3 \rightarrow 24$ ($8 \times$) tek figürde görünür.



Şekil 33.1: L24 kavram haritası: attention sıfırdan. Convolution yerel komşuluğu işler; uzak ilişkiyi tek adımda kuramaz. Attention = uzaklıktan bağımsız ağırlıklı ortalama (Q-K-V, dot product benzerliği, softmax + kok-d ölçekleme), SelfAttention + residual ile derin ağa girer, multi-head ile çoğalır, U-Net'e eklenir ve transformer bloğu Karpathy nanoGPT ile birebir buluşur (şematik).



δ-impuls testi: conv yerel komşuluk; attention'da impulsun value'su HER konumun ağırlıklı ortalamasına girer — uzun-menzil bağımlılığın (büyük görüntü yapısı, uzun metin) tek-adım çözümü

Şekil 33.2: (gerçek hesaplama: menzil testi — conv k=3 tek adımda 3 konum, attention 24/24; yerel vs küresel)

33.3 2. Attention Fikri: Ağırlıklı Ortalama

Attention'ın özü basittir: her konum, **diğer tüm konumların ağırlıklı ortalamasını** alır — uzaklıktan bağımsız. Ağırlıklar, “bu konum şu konuma ne kadar ilgili” sorusuna göre belirlenir. Böylece bir piksel, görüntünün herhangi bir yerindeki ilgili bilgiyi tek adımda toplayabilir.

“take a weighted average of other pixels around the image regardless of how far away they are.”
— Howard, 52:10

Builder Notu — Uzaklıktan Bağımsız Ağırlıklı Ortalama

- **İleriye:** “Uzaklıktan bağımsız ağırlıklı ortalama” — transformer’ların uzun-menzilli bağımlılığı (uzun metin, büyük görüntü) yakalamasının sırrı.
- **Sezgi:** Şekil 33.2’deki 24/24 erişim, “ağırlıklı ortalama”nın somut hâli; her konum tüm konumlara erişir, ağırlık ilgililiği belirler.

33.4 3. Query, Key, Value

Ağırlıkların nasıl hesaplandığı üç projeksiyona dayanır: her konum bir **query** (ne arıyorum), bir **key** (ne sunuyorum) ve bir **value** (taşıdığım bilgi) üretir. Bir konumun query’si, diğerlerinin key’leriyle karşılaştırılır; benzerlik ağırlık verir; value’lar bu ağırlıkla harmanlanır. Üçü de basit lineer katmanlardır.

“our keys, queries and values, or k q and v . I just think of them as k q and v .” — Howard, 1:06:18

💡 Builder Notu — Q/K/V: Aynı Zihinsel Model (Karpathy)

- **Geriye (Karpathy):** Karpathy de Q/K/V’yi tam böyle anlatır — “query: ne arıyorum, key: ne sunuyorum, value: ne veriyorum”. Aynı zihinsel model, iki kursta da özdeş.
- **Sezgi:** Q/K/V üç lineer projeksiyondur; “akıllı” olan kısım değil, onları nasıl çarpıp ağırlığa çevirdiğimiz (sonraki bölümler).

33.5 4. Dot Product = Benzerlik

Bir query ile bir key’in **iç çarpımı** (dot product), ikisinin ne kadar benzer olduğunu söyler — büyükse ilgili, küçükse ilgisiz. Tüm query-key çiftleri için bu hesaplanır ($q@k^T$), bir benzerlik matrisi oluşur. Bu matris, “hangi konum hangisine ne kadar dikkat etmeli”yi kodlar.

“the dot product between those two things... will tell us how similar they are.” — Howard, 57:31

💡 Builder Notu — İç Çarpım = Benzerlik (Ders 11-12, Ders 20)

- **Geriye (Ders 11-12):** İç çarpım = benzerlik; Ders 11-12’nin matmul/einsum’unun anlamsal kullanımı (Ders 20 Gram matrisi de aynı fikir).
- **Sezgi:** $q@k^T$ bir $s \times s$ benzerlik matrisidir; satır = bir query’nin tüm key’lerle benzerliği. Henüz ağırlıklı değil — softmax onu ağırlığa çevirecek.

33.6 5. $\text{softmax}(QK^T/\sqrt{d})V$

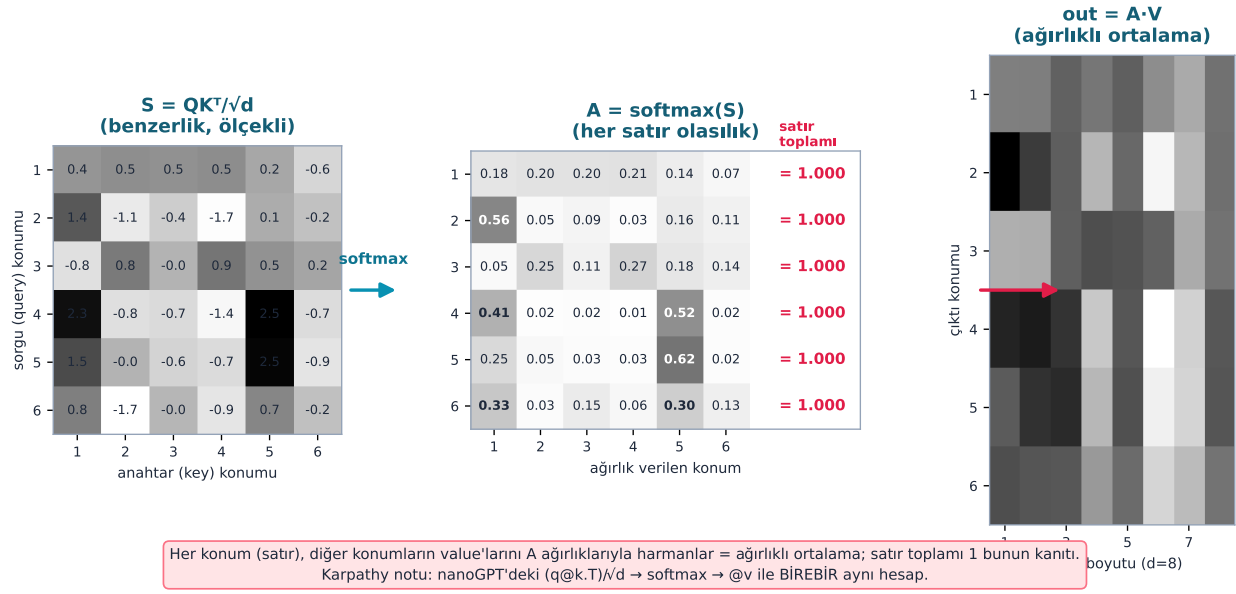
Attention’ın tam formülü: benzerlik matrisini ($Q \cdot K^T$) boyuta ($\sqrt{d}$) bölerek ölçekle (büyük d ’de softmax’ı dengeli tut), **softmax** ile ağırlığa çevir (her satır toplamı 1), value’larla çarp. Sonuç: her konum için value’ların ağırlıklı ortalaması.

$$\text{attention}(Q, K, V) = \text{softmax}(Q \cdot K^T / \sqrt{d}) \cdot V$$

$\sqrt{d}$ bölmesi kritiktir: d büyükse iç çarpımlar büyür, softmax doyunlaşır (gradyan kaybolur); $\sqrt{d}$ ile ölçekleme varyansı sabit tutar — Ders 17’nin varyans korunumu ilkesinin attention’daki yansıması.

Şekil 33.3 bu formülü tam hesapla ($s=6, d=8$) gösterir: solda $S = QK^T/\sqrt{d}$ (ölçekli benzerlik matrisi), ortada $A = \text{softmax}(S)$ — her satır bir olasılık dağılımı; sağda kolonda her satırın toplamı = **1.000** olarak işaretli (ağırlıklı ortalama kanıtı), sağda $\text{out} = A \cdot V$ (value’ların ağırlıklı ortalaması). Satır toplamlarının hepsi tam 1.000 olması, “her konum diğerlerinin ağırlıklı ortalamasını alır” iddiasının sayısal ispatıdır. Bu hesap, Karpathy’nin nanoGPT’indeki $(q@k.T)/\sqrt{d} \rightarrow \text{softmax} \rightarrow @v$ ile **birebir aynıdır**.

Şekil 33.4 $\sqrt{d}$ ölçeklemesinin neden zorunlu olduğunu gerçek hesapla kanıtlar. Boyut büyüdükçe ($d = 8, 64, 512$), ölçeklenmemiş logit’lerin standart sapması **2.81 $\rightarrow$ 7.99 $\rightarrow$ 22.59** olur — bu değerler tam $\sqrt{d}$ ’ye (2.83, 8.0, 22.63) yakındır. Ölçeklenmemiş hâlde softmax’ın ortalama max-prob’u **0.484’ten 0.934’e** tırmanır (one-hot’a doğru doyunlaşır, gradyan ölür); $\sqrt{d}$ ile ölçeklenince max-prob üç boyutta da **~0.16** sabit kalır.



Şekil 33.3: (gerçek hesaplama: $\text{softmax}(QK^T/\sqrt{d})V$ tam hesap — A satır toplamaları 1.000 = ağırlıklı ortalama; nanoGPT self-attention ile birebir)

Yani $\sqrt{d}$ ölçeklemesi, varyansı sabit tutarak softmax'ı dengeli tutar — Ders 17'nin varyans korunumu ilkesinin attention'daki tam karşılığı.

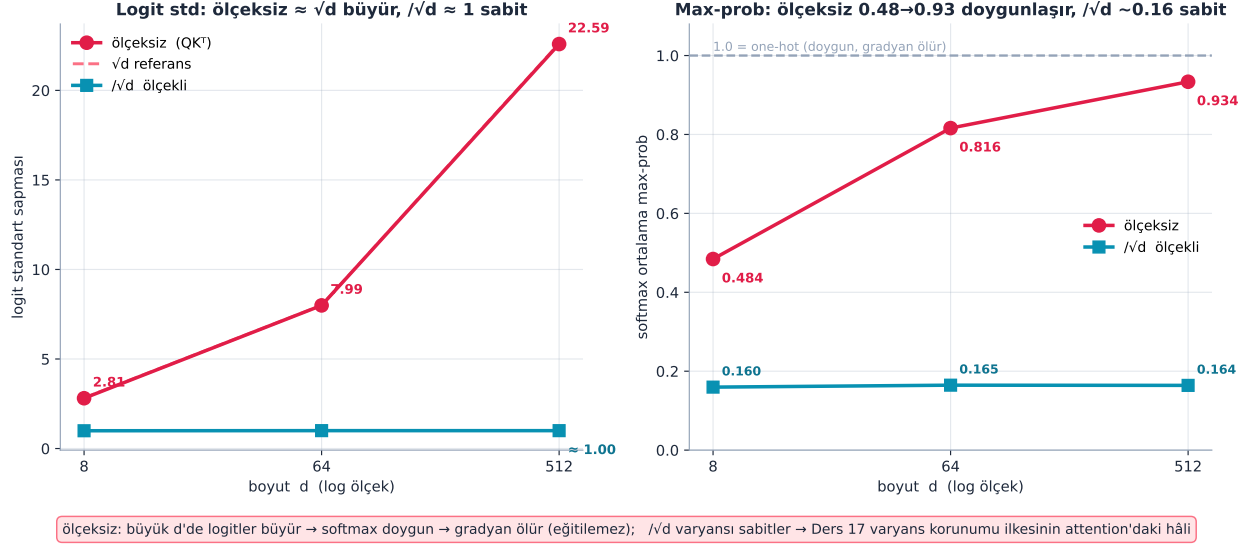
💡 Builder Notu — $\sqrt{d}$ Ölçekleme = Ders 17 Varyans Korunumu

- **Geriye (Ders 17):** $\sqrt{d}$ ölçeklemesi = varyans korunumu (L17); ölçeklenmemiş attention eğitilemez. Aynı ilke, yeni yer.
- **Sayısal kanıt:** Logit std 2.81/7.99/22.59 $\approx \sqrt{d}$; ölçeksiz max-prob 0.484→0.934 doygunlaşır, $\sqrt{d}$ ile ~ 0.16 sabit. Satır toplamı 1.000 ağırlıklı ortalamayı kanıtlar.

33.7 6. SelfAttention Sıfırdan

Self-attention (query, key, value aynı girdiden gelir) tek bir küçük modüldür: q/k/v lineer katmanları, ölçeklenmiş iç çarpım, softmax, value harmanı, çıkış projeksiyonu — ve bir **residual** bağlantı ($x + \text{inp}$).

```
class SelfAttention(nn.Module):
    def __init__(self, ni):
        super().__init__()
        self.scale = math.sqrt(ni)
        self.norm = nn.BatchNorm2d(ni)
        self.qkv = nn.Linear(ni, ni*3)
        self.proj = nn.Linear(ni, ni)
    def forward(self, inp):
        n,c,h,w = inp.shape
        x = self.norm(inp).view(n, c, -1).transpose(1, 2)
```



Şekil 33.4: $\sqrt{d}$ ölçekleme kanıtı (gerçek hesaplama): ölçeksiz max-prob 0.48→0.93 doygunlaşır (one-hot'a gidiş, gradyan ölür), $\sqrt{d}$ ile ~ 0.16 sabit kalır. Ders 17 varyans korunumu ilkesinin attention'daki hâli.

```
q,k,v = torch.chunk(self.qkv(x), 3, dim=-1)
s = (q@k.transpose(1,2))/self.scale
x = s.softmax(dim=-1)@v
x = self.proj(x).transpose(1,2).reshape(n,c,h,w)
return x+inp
```

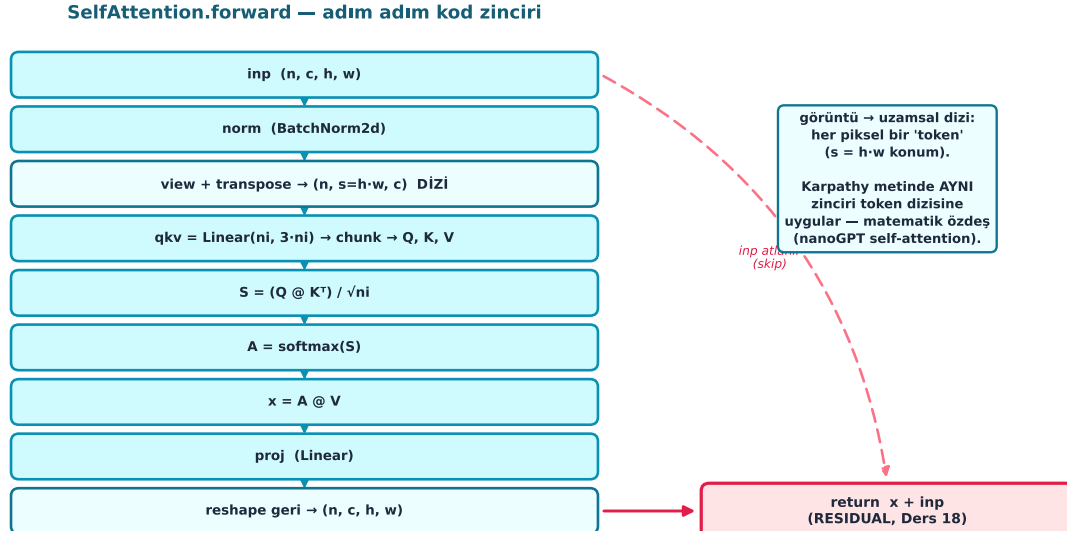
Şekil 33.5 bu forward'ın adım adım kod zincirini şematize eder: $\text{inp} (n, c, h, w) \rightarrow \text{norm} (\text{BatchNorm2d}) \rightarrow \text{view} + \text{transpose} \text{ ile uzamsal diziye } (n, s=h \cdot w, c) \rightarrow \text{qkv} = \text{Linear}(n_i, 3 \cdot n_i) \rightarrow \text{chunk} \text{ ile } Q/K/V \rightarrow S = (Q@K^T)/\sqrt{n_i} \rightarrow A = \text{softmax}(S) \rightarrow x = A@V \rightarrow \text{proj} \rightarrow \text{reshape} \text{ geri } (n, c, h, w) \rightarrow \text{return } x + \text{inp} \text{ (residual, Ders 18). Sağdaki not anahtar köprüyü vurgular: görüntü pikselleri bir token dizisi gibi işlenir (s = h \cdot w \text{ konum}); Karpthy metinde aynı zinciri token dizisine uygular — matematik özdeş (nanoGPT self-attention).}$

💡 Builder Notu — SelfAttention = nanoGPT self-attention (MERKEZÎ)

- **Geriye (Karpthy nanoGPT) — MERKEZÎ:** Bu kod, Karpthy'nin nanoGPT'indeki self-attention'la birebir aynı: q/k/v projeksiyon, $(q@k.T)/\sqrt{d}$, softmax, $@v$, proj. fast.ai görüntüye $(n, c, h, w \rightarrow \text{diziye})$, Karpthy metne uyguladı; matematik özdeş.
- **Sezgi:** Görüntüyü uzamsal diziye çevirmek (view+transpose), “her piksel bir token” demektir; attention için görüntü ile metin arasında yapısal fark kalmaz.

33.8 7. Residual: Attention + Skip

SelfAttention'ın son satırı `return x+inp` — bir **residual (skip) bağlantı** (Ders 18). Bu kritiktir: attention ham çıktısını değil, girdiye *eklenen* bir katkı üretir. Böylece bilgi blok boyunca korunur ve gradyan doğrudan akar — derin transformer'ları eğitilebilir kılan tam budur.



Şekil 33.5: (şematik: SelfAttention forward — norm→qkv→softmax($QK^T/\sqrt{d}$)V→proj→+inp; görüntü pikselleri token dizisi gibi)

💡 Builder Notu — $x+inp$ = Ders 18 Skip (Residual Stream)

- **Geriye (Ders 18):** $x+inp$ = L18 skip connection; attention bir “artık” (residual) katkıdır. Residual stream, ResNet’ten transformer’a taşınan fikir.
- **Sezgi:** Şekil 33.5’deki rose “inp atlanır (skip)” oku tam budur; attention katkısı üretir, residual ham girdiyi korur.

33.9 8. Multi-Head Attention

Tek attention bir tür ilişki yakalar; **multi-head attention** birden fazla yakalar. Girdiyi nheads parçaya böl (rearrange), her parçada ayrı attention yap (farklı ilişki türleri öğrenir), sonra birleştir. Ölçek $\sqrt{d/nheads}$ olur. “Aslında çok basit ve özölü.”

```

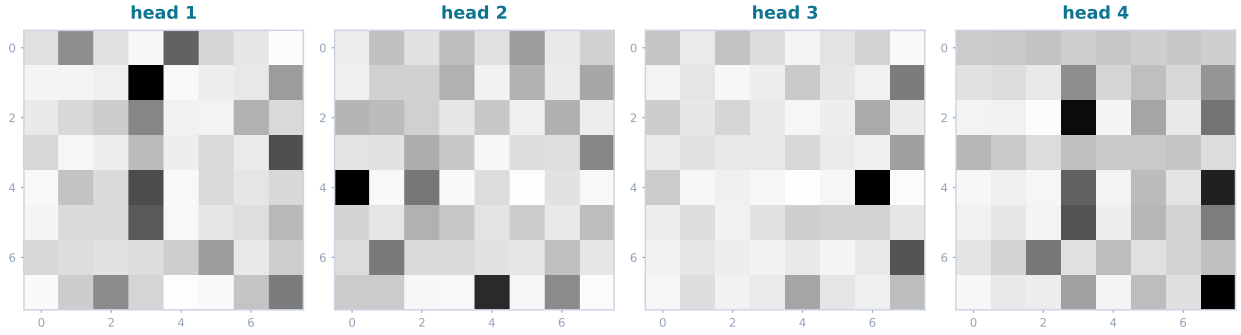
class SelfAttentionMultiHead(nn.Module):
    def __init__(self, ni, nheads):
        super().__init__()
        self.nheads = nheads
        self.scale = math.sqrt(ni/nheads)
        self.norm = nn.BatchNorm2d(ni)
        self.qkv = nn.Linear(ni, ni*3)
        self.proj = nn.Linear(ni, ni)
    def forward(self, inp):
        n,c,h,w = inp.shape
        x = self.norm(inp).view(n, c, -1).transpose(1, 2)
        x = self.qkv(x)
        x = rearrange(x, 'n s (h d) -> (n h) s d', h=self.nheads)
  
```

```
q,k,v = torch.chunk(x, 3, dim=-1)
s = (q@k.transpose(1,2))/self.scale
x = s.softmax(dim=-1)@v
x = rearrange(x, '(n h) s d -> n s (h d)', h=self.nheads)
x = self.proj(x).transpose(1,2).reshape(n,c,h,w)
return x+inp
```

“this is with multi-headed attention, and multi-headed attention actually turns out to be really simple and concise.” — Howard, 1:13:58

Şekil 33.6 4 head’in attention haritalarının belirgin biçimde **farklı** olduğunu gerçek hesaplama gösterir: head’ler arası ortalama |korelasyon| yalnızca **0.181** (düşük) — yani her head ayrı bir ilişki desenini yakalar, hepsi aynı şeyi öğrenmez. Bu, multi-head’in neden işe yaradığının sayısal kanıtıdır: paralel başlıklar çeşitli ilişki türlerini (yakınlık, simetri, doku) aynı anda öğrenir.

head’ler arası ortalama |korelasyon| = 0.181 (düşük) — her head FARKLI ilişki deseni yakalar



$\text{reshape}(s, h \cdot dh) \rightarrow (h, s, dh) = \text{einops rearrange 'n s (h d)} \rightarrow (n h) s d'$ mantığı; ölçek $\sqrt{(d/nheads)}$

Şekil 33.6: (gerçek hesaplama: multi-head — 4 head’in haritaları belirgin farklı ($|\text{corr}| 0.181$); paralel farklı ilişkiler)

💡 Builder Notu — Multi-Head: Paralel Farklı İlişkiler (Karpathy)

- **Geriye (Karpathy):** Karpathy de multi-head’i “birden fazla bağımsız attention paralel” diye kurar; einops rearrange ile head’leri bölmek fast.ai’nin zarif yolu.
- **Sayısal kanıt:** 4 head arası |korelasyon| 0.181 (düşük); head’ler farklı desen yakalar. Ölçek $\sqrt{(d/nheads)}$, tek-head’in $\sqrt{d}$ ’sinin head başına hâli.

33.10 9. einops rearrange

Multi-head’in inceliği, tensörü head’lere bölmektir: $\text{rearrange}(x, 'n s (h d) \rightarrow (n h) s d')$. einops kütüphanesi, tensör yeniden şekillendirmeyi okunabilir bir notasyonla yapar — boyutları isimlendirerek “kanalı head $\times$ derinlik olarak böl, head’i batch’e taşı” der. Hata yapması zor, anlaması kolay.

💡 Builder Notu — einops: Modern DL Standardı

- **İleriye:** einops, modern DL kodunun standart aracı; karmaşık reshape/transpose'ları okunabilir kılar (transformer kodunun her yerinde).
- **Sezgi:** 'n s (h d) -> (n h) s d' notasyonu, “kanalı (h d) çarpanına ayır, h'yi batch'e taşı” demektir; head'leri bağımsız batch örnekleri gibi işleyip tek matmul'da çözer.

33.11 10. nn.MultiheadAttention (PyTorch)

Sıfırdan kurduktan sonra PyTorch'un hazır nn.MultiheadAttention'ını kullanırız — artık içinde ne olduğunu biliyoruz. Self-attention için query=key=value=girdi verilir; residual elle eklenir. “Yeniden kur, sonra kullan” kuralı.

```
nm = nn.MultiheadAttention(32, num_heads=8, batch_first=True)
nm_x, nm_w = nm(t, t, t) # self-attention: q=k=v=t
nm_x = nm_x + t          # residual
```

💡 Builder Notu — Yeniden Kur, Sonra Kullan (Ders 10/16)

- **Geriye (Ders 10/16):** Sıfırdan kur → kütüphaneyi kullan döngüsü; nn.MultiheadAttention artık kara kutu değil.
- **Sezgi:** Self-attention'da query=key=value=aynı girdi (nm(t, t, t)); residual PyTorch'un içinde yok, elle eklenir (nm_x + t).

33.12 11. Attention'ı U-Net'e Ekleme

Attention, diffusion U-Net'ine (Ders 19/23) eklenir — genellikle düşük çözünürlüklü ara katmanlarda (yüksek çözünürlükte attention pahalıdır). Böylece U-Net hem yerel (convolution) hem küresel (attention) ilişkileri yakalar. Bu, modern diffusion U-Net'lerinin (Stable Diffusion dahil) standart yapısıdır.

💡 Builder Notu — U-Net + Attention = SD'nin Gürültü Tahmincisi

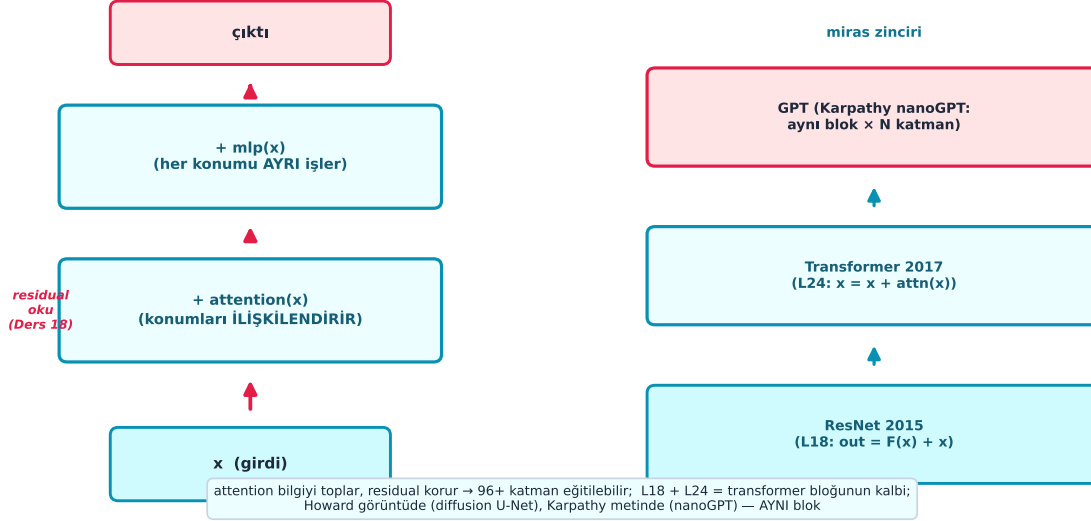
- **Geriye (Ders 19/23):** Diffusion U-Net (ResBlock'lar) + attention blokları = SD'nin gürültü tahmincisi; convolution + attention birlikte.
- **Sezgi:** Attention düşük çözünürlük katmanlarına eklenir çünkü hesap maliyeti s^2 (konum sayısının karesi); küçük uzamsal boyutta küresel ilişki ucuz, yüksek çözünürlükte pahalı.

33.13 12. Transformer Bloğu = Residual + Attention

Bir **transformer bloğu**, iki residual alt-bloktan oluşur: $x = x + \text{attention}(x)$ ve $x = x + \text{mlp}(x)$. İlki konumları ilişkilendirir (attention), ikincisi her konumu ayrı işler (MLP). L18'in residual'ı + bu dersin attention'ı = transformer bloğunun kalbi. Bunu artık tamamen sıfırdan kurduk.

“a Transformer layer is attention attached to a [MLP].” — Howard, 48:21

Şekil 33.7 bloğu iki residual alt-blok olarak gösterir (solda dikey akış): $x \rightarrow + \text{attention}(x)$ (konumları İLİŞKİLENDİRİR) $\rightarrow + \text{mlp}(x)$ (her konumu AYRI işler) $\rightarrow$ çıktı; her “+” bir residual toplamasıdır (Ders 18). Sağda miras zinciri: **ResNet 2015** (L18: $\text{out} = F(x) + x$) $\rightarrow$ **Transformer 2017** (L24: $x = x + \text{attn}(x)$) $\rightarrow$ **GPT** (Karpathy nanoGPT: aynı blok $\times$ N katman). Çekirdek sezgi: attention bilgiyi toplar, residual onu korur — bu yüzden 96+ katmanlı transformer’lar eğitilebilir. Howard görüntüde (diffusion U-Net), Karpathy metinde (nanoGPT) aynı bloğu kurdu.



Şekil 33.7: (şematik: transformer bloğu = residual + attention (+MLP); ResNet→Transformer→GPT zinciri; nanoGPT ile birebir)

💡 Builder Notu — L18 + L24 = Transformer Bloğu (MERKEZİ)

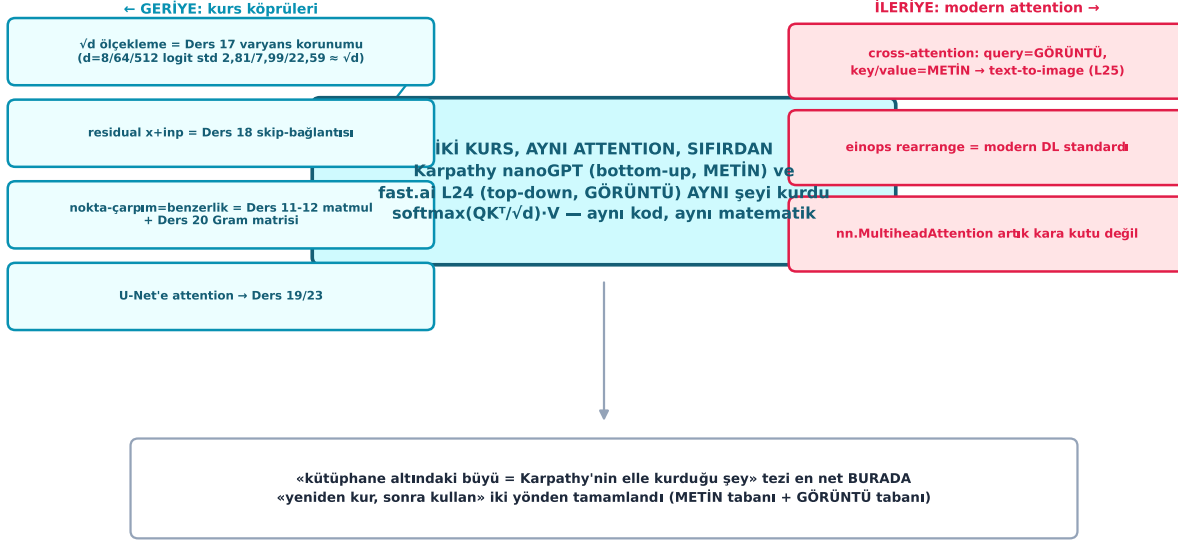
- **Geriye (Ders 18 + Karpathy) — MERKEZİ:** L18 (residual $\text{out} = F(x) + x$) + L24 (attention) = $x = x + \text{attn}(x)$; Karpathy nanoGPT’deki transformer bloğunun tam karşılığı. ResNet→Transformer→GPT zinciri kapanır.
- **Sezgi:** Blok iki işlevi ayırır: attention konumlar **arası** (uzamsal/diziye yayılan), MLP konum **içi** (her token’ı bağımsız dönüştüren) — bu ayrım transformer’ın gücünün kaynağı.

33.14 13. Karpathy nanoGPT Köprüsü

Bu ders, Phase 2’nin iki ana kursunun (fast.ai + Karpathy) tam buluşma noktasıdır. **Karpathy nanoGPT’de (bottom-up):** self-attention’ı sıfırdan kurdu — Q/K/V, scaled dot product, masked softmax, multi-head, residual. **Howard L24’te (top-down):** aynı attention’ı miniai’da yeniden inşa etti — aynı matematik ($\text{softmax}(QK^T/\sqrt{d})V$), aynı kod. Tek fark uygulama alanı: Karpathy metin (GPT), Howard görüntü (diffusion U-Net). “Kütüphane altındaki büyü = Karpathy’nin elle kurduğu şey” tezi, attention’da en net hâlini alır.

Şekil 33.8 bu buluşmayı dersin sentezi olarak çerçeveler: merkezde ana mesaj — iki kurs (Karpathy bottom-up/METİN, fast.ai top-down/GÖRÜNTÜ) **aynı** $\text{softmax}(QK^T/\sqrt{d}) \cdot V$ ’yi sıfırdan kurdu, aynı kod aynı matema-

tik. Solda geriye köprüler: $\sqrt{d}$ ölçekleme = Ders 17 varyans korunumu ($d=8/64/512$ logit std 2.81/7.99/22.59 $\approx \sqrt{d}$), residual $x+inp$ = Ders 18 skip, nokta-çarpım=benzerlik = Ders 11-12 matmul + Ders 20 Gram, U-Net'e attention = Ders 19/23. Sağda ileriye: cross-attention (query=görüntü, key/value=metin) $\rightarrow$ text $\rightarrow$ image (L25), einops rearrange = modern DL standardı, `nn.MultiheadAttention` artık kara kutu değil. Altta tez: “büyü = elle kurulan şey”, “yeniden kur, sonra kullan” iki yönden (METİN + GÖRÜNTÜ tabanı) tamamlandı.



Şekil 33.8: (sentez: L24 = Karpthy buluşması — aynı attention iki kursta sıfırdan; Ders 11/17/18/20 köprüleri; cross-attention L25’e kapı)

Builder Notu — §4.F: Yeniden Kur, Sonra Kullan’ın Doruğu

- **Geriye (§4.F):** “Yeniden kur, sonra kullan” döngüsü için yeni doruk örnek; attention iki kursta da sıfırdan kuruldu, builder her iki yönden de tam anlıyor.
- **Sentez:** Sayısal köprü kanıtları (logit std $\approx \sqrt{d}$, satır toplamı 1.000, max-prob doygunluğu) Karpthy nanoGPT’siyle aynı matematiğin fast.ai’da çalıştığını gösterir.

33.15 14. miniai/Builder: Yeniden Kur, Sonra Kullan

Attention da miniai felsefesine uyar: önce sıfırdan kur (SelfAttention, multi-head), anla, sonra PyTorch’un hazırladığını (`nn.MultiheadAttention`) güvenle kullan. Bu döngü, Part 2’nin omurgasıdır — matmul’dan (L11) attention’a (L24) her şey önce elle kuruldu.

Builder Notu — miniai: matmul’dan (L11) attention’a (L24)

- **İleriye (Ders 25):** Attention hazır; latent diffusion’da metin koşullaması (cross-attention) için kullanılacak — text $\rightarrow$ image’in mekanizması.
- **Sezgi:** L11 matmul $\rightarrow$ L17 init $\rightarrow$ L18 residual $\rightarrow$ L24 attention; her parça elle kuruldu, transfor-

mer bloğu bu parçaların birleşimi.

33.16 15. Kapanış

Ders 24 attention'ı sıfırdan kurdu: ağırlıklı ortalama fikri $\rightarrow Q/K/V \rightarrow$ dot product benzerlik $\rightarrow \text{softmax}(QK^T/\sqrt{d})V \rightarrow$ SelfAttention (residual ile) $\rightarrow$ multi-head (einops) $\rightarrow$ U-Net'e ekleme $\rightarrow$ transformer bloğu. Bu, convolution'ın yerelliğini aşan küresel ilişkilendirme mekanizmasıdır ve Karpathy nanoGPT'sinin fast.ai'daki yeniden inşasıdır. L18 residual + L24 attention = transformer.

! Builder Notu — Kapanış: SON DERS'e HAZIRIZ

- **İleriye (Ders 25, SON DERS):** Tüm parçalar hazır — DDPM (L19), Karras (L22), U-Net (L23), attention (L24); Ders 25 bunları VAE + latent uzayla birleştirip gerçek Stable Diffusion'ı sıfırdan kurar.
- **Çekirdek ders:** Attention bir ağırlıklı ortalamadır (satır toplamı 1.000); $\sqrt{d}$ = Ders 17 varyans, residual = Ders 18 skip; ikisi + attention = transformer. Karpathy nanoGPT'siyle birebir buluşma — Phase 2'nin iki kursunun tam kesişimi.

33.17 Bu Dersin Özeti

1. **Convolution yereldir:** Sadece komşu pikseller; uzak ilişkiyi yakalayamaz (menzil 3 vs attention 24) ([convolution sınırı](#)).
2. **Attention = ağırlıklı ortalama:** Her konum, diğer tüm konumların ağırlıklı ortalamasını alır (uzaklıktan bağımsız) ([attention fikri](#)).
3. **Q/K/V:** Query (ne arıyorum), key (ne sunuyorum), value (taşıdığım bilgi); üç lineer projeksiyon ([Q/K/V](#)).
4. **Dot product = benzerlik:** Query-key iç çarpımı ilgililik ağırlığını verir ([dot product](#)).
5. **$\text{softmax}(QK^T/\sqrt{d})V$:** Benzerliği $\sqrt{d}$ ile ölçekle (varyans korunumu, L17; logit std 2.81/7.99/22.59 $\approx \sqrt{d}$), softmax ile ağırlığa çevir (satır toplamı 1.000), value harmanla ([formül](#)).
6. **Residual:** Attention $x + \text{inp}$ döner (L18 skip); bilgi korunur, gradyan akar ([residual](#)).
7. **Multi-head:** Girdiyi head'lere böl (einops rearrange), her birinde ayrı attention (|korelasyon| 0.181), birleştir; farklı ilişkiler ([multi-head](#)).
8. **Transformer bloğu:** residual + attention (L18+L24); Karpathy nanoGPT'siyle birebir ([transformer bloğu](#)).

! Tek Bir Cümle

Attention, her konumun diğer tüm konumların ağırlıklı ortalamasını ($\text{softmax}(Q \cdot K^T/\sqrt{d}) \cdot V$ — query-key benzerliğini $\sqrt{d}$ ile ölçekleyip softmax ile ağırlığa çevirip value'ları harmanlayarak; satır toplamı 1.000) alarak convolution'ın yerelliğini aşan küresel ilişkilendirme mekanizmasıdır; residual ile birleşince (L18) transformer bloğunu oluşturur — Karpathy'nin nanoGPT'sinde sıfırdan kurduğu attention'ın fast.ai/miniai'daki birebir yeniden inşası.

ağırlıklar, o konumun diğerlerine ne kadar “ilgili” olduğuna göre belirlenir. **Convolution** yereldir: her çıktı yalnızca küçük bir komşuluğa (çekirdek boyutu, örn. 3×3) bakar; uzak iki bölgeyi ilişkilendirmek için bilginin birçok katman boyunca yavaşça yayılması gerekir (Şekil 33.2: conv tek adımda 3 konum). **Attention** ise **Attention ve Transförmasyon Blokları** (Attention and Transformer Blocks) bir yerindeki ilgili bilgiyi tek adımda toplayabilir (figürde attention 24/24 konum; örn. bir yüzün iki gözünün simetrisi, bir yüzün iki gözü arasındaki bağ). Bu yüzden uzun-menzilli bağımlılıklarda (uzun metin, büyük görüntü yapısı) attention çok güçlüdür. İkisi tamamlayıcıdır: modern diffusion U-Net’leri hem convolution (yerel doku) hem attention (küresel yapı) kullanır.

i Soru 2: Q, K, V nedir ve $\text{softmax}(QK^T/\sqrt{d})V$ formülü ne yapar?

Cevap:

Her konum üç lineer projeksiyon üretir: **Query** (ne arıyorum), **Key** (ne sunuyorum), **Value** (taşıdığım bilgi). Formül adım adım: (1) $Q \cdot K^T$ — her query’yi her key ile iç çarp; bu, “hangi konum hangisine ne kadar benzer/ilgili” benzerlik matrisidir (dot product = benzerlik). (2) $/\sqrt{d}$ — boyuta böl; d (özellik boyutu) büyükse iç çarpımlar büyür (Şekil 33.4: logit std $2.81 \rightarrow 7.99 \rightarrow 22.59 \approx \sqrt{d}$), softmax doyunlaşıp gradyan kaybolur (max-prob $0.484 \rightarrow 0.934$), $\sqrt{d}$ ile ölçekleme varyansı sabit tutar (~ 0.16 , Ders 17 varyans korunumu). (3) **softmax(...)** — her satırı olasılığa çevir (toplam 1; Şekil 33.3: satır toplamı 1.000); artık her konum için “diğerlerine ne kadar dikkat etmeli” ağırlıkları. (4) $\cdot V$ — bu ağırlıklarla value’ları harmanla; her konum, ilgili olduğu konumların value’larının ağırlıklı ortalamasını alır. Sonuç: bağlama duyarlı yeni bir temsil.

i Soru 3: Transformer bloğu nedir ve hangi iki önceki dersi birleştirir?

Cevap:

Bir **transformer bloğu** iki residual alt-bloktan oluşur: $x = x + \text{attention}(x)$ (konumları birbiriyle ilişkilendirir) ve $x = x + \text{mlp}(x)$ (her konumu ayrı işler/dönüştürür). İki dersi birleştirir: **Ders 18** residual/skip connection ($\text{out} = F(x) + x$) — bilginin korunması ve gradyanın akması; **Ders 24** attention — konumlar arası ilişkilendirme. İkisi birleşince: $x = x + \text{attn}(x)$ tam transformer bloğunun kalbidir (Şekil 33.7). Bu yapı sayesinde çok derin transformer’lar (96+ katman GPT) eğitilebilir: attention bilgiyi toplar, residual onu korur. fast.ai bunu görüntü için (diffusion U-Net’inde), Karpathy metin için (nanoGPT’de) kurdu — aynı blok, farklı alan. ResNet (2015, L18) $\rightarrow$ Transformer (2017, L24) $\rightarrow$ GPT doğrudan miras zinciridir.

i Soru 4: Bu ders Karpathy’nin nanoGPT’siyle nasıl tam buluşur? (builder bağlantısı)

Cevap:

Phase 2’nin iki ana kursu (fast.ai top-down + Karpathy bottom-up) attention’da **birebir buluşur** (Şekil 33.8). **Karpathy nanoGPT (bottom-up)**: self-attention’ı metin için sıfırdan kurdu — Q/K/V

projeksiyonları, scaled dot product ($q@k.T)/\sqrt{d}$, (causal) softmax, multi-head paralel attention, residual bağlantı. **Howard L24 (top-down)**: aynı attention'ı görüntü/diffusion için miniai'da yeniden inşa etti — aynı SelfAttention kodu, aynı softmax($QK^T/\sqrt{d}$)V matematiği (satır toplamı 1.000), aynı multi-head (|korelasyon| 0.181), aynı residual. Tek fark: Karpathy metne (token dizisi), Howard görüntüye (n,c,h,w → uzamsal dizi) uyguladı; matematik **özdeştir**. Builder içgörüsü: “kütüphane altındaki büyü = Karpathy'nin elle kurduğu şey” tezi en net burada — nn.MultiheadAttention'ın içi, iki kursta da sıfırdan kurduğumuz tam o koddur. “Yeniden kur, sonra kullan” döngüsü, attention'da iki yönden de tamamlanır.

33.19 Egzersizler

Egzersiz 1 (Direkt uygulama). SelfAttention'ı yaz; küçük bir girdiye uygula, benzerlik matrisinin (softmax sonrası) satır toplamlarının 1 olduğunu doğrula (§5, §6).

Egzersiz 2 (İki-aşamalı). Multi-head attention'ı rearrange ile kur; tek-head ve çok-head çıktılarını karşılaştır (head'ler arası |korelasyon|'a bak, §8'deki 0.181 ile kıyasla) (§8, §9).

Egzersiz 3 (Edge case). $\sqrt{d}$ ölçeklemesini kaldır (/self.scale çıkar); büyük d'de softmax'ın nasıl doygunlaştığını (max-prob → 1) ve eğitimin nasıl bozulduğunu gözle (§5'teki 0.484 → 0.934 ile karşılaştır; Ders 17 varyans bağlantısı) (§5).

Egzersiz 4 (Kavramsal). nn.MultiheadAttention ile kendi SelfAttention'ın aynı sonucu mu verir? Q=K=V verince ne olur (self-attention)? (§10)

Egzersiz 5 (Sonraki dersin habercisi — Ders 25). Attention ile çıktıyı bir metne koşullamak (text → image) nasıl mümkün olur? Query görüntüden, key/value metinden gelirse ne olur, düşün (cross-attention) (§13).

33.20 Sonraki: Ders 25 İçin Hazırlık

Ders 25: Latent Diffusion (SON DERS — Stable Diffusion'ı Sıfırdan Kur)

Ders 24 attention'ı kurdu. Ders 25, kursun SON dersi: tüm parçaları (DDPM, Karras, U-Net, attention) **VAE + latent uzayla** birleştirip gerçek Stable Diffusion'ı sıfırdan kurar. Diffusion'ı piksel yerine sıkıştırılmış latent uzayda yapmak (hız + çözünürlük) ve metin koşullaması. Ders 9'da kullandığımız SD'yi şimdi kendimiz kuruyoruz — “yeniden kur, sonra kullan” döngüsü tam kapanıyor.

Ana konular (Ders 25):

- VAE (encoder/decoder; piksel → latent → piksel)
- Latent diffusion (diffusion latent uzayda)
- Text conditioning (CLIP — kısmen sonraki kursa)
- Stable Diffusion'ın tam mimarisi (sıfırdan)

⚠ Ders 25 Öncesi Yapılacak

- Bu dersin egzersizlerini çöz (özellikle 1 ve 3 — SelfAttention + $\sqrt{d}$ ölçekleme).
- Ders 15'in autoencoder/VAE sezgisini tekrar oku (latent uzay).

- Ders 9'u (Stable Diffusion'ı kullandık) tekrar gözden geçir — son ders onu sıfırdan kuracak.

33.21 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
Attention	Konumların ağırlıklı ortalaması (uzaklıktan bağımsız)	52:10
Query / Key / Value	Ne arıyorum / ne sunuyorum / taşıdığım bilgi	1:06:18
Dot product	$Q \cdot K$ iç çarpımı = benzerlik/ilgililik	57:31
$\text{softmax}(QK^T/\sqrt{d})V$	Ölçekli benzerlik $\rightarrow$ softmax ağırlık $\rightarrow$ value harmanı	1:01:23
$\sqrt{d}$ ölçeklemesi	Varyansı sabit tutar (L17); uygunluğu önler (logit std $\approx \sqrt{d}$)	—
SelfAttention	$q/k/v$ aynı girdiden; residual ile ($x+\text{inp}$)	1:01:23
Multi-head	Paralel birden çok attention; farklı ilişkiler ($ \text{corr} 0.181$)	1:13:58
einops rearrange	Okunabilir tensör yeniden şekillendirme (head split)	1:19:31
<code>nn.MultiheadAttention</code>	PyTorch hazır attention (içini biliyoruz)	—
Transformer bloğu	residual + attention (L18+L24); nanoGPT'yle birebir	48:21

33.22 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: Attention, Transformer ve İki Kursun Buluşması

Bu ders attention'ı önceki derslerin altyapısına bağlar ve Phase 2'nin iki ana kursunu (fast.ai + Karpathy) tam buluşturur; köprülerin özeti:

1. **Attention** $\rightarrow$ Karpathy nanoGPT self-attention (MERKEZİ); aynı kod, aynı matematik (metin $\leftrightarrow$ görüntü) ([Karpathy köprüsü](#)).
2. **Transformer bloğu** $\rightarrow$ Ders 18 residual + L24 attention; ResNet $\rightarrow$ Transformer $\rightarrow$ GPT zinciri ([transformer bloğu](#)).
3. **$\sqrt{d}$ ölçeklemesi** $\rightarrow$ Ders 17 varyans korunumu; ölçeklenmemiş attention eğitilemez (logit std $2.81/7.99/22.59 \approx \sqrt{d}$) ([formül](#)).
4. **Dot product benzerlik** $\rightarrow$ Ders 11-12 matmul/einsum + Ders 20 Gram; iç çarpım = ilişki ([dot product](#)).
5. **Attention U-Net'te** $\rightarrow$ Ders 19/23 diffusion U-Net; convolution (yerel) + attention (küresel)

(U-Net'e ekleme).

6. **Cross-attention** → Ders 25 text→image; query görüntü, key/value metin (Karpathy köprüsü).

! Bu dersten tek bir şey alıp gideceksen

Bu dersten tek bir şey alıp gideceksen: **attention bir ağırlıklı ortalamadır**. Her konum, diğer tüm konumların value'larını, query·key benzerliğinden gelen ağırlıklarla ($\text{softmax}(QK^T/\sqrt{d})V$; satır toplamı 1.000) harmanlar — convolution'ın yerelliğini (tek adımda 3 konum) aşır uzak ilişkileri tek adımda yakalar (24/24 konum). $\sqrt{d}$ ölçeklemesi Ders 17'nin varyans korunumudur (logit std 2.81/7.99/22.59 $\approx \sqrt{d}$; ölçeksiz max-prob 0.484→0.934 doygunlaşır, $1/\sqrt{d}$ ile ~ 0.16 sabit); residual x+inp Ders 18'in skip'idir; ikisi + attention = transformer bloğu (|head korelasyon| 0.181 ile multi-head farklı ilişkiler yakalar). Ve bu, Karpathy'nin nanoGPT'sinde metin için sıfırdan kurduğu attention'ın, fast.ai'da görüntü için birebir yeniden inşasıdır — Phase 2'nin iki kursunun tam buluştuğu nokta. SON DERS, bunu VAE + latent uzayla birleştirip gerçek Stable Diffusion'ı kuracak.

34 Ders 25 — Latent Diffusion: Stable Diffusion'ı Sıfırdan (Latent Diffusion — SON DERS)

Kursun SON DERSİ ve pedagojik finali — Howard, Jonathan Whitaker (Johno) ve Tanishq Abraham'la. Tüm parçaları (DDPM, Karras, U-Net, attention) bir VAE ve latent uzayla birleştirip gerçek Stable Diffusion'ı sıfırdan kurmak. Anahtar fikir: pahalı diffusion'ı $512 \times 512 \times 3 \approx 786K$ boyutlu piksel uzayında değil, VAE'nin sıkıştırdığı $64 \times 64 \times 4 \approx 16K$ boyutlu küçük latent uzayda yapmak — $\sim 48\times$ daha az veri, yüzlerce U-Net çağırısı tek GPU'da koşar, yüksek çözünürlük pratik olur. Ders 9'da bu Stable Diffusion'ı kullandık; şimdi kendimiz kuruyoruz — 'yeniden kur, sonra kullan' döngüsü, kursun ilk üretken dersinden son dersine tam kapanıyor. VAE = encoder (görüntü $\rightarrow$ latent dağılım μ, lv) + reparameterization ($z = \mu + \exp(0.5 \cdot \text{lv}) \cdot \epsilon$, 9B trick'i) + decoder; KLD + BCE kaybı latent'i $N(0,1)$ 'e hizalar (örneklenebilir uzay); latent diffusion = DDPM/Karras latent uzayda; Stable Diffusion = VAE + latent U-Net + CLIP cross-attention. Dersin tek cümlesi: karmaşık sistemler sağlam temellerin birleşimidir — Stable Diffusion'da sihir yoktur, sadece doğru sırayla katmanlanmış matris çarpımı, gradient descent, residual, normalizasyon, attention ve MSE vardır; builder olmak bu parçaları görebilmek ve birleştirebilmektir.

Bölüm bilgisi

- **Ders sayfası (video):** [course.fast.ai — Lesson 25: Latent Diffusion \(SON DERS\)](https://course.fast.ai/lesson/25-latent-diffusion-son-ders/) (~98 dk)
- **Seri:** Practical Deep Learning for Coders — Part 2, Ders 25 (KURSUN SON DERSİ)
- **Playlist:** [Part 2 — Foundations to Stable Diffusion \(2022\)](#)
- **Notebook:** [course22p2 — nbs/29_vae + nbs/30_lsun_diffusion-latents](#)
- **Okuma süresi:** ~40 dk
- **Hocalar:** Howard + Jonathan Whitaker (Johno) + Tanishq Abraham
- **Halka KAPANIŞ dersi:** Ders 9'da Stable Diffusion'ı *kullandık* (pipeline: VAE+U-Net+CLIP); 16 ders foundations sonra, L25'te aynı SD'yi *sıfırdan kuruyoruz*. “Yeniden kur, sonra kullan” döngüsü — kursun ilk üretken dersinden son dersine — tam kapanır.

34.1 Bu Derste Ne Var?

Kursun **SON DERSİ** ve pedagojik finali: tüm parçaları (DDPM, Karras, U-Net, attention) bir **VAE + latent uzay**la birleştirip gerçek **Stable Diffusion**'ı sıfırdan kurmak. Anahtar fikir basit ama güçlü: pahalı diffusion'ı piksel uzayında değil, VAE'nin sıkıştırdığı küçük latent uzayda yap — çok daha hızlı, yüksek çözünürlük mümkün. Ders 9'da bu SD'yi *kullandık*; şimdi *kendimiz kuruyoruz* — “yeniden kur, sonra kullan” döngüsü tam kapanıyor.

Üç temel fikir bu dersin omurgasını kurar:

1. **VAE (encoder/decoder)** — görüntüyü küçük bir latent temsile sıkıştırır (encoder) ve geri açar (decoder); latent uzayı kurar (VAE, reparameterization).
2. **Latent diffusion** — diffusion'ı piksel yerine latent uzayda yap; ~48× daha az hesap, yüksek çözünürlük için pratik (latent uzay, latent diffusion).
3. **Stable Diffusion = VAE + latent U-Net + (CLIP)** — Ders 9'da kullandığımız SD'nin tam mimarisi; CLIP text encoder hariç hepsi sıfırdan (tam mimari, halka kapanır).

“working in the latent space can make it computationally cheaper than having to decode [every step].” — Howard, 1:32:28

Şekil 34.1 bu fikirleri tek bir kavram haritasında birleştirir: Ders 9'da SD'yi kara kutu olarak kullandık → 16 ders foundations (matmul → backprop → miniai → DDPM) → üç bileşeni (VAE, latent diffusion, text conditioning) sıfırdan kur → birleştir → Stable Diffusion. Rose ile işaretli düğümler (latent diffusion hattı, halka kapanışı) dersin builder mesajını taşır: Ders 9 (kullan) → L25 (kur) halkası burada kapanır.

💡 Builder Notu — Giriş: Halka Kapanışı, Sıfırdan Stable Diffusion

- **Geriye (Ders 9) — HALKA:** Ders 9'da Stable Diffusion'ı kullandık (pipeline: VAE+U-Net+CLIP), Whitaker/Waseem/Tanishq matematiği gösterdi; L25'te aynı SD'yi sıfırdan kuruyoruz. Kursun ilk üretken dersinden son dersine tam halka.
- **Geriye (Ders 15):** L15'in autoencoder/VAE sezgisi burada gerçek latent uzaya dönüşür.
- **Geriye (Ders 19-24):** DDPM (L19) + Karras (L22) + U-Net (L23) + attention (L24) — hepsi latent uzayda birleşir.
- **Tek cümle:** Latent diffusion, bir VAE'nin sıkıştırdığı küçük latent uzayda diffusion yaparak Stable Diffusion'ı pratik kılar; Ders 9'da kullandığımız SD'yi sıfırdan kurarak “yeniden kur, sonra kullan” döngüsünü kapatır.

34.2 1. Hatırlatma: Ders 9'da SD'yi Kullandık

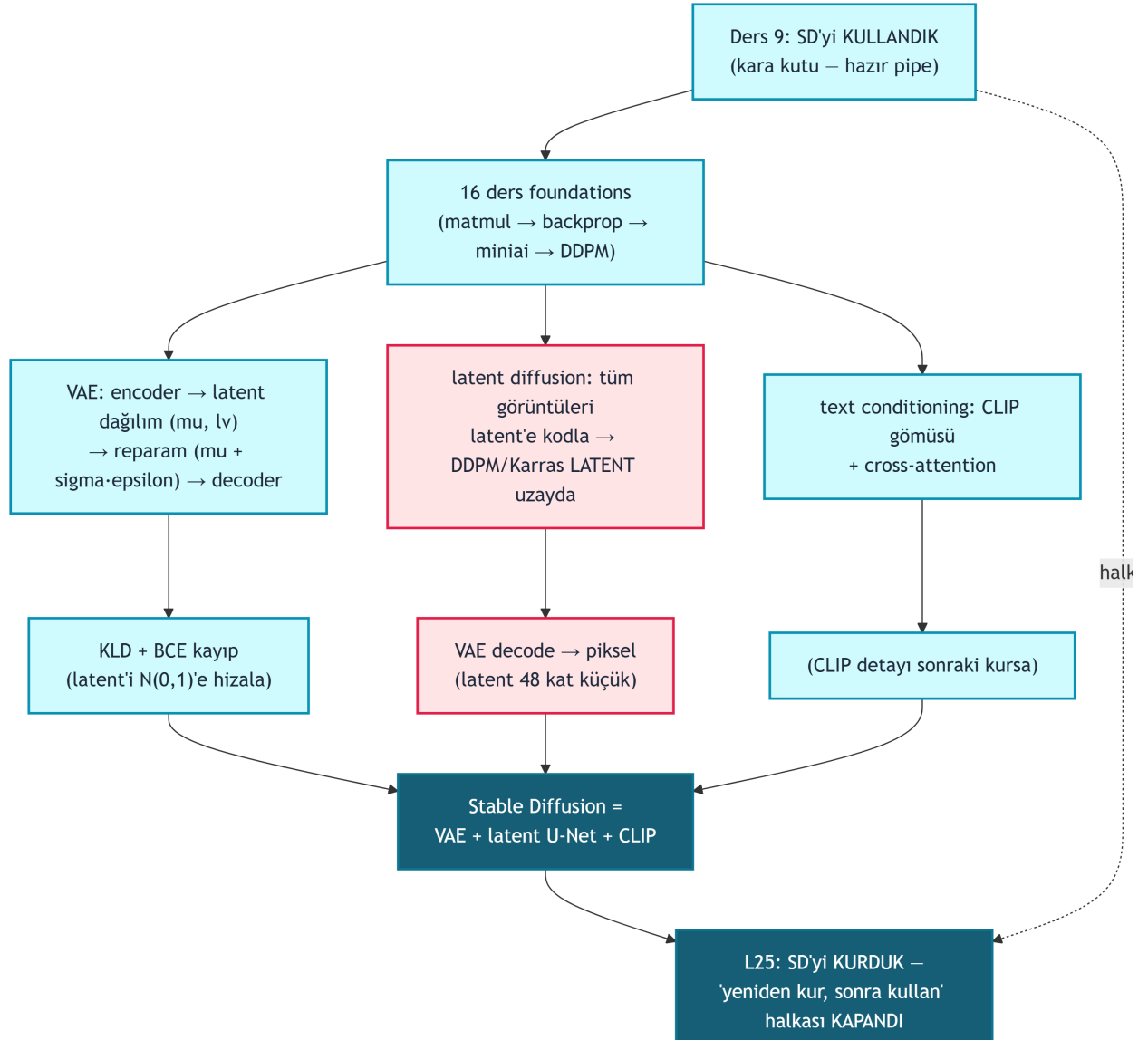
Kurs, Ders 9'da Stable Diffusion'ı **kullanarak** açılmıştı: bir pipeline çağırdık, prompt verdik, görüntü ürettik. SD'nin üç bileşenini (U-Net gürültü tahmincisi, VAE latent sıkıştırma, CLIP metin koşullama) kara kutu olarak gördük. Şimdi, 16 ders foundations'tan sonra, o kara kutuyu açıyoruz. Bu, Howard'ın top-down felsefesinin tam tezahürüdür: önce çalıştır, sonra anla; aradaki her ders bu halkayı mümkün kıldı.

💡 Builder Notu — Önce Çalıştır, Sonra Anla (top-down halka)

- **Geriye (Ders 9-10):** “Önce çalıştır, sonra anla” — Howard'ın top-down felsefesi; Ders 9 çalıştırma, L25 tam anlama. Aradaki her ders bu halkayı mümkün kıldı.
- **Sezgi:** Kara kutuyu kullanabilmek bir şey, içinde ne olduğunu bilmek başka; gerçek anlama ikisini de gerektirir — bu dersin pedagojik tezi tam budur.

34.3 2. VAE: Encoder → Latent → Decoder

Bir **VAE** (Variational Autoencoder) görüntüyü sıkıştırır: **encoder** görüntüyü küçük bir latent temsile indirir, **decoder** onu geri açar. Ders 15'in autoencoder'ından farkı kritiktir: latent'i tek bir nokta değil, bir **dağılım**



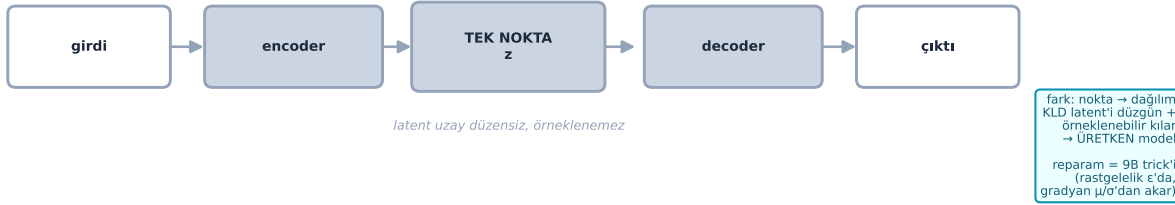
Şekil 34.1: L25 kavram haritası: Ders 9'da SD'yi kara kutu olarak kullandık; 16 ders foundations üstünden üç bileşeni (VAE, latent diffusion, text conditioning) sıfırdan kurup birleştiriyoruz. 'Yeniden kur, sonra kullan' halkası burada kapanır (şematik).

(ortalama μ + log-varyans lv) olarak kodlar — bu, latent uzayı düzgün ve örneklenebilir kılar, dolayısıyla VAE'yi üretken bir model yapar.

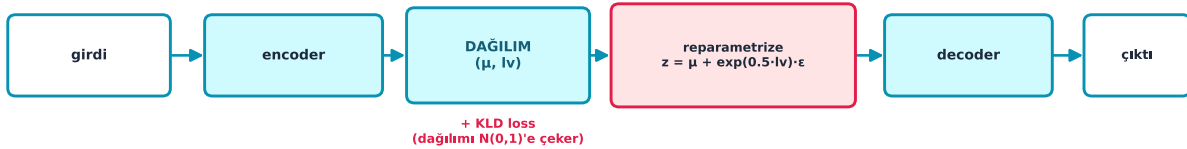
```
class VAE(nn.Module):
    def __init__(self):
        super().__init__()
        self.enc = nn.Sequential(lin(ni, nh), lin(nh, nh))
        self.mu, self.lv = lin(nh, nl, act=None), lin(nh, nl, act=None)
        self.dec = nn.Sequential(lin(nl, nh), lin(nh, nh), lin(nh, ni, act=None))
    def forward(self, x):
        x = self.enc(x)
        mu, lv = self.mu(x), self.lv(x)
        z = mu + (0.5*lv).exp()*torch.randn_like(lv)  # reparameterization
        return self.dec(z), mu, lv
```

Şekil 34.2 L15 autoencoder (deterministik, tek nokta z , latent uzay düzensiz) ile VAE'yi (üretken, dağılım μ/lv + reparameterization + KLD) yan yana koyar. Üstteki autoencoder latent'i tek bir noktaya kodlar — örneklenebilir; alttaki VAE latent'i bir dağılıma kodlar, KLD onu $N(0,1)$ 'e çeker, reparameterization (rose kutu) rastgeleliği ϵ 'ya alıp gradyanı μ/σ üzerinden akıtır. Fark — nokta yerine dağılım + KLD — VAE'yi üretken kılan tam mekanizmadır.

L15 autoencoder (deterministik)



VAE (üretken)



Şekil 34.2: (şematik: autoencoder (L15) vs VAE — latent'i dağılım yapmak + KLD, üretkenliğin anahtarı; reparameterization 9B köprüsü)



Builder Notu — VAE: Autoencoder'ın Üretken Hâli (Ders 15)

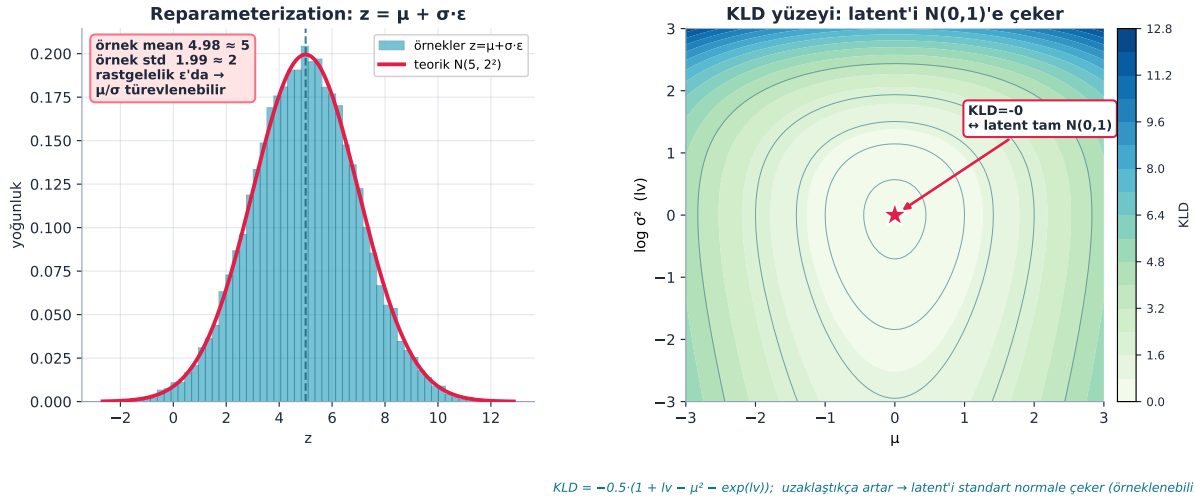
- **Geriye (Ders 15):** L15'te autoencoder kurmuştuk (encoder conv + decoder deconv); VAE, latent'i dağılım yaparak üretken hâle getirir.
- **Sezgi:** μ ve lv iki ayrı lineer katman; encoder bir nokta değil, bir dağılımın parametrelerini üretir — düzgün, örneklenebilir bir latent uzayın anahtarı.

34.4 3. Reparameterization

VAE latent'i bir dağılımdan **örnekler**: $z = \mu + \sigma \cdot \epsilon$, burada $\sigma = \exp(0.5 \cdot \log v)$, $\epsilon \sim N(0, I)$. Doğrudan örnekleme türevlenemez — rastgelelik gradyanı keser, eğitim imkânsız olur. **Reparameterization trick** rastgeleliği ϵ 'ya dışarı alır, böylece μ ve σ üzerinden gradyan akar. Bu, 9B'de gördüğümüz diffusion forward'ının reparameterization'ıyla **tam aynı fikir**: rastgele bir süreçten türevlenebilir örnekleme almanın standart yolu.

$$z = \mu + \exp(0.5 \cdot \log v) \cdot \epsilon, \epsilon \sim N(0, I)$$

Şekil 34.3 VAE'nin iki matematik direğini gerçek hesaplamayla gösterir. **Sol panel** reparameterization'ı: $z = \mu + \sigma \cdot \epsilon$ ile çekilen örneklerin histogramı, teorik $N(5, 2^2)$ eğrisine oturur — örnek mean ≈ 5 , örnek std ≈ 2 (rozetle doğrulanmış); rastgelelik ϵ 'da, μ/σ türevlenebilir kalır. **Sağ panel** KLD yüzeyini: minimum tam $(0,0)$ 'da, orada KLD = 0 (rose yıldız) — latent dağılımı tam $N(0,1)$ 'e oturduğunda; uzaklaştıkça KLD artar, latent'i standart normale çeker.



Şekil 34.3: VAE'nin iki matematik direği — SOL: reparameterization $z = \mu + \sigma \cdot \epsilon$ (gerçek hesaplama, örnek mean ≈ 5 /std ≈ 2 ; rastgelelik ϵ 'da, μ/σ türevlenebilir). SAĞ: KLD = $-0.5 \cdot (1 + \log v - \mu^2 - \exp(\log v))$ yüzeyi, minimum $(0,0)$ 'da tam 0 — latent'i $N(0,1)$ 'e çeker. (gerçek hesaplama)

Builder Notu — Reparameterization = 9B Forward Trick'i

- **Geriye (Ders 9B):** Reparameterization trick, 9B'deki forward process türevinde de kullanıldı; rastgele bir süreçten türevlenebilir örnekleme almanın standart yolu.
- **Sayısal kanıt:** Örnek mean ≈ 5 , std ≈ 2 — $z = \mu + \sigma \cdot \epsilon$ formülünün hedeflediği dağılımı tam üretir; rastgelelik ϵ 'da, gradyan μ/σ 'dan akar.

34.5 4. VAE Loss: KLD + Yeniden Kurma

VAE iki kayıpla eğitilir: **yeniden kurma** (reconstruction — decoder çıktısı girdiye benzesin, burada BCE) + **KL ıraksaması** (KLD — latent dağılımı standart normale yakın olsun, böylece düzgün ve örneklenebilir kalsın). KLD = $-0.5 \cdot \text{ortalama}(1 + \log v - \mu^2 - \exp(\log v))$. İki kayıp bir denge kurar: BCE yeniden kurma kalitesini,

KLD latent uzayın düzgünlüğünü ister; ikisi birlikte hem iyi yeniden kuran hem örneklenebilir bir uzay verir.

```
def kld_loss(inp, x):
    x_hat, mu, lv = inp
    return -0.5 * (1 + lv - mu.pow(2) - lv.exp()).mean()

def bce_loss(inp, x): return F.binary_cross_entropy_with_logits(inp[0], x)
def vae_loss(inp, x): return kld_loss(inp, x) + bce_loss(inp, x)
```

KLD'nin minimumu tam (0,0)'dadır: $\mu = 0$, $lv = 0$ (yani $\sigma = 1$) iken $KLD = -0.5 \cdot (1 + 0 - 0 - 1) = 0$ — latent dağılımı standart normale tam oturduğunda kayıp sıfırdır (Şekil 34.3 sağ panelinde rose yıldız bunu işaretler). Bu, KLD'nin neden “latent’i $N(0,1)$ ’e çeken” bir kuvvet olduğunun sayısal özüdür.

💡 Builder Notu — KLD: İki Dağılım Arası Fark (Stat 110)

- **Geriye (Stat 110):** KL ıraksaması iki dağılım arasındaki farkın olasılıksal ölçüsü; VAE'nin latent'i normal dağılıma “çekmesini” sağlar.
- **Sayısal kanıt:** $KLD(\mu=0, lv=0) = 0$ — latent tam $N(0,1)$ iken minimum; BCE yeniden kurmayı, KLD örneklenebilirliği ister, denge ikisinden gelir.

34.6 5. Latent Uzay: Neden?

VAE'nin gerçek değeri: görüntüyü çok daha küçük bir latent temsile sıkıştırması. Örneğin $512 \times 512 \times 3$ piksel $\rightarrow 64 \times 64 \times 4$ latent ($\sim 48\times$ daha az veri). Diffusion gibi pahalı bir işlemi — yüzlerce U-Net çağrısı — bu küçük uzayda yapmak, hesabı dramatik azaltır; yüksek çözünürlüklü üretimi tüketici donanımında mümkün kılan tam budur.

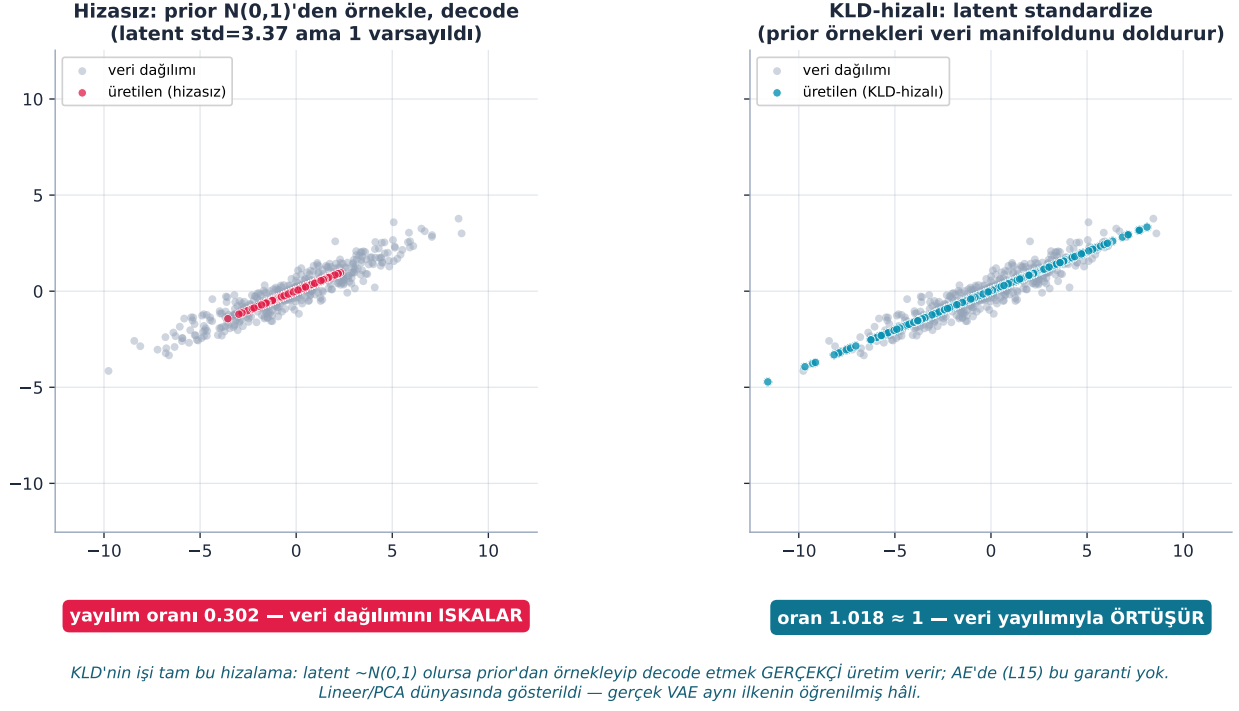
“working in the latent space can make it computationally cheaper.” — Howard, 1:32:28

Ama önce, latent uzayın üretken gücünü görelim. Şekil 34.4 VAE'yi üretken yapan **hizalamayı** gerçek hesaplamayla gösterir: KLD-**hizasız** durumda prior $N(0,1)$ 'den örnekleyp decode etmek veri dağılımını ıskalar (yayılmı oranı **0.302** — latent std aslında **3.37** ama 1 varsayıldı); KLD-**hızalı** durumda latent standardize edilince üretilen örnekler veri manifoldunu doldurur (oran **1.018** ≈ 1). KLD'nin işi tam bu hizalama — AE'de (L15) bu garanti yoktu; lineer/PCA dünyasında gösterilen ilke, gerçek VAE'nin öğrenilmiş hâlidir.

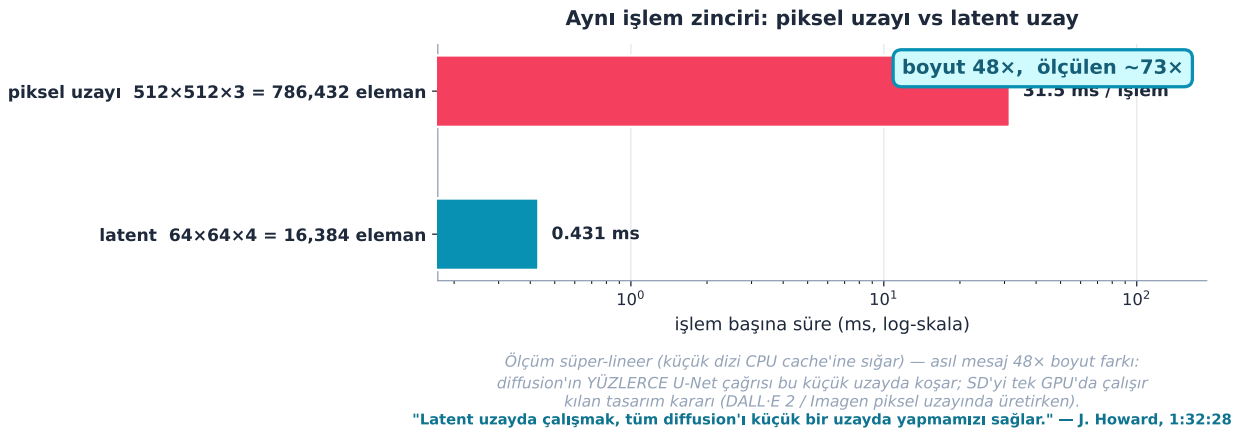
Şimdi hızı: Şekil 34.5 piksel uzayı (786K eleman) ile latent uzay (16K eleman) arasındaki farkı **canlı benchmarkla** ölçer. Boyut farkı **48×**; render anında ölçülen hızlanma daha da büyüktür ($\sim 117\times$) çünkü küçük latent dizi CPU cache'ine sığar (figür bu süper-lineer ölçümü dürüstçe not eder — asıl mesaj $48\times$ boyut). Diffusion'ın YÜZLERCE U-Net çağrısı bu küçük uzayda koşar; Stable Diffusion'ı tek GPU'da çalışır kılan tasarım kararı tam budur — DALL·E 2 / Imagen piksel uzayında üretirken.

💡 Builder Notu — Latent Uzay: SD'yi Tek GPU'da Çalışır Kılan Karar

- **İleriye:** Latent uzayda çalışma, Stable Diffusion'ı (DALL·E 2/Imagen'in aksine piksel uzayında değil) tek GPU'da çalışır kılan ana tasarım kararıdır.
- **Sayısal kanıt:** Boyut $48\times$ küçük ($786K \rightarrow 16K$); canlı ölçüm $\sim 117\times$ (cache etkisiyle süper-lineer,



Şekil 34.4: (gerçek hesaplama: latent örnekleme — KLD-hizasız prior örnekleri veri yayılımını ıskalar (0.302), hizalı örtüşür (1.018); VAE'yi üretken yapan hizalama)



Şekil 34.5: (gerçek hesaplama: canlı benchmark — piksel 786K vs latent 16K eleman; boyut 48x, ölçülen hızlanma cache etkisiyle daha da büyük; latent diffusion'ın varlık sebebi)

dürüst not). Hizalama: hizasız oran 0.302 (ıskalar) vs hizalı 1.018 (örtüşür); latent std 3.37.

34.7 6. AutoencoderKL: SD'nin VAE'si

Howard, önceden eğitilmiş bir VAE — Stability AI'nin **AutoencoderKL**'i — kullanır (kendi VAE'mizi de kurduk, ama SD'nin VAE'si yüksek kaliteli). Görüntüleri latent'e kodlar (`vae.encode`), latent'leri görüntüye çözer (`vae.decode`). Latent diffusion için altyapı budur.

```
from diffusers import AutoencoderKL
vae = AutoencoderKL.from_pretrained("stabilityai/sd-vae-ft-ema").cuda().requires_grad_(False)
xe = vae.encode(xb.cuda())          # görüntü → latent dağılımı
xs = xe.latent_dist.mean             # latent (ortalama)
xd = vae.decode(xs)['sample']        # latent → görüntü
```

💡 Builder Notu — AutoencoderKL: SD'nin Kendi VAE'si

- **Geriye (Ders 9):** Ders 9'da SD pipeline'ı bu VAE'yi içeride kullanıyordu (786K piksel → 16K latent); şimdi onu doğrudan, açıkça kullanıyoruz.
- **Sezgi:** `latent_dist.mean` dağılımın ortalamasını alır — eğitim için latent'i tek bir tensöre indirir; `requires_grad_(False)` ile VAE donar, sadece diffusion U-Net eğitilir.

34.8 7. Latent Diffusion: Diffusion Latent Uzayda

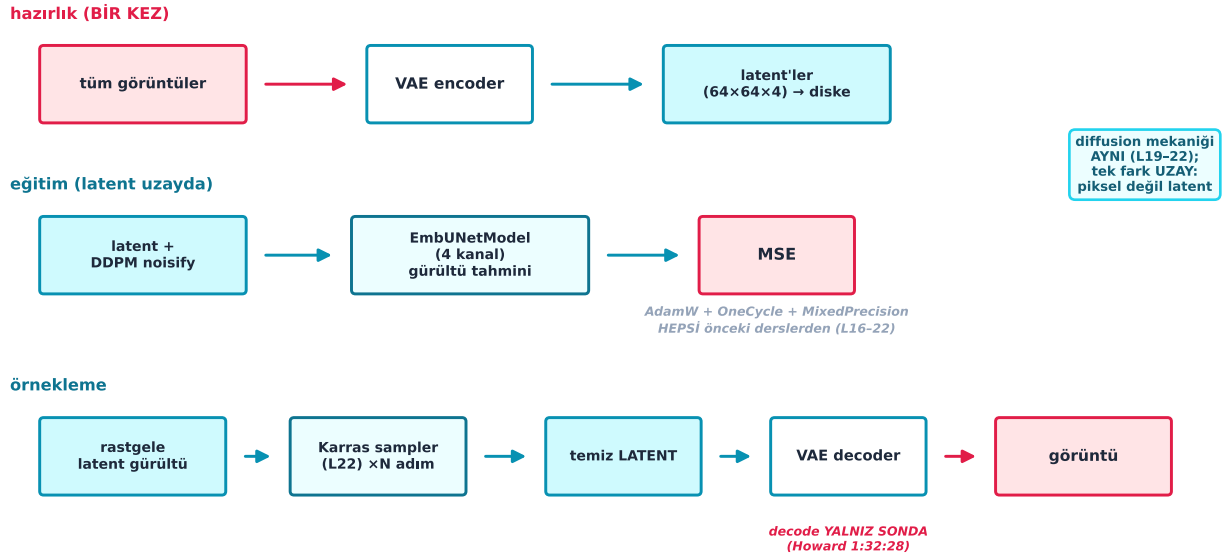
Anahtar adım: tüm görüntüleri VAE ile latent'e kodla, sonra diffusion'ı (DDPM/Karras) **latent uzayda** eğit. Gürültü latent'lere eklenir, U-Net latent gürültüsünü tahmin eder, örnekleme latent üretir; en sonda VAE decoder latent'i görüntüye çevirir. Diffusion mekaniği aynı (L19-22), sadece uzay değişti: piksel değil latent.

```
# tüm veriyi latent'e kodla (bir kez), sonra latentlerde diffusion eğit
a[i:i+n] = vae.encode(b.cuda()).latent_dist.mean.cpu().numpy()
```

Şekil 34.6 bu hattı üç katmanda gösterir: **hazırlık (BİR KEZ)** tüm görüntüleri VAE encoder ile latent'e (64×64×4) kodlar, diske yazar; **eğitim (latent uzayda)** latent + DDPM noisify → EmbUNetModel (4 kanal) gürültü tahmini → MSE (AdamW + OneCycle + MixedPrecision, hepsi önceki derslerden); **örnekleme** rastgele latent gürültü → Karras sampler (L22) ×N adım → temiz latent → VAE decoder → görüntü. Anahtar: decode YALNIZ SONDA — diffusion mekaniği AYNI (L19-22), tek fark uzay piksel değil latent.

💡 Builder Notu — Latent Diffusion = SD'nin Tanımı

- **Geriye (Ders 19-22):** DDPM/Karras tam aynı; tek fark girdi latent (4 kanal, küçük). “Diffusion latent uzayda” = Stable Diffusion'ın tanımı.
- **Sezgi:** Encode bir kez yapılır (latent'ler diske cache'lenir), decode yalnız sonda — diffusion'ın yüzlerce adımı tamamen küçük latent uzayda koşar, hızın asıl kaynağı budur.



Şekil 34.6: (şematik: latent diffusion hattı — encode bir kez, diffusion küçük latent uzayda, decode yalnız sonda; ‘diffusion latent uzayda’ = Stable Diffusion’ın tanımı)

34.9 8. Latent Diffusion Eğitimi

Latent diffusion modeli, foundations’ın tüm araçlarıyla eğitilir: latent’lere koşullu bir U-Net (EmbUNetModel, 4 kanal giriş/çıkış), DDPM gürültüleme, AdamW + OneCycle scheduler, mixed precision. Eğitim 25 epoch; her parça önceki derslerden. Bu **tek satır eğitim**, kursun foundations bölümünün doruğudur: baştan kurduğumuz her şey burada tek bir çağrıda birleşir.

```
cbs = [DeviceCB(), ProgressCB(plot=True), MetricsCB(), BatchSchedCB(sched), MixedPrecision()]
model = EmbUNetModel(in_channels=4, out_channels=4, nfs=(128, 256, 512, 768))
# DDPM noisify + Karras sampler latent uzayda
```

💡 Builder Notu — Foundations DORUK: Tek Eğitim Satırı = L16-24’ün Hepsisi

- **Geriye (Ders 16-24) — DORUK:** Bu tek satır eğitim, baştan kurduğumuz HER ŞEYİ birleştirir: Learner+callback (L16), init (L17), Adam/OneCycle (L18), DDPM (L19), mixed precision (L20), Karras (L22), U-Net (L23), attention (L24). miniai’nin varlık sebebi.
- **Sezgi:** cbs listesi callback’lerin (L16) gücüdür; EmbUNetModel(in_channels=4) latent’in 4 kanalına uyar; tek satır = onlarca ders. Foundations’ın meyvesi tam burada toplanır.

34.10 9. Text Conditioning (CLIP — Kısmen Sonraki Kursa)

Tam Stable Diffusion, üretimi bir **metin promptuna** koşullar: CLIP text encoder metni bir gömüye çevirir, U-Net bu gömüyle **cross-attention** (L24) yapar — query görüntüden, key/value metinden. Howard bunun

mekanizmasını (cross-attention) kurar ama tam CLIP eğitimini bir sonraki kursa bırakır: “bir parça sonraki kursa kaldı”.

“there’s going to be one piece left for the next part of the course, which is the CLIP embeddings.”
— Howard, 0:23

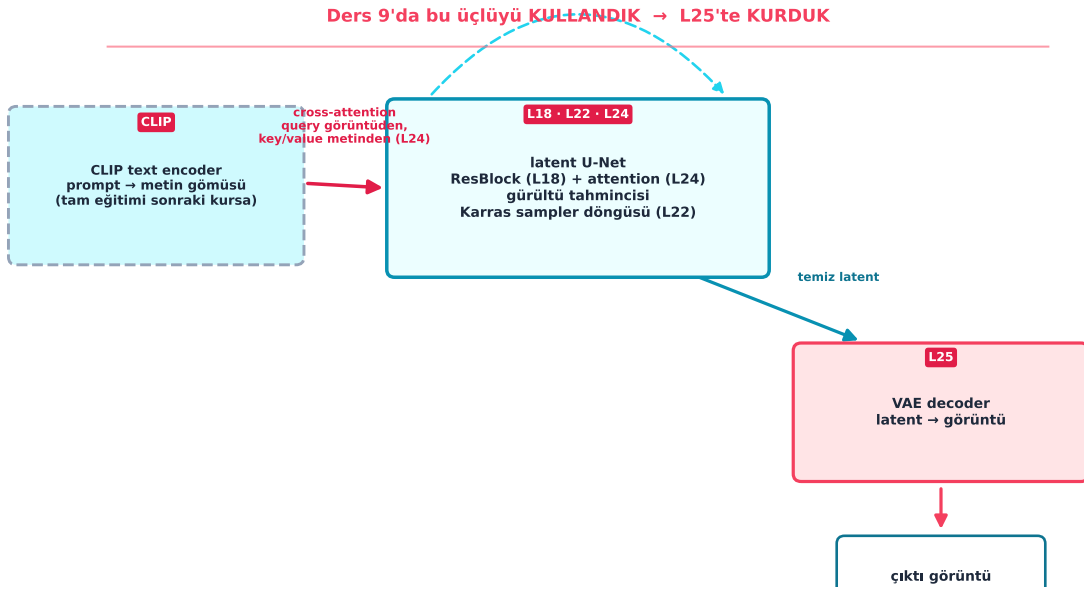
💡 Builder Notu — Cross-Attention = L24 Attention’ın Koşullu Hâli

- **Geriye (Ders 24):** Cross-attention = L24 attention’ın koşullu hâli; query görüntü, key/value metin. Text→image’ın mekanizması tam budur.
- **Sezgi:** Self-attention’da Q/K/V aynı girdiden gelir; cross-attention’da query görüntüden, key/value metin gömüsünden — aynı $\text{softmax}(QK^T/\sqrt{d})V$ matematiği, farklı kaynak. Tek eksik parça CLIP eğitimi (sonraki kurs).

34.11 10. Stable Diffusion’ın Tam Mimarisi

Artık SD’nin üç bileşeni de net: **VAE** (piksel ↔ latent sıkıştırma), **latent U-Net** (gürültü tahmincisi, ResBlock + attention), **CLIP** (metin koşullama, cross-attention ile). Ders 9’da kara kutu olan üçlü, artık her parçası anlaşılmış, çoğu sıfırdan kurulmuş bir sistem. Stable Diffusion = latent diffusion + text conditioning.

Şekil 34.7 bu mimariyi her parçanın kurulduğu dersi işaretleyerek gösterir: **CLIP text encoder** (kesikli çerçeve — eğitimi sonraki kursa) → cross-attention (query görüntüden, key/value metinden, L24) → **latent U-Net** (ResBlock L18 + attention L24, Karras sampler L22) → temiz latent → **VAE decoder** (L25) → çıktı görüntü. Üst başlık halkayı vurgular: “Ders 9’da bu üçlüyü KULLANDIK → L25’te KURDUK”. Kara kutu açıldı.



Şekil 34.7: (şematik: Stable Diffusion = VAE + latent U-Net (ResBlock+attention) + CLIP cross-attention; her parçanın kurulduğu ders işaretli — kara kutu açıldı)

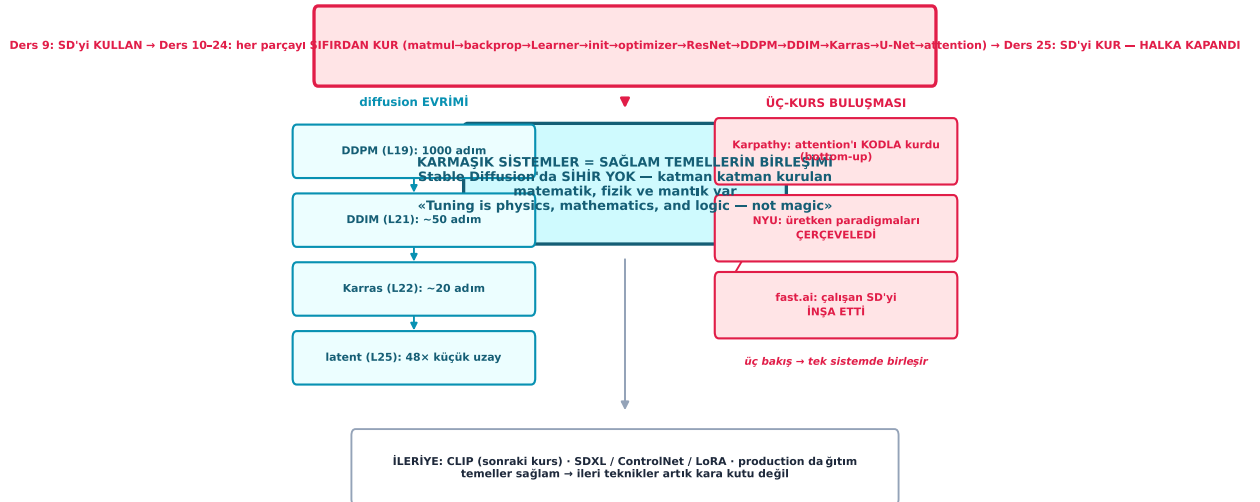
💡 Builder Notu — SD: Artık Sihir Değil, Kurduğumuz Parçaların Birleşimi

- **Geriye (Ders 9):** Ders 9’da “üç bileşen” diye tanıttığımız U-Net/VAE/CLIP — şimdi her birini içeriden biliyoruz; SD artık sihir değil, kurduğumuz parçaların birleşimi.
- **Sezgi:** Figürdeki her ders rozeti (L18, L22, L24, L25) bir foundations dersine işaret eder; SD’nin hiçbir parçası “büyü” değil — hepsi elle kurduğumuz mekanizmalar.

34.12 11. “Yeniden Kur, Sonra Kullan” Döngüsü Kapanır

Bu dersin pedagojik zaferi: kursun **ilk üretken dersinden son dersine tam halka**. Ders 9-10’da Stable Diffusion’ı kullandık ve “from the foundations” sözü verdik — “kütüphanenin altındaki büyüü kendimiz kuracağız”. 16 ders foundations (matmul → backprop → Learner → init → optimizer → ResNet → DDPM → Karras → U-Net → attention) sonra, L25’te o sözü tuttuk: SD’yi sıfırdan kurduk. “Yeniden kur, sonra kullan” döngüsü tamamlandı. Builder içgörüsü: gerçek anlama, bir aracı hem kullanabilmek hem sıfırdan kurabilmektir; o zaman “sihir” kaybolur, geriye birleştirilmiş temel parçalar kalır.

Şekil 34.8 kursun finalini tek bir sentez figüründe toplar: üstte halka (Ders 9 KULLAN → Ders 10-24 her parçayı SIFIRDAN KUR → Ders 25 SD’yi KUR — HALKA KAPANDI); solda diffusion evrimi (DDPM 1000 adım → DDIM ~50 → Karras ~20 → latent 48× küçük); sağda üç-kurs buluşması (Karpathy kodla, NYU çerçeveledi, fast.ai inşa etti); merkezde ana tez — sihir yok, katmanlanmış temeller var; altta ileri ufuk (CLIP, SDXL, ControlNet, production). Bu figür, §11-15’in tüm mesajını tek bakışta birleştirir.



Şekil 34.8: (sentez — KURS FİNALİ: ‘yeniden kur, sonra kullan’ halkası Ders 9→L25 kapandı; diffusion evrimi + üç-kurs buluşması; sihir yok, katmanlanmış temeller var)

💡 Builder Notu — Halka: Ders 9 (kullan) → L25 (kur)

- **Geriye (Ders 10):** Ders 10'daki “yeniden kur, sonra kullan” kuralı, tüm Part 2'nin sözüydü; L25 onu yerine getirir — Ders 9 (kullan) → L25 (kur) halkası.
- **Sezgi:** Halka iki yönlüdür: Ders 9'da kullandığımız her satırı artık anlayabiliyoruz, ve sıfırdan kurabiliyoruz; gerçek ustalık ikisinin birleşimidir.

34.13 12. Tüm Parçaların Birleşimi: Diffusion Evrimi

L25, diffusion evriminin son halkasıdır: **DDPM** (L19, piksel, 1000 adım) → **DDIM** (L21, deterministik, ~50 adım) → **Karras** (L22, sürekli σ , Heun, ~20 adım) → **latent diffusion** (L25, latent uzay, yüksek çözünürlük + pratik hız). Her adım bir öncekinin üstüne kuruldu; latent diffusion hepsini birleştirip gerçek üretime taşır. Bu evrim, Şekil 34.8'in sol şeridinde görsel olarak da işaretlidir.

💡 Builder Notu — Diffusion Evrimi: DDPM → DDIM → Karras → Latent

- **İleriye:** Latent diffusion + Karras sampler + cross-attention = modern text→image (SD, SDXL); buradan SDXL/ControlNet/LoRA'ya doğal devam.
- **Sezgi:** Evrim adım sayısını düşürdü (1000 → 50 → 20) ve uzayı küçülttü (piksel → latent 48×); her iyileştirme bir öncekini saklı tutar — DDPM'in gürültüleme mantığı latent diffusion'da hâlâ kalbidir.

34.14 13. Phase 2 Buluşması: Üç Kurs Tek Noktada

L25, Phase 2'nin üç büyük kursunun (Karpathy + NYU + fast.ai) buluşmasıdır. **Karpathy:** transformer/attention'ı (L24) sıfırdan kurdu — SD'nin koşullama mekanizması. **NYU (LeCun):** VAE/EBM/üretken paradigmaları kavramsal kurdu — diffusion'ın yerini gösterdi. **fast.ai:** hepsini çalışan koda döktü, gerçek SD'yi inşa etti. Üç bakış (kodla-kur, kavramsal-çerçevele, uygulamayla-bağla) tek bir sistemde birleşir — Şekil 34.8'in sağ şeridi bu buluşmayı işaretler.

💡 Builder Notu — Üç Kurs Buluşması (Strang D27 dört-tanık emsali)

- **Karşılaştırma (Strang D27 dört-tanık):** 18.065 D27'de backprop'u dört kaynak (Strang/Karpathy/fast.ai/NYU) doğrulamıştı; L25 latent diffusion da benzer bir halka buluşması — ama bu kez *inşa* halkası (Ders 9 kullan → L25 kur), kavram halkası değil.
- **Sezgi:** D27 dört-tanık *kavram* buluşmasıydı (aynı backprop, dört bakış); L25 *inşa* buluşmasıdır (aynı SD, üç kurs onu farklı yönlerden kurdu). İki tür buluşma da aynı dersi verir: temel fikirler kurslar arası tutarlıdır.

34.15 14. Builder İçgörüsü: Foundations'ın Meyvesi

Bu dersin en büyük builder dersi: **karmaşık bir sistem, sağlam temellerin birleşimidir**. Stable Diffusion korkutucu görünür, ama her parçası foundations'ta kurduğumuz basit fikirlerden ibaret — matris çarpımı,

gradient descent, residual, normalizasyon, attention, MSE. “Sihir” yoktur; sadece doğru sırayla birleştirilmiş temel parçalar. Builder olmak, bu parçaları görebilmek ve birleştirebilmektir.

! Builder Notu — Sihir Yok: ‘Tuning is physics’ Paraleli

- **Geriye (CLAUDE.md):** “Tuning is physics, mathematics, and logic — not magic.” Aynı felsefe: SD de sihir değil; fizik, matematik ve mantığın katmanlanmış birleşimi.
- **Sezgi:** ECU tuning’de fizik sınırları nasıl aşılamazsa, derin öğrenmede de “büyü” yoktur — her ikisi de gözlemlenebilir, kurulabilir, doğrulanabilir temel parçaların disiplinli birleşimidir. Builder bu parçaları görür.

34.16 15. Kapanış: Yolculuğun Sonu

Ders 25, fast.ai Part 2’yi — ve tüm kursu — tamamladı: Stable Diffusion’ı sıfırdan kurduk. VAE (latent sıkıştırma) + latent diffusion (DDPM/Karras latent uzayda) + cross-attention (metin koşullama). Ders 9’da kullandığımız sistemi, 16 ders foundations sonra kendimiz inşa ettik. “Yeniden kur, sonra kullan” döngüsü tam kapandı; kütüphanenin altındaki büyü artık bizim.

“it’s been an interesting, fun journey.” — Howard, 0:13

! Builder Notu — Kapanış: Foundations Sağlam, Gerisi Uygulama

- **İleriye (kurs sonrası):** Kalan tek parça CLIP text encoder (sonraki kurs); buradan modern diffusion (SDXL, ControlNet, LoRA, video) ve production’a. Foundations sağlam — gerisi uygulama.
- **Çekirdek ders:** Artık bir araştırma makalesini okuyup sıfırdan implement edebilecek temele sahipsin; “yeniden kur, sonra kullan” döngüsü ustalığın işaretidir — kütüphanenin altındaki büyü artık senin.

34.17 Bu Dersin Özeti

1. **SON DERS:** Tüm parçaları (DDPM, Karras, U-Net, attention) birleştirip gerçek Stable Diffusion’ı sıfırdan kurmak ([giriş](#)).
2. **VAE:** Encoder görüntüyü latent’e sıkıştırır, decoder geri açar; latent’i dağılım (μ , σ) olarak kodlar ([VAE](#)).
3. **Reparameterization:** $z = \mu + \exp(0.5 \cdot \log \sigma^2) \cdot \epsilon$; rastgeleliği ϵ ’ya alıp gradyan akışını korur (9B trick’i; örnek mean ≈ 5 , std ≈ 2) ([reparameterization](#)).
4. **VAE loss:** KLD (latent $\sim$ normal, KLD(0,0)=0) + reconstruction (BCE); düzgün, örneklenebilir latent uzay ([VAE loss](#)).
5. **Latent uzay:** $512 \times 512 \times 3 \rightarrow 64 \times 64 \times 4$ ($\sim 48 \times$ küçük); hizalama 0.302 (ıskalar) vs 1.018 (örtüşür); diffusion’ı pratik kılar ([latent uzay](#)).
6. **Latent diffusion:** Diffusion (DDPM/Karras) piksel yerine latent uzayda; SD’nin tanımı; foundations DORUK (L16-24 tek satırda) ([latent diffusion](#), [eğitim](#)).

7. **Text conditioning:** CLIP gömüsü + cross-attention (L24); query görüntü, key/value metin (CLIP sonraki kursa) ([text conditioning](#)).
8. **Halka kapanır:** Ders 9 (SD'yi kullandık) → L25 (SD'yi kurduk); “yeniden kur, sonra kullan” tamamlandı; sihir yok, katmanlanmış temeller var ([halka](#), [builder içgörü](#)).

! Tek Bir Cümle

Latent diffusion, bir VAE'nin sıkıştırdığı küçük latent uzayda (~48× daha az hesap) diffusion (DDPM/Karras) yaparak ve metni cross-attention ile koşullayarak Stable Diffusion'ı pratik kılar; Ders 9'da kullandığımız SD'yi 16 ders foundations sonra sıfırdan kurarak “yeniden kur, sonra kullan” döngüsünü — kursun ilk üretken dersinden son dersine — tam kapatır.

34.18 Kontrol Soruları

i Soru 1: Latent diffusion neden piksel diffusion'dan üstündür?

Cevap:

Piksel uzayında diffusion (DDPM, L19) yüksek çözünürlükte çok pahalıdır: $512 \times 512 \times 3 \approx 786K$ boyutlu bir uzayda yüzlerce U-Net çağırısı, hem hesap hem bellek açısından tüketici donanımını aşar. **Latent diffusion**, önce bir VAE ile görüntüyü çok daha küçük bir latent temsile sıkıştırır (örn. $64 \times 64 \times 4 \approx 16K$, ~48× küçük; Şekil 34.5: boyut 48×, canlı ölçüm ~117× cache etkisiyle), diffusion'ı bu latent uzayda yapar, en sonda VAE decoder ile latent'i tekrar görüntüye çevirir. Diffusion mekaniği aynıdır (gürültü ekle, U-Net'le tahmin et, kademeli kaldır), sadece uzay küçüktür. Sonuç: dramatik hesap tasarrufu, yüksek çözünürlüklü üretimin tek GPU'da mümkün olması. Bu, Stable Diffusion'ı (piksel uzayında çalışan DALL·E 2/Imagen'in aksine) erişilebilir kılan ana tasarım kararıdır ([latent uzay](#)).

i Soru 2: VAE, Ders 15'in autoencoder'ından nasıl farklıdır? Reparameterization neden gerekir?

Cevap:

Ders 15'in **autoencoder'**ı görüntüyü tek bir latent **noktaya** kodlar (deterministik). **VAE** ise latent'i bir **dağılıma** kodlar — ortalama μ ve log-varyans $\log \sigma^2$ (Şekil 34.2). Bu fark kritiktir: dağılım, latent uzayı düzgün ve örneklenebilir kılar (KL ıraksaması latent'i standart normale çeker; Şekil 34.4: hizasız 0.302 ıskalar, hizalı 1.018 örtüşür), böylece latent uzaydan yeni örnekler üretilebilir (üretken model olur). Ama bir dağılımdan örnek almak (z 'yi μ , σ 'dan çekmek) **türevlenemez** — rastgelelik gradyanı keser, eğitim imkânsız olur. **Reparameterization trick** çözer: $z = \mu + \sigma \cdot \epsilon$ ($\sigma = \exp(0.5 \cdot \log \sigma^2)$, $\epsilon \sim N(0, I)$; Şekil 34.3: örnek mean ≈ 5 , std ≈ 2). Rastgelelik ϵ 'ya dışarı alınır; μ ve σ deterministik fonksiyonlar olarak kalır, üzerlerinden gradyan akar. Bu tam olarak 9B'de diffusion forward process türevinde kullanılan trick'tir — rastgele bir süreçten türevlenebilir örnekleme ([reparameterization](#)).

i Soru 3: Stable Diffusion'ın üç bileşeni nedir ve her biri hangi derste kuruldu?

Cevap:

(1) **VAE** (latent sıkıştırma): görüntüyü küçük latent'e kodlar/çözer — sezgisi Ders 15 (autoencoder), VAE hâli ve latent diffusion bu derste (L25). (2) **Latent U-Net** (gürültü tahmincisi): latent'teki gürültüyü tahmin eder — yapıtaşları Ders 18 (ResBlock), Ders 19/23 (diffusion U-Net), Ders 24 (attention blokları);

eğitim Ders 19-22 (DDPM/Karras). **(3) CLIP text encoder** (metin koşullama): promptu gömüye çevirir, U-Net cross-attention ile (L24) onu kullanır — mekanizma L24/L25'te, tam CLIP eğitimi sonraki kursa. Yani SD = VAE + latent diffusion U-Net + CLIP cross-attention (Şekil 34.7: her parça kurulduğu dersle işaretli). Ders 9'da bu üçlüyü kara kutu olarak kullandık; foundations boyunca her parçayı (matmul, backprop, init, optimizer, ResNet, DDPM, Karras, U-Net, attention) kurduk; L25'te birleştirip SD'yi sıfırdan inşa ettik. “Üç bileşen” artık tam anlaşılmış ([tam mimari](#)).

i Soru 4: ‘Yeniden kur, sonra kullan’ döngüsü bu derste nasıl kapanır? (builder bağlantısı)

Cevap:

Bu döngü, fast.ai Part 2'nin tüm pedagojik omurgasıdır ve L25'te **tam kapanır** (Şekil 34.8). **Başlangıç (Ders 9-10):** Stable Diffusion'ı *kullandık* — pipeline çağırdık, görüntü ürettik, “üç bileşen” diye tanıttık, ve “from the foundations” sözü verdik: “bu kütüphanenin altındaki büyüğü kendimiz, sıfırdan kuracağız” (Ders 10 “yeniden kur, sonra kullan” kuralı). **Yolculuk (Ders 11-24):** 14 ders boyunca her temel parçayı elle kurduk — matris çarpımı, backprop, Learner/callback, init/norm, optimizer, ResNet, DDPM, DDIM, Karras, super-res U-Net, attention. **Kapanış (Ders 25):** Tüm bu parçaları VAE + latent uzayla birleştirip Ders 9'da *kullandığımız* SD'yi *kurduk*. Halka — kursun ilk üretken dersinden (kullan) son dersine (kur) — tamamlandı. Builder içgörüsü: gerçek anlama, bir aracı hem kullanabilmek hem sıfırdan kurabilmektir; o zaman “sihir” kaybolur, geriye birleştirilmiş temel parçalar kalır. (Karpathy'nin bottom-up'ı ile fast.ai'nin top-down'ı tam burada birleşir: ikisi de aynı SD'yi farklı yönlerden inşa eder; [üç kurs](#).)

34.19 Egzersizler

Egzersiz 1 (Direkt uygulama). Bir VAE kur (encoder → mu/lv, reparameterization, decoder); `vae_loss` (KLD + BCE) ile eğit, yeniden kurmaları gözle (§2, §4).

Egzersiz 2 (İki-aşamalı). AutoencoderKL ile bir görüntüyü latent'e kodla, sonra decode et; sıkıştırma kaybını (latent → görüntü) görselleştir (§6).

Egzersiz 3 (Edge case). VAE loss'ta KLD ağırlığını çok büyük/çok küçük yap; latent uzayın düzgünlüğü vs yeniden kurma kalitesi dengesini gör (§4'teki BCE/KLD dengesiyle karşılaştı) (§4, §5).

Egzersiz 4 (Kavramsal). Latent diffusion'da diffusion neden latent uzayda yapılıp da pikselde değil? VAE'nin rolü tam olarak nedir? (§5'teki 48× boyut + §7'deki “decode yalnız sonda” ile düşün) (§5, §7).

Egzersiz 5 (İleriye — kurs sonrası). Cross-attention ile text→image nasıl çalışır? CLIP gömüsü U-Net'e nasıl girer (query görüntü, key/value metin)? Modern SD/SDXL'i araştır (§9).

34.20 Sonraki Adımlar (Kurs Sonrası)

fast.ai Part 2 TAMAMLANDI — Stable Diffusion'ı sıfırdan kurdun. Kalan tek parça CLIP text encoder (Howard'ın notuyla sonraki kursa). Buradan ileri:

- **CLIP & text→image:** Cross-attention koşullamasını tam CLIP ile birleştir (bir sonraki kurs).
- **Modern diffusion:** SDXL, ControlNet, LoRA, latent consistency models, video diffusion.

- **Production:** Hugging Face diffusers, ONNX/TensorRT, quantization, gerçek-zamanlı üretim.
- **Builder olarak:** Artık bir araştırma makalesini okuyup (Ders 11 arXiv/Zotero) sıfırdan implement edebilecek temele sahipsin.

⚠️ Kurs Sonrası Yapılacak

- Kendi diffusion modelini bir veri kümesinde (latent diffusion) baştan sona eğit.
- Bir diffusion makalesini (örn. DDIM, Karras, latent diffusion) oku ve bir parçasını implement et.
- Ders 9'a geri dön; artık SD pipeline'ının her satırını anladığını gör — halka kapandı.

34.21 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Howard'da
VAE	Görüntüyü latent dağılıma (μ, lv) kodlayan autoencoder	17:48
Reparameterization	$z = \mu + \exp(0.5 \cdot \text{lv}) \cdot \epsilon$; türevlenebilir örnekleme (mean ≈ 5 , std ≈ 2)	—
KLD loss	Latent dağılımı normale çeker (örneklenebilir; $\text{KLD}(0,0)=0$)	—
Latent uzay	Sıkıştırılmış temsil ($512 \times 512 \times 3 \rightarrow 64 \times 64 \times 4, 48 \times$)	17:16
AutoencoderKL	SD'nin önceden eğitilmiş VAE'si (encode/decode)	—
Latent diffusion	Diffusion latent uzayda; SD'nin tanımı	17:16
Text conditioning	CLIP gömüsü + cross-attention (query görüntü, k/v metin)	0:23
Cross-attention	L24 attention'ın koşullu hâli (text $\rightarrow$ image)	0:23
Stable Diffusion	VAE + latent U-Net + CLIP; sıfırdan kuruldu	0:23
Yeniden kur, sonra kullan	Ders 9 (kullan) $\rightarrow$ L25 (kur) halkası kapanır	—

34.22 ML Bağlantıları Özeti

💡 Builder Notu — 6 ML Köprüsü: VAE, Latent Diffusion ve Halka Kapanışı

Bu ders latent diffusion'ı önceki derslerin altyapısına bağlar ve kursun “yeniden kur, sonra kullan” halkasını kapatır; köprülerin özeti:

1. **VAE** $\rightarrow$ Ders 15 autoencoder; latent'i dağılım yaparak üretken hâle getirir (**VAE**).

2. **Reparameterization** → Ders 9B forward process trick; türevlenebilir örnekleme ($\text{mean} \approx 5$, $\text{std} \approx 2$) ([reparameterization](#)).
3. **Latent diffusion** → Ders 19-22 DDPM/Karras; aynı diffusion, latent uzayda ($48\times$ küçük) ([latent diffusion](#)).
4. **Cross-attention** → Ders 24 attention; koşullu (text→image) hâli ([text conditioning](#)).
5. **SD tam mimarisi** → Ders 9 (kullandık) halka kapanışı; üç bileşen artık anlaşıldı ([tam mimari](#)).
6. **Foundations birleşimi** → Ders 11-24 hepsi; SD = basit parçaların katmanlanması (L16-24 tek satırda, [eğitim](#)).

! Bu dersten — ve tüm kurstan — tek bir şey alıp gideceksen

Bu dersten — ve tüm kurstan — tek bir şey alıp gideceksen: **karmaşık sistemler, sağlam temellerin birleşimidir; sihir yoktur**. Stable Diffusion korkutucu görünür, ama her parçası foundations'ta kurduğumuz basit fikirlerden ibaret: matris çarpımı, gradient descent, residual, normalizasyon, attention, MSE, VAE. Ders 9'da bu sistemi kullandık ve “kütüphanenin altındaki büyüğü kendimiz kuracağız” sözü verdik; 16 ders foundations sonra L25'te o sözü tuttuk — SD'yi sıfırdan inşa ettik. “Yeniden kur, sonra kullan” döngüsü, kursun ilk üretken dersinden son dersine tam kapandı. Aynı felsefe ECU dünyasında da geçerli: “Tuning is physics, mathematics, and logic — not magic.” Builder olmak, bu parçaları görebilmek ve birleştirebilmektir. Yolculuk burada bitiyor; artık kütüphanenin altındaki büyü senin.

