Sıfırdan Sinir Ağları — Karpathy Neural Networks: Zero to Hero'dan

ML Builder için Türkçe Notlar

Phase 2

2026-06-07

İçindekiler

1 Önsöz

2 Bu kitap nedir?

Bu, **Andrej Karpathy — Neural Networks: Zero to Hero** ders serisinin Türkçe ders notlarıdır. Hedef, videoları izlerken paralel okunabilecek; sonradan tek başına da yeterli olabilecek bir referans seti üretmek.

Serinin tek bir iddiası var ve her ders onu biraz daha somutlaştırır: bir sinir ağı **sihir değildir**. Boş bir Jupyter notebook'tan başlayıp, GPT ölçeğine kadar, her parçayı kendi elinle kurarsın. Karpathy'nin sözüyle: *"micrograd is what you need to train your networks, and everything else is just efficiency."* Yani bir ağı eğiten matematiğin tamamı yaklaşık 100 satırlık bir motorda; gerisi — tensörler, GPU, milyar parametre — yalnızca verimliliktir.

Her bölüm bir **Builder Notu** katmanı taşır: kavramın hem **geriye** (calculus zincir kuralı, lineer cebir dot product, olasılık) hem de **ileriye** (PyTorch, GPU, üretim) köprüsü. micrograd'ın `backward()`'ı PyTorch autograd'ın skaler prototipidir; `Neuron / Layer / MLP` API'si `torch.nn.Module` ile birebir hizalanır. Bu seriyi "tek başına ML teorisi" olarak değil, **kendi elinle kurarak öğrenme** disipliniyle okuyoruz: yeniden kur, sonra kullan.

> **Kaynak**
>
> - **Seri:** Neural Networks: Zero to Hero (YouTube playlist) — Andrej Karpathy
> - **Yazar:** Andrej Karpathy — eski Tesla AI direktörü, OpenAI kurucu ekibi
> - **Ders 1 repo:** github.com/karpathy/micrograd
> - **Ders depoları:** github.com/karpathy/nn-zero-to-hero
> - **Çeviri ve genişletme:** Phase 2 (TR + ML Builder köprüleri)

3 Nasıl Okumalı

Sıralı oku. Seri kümülatiftir — her ders bir öncekinin kurduğu motoru ve **dili** kullanır. micrograd (Ders 1) bütün serinin temelidir; makemore dersleri (2-6) dil modellemeyi sıfırdan kurar; sonraki dersler GPT'ye kadar tırmanır.

Karpathy'nin önerdiği akış: önce videoyu izle, sonra ilgili dersi oku, en sonunda **kodu kendin yaz**. Bu set videoyu **destekler**, ikame etmez.

> Pratik bir tavsiye
>
> Her bölüm sonundaki **egzersizleri** atlama. Karpathy'nin felsefesi tek cümle: kütüphaneyi kapağı açıkken gör. `Value` sınıfını sıfırdan yaz, gradient check yap, küçük bir MLP'yi elle eğit. Aynı forward → backward → update döngüsünü ileride PyTorch'ta yıllarca farklı kılıklarda yazacaksın; bir kez sıfırdan kurmak onu kara kutu olmaktan çıkarır.

4 10 Ders

Seri micrograd'dan başlayıp GPT'ye doğru tırmanır: önce autograd motoru, sonra karakter-düzeyli dil modelleri (makemore), en sonunda transformer.

#	Ders	Ana Fikir
1	micrograd	Sıfırdan autograd ve geri yayılım (≈100 satır)
2	makemore 1 — Bigram	Karakter dil modeli; sayım + tek katmanlı ağ
3	makemore 2 — MLP	Bengio 2003; gömü (embedding) + gizli katman
4	makemore 3 — BatchNorm	Aktivasyon/gradyan istatistikleri; başlatma
5	makemore 4 — Backprop Ninja	Backprop'u elle, tensör düzeyinde türetmek
6	makemore 5 — WaveNet	Hiyerarşik (dilated) konvolüsyon; derinleşme
7	GPT'yi sıfırdan kur	Self-attention + transformer (nanoGPT)
8	Tokenizer (BPE)	GPT'nin byte-pair encoding tokenizer'ı
9	reproducing GPT-2	GPT-2'yi (124M) sıfırdan yeniden üretmek
10	LLM'ler — genel bakış	Pretraining → finetuning → RLHF büyük resim

Not: Phase 2 pilotu Ders 1 (micrograd) ile başlar. Dersler 2-10 aynı şablonla eklenecektir.

5 Notasyon

- **Skaler değer kutusu:** `Value(data)` — micrograd'ın temel yapı taşı
- **Gradyan:** $\partial L/\partial x$ — x'in çıktı L'ye etkisi (kısmi türev)
- **Yerel türev:** bir düğümün çıkışının girişine göre türevi (toplama $\rightarrow$ 1, çarpma $\rightarrow$ diğer operand, tanh $\rightarrow 1 - o^2$)
- **Zincir kuralı:** $\dfrac{dz}{dx} = \dfrac{dz}{dy} \cdot \dfrac{dy}{dx}$ — backprop'un tek aracı
- **Nöron:** $o = \tanh(\mathbf{w} \cdot \mathbf{x} + b)$ — dot product + bias + aktivasyon
- **MSE kaybı:** $L = \sum_i (y_i^{\text{pred}} - y_i^{\text{gt}})^2$
- **Güncelleme:** $p \leftarrow p - \eta \dfrac{\partial L}{\partial p}$ — gradyanın tersine adım (η = learning rate)

Tüm matematik MathJax 3 ile render ediliyor.

6 Builder Eksen — micrograd → Üretim

Her ders bu bağlantı katmanını taşır

micrograd kavramı	Üretim karşılığı
`Value` + `backward()`	`torch.Tensor` + `loss.backward()` (autograd)
İfade grafiği (`_prev`, `_op`)	PyTorch computation graph (`grad_fn`)
İşleme gömülü `_backward`	`torch.autograd.Function` (forward/backward çifti)
Topolojik sıralama	reverse-mode autodiff
`Neuron / Layer / MLP` + `parameters()`	`torch.nn.Module`, `model.parameters()`
$\mathbf{w} \cdot \mathbf{x} + b$ (dot product + bias)	`nn.Linear`; donanımda GEMM (GPU/TPU)
backprop = zincir kuralı	Calculus zincir kuralı (geriye köprü)
`zero_grad` + gradyan biriktirme (`+=`)	`optimizer.zero_grad()` ("unutursan bug")
"Gerisi sadece verimlilik"	skalerden tensöre, oradan GPT ölçeğine

Bir tek şey

Bir sinir ağını eğitmek tek bir döngüye iner: bir ifade grafiğinde **ileri geçişle** loss'u hesapla, **zincir kuralını geriye uygula** (backprop) ve her parametrenin gradyanını bul, sonra **gradyanın tersine küçük bir adım at**. micrograd bu çekirdeği 100 satırda gösterir; GPT yalnızca aynı şeyi devasa ölçekte tekrarlar.

7 Yazım Kuralları

- **Türkçe terminoloji + parantez içinde İngilizce orijinal** ilk geçtiğinde: “otomatik gradyan (autograd)”, “ifade grafiği (expression graph)”, “geri yayılım (backpropagation)”.
- **Karpathy’den alıntılar** İngilizce orijinal hâliyle, blockquote içinde, zaman damgasıyla verilir.
- **Builder Notu** callout’ları her ana bölüm sonunda; ML köprüsünü (geriye + ileriye) buraya yazıyoruz.
- **Öğretim kodu** (Karpathy’nin notebook’ta yazdığı `Value`, `_backward`, training loop) görünür `python` bloklarında; **figürler** ise tam motoru kullanan executable hücrelerdir (`echo:true` — kaynak figürle birlikte görünür).
- **Kontrol Soruları** collapse’lu — cevap kapalı başlar, okur kendi düşündükten sonra açar.
- **Egzersizler** cevapsız — en az bir Python/elle-türetme egzersizi.

> ⚠ Bu kitap Karpathy’nin yerine geçmez
>
> Tek başına bu set yetmez — Karpathy’nin satır satır canlı kodlama anlatımının ve sezgisinin yerine geçemez. Önce videoyu izle, sonra ilgili dersi oku, son olarak kodu **kendin yaz**. Set videoyu **destekler**, ikame etmez.

8 Sıfırdan Autograd ve Geri Yayılım (micrograd)

Bir sinir ağını eğiten makineyi 100 satırda kur: Value, ifade grafiği, backprop ve gradient descent

> **Bölüm bilgisi**
>
> - **Karpathy'nin videosu:** YouTube — The spelled-out intro to neural networks and backpropagation: building micrograd (≈146 dk)
> - **Seri:** Neural Networks: Zero to Hero — Ders 1
> - **Hoca:** Andrej Karpathy
> - **Kaynak repo:** github.com/karpathy/micrograd
> - **Okuma süresi:** ≈35 dk

8.1 Bu Derste Ne Var?

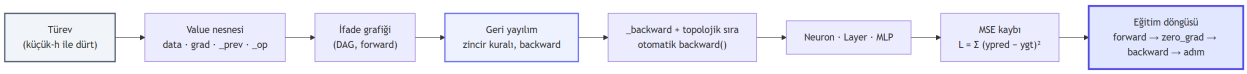
Bu, Karpathy'nin **Neural Networks: Zero to Hero** serisinin ilk dersi — ve serinin geri kalanının üstüne kurulacağı temel. Karpathy boş bir Jupyter notebook açıp, bir sinir ağını eğiten makinenin tam olarak nasıl çalıştığını **sıfırdan** kurar: parça parça, her satırı yorumlayarak. Ortaya çıkan şey **micrograd** — yaklaşık 100 satırlık küçük bir autograd (otomatik gradyan) motoru.

Dersin sürpriz iddiası şu: bir sinir ağını eğitmek için gereken matematiğin tamamı bu küçük motorda. Gerisi — tensörler, GPU, milyar parametre — yalnızca *verimliliktir*.

> *"Okay so here's the fun part. My claim is that micrograd is what you need to train your networks, and everything else is just efficiency."* — Karpathy, 6:44

Dersin üç büyük fikri:

1. **Türev ve gradyan** — bir sayının çıktıyı ne kadar etkilediğini ölçmek (sayısal sezgiyle başlayıp analitik backprop'a geçeriz).
2. **İfade grafiği + geri yayılım** — her hesabı düğümlerden oluşan bir grafiğe dökmek, sonra zincir kuralını grafikte geriye doğru uygulamak.
3. **Sinir ağı = ifade grafiğinin özel hâli** — Neuron / Layer / MLP'yi bu motorun üstüne kurup, bir loss'u gradient descent ile minimize ederek eğitmek.



Şekil 8.1: Ders 1'in kavram haritası: türev sezgisinden, ifade grafiği + geri yayılıma, oradan eğitilen bir sinir ağına.

Builder Notu — ML Köprüleri

Bu ders neredeyse tamamen önceki üç kursun (calculus, lineer cebir, olasılık) **matematik temeli** üstünde durur — köprüler zorlama değil, dersin kendi dili:

- **Geri yayılım = Calculus zincir kuralı.** Karpathy'nin kendi sözüyle "nothing more than the chain rule" (Calculus Ders 4).
- **Sayısal türev = Calculus türev tanımı.** Küçük bir h ile $(f(x+h) - f(x))/h$, türevin limit tanımının ta kendisi (Calculus Ders 2).
- **Nöron = 18.06 dot product.** $\mathbf{w} \cdot \mathbf{x} + b$ tam olarak matris-vektör çarpımı + öteleme (18.06 Ders 30).
- **İfade grafiği = yönlü çevrimsiz grafik (DAG).** Topolojik sıralama bir graf teorisi kavramıdır.

İleriye (production) köprüler de derse içkin: micrograd'ın `backward()`'ı **PyTorch autograd**'ın skaler prototipidir; `Neuron/Layer/MLP` API'si **`torch.nn.Module`** ile birebir hizalanır; skalerden tensöre geçiş **GPU GEMM** ve paralellik demektir.
Tek cümleyle: Bir sinir ağını eğitmek = bir ifade grafiğinde ileri geçişle loss'u hesaplamak, sonra zincir kuralını geriye uygulayıp (backprop) her parametrenin gradyanını bulmak ve gradyanın tersine küçük adım atmak — micrograd bu çekirdeği 100 satırda gösterir.

8.2 micrograd Nedir?

Karpathy işe tanımla başlıyor: micrograd bir **autograd motorudur**. "Autograd", *automatic gradient* (otomatik gradyan) kısaltması. Görevi: bir matematiksel ifadeyi otomatik olarak türevlemek — yani her girdinin çıktıyı ne kadar etkilediğini (gradyanını) hesaplamak.

> *"Micrograd is basically an autograd engine. Autograd is short for automatic gradient."* — Karpathy, 0:47

Micrograd iki dosyadan ibarettir: `engine.py` (autograd çekirdeği) ve `nn.py` (üstüne kurulan minik sinir ağı kütüphanesi). `engine.py`'nin temel yapı taşı **Value** nesnesidir — tek bir skaleri (sayıyı) saran, onu hangi işlemden ve hangi başka Value'lardan üretildiğini hatırlayan bir kutu. Value'lar üzerinde toplama, çarpma gibi işlemler yaptıkça, arka planda bir **ifade grafiği** (expression graph) örülür.

İki yön var. **İleri geçiş** (forward pass): girdilerden çıktıyı hesaplamak — düz, sıradan aritmetik. **Geri geçiş** (backward pass): çıktıdan başlayıp her girdinin çıktıya etkisini (gradyanı) grafikte *geriye* doğru taşımak. İşte tüm zorluk ve tüm güç, bu ikinci yöndedir.

Önemli bir basitleştirme: micrograd **skaler** değerlerle çalışır (tek tek sayılar), tensörlerle değil. Bu kasıtlı — Karpathy önce mekanizmayı en sade hâlinde gösteriyor; PyTorch'un tensörleri yalnızca aynı işlemleri paralel yapan bir verimlilik katmanıdır.

> Builder Notu — PyTorch autograd
>
> **İleriye:** `engine.py`'deki `Value` + `backward()`, PyTorch'taki `torch.Tensor` + `loss.backward()`'ın birebir kavramsal karşılığıdır. Fark: PyTorch tensör (çok boyutlu dizi) üzerinde çalışıp işlemleri GPU'da paralelleştirir; micrograd ise tek skalerle çalışır. "Gerisi sadece verimlilik" derken Karpathy tam da bunu kastediyor — algoritma aynı, ölçek farklı.

8.3 Türev Nedir? Tek Girişli Fonksiyonun Eğimi

Karpathy backprop'a girmeden önce, türevin ne olduğuna dair **sezgiyi** sağlamlaştırmak istiyor. Çünkü gradyan = türev, ve backprop = türevlerin zincirlenmesi.

> *"I'd like to make sure that you have a very good understanding intuitively of what a derivative is and exactly what information it gives you."* — Karpathy, 8:09

Uydurma bir skaler fonksiyon alalım:

$$f(x) = 3x^2 - 4x + 5$$

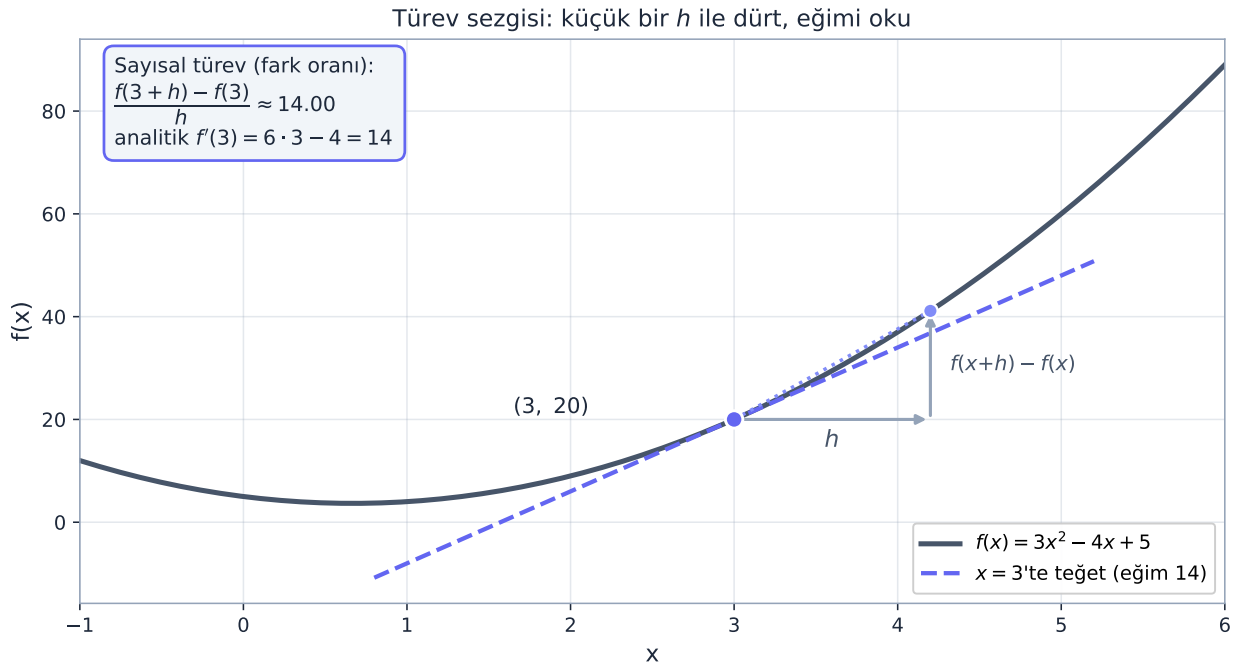
Türev, "x'i azıcık değiştirsem f ne kadar değişir?" sorusunun cevabıdır — yani fonksiyonun o noktadaki **duyarlılığı** (eğimi). Karpathy bunu sembolik formülle değil, **sayısal** olarak gösteriyor: çok küçük bir h al, fark oranını hesapla.

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Pratikte h'yi 0 yapamayız ama 0,0001 gibi küçük bir değer alıp oranı hesaplayabiliriz — bu da gerçek türeve çok yaklaşır. Örneğin $x = 3$'te eğim pozitif (f artıyor), $x = -3$'te negatif (f azalıyor), eğimin sıfır olduğu noktada ise fonksiyon tepe/dip yapar.

```python
def f(x):
    return 3*x**2 - 4*x + 5

h = 0.0001
x = 3.0
# sayisal turev: fark orani (difference quotient)
slope = (f(x + h) - f(x)) / h
print(slope)   # ~14 (analitik: 6x - 4 = 14)
```

Bu "küçük h ile dürt, ne değişti bak" fikri dersin tamamının çekirdeği: birazdan her girdiyi tek tek dürtüp gradyanı *sayısal* olarak okuyacağız, sonra aynı sonucu *analitik* olarak (zincir kuralıyla) üretip ikisinin eşleştiğini doğrulayacağız (buna **gradyan kontrolü** denir).



Şekil 8.2: Türev sezgisi: $f(x) = 3x^2 - 4x + 5$ eğrisi (slate) ve $x = 3$'te eğimi 14 olan teğet (indigo). Küçük-h sekantının fark oranı $\frac{f(3+h)-f(3)}{h} \approx 14$, analitik türev $f'(3) = 6 \cdot 3 - 4 = 14$ ile eşleşir — sayısal türev sezgisi.

> Builder Notu — Gradient Check
>
> **Geriye (Calculus):** Bu bölüm doğrudan Calculus Ders 2'nin türev tanımı. "Anlık değişim oranı" yerine Grant'ın (3Blue1Brown) tercih ettiği "en iyi sabit yaklaşım" sezgisi de aynı şeyi söyler: türev, fonksiyonu o noktada en iyi yaklaşan doğrunun eğimidir. $h \to 0$ limiti ise fark oranının (difference quotient) gerçek türeve yakınsamasıdır.
> **İleriye:** Sayısal gradyan, üretimde **gradient check** olarak yaşar: elle yazdığın (veya optimize ettiğin) analitik gradyanın doğru olduğunu, küçük bir h ile sayısal gradyana karşı test ederek doğrularsın. Karpathy bu derste tam olarak bunu yapacak.

8.4 Çok Girişli Fonksiyonda Kısmi Türevler

Tek girişli fonksiyondan, birden çok girişe geçelim — çünkü sinir ağları yüzlerce/milyonlarca girişe sahip. Karpathy basit bir örnek alır:

$$d = a \cdot b + c$$

Üç girişimiz var: $a = 2$, $b = -3$, $c = 10$. Şimdi her birini ayrı ayrı küçük bir h ile dürtüp d'nin nasıl değiştiğine bakarız — bu, her girişin **kısmi türevidir** (partial derivative). Sonuçlar analitik olarak da doğrulanabilir:

$$\frac{\partial d}{\partial a} = b = -3, \qquad \frac{\partial d}{\partial b} = a = 2, \qquad \frac{\partial d}{\partial c} = 1$$

Dikkat: a'nın kısmi türevi b'ye, b'ninki a'ya eşit — yani **çarpma**, her girişe *diğer* girişin değerini gradyan olarak geçirir. **Toplama** ise (c terimi) gradyanı 1 katsayısıyla, yani değiştirmeden geçirir. Bu iki kural — "çarpma diğerini geçirir, toplama aynen geçirir" — birazdan tüm geri yayılımın temeli olacak.

> Builder Notu — Gradyan Vektörü
>
> **Geriye (Calculus + 18.06):** Kısmi türev, çok değişkenli fonksiyonun tek bir değişkene göre türevidir (Calculus Ders 6). Tüm kısmi türevleri bir vektörde toplarsak **gradyan** (∇) elde edilir. Bir sinir ağında "her parametrenin loss'a kısmi türevi" tam olarak budur — milyonlarca kısmi türevden oluşan dev bir gradyan vektörü.

8.5 Value Nesnesi ve İfade Grafiği

Şimdi bu sayıları ve işlemleri koda dökelim. Karpathy ilk olarak **Value** nesnesini yazar:

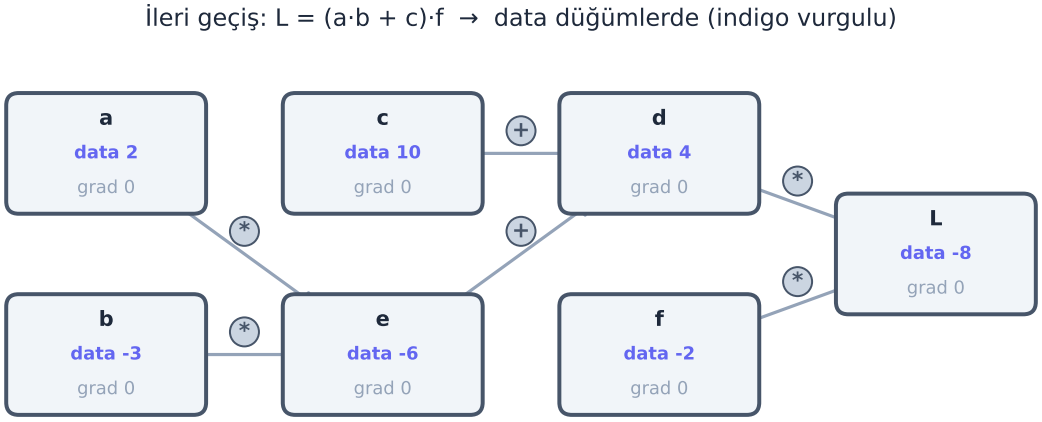
> *"First, a very simple Value object. Class Value takes a single scalar value that it wraps and keeps track of, and that's it."* — Karpathy, 19:30

Value, tek bir skaleri saran bir kutudur. Üstüne `__add__` ve `__mul__` gibi Python operatör metotlarını ekleriz; böylece `a + b` veya `a * b` yazdığımızda, sonuç yeni bir Value olur ve bu yeni Value, kendisini hangi **çocuk düğümlerden** (`_prev`) ve hangi **işlemden** (`_op`) üretildiğini hatırlar.

```
class Value:
    def __init__(self, data, _children=(), _op=''):
        self.data = data
        self.grad = 0.0                # baslangicta gradyan sifir
        self._prev = set(_children)    # bu degeri ureten cocuk dugumler
        self._op = _op                 # ureten islem etiketi: + veya *

    def __add__(self, other):
        return Value(self.data + other.data, (self, other), '+')

    def __mul__(self, other):
        return Value(self.data * other.data, (self, other), '*')

    def __repr__(self):
        return f"Value(data={self.data})"
```

`_prev` ve `_op` sayesinde, art arda yapılan işlemler bir **ifade grafiği** (expression graph) örer: yapraklar girdiler, iç düğümler ara sonuçlar, kök ise nihai çıktı. Karpathy bu grafiği `draw_dot` adlı bir Graphviz yardımcısıyla görselleştirir — her Value bir kutu (veri + gradyan), her işlem ayrı bir düğüm olarak çizilir. Grafiği görmek, geri yayılımın neden "grafikte geriye yürümek" olduğunu sezgisel kılar.

Aşağıda Karpathy'nin klasik örneğini — $L = (a \cdot b + c) \cdot f$ — kendi motorumuzla kurup ifade grafiğini çiziyoruz. İleri geçişte değerler otomatik akar: $e = -6$, $d = 4$, $L = -8$.



Şekil 8.3: İfade grafiği (forward DAG): $L = (a \cdot b + c) \cdot f$. Yapraklar (a, b, c, f) solda; çarpma (*) ve toplama (+) düğümleri kenarlarda; kök çıktı L sağda. İleri geçişte değerler ($e = -6$, $d = 4$, $L = -8$) düğüm kutularında **data** olarak indigo vurgulu; oklar girişten çıktıya akar.

> Builder Notu — Computation Graph
>
> **Geriye (18.06 / graf teorisi):** İfade grafiği bir **yönlü çevrimsiz grafiktir** (DAG): kenarlar hep girdiden çıktıya akar, döngü yoktur. Bu yapı, birazdan göreceğimiz topolojik sıralamanın ön koşuludur.
> **İleriye:** Bu `Value` + `_prev` + `_op` üçlüsü, PyTorch'un **computation graph**'ının skaler prototipidir. PyTorch da her tensör işleminde "bu tensör hangi işlemden, hangi girdilerden üretildi" bilgisini (`grad_fn`) saklar — `loss.backward()` tam olarak bu grafiği geriye yürür.

8.6 Elle Geri Yayılım #1: L = (a·b + c)·f

Şimdi grafiğin kökünden başlayıp, her düğümün gradyanını **elle** hesaplayalım. Karpathy'nin klasik örneği:

$$e = a \cdot b, \qquad d = e + c, \qquad L = d \cdot f$$

Değerler: $a = 2$, $b = -3$, $c = 10$, $f = -2$. İleri geçiş: $e = -6$, $d = 4$, $L = -8$. Amacımız her girişin L'ye etkisini, yani $\partial L/\partial a$, $\partial L/\partial b$, $\partial L/\partial c$, $\partial L/\partial f$'i bulmak.

Köke kendi gradyanını veririz: $\partial L/\partial L = 1$. Sonra geriye yürürüz. $L = d \cdot f$ bir **çarpma** olduğundan, her operanda *diğerini* geçirir:

$$\frac{\partial L}{\partial d} = f = -2, \qquad \frac{\partial L}{\partial f} = d = 4$$

> *"Now we're getting to the crux of backpropagation, so this will be the most important node to understand. Because if you understand the gradient for this node, you understand all of backpropagation and all of training of neural nets basically."* — Karpathy, 38:03

Şimdi en kritik adım: $d = e + c$ bir **toplama**. Toplama gradyanı *değiştirmeden* geçirir (yerel türev 1). Yani zincir kuralıyla:

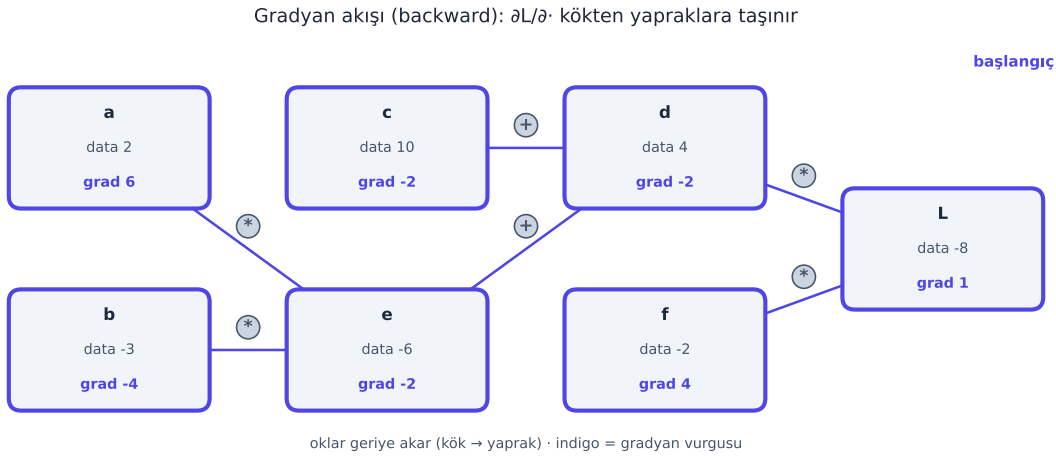
$$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial c} = (-2) \cdot 1 = -2$$

$$\frac{\partial L}{\partial e} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial e} = (-2) \cdot 1 = -2$$

Bir adım daha geriye, $e = a \cdot b$ çarpımına: gradyanı *diğer* operandla çarparak geçiririz:

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e} \cdot b = (-2)(-3) = 6, \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e} \cdot a = (-2)(2) = -4$$

İşte geri yayılımın özü: kökten başla, her düğümde **yerel türevi** hesapla, gelen gradyanla çarp (zincir kuralı), bir önceki düğüme aktar. Her gradyanı küçük bir h ile **sayısal olarak** doğrulayabiliriz (gradient check) — analitik ile sayısal eşleşmeli. Aşağıdaki figür, motorumuzun `L.backward()` ile otomatik bulduğu gradyanların elle hesabımızla birebir aynı çıktığını gösteriyor.



Şekil 8.4: Geri yayılım, ileri geçişle **aynı** ifade grafiğini ters yönde gezer: köke $\partial L/\partial L = 1$ verilir, sonra ters topolojik sırada her düğümün yerel türevi gelen gradyanla çarpılarak (zincir kuralı) yapraklara taşınır. İndigo ve kalın yazılan grad değerleri Bölüm 5'te elle hesapladığımız sonuçlardır: $\partial L/\partial a = 6$, $\partial L/\partial b = -4$, $\partial L/\partial c = -2$, $\partial L/\partial f = 4$. Toplama düğümü gradyanı aynen geçirir ($d \to c, e$), çarpma diğer operandı geçirir ($e \to a, b$).

Builder Notu — Zincir Kuralı

Geriye (Calculus): Buradaki tek araç **zincir kuralıdır** (Calculus Ders 4): bileşik bir fonksiyonun türevi, dış türev × iç türev. "Toplama gradyanı aynen geçirir, çarpma diğerini geçirir" kuralları, +/× işlemlerinin yerel türevlerinin $(1, 1)$ ve (b, a) olmasından çıkar.

İleriye: Bu elle yaptığımız işlem, PyTorch'ta `loss.backward()` çağrısının her ara tensör için otomatik yaptığı şeydir. Manuel yapmak — "motoru kapağı açıkken görmek" — tam olarak Karpathy'nin amacı: kütüphane bir kara kutu olmaktan çıkar.

8.7 Zincir Kuralı ve Özyinelemeli Sezgi

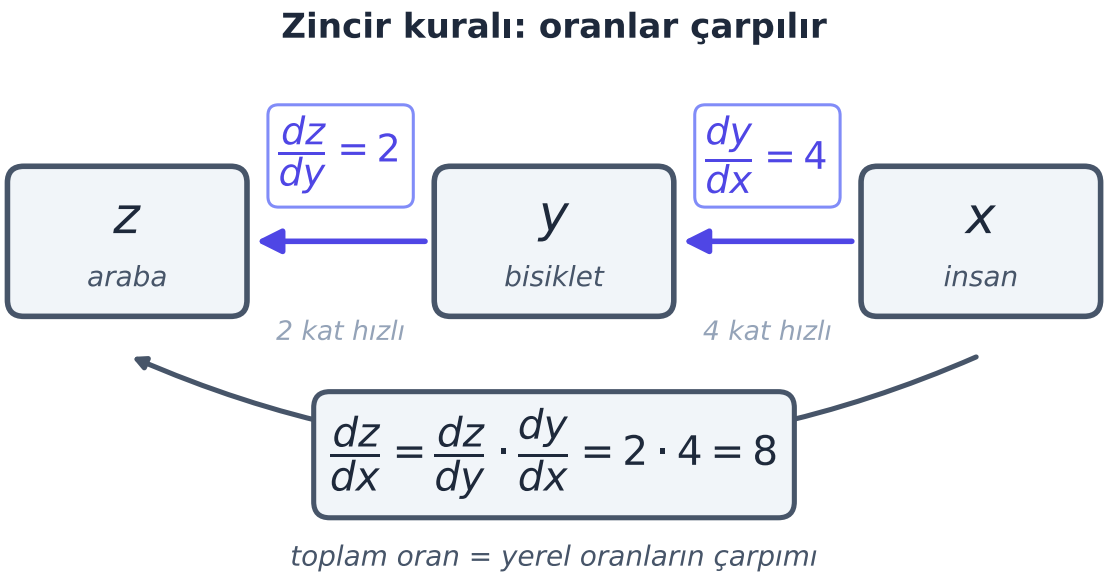
Az önce sezgisel yaptığımız şeyin resmî adı **zincir kuralıdır**. Karpathy Wikipedia'dan açıp, en sevdiği ifadeyi paylaşır:

> *"This is the way I learned chain rule and it was very confusing. I like this expression much better."*
> — Karpathy, 42:03

Sezgi, klasik araba örneğiyle gelir: bir araba bisikletten iki kat, bisiklet de yürüyen bir insandan dört kat hızlıysa, araba insandan $2 \times 4 = 8$ kat hızlıdır. "Oranlar çarpılır." Matematikte z, y'ye; y de x'e bağlıysa:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

İşte geri yayılımın özyinelemeli (recursive) doğası burada: bir düğümün loss'a etkisi = (bir sonraki düğümün loss'a etkisi) × (kendi yerel türevi). Bu çarpımı grafikte kökten yapraklara doğru zincirleyerek taşırız. **Toplama** düğümü gradyanı 1 ile çarpıp aynen "yönlendirir" (router gibi); **çarpma** düğümü diğer operandla çarpar.



Şekil 8.5: Zincir kuralı: $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$ (oran çarpımı). Araba bisikletten 2 kat, bisiklet insandan 4 kat hızlıysa araba insandan $2 \cdot 4 = 8$ kat hızlıdır; toplam oran, kökten yaprağa yerel oranların çarpımıdır.

Builder Notu — Reverse-Mode Autodiff

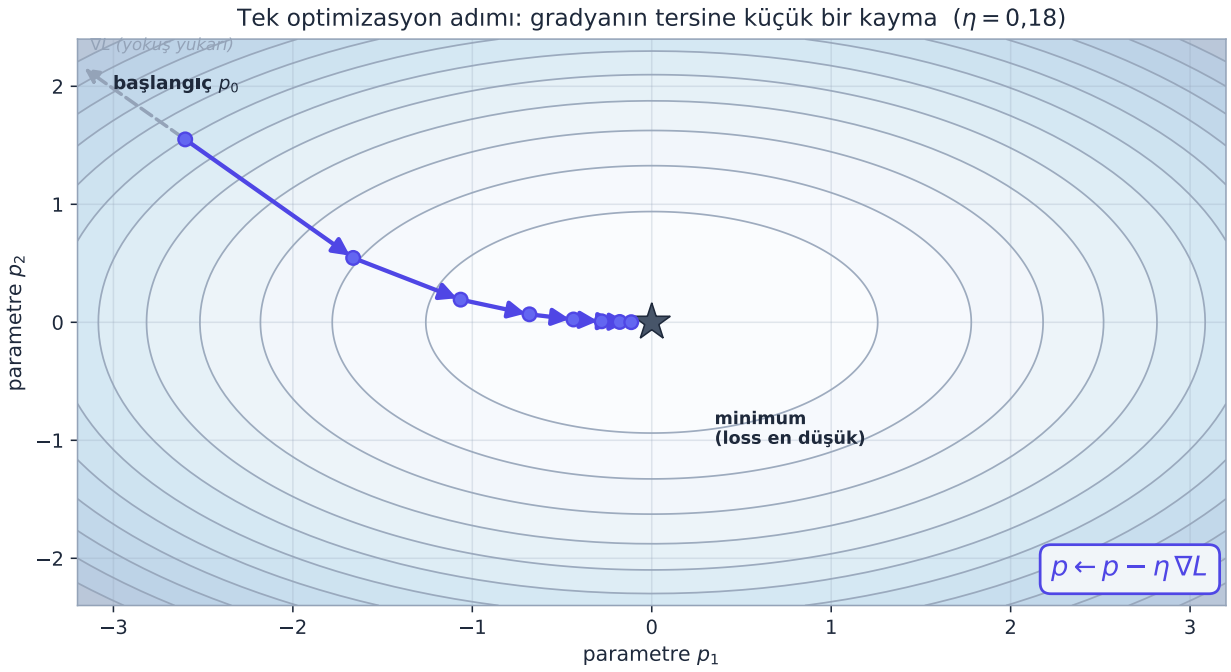
Geriye (Calculus Ders 4): Karpathy'nin dersin ilerleyen bölümünde söyleyeceği gibi, backprop "nothing more than the chain rule" — zincir kuralının bir grafik boyunca tekrar tekrar uygulanması. Calculus'ta gördüğün "dış türev çarpı iç türev" buraya birebir oturur.
İleriye: Bu özyinelemeli yapı, **reverse-mode autodiff** (ters-mod otomatik türev) adını alır. Gradyanı çıktıdan girişe doğru biriktirmek, ara türevleri yeniden hesaplamadan zincirler — bu yüzden derin ağlarda verimlidir (ileri-mod yerine ters-mod seçilmesinin sebebi).

8.8 Tek Optimizasyon Adımı (Önizleme)

Gradyanlar elimizde; peki ne işe yarar? Karpathy kısa bir önizleme yapar: girişleri gradyan yönünde azıcık iterek (nudge) L'yi *yükseltmeye* çalışır.

> *"What we're going to do is we're going to nudge our inputs to try to make L go up."* — Karpathy, 51:10

Mantık basit: $\partial L/\partial a$, a'yı artırırsak L'nin ne yönde değişeceğini söyler. L'yi yükseltmek istiyorsak her girişi *kendi gradyanı yönünde* küçük bir adım kaydırırız; sonra ileri geçişi tekrar koşar ve L'nin gerçekten arttığını görürüz. Sinir ağı eğitiminde tam tersini yapacağız — loss'u *düşürmek* için gradyanın **negatif** yönünde adım atacağız (gradient descent). Ama mekanizma birebir aynı.



Şekil 8.6: Kavramsal bir loss yüzeyinde tek optimizasyon adımı: her adımda parametre, gradyanın *tersi* yönünde küçük bir kayma yapar — $p \leftarrow p - \eta \nabla L$. İndigo oklar adımları, soluk slate kesikli ok başlangıçtaki gradyan yönünü (yokuş yukarı) gösterir; yıldız çukurun dibidir (loss en düşük). Bu kavramsal bir yüzeydir — gerçek eğitim değil (bkz. Şekil 8.8).

Builder Notu — Gradient Descent Adımı

İleriye: Bu "gradyan yönünde küçük adım" tek hücresi, optimizer'ın ta kendisidir. Eğitimde formül $p \leftarrow p - \eta \cdot$ gradyan olur; η (learning rate) adım boyudur. Karpathy burada işaretini artı yapıp L'yi yükseltiyor — Bölüm 16'da işareti eksiye çevirip loss'u düşüreceğiz.

8.9 Elle Geri Yayılım #2: tanh'lı Tek Nöron

Şimdi daha gerçekçi bir örnek: tek bir **nöron**. Karpathy iki girişli bir nöron kurar.

> *"So now I would like to do one more example of manual backpropagation using a bit more complex and useful example. We are going to backpropagate through a neuron."* — Karpathy, 52:52

Nöron, girişleri ağırlıklarla çarpıp toplar, bias ekler (ham toplam n), sonra bir **aktivasyon** fonksiyonundan geçirir. Karpathy aktivasyon olarak **tanh** seçer:

$$n = x_1 w_1 + x_2 w_2 + b, \qquad o = \tanh(n)$$

tanh'ın geri yayılımda işimizi kolaylaştıran güzel bir türevi vardır:

$$\frac{\partial o}{\partial n} = 1 - \tanh^2(n) = 1 - o^2$$

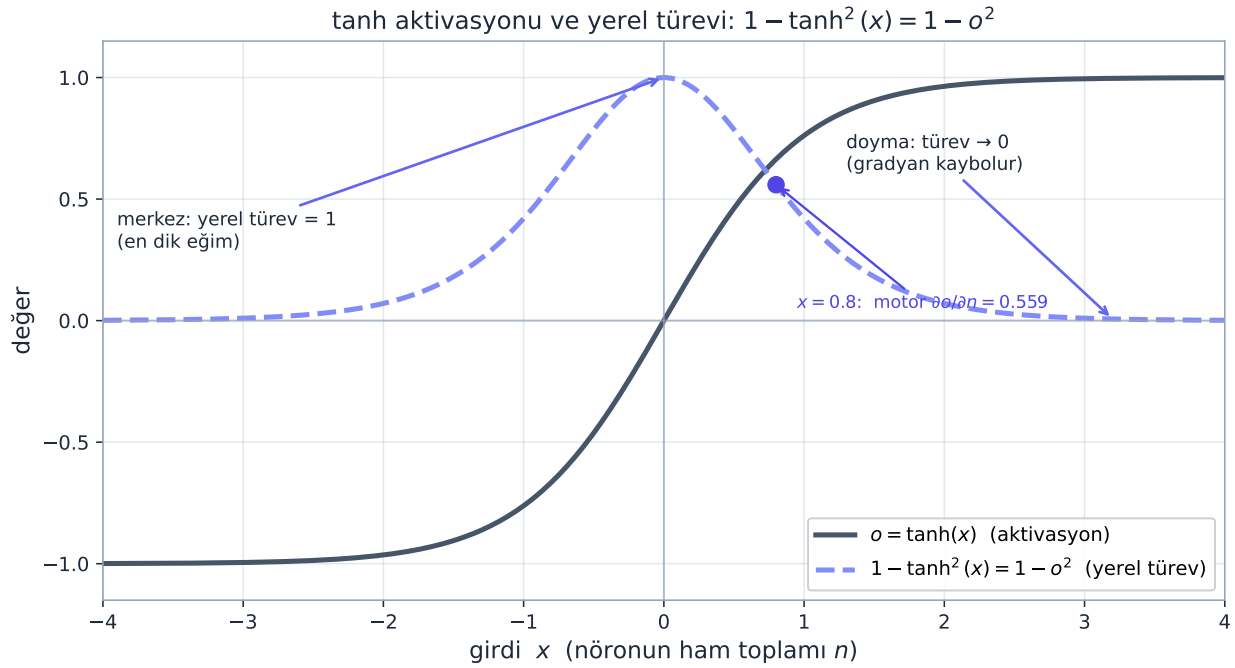
Yani $o = \tanh(n)$ düğümünün yerel türevini, çıktının kendisinden (o) hesaplayabiliriz — n'i yeniden tanh'lamaya gerek yok. Geri geçişte: `o.grad = 1`'den başlar, `n.grad = (1 - o**2) * o.grad` olur, oradan çarpma ve toplama düğümlerinden geçerek x_1, w_1, x_2, w_2, b gradyanlarına ulaşırız — Bölüm 5'teki aynı iki kuralla (toplama aynen geçirir, çarpma diğerini geçirir).

Builder Notu — tanh ve ReLU Türevi

Geriye (18.06 + Calculus): Nöronun çekirdeği $x_1 w_1 + x_2 w_2 + b$, tam olarak **dot product + bias** (18.06 Ders 30) — bir katmanda bu $\mathbf{Wx} + \mathbf{b}$ matris çarpımına genişler. tanh'ın $1 - \tanh^2$ türevi ise Calculus'tan; sigmoid'in türevinin $g(1-g)$ olmasına akrabadır.
İleriye: GitHub'daki güncel micrograd, tanh yerine **ReLU** kullanır ($\max(0, x)$; türevi 0 veya 1). Karpathy derste bilinçli tanh seçer — türevi daha "zengin" olduğu için backprop'u daha öğretici kılar. Modern derin ağlarda varsayılan çoğunlukla ReLU/GELU'dur.

8.10 Her İşlem İçin _backward {#sec-backward-kapanis}

Elle backprop yapmak öğretici ama dayanılmaz. Karpathy bunu otomatikleştirmeye geçer:



Şekil 8.7: tanh aktivasyonu (slate, düz) ve yerel türevi $1-\tanh^2(x) = 1-o^2$ (açık indigo, kesik). Yerel türev, çıktının kendisinden (o) hesaplanabilir — n'i yeniden tanh'lamaya gerek yok. Merkezde eğim en dik (türev $= 1$); $|x|$ büyüdükçe tanh doyar ve türev sıfıra iner (gradyan kaybolur). $x = 0.8$ noktasında motorun (`Value.tanh`) ürettiği $\partial o/\partial n$ analitik $1-o^2$ ile birebir eşleşir.

> *"Okay so doing the backpropagation manually is obviously ridiculous, so we are now going to put an end to this suffering and we're going to see how we can implement the backward pass a bit more automatically."* — Karpathy, 1:08:59

Fikir şu: her işlem, kendi **yerel gradyan kuralını** bilir. O hâlde her Value'ya, çıkışının gradyanını (`out.grad`) alıp girişlerine dağıtan bir **`_backward` fonksiyonu** (closure) iliştirelim. Toplama aynen geçirir, çarpma diğerini geçirir, tanh ise $(1-o^2)$ ile çarpar:

```
import math

def __add__(self, other):
    out = Value(self.data + other.data, (self, other), '+')
    def _backward():
        self.grad = 1.0 * out.grad    # toplama: gradyani aynen gecir
        other.grad = 1.0 * out.grad
    out._backward = _backward
    return out

def __mul__(self, other):
    out = Value(self.data * other.data, (self, other), '*')
    def _backward():
        self.grad = other.data * out.grad   # carpma: digerini gecir
        other.grad = self.data * out.grad
```

```
    out._backward = _backward
    return out

def tanh(self):
    t = math.tanh(self.data)
    out = Value(t, (self,), 'tanh')
    def _backward():
        self.grad = (1 - t**2) * out.grad   # tanh: (1 - o^2) ile carp
    out._backward = _backward
    return out
```

Artık elle çarpmak yerine, doğru sırada her düğümün `_backward()`'ını çağırmamız yeter. Ama "doğru sıra" ne? Bir düğümün gradyanını dağıtmadan önce, *kendi* gradyanının tamamlanmış olması gerekir — yani çıkışındaki tüm düğümler işlenmiş olmalı. İşte bu sırayı bir sonraki bölümde topolojik sıralamayla çözeceğiz.

> Builder Notu — torch.autograd.Function
>
> **İleriye:** Her işleme kendi yerel gradyanını gömmek, PyTorch'taki **`torch.autograd.Function`**'ın `forward`/`backward` çiftinin tam karşılığıdır. Yeni bir özel işlem tanımladığında PyTorch'ta da yaptığın şey budur: ileri hesabı ve onun yerel gradyanını yazmak; gerisini motor halleder.

8.11 Topolojik Sıralama ve Otomatik backward()

`_backward`'ları doğru sırada çağırmak için grafiği **topolojik olarak** sıralarız.

> *"This can be achieved using something called topological sort. Topological sort is basically a laying out of a graph such that all the edges go only from left to right."* — Karpathy, 1:18:14

Topolojik sıralama, bir düğümü ancak *tüm çocukları* listeye eklendikten sonra ekleyen özyinelemeli bir gezintidir. Sonuç: kenarların hep soldan sağa aktığı bir dizilim. Geri yayılım için bu dizilimi **tersten** gezeriz: önce köke grad = 1 veririz, sonra ters topolojik sırada her düğümün `_backward()`'ını çağırırız. Böylece her düğüm işlenirken, gradyanı çoktan tamamlanmış olur.

```
def backward(self):
    topo = []
    visited = set()
    def build_topo(v):
        if v not in visited:
            visited.add(v)
            for child in v._prev:
                build_topo(child)   # once cocuklar
            topo.append(v)          # sonra dugumun kendisi
    build_topo(self)

    self.grad = 1.0                 # kok dugumun gradyani 1
```

```
    for node in reversed(topo):     # ters topolojik sira
        node._backward()
```

Tek bir `o.backward()` çağrısı, artık tüm grafiğin gradyanlarını otomatik hesaplar. Bu, micrograd'ın kalbidir — ve PyTorch'taki `loss.backward()` ile aynı fikir. (Bölüm 5'teki Şekil 8.4 figürü, tam olarak bu `backward()` çağrısının ürettiği gradyanları çiziyordu.)

> Builder Notu — Topolojik Sıralama
>
> **Geriye (graf teorisi):** Topolojik sıralama yalnızca **DAG**'larda (yönlü çevrimsiz grafik) tanımlıdır — ifade grafiğimiz tam da öyle. Döngü olsaydı sıralama mümkün olmazdı (ve gradyan da tanımsız olurdu).
> **İleriye:** PyTorch da `backward()` çağrısında dahili computation graph'ı ters topolojik sırada gezer. Sıralama mantığı birebir aynı; tek fark, PyTorch'un bunu C++ seviyesinde ve tensörler üzerinde yapması.

8.12 Gradyan Biriktirme (+=) Hatası

Karpathy şimdi ince ama kritik bir bug gösterir — kasten. Bir değişken grafikte **birden çok kez** kullanılırsa ne olur? Örneğin $b = a + a$, ya da bir nöronun aynı girdiyi iki yola beslemesi.

`_backward` içinde `self.grad = ...` (atama) yazdık. Ama a iki kez kullanılırsa, ikinci `_backward` çağrısı birincinin yazdığı gradyanı **ezer** — oysa çok değişkenli zincir kuralına göre, bir değişken birden çok yola katkı veriyorsa gradyanları **toplanmalıdır**.

> *"But then we come back to d and call backward, and it overwrites those gradients at a and b. So that's obviously a problem."* — Karpathy, 1:25:05

Çözüm tek karakter: `=` yerine `+=`. Her `_backward`, mevcut gradyanın *üstüne ekler*, ezmez:

```
def _backward():
    self.grad += 1.0 * out.grad    # ATAMA degil, BIRIKTIRME (+=)
    other.grad += 1.0 * out.grad
```

Bu yüzden `backward()`'tan önce tüm gradyanların 0 olması gerekir (birikim sıfırdan başlasın diye) — bu da Bölüm 16'daki `zero_grad` dersinin tohumu.

> Builder Notu — zero_grad
>
> **Geriye (Calculus):** Bu, **çok değişkenli zincir kuralının** doğrudan sonucudur: bir değişken çıktıya birden çok yoldan etki ediyorsa, toplam türev bu yolların türevlerinin *toplamıdır*. `+=` tam olarak bu toplamı uygular.
> **İleriye:** Aynı mantık PyTorch'ta da geçerlidir — gradyanlar varsayılan olarak **birikir**. Bu yüzden her eğitim adımından önce `optimizer.zero_grad()` çağırırsın. Karpathy'nin burada gösterdiği bug, üretimde "zero_grad'ı unuttum" hatasının ta kendisidir (Bölüm 16).

8.13 tanh'ı Atomlara Ayırma (exp, pow, bölme)

Şimdiye dek tanh'ı **tek parça** (kompozit) bir işlem olarak yazdık — yerel türevi $1 - o^2$ olduğu sürece bu meşru. Karpathy burada güzel bir nokta gösterir: bir işlemi ne kadar "atomik" tutacağın sana kalmış; yeter ki her parçanın yerel türevini bilesin. tanh'ı atomlarından kurmak için yeni işlemler ekler: `exp`, `__pow__` (kuvvet), bölme ve çıkarma.

$$\tanh(n) = \frac{e^{2n} - 1}{e^{2n} + 1}$$

Bunu yazabilmek için e^x'in türevinin yine kendisi olduğunu ($d(e^x)/dx = e^x$) ve kuvvet kuralını kullanırız:

$$\frac{d}{dx} x^k = k\, x^{k-1}$$

Bölme de ayrı bir işlem değil — `a / b`, $a \cdot b^{-1}$ olarak yazılır (kuvvet kuralının -1 özel hâli):

> *"A value raised to the power of negative one — we have now defined that."* — Karpathy, 1:33:00

```python
def exp(self):
    out = Value(math.exp(self.data), (self,), 'exp')
    def _backward():
        self.grad += out.data * out.grad      # d(e^x)/dx = e^x
    out._backward = _backward
    return out

def __pow__(self, k):                          # k: int veya float
    out = Value(self.data ** k, (self,), f'**{k}')
    def _backward():
        self.grad += k * self.data**(k - 1) * out.grad   # kuvvet kurali
    out._backward = _backward
    return out

def __truediv__(self, other):                  # a / b = a * b**-1
    return self * other**-1
```

tanh'ı bu atomlardan kurup, hem ileri geçişin hem de gradyanların **birebir aynı** çıktığını görürüz. Ayrıca `2 * a` gibi durumlarda Python'un sağdan çağırdığı `__rmul__` / `__radd__` yedek operatörleri ve skaler sarmalama da eklenir (örn. `1 + a`).

> Builder Notu — Fused Kernel
>
> **Geriye (Calculus):** Burada üç Calculus aracı bir arada: e^x'in türevi (Ders 5), kuvvet kuralı (Ders 2), ve bölmenin negatif kuvvet olarak yazılması. Hepsi yerel türev olarak `_backward`'a girer.
> **İleriye:** "Soyutlama seviyesini sen seçersin" fikri PyTorch'ta da geçerli: tanh'ı tek `torch.tanh` çağrısı olarak da, atomlarından da yazabilirsin — motor her iki durumda da doğru gradyanı üretir. Performans için kütüphaneler bunları tek bir **fused kernel**'de birleştirir.

8.14 Aynı Şeyi PyTorch ile

Karpathy şimdi tam olarak aynı nöronu **PyTorch** ile kurar ve micrograd'ın doğruluğunu kanıtlar. Tek fark: PyTorch tensörlerle çalışır (micrograd skalerle). Yapraklarda `requires_grad = True` açılır (PyTorch verimlilik için gradyanı varsayılan kapalı tutar), `o.backward()` çağrılır ve gradyanlar **micrograd ile birebir eşleşir**.

```
import torch

x1 = torch.tensor([2.0]).double();  x1.requires_grad = True
x2 = torch.tensor([0.0]).double();  x2.requires_grad = True
w1 = torch.tensor([-3.0]).double(); w1.requires_grad = True
w2 = torch.tensor([1.0]).double();  w2.requires_grad = True
b  = torch.tensor([6.8813735870195432]).double(); b.requires_grad = True

n = x1*w1 + x2*w2 + b
o = torch.tanh(n)
o.backward()

print(x1.grad.item(), w1.grad.item())   # micrograd ile ayni cikar
```

Mesaj net: micrograd ile PyTorch aynı algoritmayı çalıştırır. PyTorch yalnızca (a) tensörlerle paralel hesap yapar, (b) bunu GPU'da hızlandırır. "Gerisi sadece verimlilik" iddiası burada kanıtlanır.

> Builder Notu — torch.Tensor API
>
> **İleriye:** `requires_grad`, `.backward()`, `.grad` — bunlar her PyTorch eğitim döngüsünde her gün kullandığın API'nin ta kendisi. Tensörlerin `.double()` ile float64'e çevrilmesi yalnızca micrograd'la birebir sayısal karşılaştırma içindir; gerçek eğitimde float32 (hatta bf16) kullanılır (bkz. Ders 10, mixed precision).

8.15 Sinir Ağı Kütüphanesi: Neuron, Layer, MLP

Elimizde karmaşık ifadeler kurup türevleyebilen bir motor var. Karpathy artık sinir ağı katmanını (`nn.py`) kurar:

> *"Okay so now that we have some machinery to build out pretty complicated mathematical expressions, we can also start building out neural nets. And as I mentioned, neural nets are just a specific class of mathematical expressions."* — Karpathy, 1:43:55

Üç sınıf, üst üste: **Neuron** (rastgele ağırlıklar + bias, dot product + tanh), **Layer** (nöron listesi), **MLP** (katman dizisi). Her birinde `parameters()` metodu, tüm öğrenilebilir Value'ları tek listede toplar.

```
import random

class Neuron:
    def __init__(self, nin):
        self.w = [Value(random.uniform(-1, 1)) for _ in range(nin)]
        self.b = Value(random.uniform(-1, 1))
    def __call__(self, x):
        act = sum((wi*xi for wi, xi in zip(self.w, x)), self.b)  # w.x + b
        return act.tanh()
    def parameters(self):
        return self.w + [self.b]

class Layer:
    def __init__(self, nin, nout):
        self.neurons = [Neuron(nin) for _ in range(nout)]
    def __call__(self, x):
        outs = [n(x) for n in self.neurons]
        return outs[0] if len(outs) == 1 else outs
    def parameters(self):
        return [p for n in self.neurons for p in n.parameters()]

class MLP:
    def __init__(self, nin, nouts):
        sz = [nin] + nouts
        self.layers = [Layer(sz[i], sz[i+1]) for i in range(len(nouts))]
    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        return x
    def parameters(self):
        return [p for layer in self.layers for p in layer.parameters()]

# 3 girisli, iki gizli katman (4,4) ve 1 cikisli bir MLP
model = MLP(3, [4, 4, 1])
```

Builder Notu — torch.nn.Module

Geriye (18.06): Bir Layer = nöron listesi; her nöron $\mathbf{w} \cdot \mathbf{x} + b$ (dot product + öteleme). Tüm katmanın ileri geçişi aslında bir **matris-vektör çarpımıdır** (18.06 Ders 30) — micrograd bunu skaler tek tek yapar, PyTorch tek `nn.Linear` çağrısında.
İleriye: `MLP.parameters()` ve `__call__` arabirimi, **`torch.nn.Module`** API'siyle birebir hizalanır: PyTorch'ta da `model.parameters()` optimizer'a verilir, `model(x)` ileri geçişi koşar. Karpathy bunu kasten aynı yapar — micrograd'dan PyTorch'a geçiş sıfır sürtünmeli olsun diye.

8.16 Veri Seti, MSE Kaybı ve Elle Eğitim

Ağı kurduk; şimdi **eğitelim**. Karpathy 4 örnekten oluşan oyuncak bir veri seti tanımlar: dört giriş vektörü (xs) ve dört hedef (ys, +1 veya −1). Ağın tahminleri (ypred) ile hedefler arasındaki farkı tek bir sayıda toplamak için **ortalama karesel hata** (MSE) kullanır:

$$L = \sum_i \left(y_i^{\text{pred}} - y_i^{\text{gt}}\right)^2$$

Her tahmin hedefe ne kadar yakınsa kare-fark o kadar küçük; toplam loss sıfıra yaklaştıkça ağ "doğru öğrenmiş" demektir. Önemli olan şu: loss da bir Value'dur — yani devasa bir ifade grafiğinin köküdür. `loss.backward()` çağırınca, **tüm ağırlıkların** gradyanı otomatik hesaplanır (Bölüm 10'daki motor). Sonra her parametreyi gradyanın *tersine* küçük bir adım kaydırırız (loss'u düşürmek için):

$$p \leftarrow p - \eta \frac{\partial L}{\partial p}$$

Karpathy bunu önce elle birkaç kez yapar: forward → backward → update. Loss'un adım adım düştüğünü, tahminlerin hedeflere yaklaştığını izleriz.

> Builder Notu — MSE ve Learning Rate
>
> **Geriye (istatistik):** MSE, tahminlerin hedeflerden ortalama karesel uzaklığıdır; Gauss gürültü varsayımı altında maximum likelihood'a denktir. (Sınıflandırmada yerini cross-entropy alır — bkz. Ders 2, makemore.)
> **İleriye:** `model.parameters()` + loss + `backward()` + güncelleme, her PyTorch eğitim döngüsünün dört temel adımıdır. η (learning rate) burada elle 0,05 seçilir; gerçek eğitimde en kritik hyperparameter'lardandır.

8.17 Eğitim Döngüsü, zero_grad Hatası ve Özet

Elle tekrar etmek yerine düzgün bir **eğitim döngüsü** yazalım:

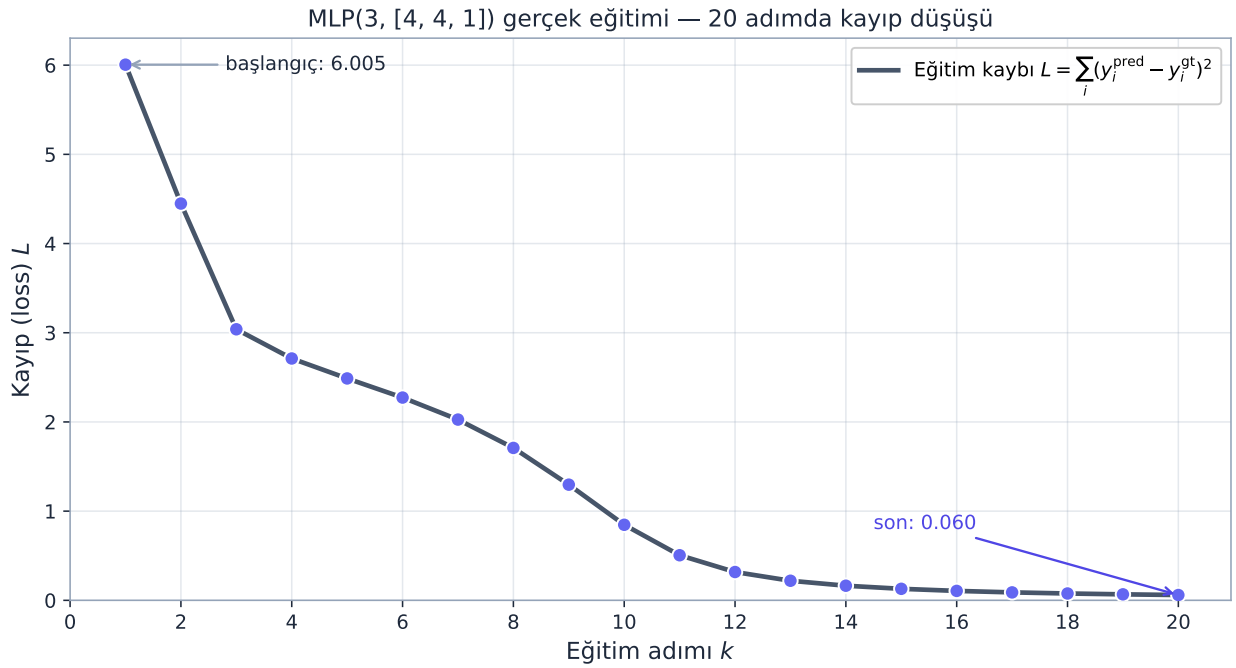
> *"Okay let's make this a tiny bit more respectable and implement an actual training loop."* — Karpathy, 2:08:35

Ve burada Karpathy en meşhur hatasını gösterir — yine kasten. Gradyanların `+=` ile *biriktiğini* (Bölüm 11) hatırla. Eğer her adımda `backward()`'tan önce gradyanları **sıfırlamazsan**, önceki adımların gradyanları üst üste birikir; adımlar bozulur, eğitim kararsızlaşır. Çözüm: her iterasyonda önce `p.grad = 0.0` (zero_grad), sonra backward.

```
for k in range(20):
    # 1) ileri gecis: tahmin + loss
    ypred = [model(x) for x in xs]
    loss = sum((yout - ygt)**2 for ygt, yout in zip(ys, ypred))
```

```
    # 2) ZERO_GRAD: gradyanlari sifirla (yoksa birikir!)
    for p in model.parameters():
        p.grad = 0.0

    # 3) geri gecis: tum gradyanlari hesapla
    loss.backward()

    # 4) guncelleme: gradyanin tersine kucuk adim
    for p in model.parameters():
        p.data += -0.05 * p.grad

    print(k, loss.data)   # loss adim adim dusmeli
```

Bu döngü, GPT ölçeğindeki ağlara aynen ölçeklenir: yapı taşı değişmez, yalnızca skaler yerine tensör, CPU yerine GPU, dört örnek yerine milyarlarca token gelir. Aşağıdaki figür, tam olarak bu döngüyü kendi motorumuzla 20 adım koşturup MSE kaybının monoton düştüğünü gösteriyor.



Şekil 8.8: MLP(3, [4, 4, 1]) gerçek eğitimi: `random.seed(1337)` ile deterministik 20 adımda kayıp eğrisi. Her adım forward → zero_grad → backward → güncelleme; MSE kaybı $L = \sum_i (y_i^{\text{pred}} - y_i^{\text{gt}})^2$ monoton düşer (başlangıç 6,005 → son 0,060). Slate çizgi, indigo işaretçiler.

Builder Notu — Eğitim Döngüsü

İleriye: "zero_grad'ı unutma" hatası üretimde gerçektir — PyTorch'ta her adımda `optimizer.zero_grad()` çağırırsın, tam olarak bu yüzden. Karpathy'nin döngüsü (forward → zero_grad → backward → update) PyTorch eğitim döngüsünün birebir iskeletidir; tek fark `optimizer.step()`'in güncellemeyi senin yerine yapması.

8.18 Bu Dersin Özeti

1. **micrograd**, bir autograd (otomatik gradyan) motorudur — yaklaşık 100 satırda backpropagation'ı gösterir. "Gerisi sadece verimlilik."
2. **Türev**, bir girişi azıcık dürtünce çıktının ne kadar değiştiğidir; küçük bir h ile *sayısal* olarak okunabilir (gradient check).
3. **Value** nesnesi bir skaleri sarar, hangi işlemden (`_op`) ve hangi çocuklardan (`_prev`) üretildiğini tutar; işlemler bir **ifade grafiği** (DAG) örer.
4. **Geri yayılım**, kökten yapraklara zincir kuralını uygulamaktır: toplama gradyanı aynen geçirir, çarpma diğer operandı geçirir, tanh ise $(1 - o^2)$ ile çarpar.
5. Her işleme bir **`_backward`** kapanışı iliştirilir; **topolojik sıralamanın tersinde** çağrılınca tek `backward()` tüm gradyanları hesaplar.
6. Bir değişken birden çok yola besleniyorsa gradyanlar **toplanır** (`+=`) — yoksa ezilir (Bölüm 11 bug'ı).
7. **Neuron / Layer / MLP**, motorun üstüne kurulan sinir ağıdır; `parameters()` tüm ağırlıkları toplar (PyTorch `nn.Module` gibi).
8. **Eğitim** = döngüde forward → **zero_grad** → backward → gradyanın tersine adım (gradient descent). zero_grad'ı atlamak en meşhur hatadır.
9. micrograd ile PyTorch **aynı algoritmayı** çalıştırır; fark yalnızca tensör + GPU + ölçek.

! Tek Bir Cümle

Bir sinir ağını eğitmek sihir değildir: her hesabı bir ifade grafiğine dök, ileri geçişle loss'u hesapla, zincir kuralını grafikte geriye uygulayıp (backprop) her parametrenin gradyanını bul, sonra gradyanın tersine küçük bir adım at — micrograd bu çekirdeği 100 satırda gösterir, GPT bunu yalnızca devasa ölçekte tekrarlar.

8.19 Kontrol Soruları

i Soru 1: a = 3, b = 2 için şu grafiği düşün: e = a·b, d = e + 5, L = d·2. Tüm gradyanları elle hesapla.

Cevap: İleri geçiş: $e = 6$, $d = 11$, $L = 22$. Kökten geriye yürüyelim ($\partial L/\partial L = 1$). $L = d \cdot 2$ bir çarpma (diğerini geçirir), $d = e + 5$ bir toplama (aynen geçirir), $e = a \cdot b$ bir çarpma:

$$\frac{\partial L}{\partial d} = 2, \qquad \frac{\partial L}{\partial e} = 2 \cdot 1 = 2$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e} \cdot b = 2 \cdot 2 = 4, \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e} \cdot a = 2 \cdot 3 = 6$$

Cevap: $\partial L/\partial d = 2$, $\partial L/\partial e = 2$, $\partial L/\partial a = 4$, $\partial L/\partial b = 6$. Doğrulamak için a'yı 0,0001 dürt: $L \approx 22 + 4 \cdot 0{,}0001$ olmalı (sayısal gradyan ≈ 4).

ℹ Soru 2: _backward içinde neden self.grad = … değil self.grad += … kullanırız? Hangi durumda fark yaratır, hangi durumda yaratmaz?

Cevap: Bir değişken grafikte **birden çok kez** kullanılırsa (örn. $b = a + a$, ya da aynı girdinin iki nörona beslenmesi), o değişkenin `_backward`'ı birden çok kez çağrılır. = (atama) kullanırsak ikinci çağrı birincinin yazdığını **ezer**; oysa çok değişkenli zincir kuralı, farklı yollardan gelen katkıların **toplanmasını** gerektirir. += bu toplamayı yapar. Bir değişken yalnızca **bir kez** kullanılıyorsa = ile += aynı sonucu verir — fark yalnızca çoklu kullanımda ortaya çıkar. Bu yüzden `backward()`'tan önce tüm gradyanların 0 olması gerekir (birikim sıfırdan başlasın diye).

ℹ Soru 3: tanh'ı hem tek kompozit işlem hem de exp/pow/bölme atomlarından kurabiliyoruz ve ikisi de aynı gradyanı veriyor. Bu nasıl mümkün? Hangisini seçmek neyi değiştirir?

Cevap: Geri yayılım yalnızca her düğümün **yerel türevinin** doğru olmasını ister. tanh'ı tek işlem yazarsan yerel türev $1 - o^2$'dir; atomlarına ayırırsan her atom (exp, pow, toplama, bölme) kendi yerel türevini taşır ve zincir kuralı bunları çarparak **aynı** sonuca ulaşır. Yani matematiksel sonuç (doğruluk) değişmez. Değişen şey grafiğin **granülerliğidir**: atomik graf daha çok düğüm/bellek demektir ama daha esnektir; tek-parça işlem daha az düğüm ve pratikte daha hızlıdır. Üretimde kütüphaneler bu atomları tek bir **fused kernel**'de birleştirip hızlandırır — soyutlama seviyesi bir performans kararıdır, doğruluk kararı değil.

ℹ Soru 4: (Builder) backward() neden düğümleri ters topolojik sırada çağırmak zorunda? Calculus zincir kuralıyla bağla.

Cevap: Bir düğümün gradyanını çocuklarına dağıtmadan önce, o düğümün **kendi gradyanının tamamlanmış** olması gerekir — yani çıkışındaki tüm düğümler çoktan işlenmiş olmalı. Zincir kuralı $\partial L/\partial x = \partial L/\partial y \cdot \partial y/\partial x$ der; burada $\partial L/\partial y$ (sonraki düğümün gradyanı) **önce** bilinmeli ki $\partial L/\partial x$'i hesaplayabilelim. Topolojik sıralama tüm kenarları soldan sağa dizer; bunu **tersten** gezmek, her düğüm işlenirken gradyanının hazır olmasını garanti eder. Döngü olsaydı (DAG değilse) ne sıralama ne de gradyan tanımlı olurdu — bu yüzden ifade grafiği çevrimsizdir.

8.20 Egzersizler

Egzersiz 1 (Value'yu sıfırdan kur). `Value` sınıfını baştan yaz: `data`, `grad`, `_prev`, `_op` ve `__add__` / `__mul__`. Sonra `a = Value(2.0)`, `b = Value(-3.0)`, `c = Value(10.0)` ile `L = (a*b + c) * Value(-2.0)` ifadesini kur. İleri geçişin Bölüm 5'teki $L = -8$ ile aynı çıktığını doğrula.

Egzersiz 2 (Elle backprop + gradient check). Egzersiz 1'in grafiğinde, her düğümün gradyanını **elle** hesapla ($\partial L/\partial a = 6$, $\partial L/\partial b = -4$ vb. çıkmalı). Sonra her girişi küçük bir $h = 0{,}0001$ ile dürtüp **sayısal gradyanı** hesapla ve analitik sonuçla karşılaştır. İki değer yaklaşık eşleşmeli (gradient check).

```
def numerical_grad(f, x, h=1e-4):
    # f: tek girisli fonksiyon, x: nokta -> sayisal turev
    return (f(x + h) - f(x - h)) / (2 * h)   # merkezi fark daha hassas
```

Egzersiz 3 (Edge case — += hatası). `a = Value(3.0)` için `b = a + a` kur ve `b.backward()` çağır. `a.grad` ne çıkmalı? `_backward`'ı = (atama) ile yazarsan ne çıkar, += (biriktirme) ile ne çıkar? Doğru cevabın ($\partial b/\partial a = 2$) neden yalnızca += ile geldiğini açıkla.

Egzersiz 4 (MLP'yi eğit). Bölüm 14-16'daki `MLP(3, [4, 4, 1])` ile 4 örnekli oyuncak veri setini eğit. Eğitim döngüsünü (forward → zero_grad → backward → update) 20 adım koştur, her adımda loss'u yazdır. Loss'un düştüğünü gözlemle. Sonra **bilerek zero_grad'ı kaldır** ve eğitimin nasıl bozulduğunu gör.

Egzersiz 5 (Sonraki dersin habercisi). micrograd skaler değerlerle çalışıyor ve örneklerimiz sayısaldı. Şimdi farklı bir problem düşün: bir **isim** üretmek (örn. "emma", "olivia"). Bir karakteri tahmin etmek için önceki karaktere bakan basit bir model nasıl kurulur? (a) Karakterleri sayıya nasıl çevirirsin (bir karakter = bir giriş)? (b) Çıktı neden bir **olasılık dağılımı** olmalı (tek bir sayı değil)? (c) micrograd'daki MSE yerine, "doğru karaktere yüksek olasılık ver" diyen bir kayıp nasıl tanımlanır? Bu üç soru, Ders 2'de (**makemore**, bigram dil modeli) kuracağımız modeli motive eder.

8.21 Sonraki Ders İçin Hazırlık

Ders 2: makemore 1 — Bigram Karakter Dil Modeli — Andrej Karpathy

Bu derste skaler bir MLP'yi elle eğittik. Ders 2'de **makemore** projesine geçiyoruz: 32 binden fazla gerçek ismi öğrenip yeni isimler *üreten* bir karakter-düzeyli dil modeli. Önce bigram'ları (ardışık karakter çiftlerini) sayarak, sonra **tam olarak aynı modeli** tek katmanlı bir sinir ağı olarak yeniden kurarak — micrograd'da gördüğümüz forward/backward/update döngüsünün birebir aynısıyla.

Ana konular:

- Karakter-düzeyli dil modeli: bir karaktere bakıp sonrakini tahmin etmek.
- Bigram sayımı, olasılık matrisi ve `torch.multinomial` ile örnekleme.
- **Negatif log olabilirlik** (NLL) kaybı — micrograd'daki MSE'nin dil modeli karşılığı.
- Aynı modelin sinir ağı hâli: one-hot girdi, softmax, gradient descent.

> ⚠ Ders 2 Öncesi Yapılacak
>
> - Egzersizleri çöz — özellikle 4 (MLP eğitimi) ve 5 (dil modeli sezgisi).
> - micrograd'ın `engine.py` ve `nn.py`'sini github.com/karpathy/micrograd üzerinden oku; 100 satırın tamamını gör.
> - Ana cümleyi tekrar oku: *"Bir sinir ağını eğitmek = ifade grafiğinde forward, sonra backprop (zincir kuralı), sonra gradyanın tersine adım."*

8.22 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Karpathy'de
micrograd / autograd	≈100 satırlık otomatik gradyan motoru; backprop'u skaler düzeyde gösterir	0m47
Türev / sayısal gradyan	Girişi h ile dürtünce çıktının değişimi; $(f(x+h)-f(x))/h$ ile sayısal okunur	8m09
Value nesnesi	Skaleri saran kutu; data, grad, _prev, _op, _backward tutar	19m30
İfade grafiği (DAG)	İşlemlerin ördüğü yönlü çevrimsiz grafik; yapraklar girdi, kök çıktı	19m30
Yerel türev kuralları	Toplama gradyanı aynen geçirir; çarpma diğer operandı geçirir	38m03
Zincir kuralı (backprop)	$\partial L/\partial x = \partial L/\partial y \cdot \partial y/\partial x$; grafikte kökten yapraklara çarpılarak taşınır	42m03
tanh aktivasyonu	Nöronun doğrusal-olmama katmanı; yerel türevi $1-o^2$	52m52
_backward kapanışı	Her işleme iliştirilen, yerel gradyanı girişlere dağıtan fonksiyon	1h09m
Topolojik sıralama	Düğümleri bağımlılık sırasına dizer; backward ters sırada gezer	1h18m
Gradyan biriktirme (+=)	Birden çok kez kullanılan düğümün gradyanları toplanır, ezilmez	1h25m
Neuron / Layer / MLP	Motorun üstüne kurulan ağ; parameters() tüm ağırlıkları toplar	1h44m
MSE kaybı	$\sum(\text{tahmin}-\text{hedef})^2$; eğitilen loss, ifade grafiğinin köküdür	1h51m
zero_grad + eğitim döngüsü	forward → zero_grad → backward → gradyanın tersine adım	2h08m

8.23 ML Builder Bağlantıları

9 köprü

Bu ders, modern derin öğrenme altyapısının skaler prototipidir — köprülerin özeti:

1. **Value + backward**() → PyTorch `torch.Tensor` + `loss.backward()` (autograd). İleriye: tensör + GPU.
2. **İfade grafiği** → PyTorch computation graph (`grad_fn`); aynı "neyden üretildim" bilgisi.
3. **İşleme gömülü _backward** → PyTorch `torch.autograd.Function`'ın forward/backward çifti.
4. **Topolojik sıralama** → reverse-mode autodiff; gradyanı çıktıdan girişe biriktirme (derin ağlarda neden verimli).
5. **Neuron / Layer / MLP + parameters**() → `torch.nn.Module` API'si; `model.parameters()` optimizer'a verilir.
6. $\mathbf{w} \cdot \mathbf{x} + b$ → 18.06 dot product / matris çarpımı (Ders 30). İleriye: GPU GEMM, throughput.
7. **backprop = zincir kuralı** → Calculus Ders 4; Karpathy'nin kendi sözüyle "nothing more than the chain rule".
8. **zero_grad + gradyan biriktirme (+=)** → PyTorch `optimizer.zero_grad()`; "unutursan bug" hatasının kaynağı.
9. **"Gerisi sadece verimlilik"** → skalerden tensöre, oradan GPT ölçeğine (Ders 7 ve 10) aynı çekirdek.

8.24 Karpathy'nin Önerdiği Kaynaklar

Karpathy'nin bu ders için verdiği kaynaklar:

- **micrograd repo:** github.com/karpathy/micrograd — dersin kurduğu kütüphanenin tamamı (engine.py + nn.py).
- **Ders deposu:** nn-zero-to-hero / lectures / micrograd — ders notebook'ları.
- **Ders Colab notebook'u:** Google Colab — dersi adım adım çalıştırabileceğin ortam.

Tek bir şey alıp gideceksen

Bir sinir ağı sihir değildir — "dot product + bias + doğrusal-olmama" yapı taşının istiflenmesidir ve onu eğitmek, bir loss'u gradient descent ile minimize edip gradyanı backpropagation (zincir kuralı) ile hesaplamaktan ibarettir. micrograd bu çekirdeği 100 satırda gösterir; GPT yalnızca aynı şeyi devasa ölçekte tekrarlar. *"Micrograd is what you need to train your networks, and everything else is just efficiency."*

9 makemore 1 — Bigram Karakter Dil Modeli

Aynı modeli iki kez kur: önce bigram'ları say-normalize et, sonra birebir aynısını tek katmanlı bir sinir ağı olarak softmax + gradient descent ile eğit

> Bölüm bilgisi
>
> - **Karpathy'nin videosu:** YouTube — The spelled-out intro to language modeling: building makemore (≈118 dk)
> - **Seri:** Neural Networks: Zero to Hero — Ders 2
> - **Hoca:** Andrej Karpathy
> - **Kaynak repo:** github.com/karpathy/makemore
> - **Okuma süresi:** ≈32 dk

9.1 Bu Derste Ne Var?

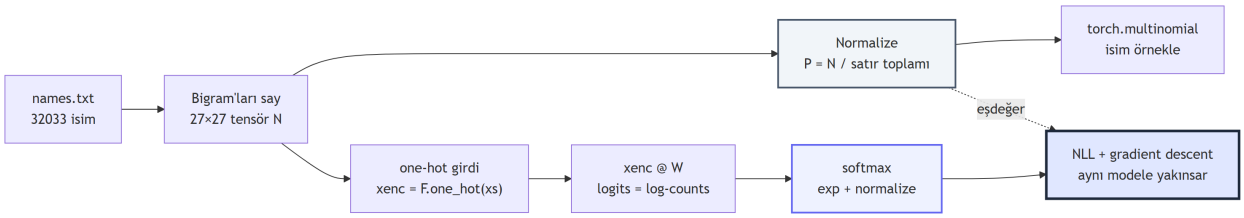
Ders 1'de skaler bir sinir ağını (micrograd) elle eğittik. Bu derste **makemore** projesine geçiyoruz: 32 binden fazla gerçek ismi (`names.txt`) öğrenip yeni, isme-benzer diziler *üreten* bir **karakter-düzeyli dil modeli**. Karpathy serinin geri kalanını da haber veriyor: bigram'dan başlayıp adım adım GPT-2 eşdeğeri bir transformer'a kadar tırmanacağız.

> *"Like micrograd before it, makemore is a repository that I have on my GitHub webpage. Just like with micrograd, I'm going to build it out step by step and I'm going to spell everything out."* — Karpathy, 0:06

Dersin büyük fikri şu: **aynı modeli iki kez** kuracağız. Önce saf sayımla (her karakter çiftinin kaç kez geçtiğini sayarak), sonra **birebir aynı modeli** tek katmanlı bir sinir ağı olarak (gradient descent ile eğitilen). İkisinin aynı sonucu vermesi, Ders 1'deki mekaniğin gerçek bir probleme nasıl oturduğunu gösterir.

Dersin üç büyük fikri:

1. **Bigram dil modeli** — bir sonraki karakteri yalnızca **önceki tek karaktere** bakarak tahmin etmek; sayım + normalize ile bir olasılık tablosu kurmak.
2. **Negatif log olabilirlik (NLL)** — modelin ne kadar iyi olduğunu ölçen tek sayı; Ders 1'deki MSE'nin dil modeli karşılığı.
3. **Aynı model = tek katmanlı sinir ağı** — one-hot girdi, `xenc @ W`, softmax, sonra Ders 1'deki forward/backward/update döngüsüyle eğitim.



Şekil 9.1: Ders 2'nin kavram haritası: aynı bigram modelini iki yoldan kur. Üst yol (say → normalize → örnekle) ile alt yol (one-hot → softmax → eğit) aynı modele ulaşır.

Builder Notu — İki Yol, Tek Model

Geriye (Ders 1 + Stat 110):

- **Sayım → olasılık = Stat 110.** Bigram sıklıklarını toplayıp normalize etmek, frekanstan olasılığa geçiştir; NLL'i minimize etmek **maximum likelihood**'dur (Stat 110, MLE).
- **Eğitim döngüsü = Ders 1 micrograd.** Bu dersin sinir ağı kısmı, micrograd'da gördüğümüz `forward → backward → update` döngüsünün birebir aynısı; Karpathy bunu derste micrograd notebook'unu açarak gösterir.
- **MSE → NLL.** Ders 1'de regresyon için MSE kullandık; burada çıktı bir **olasılık dağılımı** olduğu için kayıp NLL olur (sınıflandırmanın doğal kaybı).

İleriye: Bu bigram yalnızca **tek** karaktere bakıyor — bağlam yok. Ders 3'te **embedding tablosu + MLP** ile birkaç karakterlik bağlam ekleyeceğiz; sonunda transformer (Ders 7) ile uzun bağlam. Ama çerçeve hep aynı: "bir sonraki token'ı tahmin et, NLL'i düşür". `torch.multinomial`, broadcasting, softmax burada öğreneceğin ve tüm seri boyunca kullanacağın production araçları.
Tek cümleyle: Bir dil modeli, "bir sonraki karakterin olasılık dağılımını" üreten bir fonksiyondur; bigram bunu en basit hâliyle (tek karakter bağlamı) yapar ve ister say, ister gradient descent ile eğit — aynı modele ulaşırsın.

9.2 makemore Nedir? Veri Seti

makemore, adının söylediğini yapar: kendisine verilen şeylerden **daha fazlasını üretir**. Bu derste girdi `names.txt` — 32 binden fazla gerçek isim (emma, olivia, ava, …). Model bu isimleri "öğrenip" yeni, isme-benzer ama var olmayan diziler üretecek.

Karakter-düzeyli bir model kuruyoruz: İsmi bir **karakter dizisi** olarak görür, bir sonraki karakteri bir öncekilerden tahmin eder. Önce veriyi yükleyip tanıyalım.

```
words = open('names.txt', 'r').read().splitlines()
len(words)                 # 32033
min(len(w) for w in words) # 2  (en kisa isim)
max(len(w) for w in words) # 15 (en uzun isim)
words[:3]                  # ['emma', 'olivia', 'ava']
```

Kritik sezgi: tek bir "emma" kelimesi bile **bir sürü** eğitim örneği taşır. "e'den sonra m gelir", "m'den sonra m gelir", "m'den sonra a gelir", artı ismin nerede başlayıp nerede bittiği bilgisi. Bir karakter-düzeyli model, kelimeyi bu küçük "şu karakterden sonra şu gelir" kararlarına böler.

> Builder Notu — Diziyi Tahmin Problemine Çevirmek
>
> **İleriye:** Bu "bir diziyi, bir sonraki öğeyi tahmin etme problemine çevirme" fikri, tüm dil modellemenin temelidir — GPT de tam olarak bunu yapar, yalnızca karakter yerine **token** ve tek karakter bağlamı yerine binlerce token bağlamıyla (Ders 9: tokenizer, Ders 7/10: GPT). 32033 isim küçük görünür ama yüz binlerce bigram örneği taşır; veri bolluğu, sayım yerine öğrenmeye geçince anlam kazanır.

9.3 Bigram'ları Saymak

Bigram, ardışık iki karakterdir. Bigram modeli son derece zayıftır: yalnızca **tek** önceki karaktere bakar, ondan öncesini unutur. Ama başlamak için mükemmel.

Her kelimedeki ardışık çiftleri `zip(w, w[1:])` ile gezeriz. Bir de ismin **başını ve sonunu** işaretlemek gerek; Karpathy önce ayrı `<S>` / `<E>` token'ları kullanır, sonra ikisini tek bir nokta token'ına (`.`) sadeleştirir. Çiftleri bir Python sözlüğünde sayarız:

```
b = {}
for w in words:
    chs = ['.'] + list(w) + ['.']        # basla/bitir token'i
    for ch1, ch2 in zip(chs, chs[1:]):   # ardisik ciftler
        bigram = (ch1, ch2)
        b[bigram] = b.get(bigram, 0) + 1  # say

# en sik bigram'lar
sorted(b.items(), key=lambda kv: -kv[1])[:3]
```

`b.get(bigram, 0) + 1` deyimi önemli: sözlükte yoksa varsayılan 0 döner, varsa mevcut sayıyı alır — böylece her çifti güvenle artırırız. Sonuç: hangi karakter çiftinin kaç kez geçtiğini gösteren bir tablo. Bir sonraki adımda bunu bir sözlük yerine bir **tensöre** taşıyacağız — çünkü tensör hem hızlı hem de PyTorch'un tüm gücünü açar.

> Builder Notu — Sayım Neden Ölçeklenmez
>
> **İleriye:** Sayım, bir modeli eğitmenin en sade yoludur — "veriden istatistik çıkar". Ama ölçeklenmez: bigram (tek karakter) için 27×27 tablo yeter, ama iki karakter bağlamı $27 \times 27 \times 27$, on karakter için astronomik olur. Bu yüzden Ders 3'ten itibaren sayım yerine **öğrenilen parametreler** (sinir ağı) kullanılır. Bu sayım-patlaması, dersin sonundaki Egzersiz 5'in tam çekirdeği.

9.4 Sayımları 2B Tensöre: N ve s2i / i2s

Python sözlüğü yerine sayımları bir **2 boyutlu tensörde** tutalım. 27 karakter var (26 harf + nokta token'ı), yani 27×27'lik bir tamsayı tensörü `N`: satır = ilk karakter, kolon = ikinci karakter, hücre = o çiftin sayısı.

Önce karakterleri tamsayıya çeviren bir **arama tablosu** gerekir (`s2i` = string-to-integer) ve tersi (`i2s`). Karpathy noktayı (`.`) 0 indeksine, harfleri 1-26'ya yerleştirir:

```
import torch

chars = sorted(list(set(''.join(words))))   # a..z
s2i = {ch: i + 1 for i, ch in enumerate(chars)}   # a->1 .. z->26
s2i['.'] = 0                                       # nokta -> 0
i2s = {i: ch for ch, i in s2i.items()}             # ters tablo

N = torch.zeros((27, 27), dtype=torch.int32)
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        N[s2i[ch1], s2i[ch2]] += 1   # ilgili hucreyi artir
```

Artık tüm bigram istatistiği tek bir tensörde. `dtype=torch.int32` ile tamsayı tutuyoruz (sayımlar tam sayı). Bir sonraki adımda bu sayımları olasılığa çevirip örnekleme yapacağız.

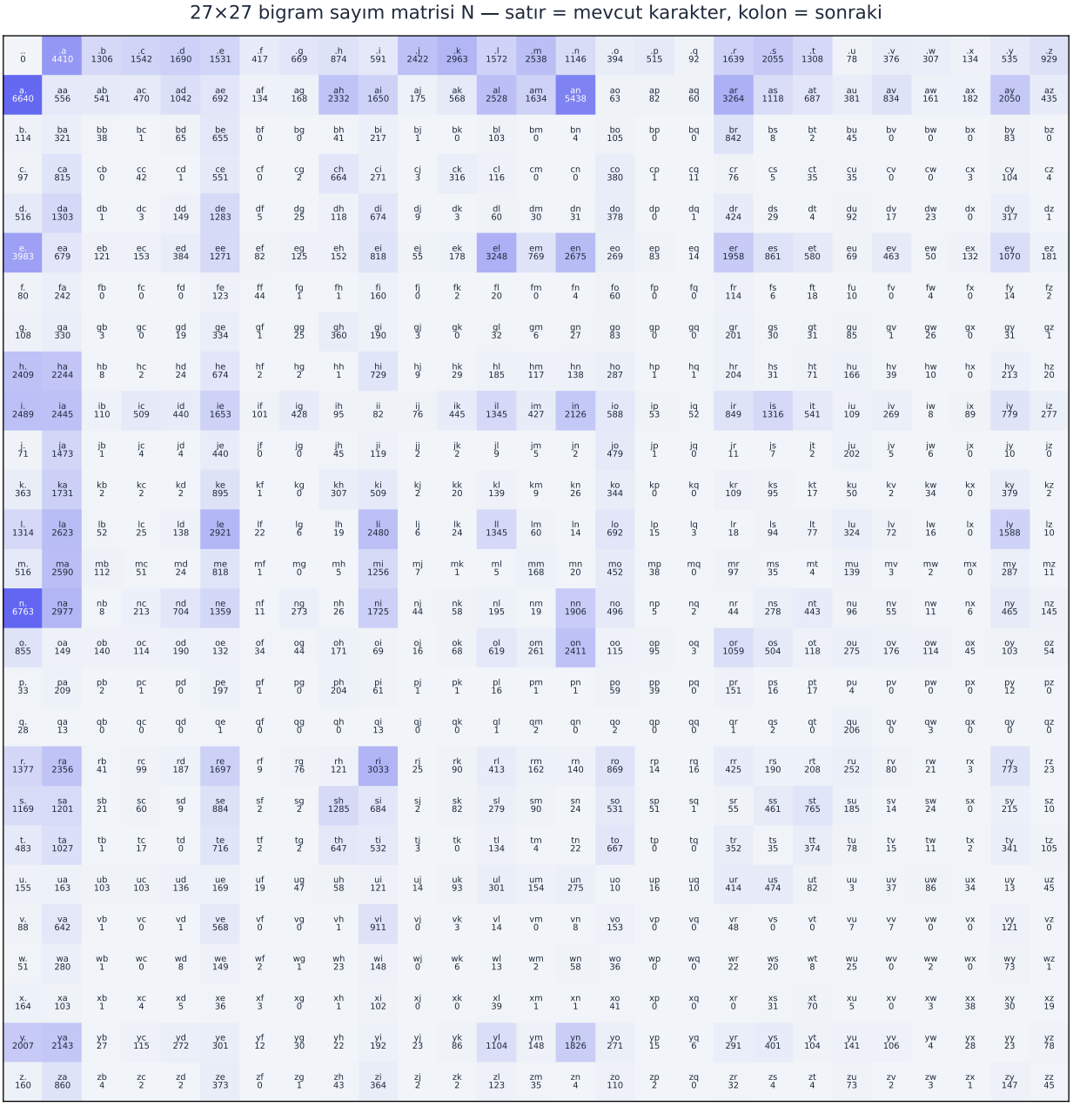
> Builder Notu — N Bir Matris, s2i Bir Tokenizer
>
> **Geriye (18.06):** `N` bir matristir; satır/kolon indeksleme tam olarak lineer cebirdeki matris elemanı erişimidir. Birazdan bu matrisi satır-satır normalize edip her satırı bir olasılık dağılımına çevireceğiz. **İleriye:** `s2i` / `i2s` arama tabloları, her dil modelinin **tokenizer**'ının çekirdeğidir — karakter/token ile tamsayı id arasındaki çeviri (Ders 9'da bunun gerçek, byte-pair encoding hâlini kuracağız).

9.5 Görselleştirme ve Tek '.' Token

Karpathy `matplotlib` ile `N`'i etiketli bir ızgara olarak çizer — her hücrede bigram ve sayısı görünür. Görselleştirme, modelin ne öğrendiğini gözle görmeyi sağlar: örneğin satır `.` (başlangıç) en çok hangi harfle başlandığını, kolon `.` ise isimlerin en çok hangi harfle bittiğini gösterir.

Başlangıçta ayrı `<S>` ve `<E>` token'ları kullanmak iki sorun yaratır: tablo gereksiz büyür ve bazı satır/kolonlar hep sıfır kalır. Çözüm: **tek bir `.` token'ı** hem başlangıç hem bitiş için (indeks 0). Bu, tabloyu temiz 27×27'ye indirir.

> Builder Notu — Modeli Görselleştir
>
> **İleriye:** "Modeli görselleştir, ne öğrendiğine bak" alışkanlığı production'da kritiktir — attention matrisleri, embedding uzayları, aktivasyon istatistikleri hep görselleştirilir (Ders 4'te aktivasyon/gradyan histogramları, Ders 7'de attention desenleri). Yukarıdaki ısı haritası, bir modeli "okumanın" en sade



Şekil 9.2: Karpathy'nin imza görseli: 27×27 bigram sayım matrisi N. Satır = mevcut karakter, kolon = sonraki karakter; her hücrede üstte bigram etiketi (ör. `em`), altta o çiftin sayısı. Renk Slate→Indigo: açık = az, indigo = çok. 0. satır (`.`) isimlerin hangi harfle **başladığını**, 0. kolon (`.`) hangi harfle **bittiğini** gösterir.

hâli: koyu hücreler sık geçişler, açık hücreler nadir veya hiç görülmemiş çiftler.

9.6 Olasılığa Çevirip Örnekleme

Modelden **isim üretmek** için: bir karakterden başla (`.`), o satırın sayımlarını olasılığa çevir, bu dağılımdan bir sonraki karakteri çek, tekrarla — ta ki `.` (bitiş) gelene dek.

Bir satırı olasılığa çevirmek = sayıları toplamlarına bölmek (normalize). Sonra `torch.multinomial` ile bu dağılımdan örnek çekeriz. Tekrarlanabilirlik için tohumlu bir `torch.Generator` kullanılır.

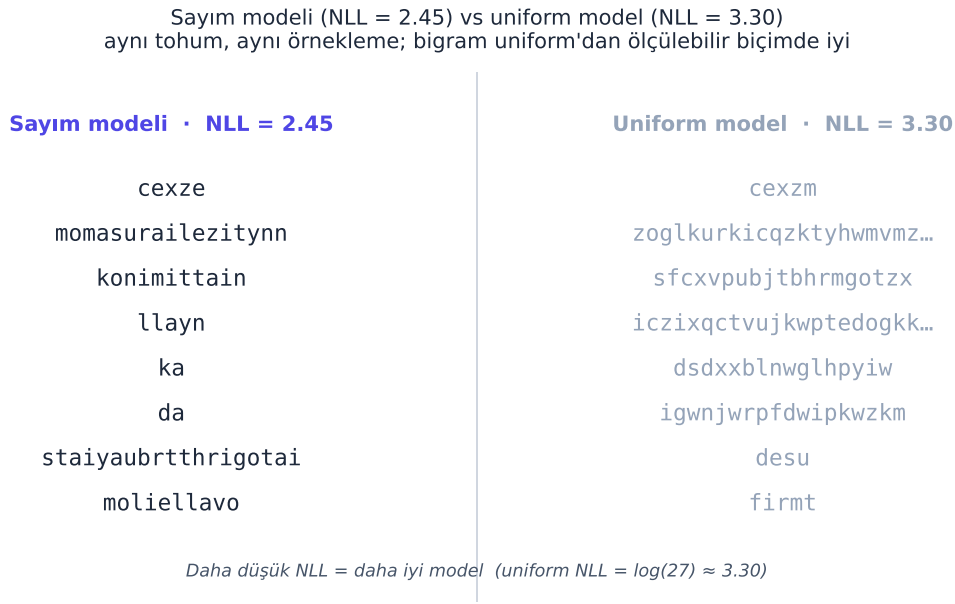
> *"To sample from these distributions we're going to use torch.multinomial."* — Karpathy, 26:29

```
g = torch.Generator().manual_seed(2147483647)   # tohumlu -> tekrarlanabilir

for _ in range(5):
    out = []
    ix = 0                          # '.' ile basla
    while True:
        p = N[ix].float()
        p = p / p.sum()             # satiri olasiliga cevir (normalize)
        ix = torch.multinomial(p, num_samples=1, replacement=True,
                               generator=g).item()
        if ix == 0:                 # '.' bitis -> dur
            break
        out.append(i2s[ix])
    print(''.join(out))
```

Aşağıda, sayım modeliyle (`P`) üretilen isimleri, her satırı eşit $(1/27)$ olan bir **uniform** modelin ürettikleriyle yan yana koyuyoruz — aynı tohum, aynı örnekleme.

Üretilen isimler korkunç görünür ("cexze", "ka", …) — ama Karpathy önemli bir noktayı vurgular: bu model **rastgele (uniform) bir modelden ölçülebilir biçimde daha iyidir**. Sayım modelinin ortalama NLL'i ≈2,45, uniform modelinki $\log 27 \approx 3{,}30$; bigram bile veriden bir şey öğrenmiştir; sadece tek karakter bağlamı çok zayıf olduğu için sonuç zayıf.

> Builder Notu — Dağılımdan Örnekle
>
> **Geriye (Stat 110):** `torch.multinomial`, bir olasılık dağılımından örnek çeker — Stat 110'daki multinomial/kategorik örneklemenin ta kendisi. Tohumlu `Generator` ise determinizm için (aynı tohum = aynı sonuç), production'da **reproducibility**'nin temeli.
> **İleriye:** "Bir olasılık dağılımı üret, ondan örnekle" döngüsü, her üretken modelin (GPT dahil) çekirdeğidir. GPT de her adımda bir sonraki token için dağılım üretip ondan örnekler (sampling, temperature, top-k — hepsi bu multinomial adımının varyasyonları).



Şekil 9.3: Sayım modeli vs uniform model: aynı tohumla örneklenen 8'er isim (metin paneli). Sol sütun bigram sayım modeli P ile, sağ sütun her satırı $1/27$ eşit olan uniform model ile üretildi. Sayım modelinin ortalama NLL'i ($\approx 2{,}45$) uniform modelinkinden ($\log 27 \approx 3{,}30$) belirgin küçüktür — bigram, tek karakter bağlamıyla bile uniform'dan ölçülebilir biçimde iyidir; sol isimler isme-benzer, sağdakiler rastgele çöptür.

9.7 Vektörel Normalizasyon ve Broadcasting

Her örnekte satırı tek tek normalize etmek yavaş. Bunun yerine **tüm satırları bir kerede** normalize edip bir olasılık matrisi `P` önceden hesaplarız. İşte burada PyTorch'un **broadcasting** (yayım) kuralları devreye girer — ve Karpathy en sık yapılan hatayı kasten gösterir.

```
P = N.float()
P = P / P.sum(1, keepdim=True)   # her satiri kendi toplamina bol
```

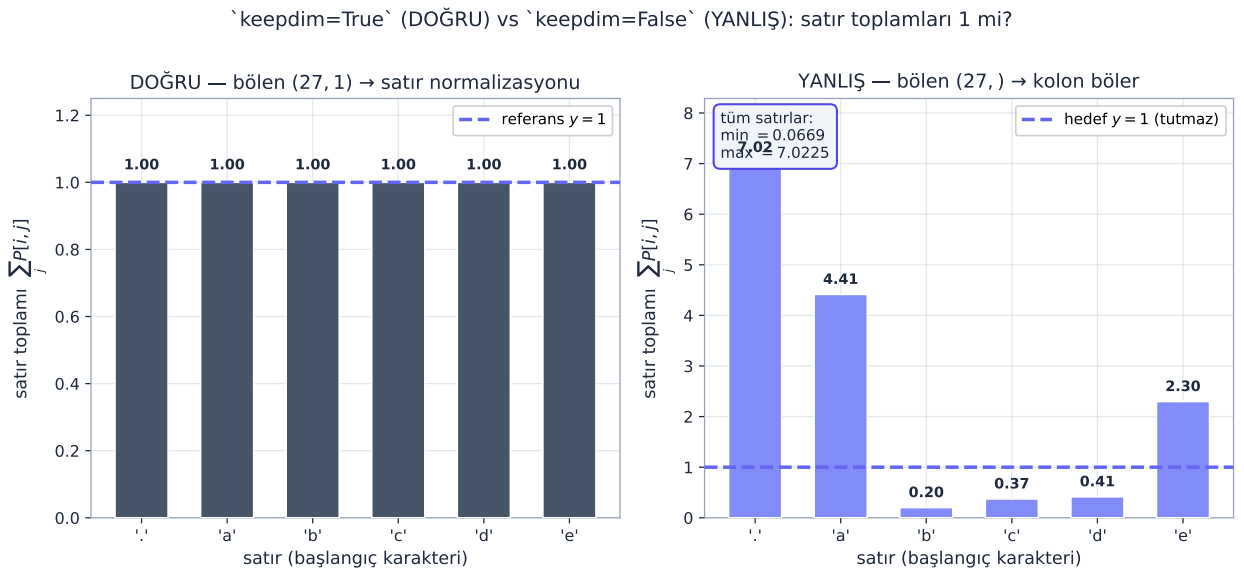
İncelik `keepdim` argümanında. `N.sum(1)` satır toplamlarını verir ama şekli `(27,)` — bu, broadcasting'de bir **satır vektörü** gibi hizalanır ve yanlışlıkla her **kolonu** böler. `N.sum(1, keepdim=True)` ise şekli `(27, 1)` tutar — bu, her **satır** boyunca yayılır ve doğru olanı (satır normalizasyonu) yapar.

> *"I'll be honest with you, this doesn't look right, so I spent a few minutes to convince myself that it actually is right."* — Karpathy, 34:14

Aşağıdaki figür ikisini yan yana koyuyor: `keepdim=True` ile satır toplamları tam 1 (doğru), `keepdim=False` ile satır toplamları 1 değil (yanlış — kod hata fırlatmadan sessizce yanlış eksende böler).

Karpathy broadcasting'e saygı duyulması gerektiğini özellikle vurgular; sessizce yanlış sonuç veren ama hata fırlatmayan bu tür buglar en tehlikelisidir:

> *"Now I would like to scare you a little bit… I encourage you to treat this with respect, and it's not something to play fast and loose with."* — Karpathy, 44:19



Şekil 9.4: Broadcasting tuzağı yan yana: solda `keepdim=True` bölen şekli $(27, 1)$ — her satır kendi toplamına bölünür, satır toplamları tam 1 (DOĞRU); sağda `keepdim=False` bölen şekli $(27,)$ — vektör kolonlara hizalanır, satır toplamları 1 değil (YANLIŞ; min / max $= 0{,}0669 / 7{,}0225$). İlk altı satırın `.sum(1)` değerleri bar olarak; indigo yatay çizgi $y = 1$ referansı.

> **Builder Notu — Sessiz Şekil Hataları**
>
> **İleriye:** Broadcasting, tüm tensör kodunun (NumPy, PyTorch, JAX) temelidir ve **sessiz hata** kaynağı bir numara: şekiller "uyduğu" için kod çalışır ama yanlış eksende işlem yapar. Kural: kritik işlemlerden sonra `.shape` yazdır (Ders 1'deki `print(x.shape)` debug alışkanlığı). Bu bug sınıfı production'da modelinizi sessizce bozabilir — yukarıdaki sağ panelde satır toplamları $0{,}0669$ ile $7{,}0225$ arasında savrulur, yani hiçbiri olasılık dağılımı değildir.

9.8 Loss: Negatif Log Olabilirlik (NLL)

Model "iyi mi"? Tek bir sayıya ihtiyacımız var. Sezgi: iyi bir model, **veride gerçekten görülen** bigram'lara yüksek olasılık atamalı. Tüm bu olasılıkların **çarpımı** (likelihood) modelin veriyi ne kadar iyi açıkladığını ölçer — ama çarpım çok küçük sayılara iner, bu yüzden **logaritmasını** alırız (çarpım → toplam):

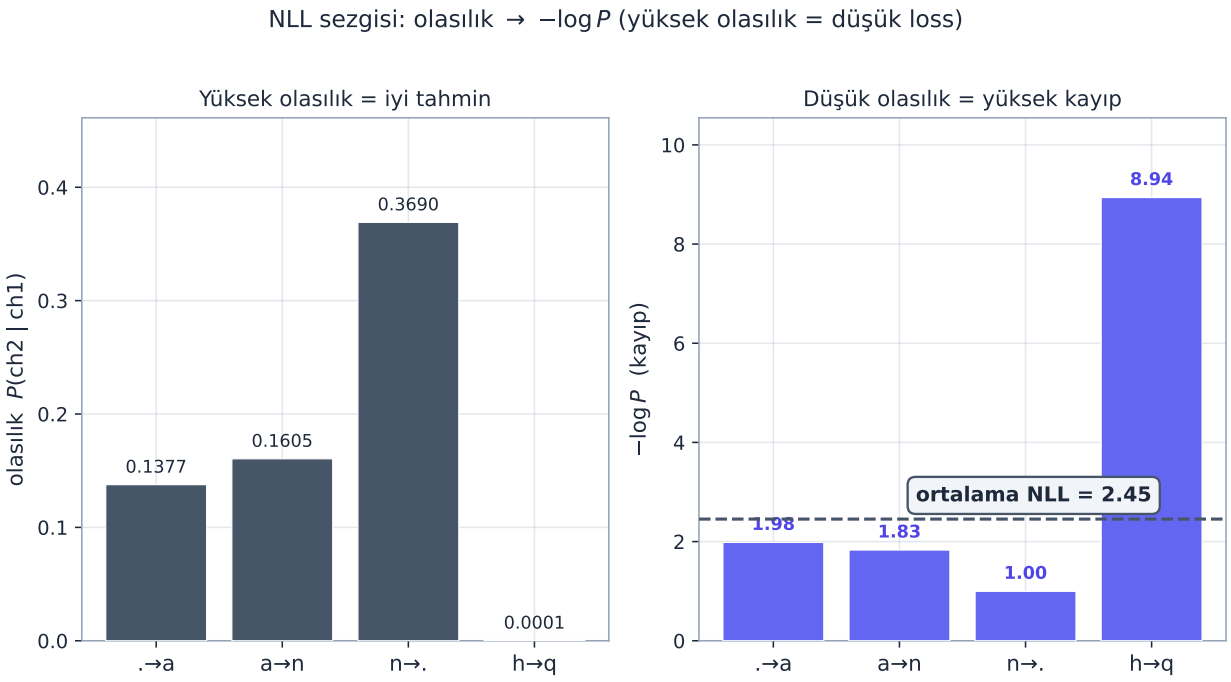
$$\text{likelihood} = \prod_i P(x_i), \qquad \log(\text{likelihood}) = \sum_i \log P(x_i)$$

Loss'u "düşük = iyi" yapmak için negatifini alır ve örnek sayısına böleriz — **ortalama negatif log olabilirlik**:

$$\text{NLL} = -\frac{1}{n} \sum_i \log P(x_i)$$

"The loss function: the negative log likelihood of the data under our model." — Karpathy, 50:09

Aşağıdaki figür sezgiyi somutlaştırır: yüksek olasılıklı bir bigram düşük $-\log P$ (düşük kayıp) verir; nadir bir çift çok yüksek kayıp verir.



Şekil 9.5: NLL sezgisi: olasılık ile o olasılığın $-\log P$ kaybı yan yana. **Sol (slate):** birkaç bigram'ın $P(\text{ch2} \mid \text{ch1})$ olasılığı. **Sağ (indigo):** aynı bigram'ların $-\log P$ değeri — modelin o çifte verdiği *kayıp*. Yüksek olasılık (ör. $P(. \to a) = 0{,}14$) düşük kayıp ($\approx 1{,}98$) verir; nadir bir çift ($P(h \to q) \approx 0{,}0001$, veride 1 kez) çok yüksek kayıp ($\approx 8{,}94$) verir. Tüm bigram'ların $-\log P$ ortalaması = sayım modelinin **ortalama NLL'i** ($\approx 2{,}45$). $-\log$ monoton azalandır: olasılık 1'e giderken kayıp 0'a, 0'a giderken $+\infty$'a gider.

Bigram modeli için bu ≈2,45 çıkar. Karpathy maximum likelihood ile bağını net kurar:

> *"Our goal is to maximize likelihood, the product of all the probabilities assigned by the model… maximizing the likelihood is equivalent to maximizing the log likelihood because log is a monotonic [function]."* — Karpathy, 59:08

Builder Notu — NLL = Maximum Likelihood = Cross-Entropy

Geriye (Stat 110 + Ders 1): NLL'i minimize etmek = log-likelihood'u maksimize etmek = **maximum likelihood estimation** (Stat 110). Log'un monotonluğu, çarpım yerine toplamla çalışmayı (sayısal kararlılık + türevlenebilirlik) sağlar. Bu, Ders 1'deki **cross-entropy**'nin ta kendisidir — orada Bernoulli, burada karakterler üzerinden multinomial.
İleriye: NLL / cross-entropy, neredeyse tüm sınıflandırma ve dil modellerinin standart kaybıdır. GPT'nin eğitildiği loss da budur: bir sonraki token'a atanan olasılığın negatif log'u.

9.9 Model Yumuşatma (Fake Counts)

Bir sorun: veride hiç görülmemiş bir bigram'a model **sıfır** olasılık atar; o bigram bir isimde geçerse $\log(0) = -\infty$ olur, NLL patlar. Çözüm basit ve klasik: her sayıma küçük bir **sahte sayı** ekle (örn. +1), böylece hiçbir olasılık sıfır olmaz.

```
P = (N + 1).float()              # +1 = yumusatma (smoothing)
P = P / P.sum(1, keepdim=True)
```

Ne kadar çok eklersen model o kadar "düzleşir" (her şey eşit olasılığa yaklaşır); ne kadar az, gerçek sayımlara o kadar sadık kalır. Bu, modeli aşırı keskinlikten koruyan bir denge ayarıdır. Pratikte +1 yumuşatma, sayım NLL'ini neredeyse hiç bozmaz (2,4540'tan 2,4546'ya), ama $-\infty$ riskini ortadan kaldırır.

> Builder Notu — Smoothing = Laplace = Uniform Prior
>
> **Geriye (Stat 110):** Sahte sayı eklemek, Stat 110'daki **Laplace ardışıklık kuralının** (Laplace smoothing) ta kendisidir: uniform bir prior eklemek gibi. Birazdan göreceğiz ki bu, sinir ağı versiyonunda **L2 düzenlileştirme** (regularization) ile birebir aynı işi yapar (§14).
> **İleriye:** "Görülmemiş duruma sıfır verme" problemi her olasılıksal modelde vardır; smoothing/regularization, modelin eğitim verisine aşırı güvenmesini (overfitting) engelleyen genel bir araçtır.

9.10 Bölüm 2: Bigram'ı Sinir Ağı Olarak Görmek

Şimdi dersin asıl güzelliği: **tam olarak aynı bigram modelini** bir sinir ağı olarak yeniden kuralım. Sayım yerine, modeli **gradient descent ile** eğiteceğiz (Ders 1 micrograd mekaniği).

Yeni çerçeve: ağ bir karakter alır (girdi), bir ağırlık matrisi `W` vardır, çıktı olarak **bir sonraki karakterin olasılık dağılımını** üretir. Eğitim = NLL kaybını gradient descent ile minimize etmek. Sonuçta öğrenilen `W`, sayım tablosuyla **aynı** modeli verecek — ama bu kez yöntem her ölçeğe genişleyebilir.

> Builder Notu — Aynı Fikir, Tensör Hâli
>
> **Geriye (Ders 1):** Bu, micrograd'da kurduğumuz "parametreleri loss'u düşürecek şekilde ayarla" fikrinin gerçek bir probleme uygulanması. Tek fark: skaler yerine tensör, elle yazılmış MLP yerine PyTorch.
> **İleriye:** "Girdi → ağırlıklar → olasılık dağılımı → NLL ile eğit" şablonu, tüm dil modellerinin iskeletidir. Ders 3'te `W`'nin yerini embedding + gizli katman alır; GPT'de devasa bir transformer — ama çerçeve sabit.

9.11 Bigram Veri Seti ve One-Hot Kodlama

Önce eğitim verisini hazırlarız: her bigram bir (girdi, hedef) çiftidir. `xs` girdi karakterlerinin tamsayı id'leri, `ys` hedef (bir sonraki) karakterlerin id'leri.

```
xs, ys = [], []
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        xs.append(s2i[ch1])   # girdi karakter id
        ys.append(s2i[ch2])   # hedef (sonraki) karakter id
xs = torch.tensor(xs)         # KUCUK harf torch.tensor (dtype korur)
ys = torch.tensor(ys)
```

Karpathy bir tuzağa dikkat çeker: `torch.tensor` (küçük t) girdinin dtype'ını korur (tamsayı kalır); `torch.Tensor` (büyük T) float32'e çevirir. Küçük harf tercih edilir.

Bir tamsayı id'sini doğrudan sinir ağına **besleyemezsin** (ağ sayılarla aritmetik yapar, "13" karakterin sırası değil sayısal değeri olur). Çözüm **one-hot kodlama**: id'yi, yalnızca o indekste 1 olan bir sıfır vektörüne çevir.

```
import torch.nn.functional as F
xenc = F.one_hot(xs, num_classes=27).float()   # (n, 27), float'a cevir
```

`xenc`'in her satırı tek bir 1 içeren 27-uzunlukta bir vektör. `.float()` şart: ağ ondalık aritmetik yapacak.

> Builder Notu — One-Hot = Baz Vektörü
>
> **Geriye (18.06):** One-hot vektör, standart **baz vektörüdür** (e_i) — yalnızca i'inci bileşeni 1. Birazdan göreceğin gibi, bir baz vektörüyle matris çarpımı, matrisin tam o satırını/kolonunu seçer (18.06 Ders 30).
> **İleriye:** One-hot, küçük sözlükler için iyidir ama 50.000 token'lık bir sözlükte 50.000-uzunlukta seyrek vektör israftır. Ders 3'te bunun yerini **embedding tablosu** alır (doğrudan id ile satır seçimi) — ama Karpathy birazdan one-hot @ W'nin zaten bir satır seçimi olduğunu gösterecek (§14), yani embedding'in habercisi.

9.12 Tek Lineer Katman + Softmax

Ağırlık matrisini rastgele başlatırız: `W = torch.randn((27, 27))`. Girdiyi (one-hot) `W` ile çarparız: `xenc @ W`. Bu, 27 nöronun aynı girdiye paralel ateşlemesidir — çıktı her olası sonraki karakter için bir ham sayı.

Bu 27 ham sayı **logit** (log-counts) olarak yorumlanır:

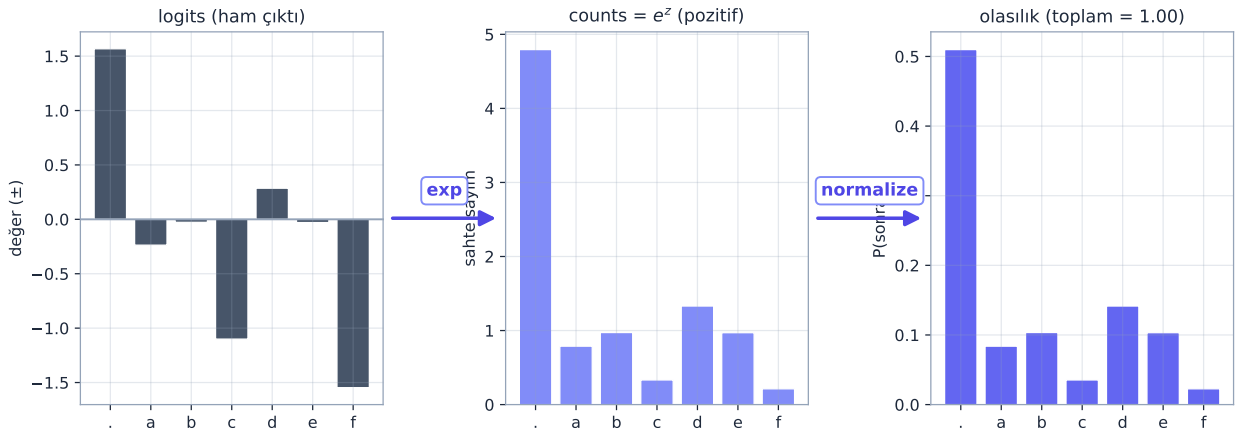
> *"These 27 numbers are giving us log counts basically. So instead of giving us counts directly… and to get the counts we're going to exponentiate them."* — Karpathy, 1:20:42

Yani: logitleri **üstel** alıp (pozitif "fake counts" elde et), sonra normalize et → olasılık dağılımı. Bu iki adım (exp + normalize) tam olarak **softmax**'tır:

$$z = x_{enc}\, W, \qquad P = \frac{e^{z}}{\sum_{j} e^{z_j}}$$

```
W = torch.randn((27, 27), requires_grad=True)   # ogrenilecek parametre

xenc = F.one_hot(xs, num_classes=27).float()
logits = xenc @ W                # ham log-counts (n, 27)
counts = logits.exp()            # fake counts (pozitif)
probs = counts / counts.sum(1, keepdim=True)   # softmax -> olasilik
```

`logits.exp()` neden? Çünkü ham logitler negatif olabilir, ama sayım/olasılık pozitif olmalı; üstel her şeyi pozitife taşır. `counts.sum(1, keepdim=True)` — yine §6'daki broadcasting kuralı (satır normalizasyonu için `keepdim=True`). Aşağıdaki figür, tek bir örnek satır için softmax'ın üç aşamasını gösterir.



Şekil 9.6: Softmax akışı tek bir örnek satır için (W'nin bir satırı, tohumlu): ham **logits** (negatif/pozitif) $\xrightarrow{\text{exp}}$ **counts** $= e^z$ (hepsi pozitif, sahte sayım) $\xrightarrow{\text{normalize}}$ **olasılık** $P = e^z / \sum_j e^{z_j}$ (toplamı 1). İki adım birlikte softmax'tır.

> Builder Notu — Softmax = exp + normalize
>
> **Geriye (18.06 + Calculus + Ders 1):** `xenc @ W`, bir one-hot (baz vektörü) ile matris çarpımı olduğundan `W`'nin ilgili **satırını seçer** (18.06). `exp` ise Calculus Ders 5'in e^x'i. Softmax = exp + normalize, Ders 1'deki sigmoid'in çok-sınıflı genellemesidir.
> **İleriye:** Softmax, her çok-sınıflı sınıflandırıcının ve her dil modelinin son katmanıdır. GPT de son katmanda sözlük-boyutu kadar logit üretip softmax'tan geçirir. "Logit = log-counts" sezgisi tüm seride geçerli kalır.

9.13 Forward / Backward / Update

Şimdi modeli **eğitelim**. Loss = doğru hedeflere atanan olasılıkların ortalama negatif log'u. Vektörel olarak: her örnek için `probs[satır, ys]` ile doğru karakterin olasılığını seç, log'unu al, ortalamasının negatifini hesapla:

```
loss = -probs[torch.arange(len(xs)), ys].log().mean()   # ortalama NLL
```

Sonra Ders 1'deki micrograd döngüsünün **birebir aynısı**: gradyanları sıfırla, geri yayılım, gradyanın tersine adım.

```
W.grad = None        # zero_grad (Ders 1: gradyanlar birikir!)
loss.backward()      # autograd -> W.grad doldurulur
W.data += -50 * W.grad   # gradient descent adimi
```

Karpathy neden rastgele deneme değil de gradient descent kullandığımızı net söyler:

> *"What I'm doing here, which is just guess and check of randomly assigning parameters and seeing if the network is good — that is amateur hour. That's not how you optimize a neural net."* — Karpathy, 1:32:29

Ve eğitim döngüsünün Ders 1 ile aynı olduğunu göstermek için micrograd notebook'unu açar:

> *"Identical to what we had with micrograd. So here I pulled up the lecture from micrograd… we had something very very similar."* — Karpathy, 1:33:14

Builder Notu — Dört Adım, Tensör Üzerinde

Geriye (Ders 1): `loss.backward()` + `W.data += -lr * W.grad`, micrograd'daki `backward()` + güncelleme döngüsünün tensör versiyonu. `W.grad = None` ise Ders 1'in **zero_grad** dersi (gradyanlar birikir, sıfırlamazsan bozulur). Tek fark: PyTorch tüm gradyanları tensör üzerinde otomatik hesaplar. **İleriye:** Bu dört adım (forward → loss → backward → update) her PyTorch eğitiminin çekirdeği. Öğrenme oranı (≈50) burada elle ayarlanır; Ders 4'te bunun nasıl prensipli seçileceğini (init + LR taraması), Ders 10'da production schedule'larını göreceğiz.

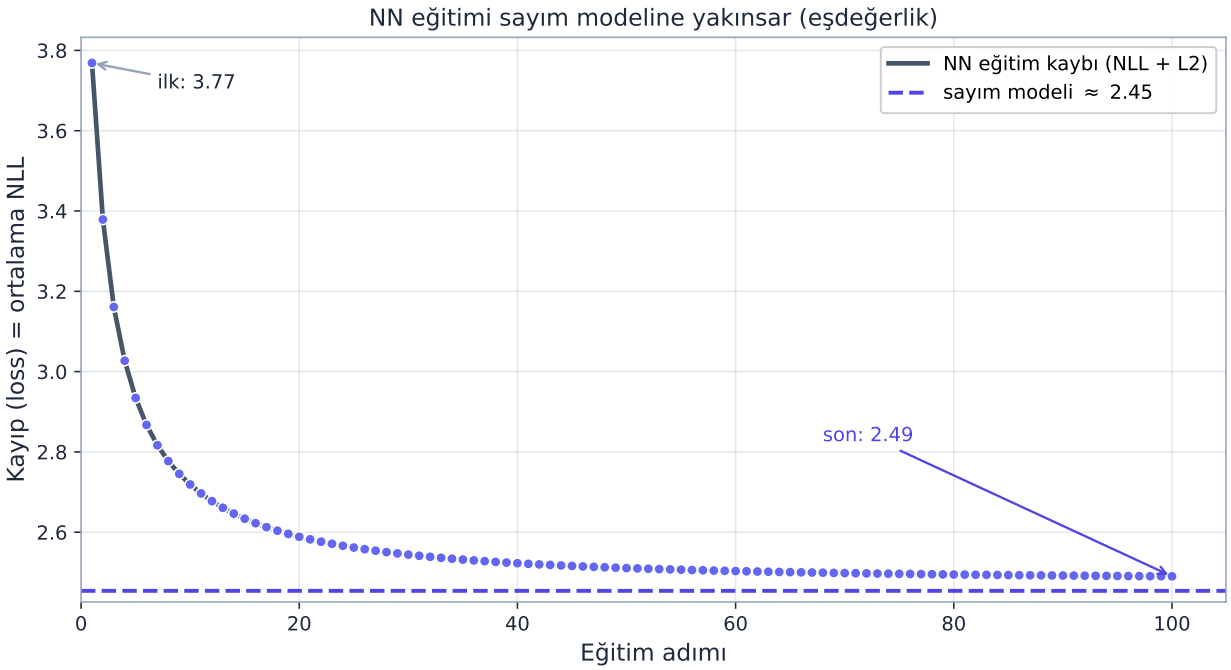
9.14 Tam Veri Seti ve Eşdeğerlik

Modeli tüm veri setine ölçeklendiririz: 228 binden fazla bigram örneği. Öğrenme oranını yükseltip (≈50) yeterince adım koşunca eğitim eğrisindeki loss (L2 reg dahil) **≈2,49**'a iner; sayım modeliyle adil kıyas için reg'siz ölçülen saf NLL **≈2,47**, sayım modelinin **≈2,45**'ine çok yakındır.

Kritik gözlem: bu, **saf sayım yöntemiyle aldığımız ≈2,45 ile neredeyse aynı**. Yani iki yöntem — sayım+normalize ve gradient descent — **aynı modele** ulaşır. Sayım yöntemi bigram için doğrudan optimal çözümü verir; gradient descent ise onu yinelemeli olarak bulur. Eğitim eğrisinde okunan ≈2,49 değeri `0.01 * (W**2).mean()` L2 reg cezasını da içerir; sayım modelinin reg'i olmadığından adil kıyas, ağın reg'siz saf NLL'idir (≈2,47) — bu da sayımın ≈2,45'ine 0,02 farkla yakındır. Fark: gradient descent her ölçeğe (çok daha karmaşık modellere) genişler, sayım yöntemi genişlemez.

Builder Notu — İki Yöntem, Tek Optimum

İleriye: "İki farklı yöntem aynı sonuca ulaşıyor" — bu, gradient descent'in doğruluğuna güven verir. Bigram'da sayımla doğrulayabildik; ama gerçek modellerde (GPT) kapalı-form çözüm YOKTUR,



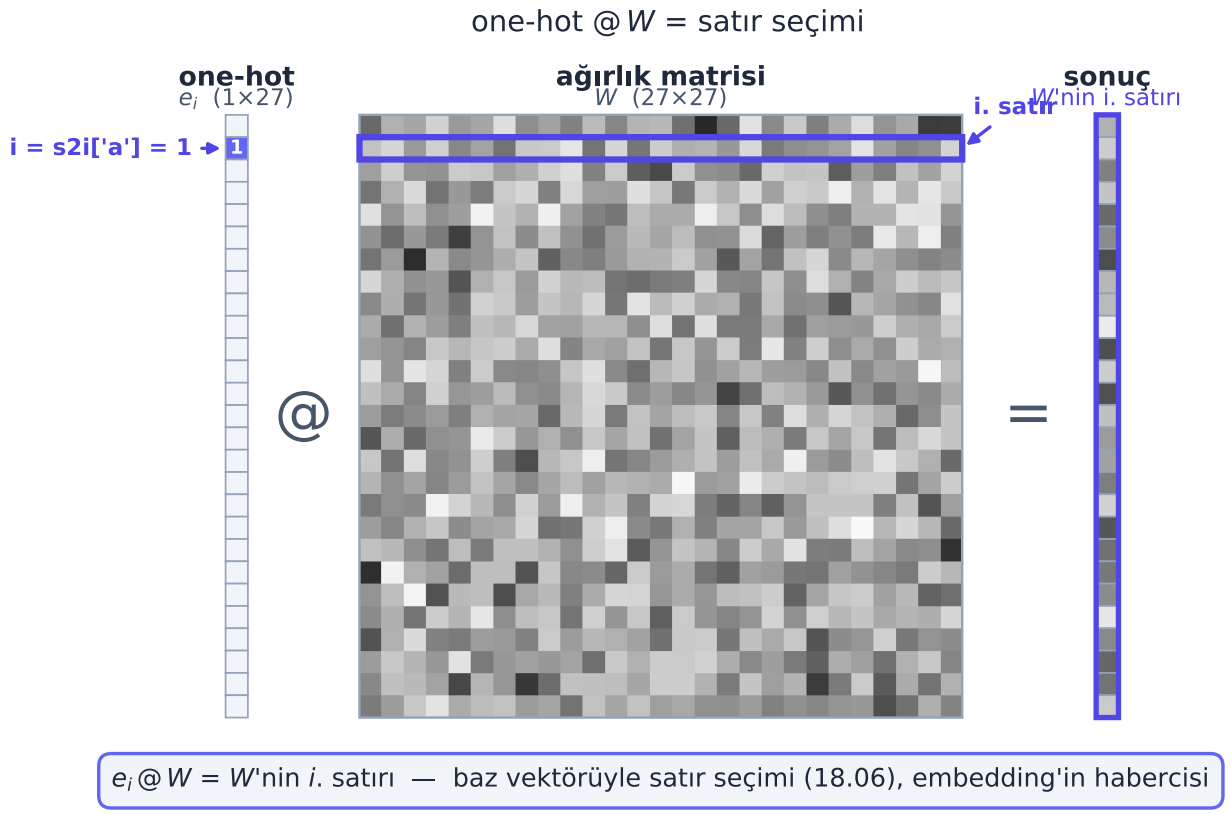
Şekil 9.7: Tek katmanlı sinir ağının (`train_nn`, 100 adım, lr=50, seed sabit) ortalama NLL kaybı: ilk adım $\approx$ 3,77, son adım $\approx$ 2,49. Slate çizgi + indigo işaretçiler. Kesikli indigo yatay çizgi, sayım modelinin tüm veri üzerindeki NLL'i ($\approx$ 2,45); gradient descent ile eğitilen ağ bu değere **yakınsar** — yani sayım yöntemiyle gradient descent **aynı modele** ulaşır (eşdeğerlik). x = eğitim adımı, y = kayıp (loss).

yalnızca gradient descent kalır. Bu ders, o güveni küçük, doğrulanabilir bir örnekte inşa eder.

9.15 İki Not: One-Hot = Satır Seçimi, L2 = Yumuşatma

Karpathy iki güzel bağlantıyla kapatır:

One-hot @ W = satır seçimi. Bir one-hot vektörünü `W` ile çarpmak, aslında `W`'nin tam o indeksteki **satırını çekip almaktır** (çarpma masrafı boşa). Yani `W`'nin satırları, sayım tablosundaki log-sayımların öğrenilmiş hâlidir. Bu gözlem, Ders 3'teki **embedding tablosunun** doğrudan habercisi: id ile satır seçimi, matris çarpımı olmadan.



Şekil 9.8: One-hot @ W = satır seçimi: 'a' karakterinin one-hot vektörü baz vektörü e_i'dir ($i = $ `s2i['a']` $= 1$; yalnız 1. hücre dolu, gerisi 0). Bunu 27×27 ağırlık matrisi W ile çarpmak, W'nin tam i. **satırını** çekip alır — 27 çarpma-toplama boşa, sonuç sadece o satır. Bu, 18.06'daki baz vektörüyle satır seçiminin ta kendisi ve Ders 3'teki embedding tablosunun habercisidir.

L2 düzenlileştirme = yumuşatma. Loss'a `(W**2).mean() * strength` eklemek, `W`'yi sıfıra doğru iter; $W = 0$ ise tüm logitler eşit, yani **uniform** dağılım. Bu, §8'deki sahte sayı eklemenin (smoothing) birebir sinir ağı karşılığıdır — ne kadar güçlü, o kadar düz model.

```
loss = -probs[torch.arange(len(xs)), ys].log().mean() + 0.01 * (W**2).mean()
```

Ağdan örneklenen isimler, sayım modelinden örneklenenlerle **aynı** çıkar — çünkü aynı model. Karpathy sonra bir sonraki adımı haber verir:

> *"We can expand this approach and complexify the neural net. So currently we're just taking a single character and feeding it into a neural net… but we're about to iterate on this substantially."*
> — Karpathy, 1:46:09

Builder Notu — İki İleri Kavram, Aslında Tanıdık

Geriye (Stat 110 + 18.06): L2 regularization = Laplace smoothing = uniform prior eklemek (Stat 110). One-hot @ W = baz vektörüyle satır seçimi (18.06). İki "ileri seviye" kavram, aslında daha önce gördüğün basit şeyler.
İleriye: "Karmaşıklaştır" sözü serinin yol haritası: tek karakter → çok karakter bağlam (Ders 3 MLP), sonra dikkat mekanizması (Ders 7 transformer). Her adımda model büyür ama loss/eğitim çerçevesi aynı kalır.

9.16 Bu Dersin Özeti

1. **makemore**, karakter-düzeyli bir dil modelidir: `names.txt`'ten öğrenip yeni isimler üretir. **Bigram** modeli sonraki karakteri yalnızca önceki tek karakterden tahmin eder.
2. Bigram'ları sayıp (27×27 tensör `N`), satır-satır normalize ederek bir **olasılık matrisi** elde ederiz.
3. **Örnekleme**: bir satırı olasılığa çevir, `torch.multinomial` (tohumlu `Generator`) ile sonraki karakteri çek, `.` gelene dek tekrarla.
4. **Broadcasting** kurallarına saygı (özellikle `keepdim=True`) — sessiz satır/kolon hatasından kaçın.
5. **Loss = ortalama NLL** $= -\frac{1}{n}\sum_i \log P(x_i)$; minimize etmek = maximum likelihood (Stat 110). Bigram için ≈2,45.
6. **Yumuşatma** (sahte sayı +1) sıfır olasılığı önler ($\log(0) = -\infty$ patlamasını).
7. **Aynı model = tek katmanlı sinir ağı**: one-hot girdi → `xenc @ W` (logit/log-counts) → exp → normalize (**softmax**) → olasılık.
8. **Eğitim** = Ders 1 micrograd döngüsü: forward → NLL → `loss.backward()` → `W.data += -lr * W.grad` (zero_grad'ı unutma).
9. Gradient descent ≈2,47'ye (reg'siz saf NLL; eğitim eğrisi reg dahil ≈2,49) yakınsar — sayımın ≈2,45'iyle **aynı model**. One-hot @ W = satır seçimi (embedding habercisi); L2 reg = yumuşatma.

Tek Bir Cümle

Bir dil modeli, "bir sonraki karakterin olasılık dağılımını" üreten bir fonksiyondur; bigram bunu tek karakter bağlamıyla yapar ve ister say-normalize et ister one-hot + softmax + gradient descent ile eğit — aynı modele ulaşırsın, ama yalnızca ikincisi GPT ölçeğine genişler.

Satırı normalize ederiz:

$$P(a \mid .) = \frac{N[., a]}{\sum_j N[., j]} = \frac{4000}{32000} = 0{,}125$$

NLL katkısı bu olasılığın negatif log'udur:

9.17 Kontrol Soruları

$$-\log P(a \mid .) = -\log(0{,}125) \approx 2{,}08$$

Cevap: $P(a \mid .) = 0{,}125$ (yani %12,5 olasılıkla isimler 'a' ile başlar). $-\log(0{,}125) \approx 2{,}08$. Model bu karaktere ne kadar yüksek olasılık atarsa, $-\log$ o kadar küçük (loss o kadar iyi) olur; olasılık 1'e giderse katkı 0'a, 0'a giderse $+\infty$'a gider.

i Soru 2: Sayım ≈2,45, gradient descent (reg'siz) ≈2,47 loss veriyor — neredeyse aynı. Bu neden beklenir? Neden yine de gradient descent öğreniriz?

Cevap: İkisi de **aynı amacı** optimize eder: bigram olasılıklarını veriye en iyi uyduran değerler. Sayım+normalize, bigram için bu optimumun **kapalı-form (doğrudan) çözümüdür**; gradient descent ise aynı optimuma yinelemeli yaklaşır — bu yüzden neredeyse aynı loss'a varırlar (sayım ≈2,45, ağın reg'siz saf NLL'i ≈2,47, fark 0,02). Bir incelik: ağın eğitim eğrisinde okunan değer (≈2,49) `0.01 * (W**2).mean()` L2 reg cezasını da içerir; sayım modelinin reg'i olmadığından adil kıyas, ağın reg'siz saf NLL'idir (≈2,47). Yani ağın biraz daha yüksek görünmesinin bir kısmı yinelemeli yaklaşım, bir kısmı da reg terimidir. Gradient descent'i öğrenmemizin sebebi: bigram'dan daha karmaşık modellerde (MLP, transformer) **kapalı-form çözüm yoktur** — sayamazsın. Gradient descent her ölçeğe genişleyen tek yöntemdir. Bigram, gradient descent'i sayımla doğrulayabildiğimiz son basit duraktır.

i Soru 3: P = N.float() / N.sum(1, keepdim=True) ile P = N.float() / N.sum(1) farkı nedir? Hangisi satırları doğru normalize eder, neden?

Cevap: `N.sum(1)` her satırın toplamını verir ama şekli `(27,)` — broadcasting'de bir **satır vektörü** gibi hizalanır ve $(27, 27)$ matrisin her **kolonunu** bu vektöre böler (YANLIŞ — kolon normalizasyonu). `N.sum(1, keepdim=True)` şekli `(27, 1)` tutar; bu, her **satır** boyunca yayılır ve her satırı kendi toplamına böler (DOĞRU — satır normalizasyonu, her satır toplamı 1 olur). Tehlikeli olan: ikisi de hata fırlatmaz, kod çalışır — ama `keepdim=False` sessizce yanlış eksende normalize eder (satır toplamları $0{,}0669$–$7{,}0225$ arası savrulur). Kural: şekilleri `.shape` ile kontrol et.

i Soru 4: (Builder) xenc @ W işleminde xenc bir one-hot vektörse, sonuç neden W'nin tam bir satırına eşittir? 18.06 ile bağla.

Cevap: One-hot vektör, standart **baz vektörüdür** e_i — yalnızca i'inci bileşeni 1, gerisi 0. Bir baz vektörü e_i ile matris çarpımı $(e_i \cdot W)$, W'nin i**'inci satırını** seçip döndürür (18.06 Ders 30: matris-

vektör çarpımı, satırların/kolonların seçilmesi). Yani 27 çarpma-toplama yapıyor gibi görünse de, aslında `W`'nin bir satırını "çekip alıyoruz". Bu yüzden `W`'nin her satırı, o girdi karakteri için 27 log-count'tur — tıpkı sayım tablosunun bir satırı gibi. Bu gözlem, Ders 3'teki **embedding tablosunun** temelidir: id ile doğrudan satır seçimi (one-hot + matris çarpımı israfı olmadan).

9.18 Egzersizler

Egzersiz 1 (Sayım modelini kur). `names.txt`'i indir (github.com/karpathy/makemore), bigram sayım tensörü `N`'i (27×27) doldur, olasılık matrisi `P`'yi (satır normalize, `keepdim=True`) hesapla. Tohumlu `torch.multinomial` ile 10 isim örnekle. Üretilen isimleri gözlemle.

Egzersiz 2 (NLL'i hesapla). Sayım modelinin tüm veri seti üzerindeki **ortalama negatif log olabilirliğini** hesapla (her bigram için $-\log P(\text{ch2} \mid \text{ch1})$, ortalamasını al). ≈2,45 çıkmalı. Sonra +1 yumuşatma ekleyip nasıl değiştiğine bak.

Egzersiz 3 (keepdim bug'ını gör). `P = N / N.sum(1, keepdim=True)` ile `P = N / N.sum(1)` çıktılarını karşılaştır. İkincide her **satırın** toplamının 1 olmadığını (yanlış normalizasyon) `P.sum(1)` ile doğrula. Şekilleri (`N.sum(1).shape` vs `N.sum(1, keepdim=True).shape`) yazdır.

Egzersiz 4 (Sinir ağı = sayım, doğrula). `W = torch.randn((27,27), requires_grad=True)` ile sinir ağı versiyonunu kur (one-hot → `xenc @ W` → exp → normalize → NLL). Gradient descent ile eğit (lr≈50); reg'siz saf NLL'in ≈2,47'ye — yani sayım modelinin ≈2,45'ine — indiğini gözlemle. `0.01 * (W**2).mean()` regularization ekleyip etkisine bak: eğitim eğrisinde okunan değer reg cezası dahil ≈2,49 olur.

Egzersiz 5 (Sonraki dersin habercisi). Bigram yalnızca **tek** karaktere bakıyor. Bağlamı 3 karaktere çıkarmak istesen, sayım tablosu kaç hücre olur? (a) $27^3 = ?$ hesapla; 10 karakter bağlam için 27^{10}'un neden imkânsız olduğunu açıkla. (b) Bir sinir ağı (her karakteri küçük bir vektöre — embedding — gömüp birleştiren) bu patlamayı nasıl önler? Bu iki gözlem, Ders 3'te (**makemore 2: MLP, Bengio 2003**) sayım yerine neden öğrenilen embedding + gizli katmana geçtiğimizi motive eder.

9.19 Sonraki Ders İçin Hazırlık

Ders 3: makemore 2 — MLP (Bengio 2003) — Andrej Karpathy

Bigram tek karakter bağlamıyla sınırlıydı ve sayım tablosu bağlamla üstel büyüyor. Ders 3'te 2003 tarihli ünlü Bengio makalesini izleyip bir **çok katmanlı algılayıcı (MLP)** kuracağız: birkaç karakterlik bağlamı **embedding tablosuyla** küçük vektörlere gömüp, bir gizli `tanh` katmanından geçirip `F.cross_entropy` ile eğiteceğiz. Bu, bigram'ın sayım-patlamasını öğrenilen parametrelerle aşar.

Ana konular:

- Embedding arama tablosu C (bu dersin "one-hot @ W = satır seçimi" gözlemi somutlaşır).
- Bağlam penceresi (`block_size`), gizli `tanh` katmanı, `F.cross_entropy`.
- Minibatch SGD, öğrenme oranı taraması, train/dev/test bölmesi.

⚠ Ders 3 Öncesi Yapılacak

- Egzersizleri çöz — özellikle 4 (NN = sayım doğrulaması) ve 5 (bağlam patlaması sezgisi).
- "One-hot @ W = W'nin bir satırını seçer" gözlemini kendi cümlenle yaz (Ders 3'te embedding tam budur).
- Ana cümleyi tekrar oku: *"Bir dil modeli, bir sonraki karakterin olasılık dağılımını üreten bir fonksiyondur."*

9.20 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Karpathy'de
makemore / bigram	Karakter-düzeyli dil modeli; sonraki karakteri önceki tek karakterden tahmin eder	0m06
'.' token	Başlangıç ve bitişi işaretleyen tek özel token (indeks 0)	18m19
s2i / i2s	Karakter ile tamsayı id arasındaki arama tabloları (tokenizer çekirdeği)	12m45
Sayım tensörü N	27×27 tamsayı tensör; N[i,j] = i'den sonra j'nin geçiş sayısı	12m45
Normalize + keepdim	P = N / N.sum(1, keepdim=True); satır normalizasyonu, broadcasting tuzağı	36m17
torch.multinomial	Olasılık dağılımından örnek çekme; tohumlu Generator ile tekrarlanabilir	26m29
Negatif log olabilirlik	Loss $= -\frac{1}{n}\sum_i \log P(x_i)$; minimize = maximum likelihood; bigram ≈2,45	50m09
Yumuşatma (smoothing)	Sahte sayı (+1) ekleyip sıfır olasılığı ($\log(0) = -\infty$) önleme	1h00m
One-hot kodlama	Tamsayı id'yi tek-1'li sıfır vektörüne çevirme (F.one_hot); baz vektörü	1h05m
logits / softmax	xenc@W ham çıktı = log-counts; exp + normalize = olasılık	1h20m

Kavram	Tanım	Karpathy'de
Eğitim döngüsü	forward → NLL → loss.backward() → W.data += -lr·W.grad (zero_grad)	1h35m
Count = gradient eşdeğerliği	Sayım (≈2,45) ile gradient descent (reg'siz ≈2,47; eğitim eğrisi ≈2,49) aynı modele varır	1h42m
L2 reg = smoothing	(W^2).mean() cezası W'yi 0'a iter (uniform); sahte sayımın NN karşılığı	1h47m

9.21 ML Builder Bağlantıları

9 köprü

1. **NLL / cross-entropy** → Stat 110 maximum likelihood + Ders 1 cross-entropy (Bernoulli'nin multinomial genellemesi). İleriye: GPT'nin eğitim kaybı.
2. **softmax (exp + normalize)** → Calculus e^x (Ders 5) + Stat 110 multinomial. İleriye: her sınıflandırıcının/dil modelinin son katmanı.
3. **one-hot @ W = satır seçimi** → 18.06 baz vektörü/matris-vektör çarpımı (Ders 30). İleriye: **Ders 3 embedding tablosu**.
4. **forward/backward/update + zero_grad** → Ders 1 micrograd döngüsünün tensör hâli. İleriye: tüm PyTorch eğitimi.
5. **torch.multinomial (tohumlu)** → Stat 110 kategorik örnekleme + reproducibility. İleriye: GPT sampling (temperature, top-k).
6. **broadcasting / keepdim** → tensör mekaniği. İleriye: sessiz şekil hatalarından kaçınma (production debug).
7. **L2 regularization = smoothing** → Stat 110 Laplace/uniform prior. İleriye: weight decay (Ders 4, AdamW).
8. **sayım = gradient eşdeğerliği** → bigram'da kapalı-form var; GPT'de YOK, yalnızca gradient descent kalır.
9. **"sonraki token'ı tahmin et" çerçevesi** → tüm dil modellemenin çekirdeği; bigram → MLP → transformer hep aynı amaç.

9.22 Karpathy'nin Önerdiği Kaynaklar

Karpathy'nin bu ders için verdiği kaynaklar:

- **makemore repo:** github.com/karpathy/makemore — `names.txt` ve tam proje.
- **Ders notebook'u:** makemore_part1_bigrams.ipynb — dersin adım adım kodu.

- **Ders Colab notebook'u:** Google Colab — çalıştırılabilir ortam.

! Tek bir şey alıp gideceksen

Bir dil modeli, "bir sonraki karakterin olasılık dağılımını" üreten bir fonksiyondur. Bigram bunu en sade hâliyle kurar — ve ister say-normalize et ister one-hot + softmax + gradient descent ile eğit, aynı modele ulaşırsın. Fark şu: yalnızca gradient descent yaklaşımı, bağlamı büyüttükçe (Ders 3 MLP, Ders 7 transformer) ölçeklenir. *"That is amateur hour"* — Karpathy, rastgele değil, gradient'le optimize et.

10 makemore 2 — MLP (Bengio 2003)

Bağlamı say-tablosunda tutmak üstel patlar; her karakteri öğrenilen bir vektöre gömüp (embedding) bir MLP'den geçir — aynı 'sonraki karakteri tahmin et' problemi, bu kez ölçeklenebilir

> **Bölüm bilgisi**
>
> - **Karpathy'nin videosu:** YouTube — Building makemore Part 2: MLP (≈76 dk)
> - **Seri:** Neural Networks: Zero to Hero — Ders 3
> - **Hoca:** Andrej Karpathy
> - **Kaynak repo:** github.com/karpathy/makemore
> - **Okuma süresi:** ≈30 dk

10.1 Bu Derste Ne Var?

Ders 2'de bigram modelini kurduk — ama yalnızca **tek** karaktere bakıyordu ve bağlamı büyütmek istediğimizde sayım tablosu üstel patlıyordu (27^3, 27^4, …). Bu derste çözümü kuruyoruz: 2003 tarihli ünlü **Bengio ve ark.** makalesini izleyerek bir **çok katmanlı algılayıcı (MLP)** dil modeli.

> *"In the last lecture we implemented the bigram language model, and we implemented it both using counts and also using a super simple neural network that had a single linear layer."* — Karpathy, 0:00

Büyük fikir: karakterleri sayım tablosunda tutmak yerine, her karakteri küçük bir **vektöre** (embedding) gömeriz; birkaç karakterlik bağlamı bu vektörlerle bir **gizli tanh katmanından** geçirip bir sonraki karakteri tahmin ederiz. Sayım patlaması yok — çünkü artık sayıyor değil, **öğreniyoruz** (gradient descent).

Dersin üç büyük fikri:

1. **Embedding arama tablosu C** — Ders 2'deki "one-hot @ W = satır seçimi" gözlemi burada somutlaşır: her karakter id'si, C tablosunun bir satırı (küçük bir vektör).
2. **MLP (Bengio 2003)** — bağlam vektörlerini birleştir → gizli `tanh` katmanı → çıktı logitleri → softmax. Ders 1'in Neuron/Layer/MLP'sinin gerçek bir dil modeli hâli.
3. **Eğitim pratikleri** — `F.cross_entropy`, minibatch SGD, öğrenme oranı taraması, train/dev/test bölmesi: ilk kez "düzgün" bir eğitim hattı.

Şekil 10.1: Ders 3'ün kavram haritası: sayım patlaması yerine öğrenilen embedding + MLP. Bağlam patlamasından (üstel say-tablosu) yola çıkıp, embedding tablosu C ile karakterleri vektöre gömer, MLP'den (birleştir → tanh → logit) geçirir, cross-entropy ile eğitir ve örnekler.

Builder Notu — Embedding, Ders 2'nin Ödülü

Geriye (Ders 1-2 + Stat 110):

- **Embedding = Ders 2'nin ödülü.** Ders 2'de "one-hot @ W, W'nin bir satırını seçer" demiştik; embedding tablosu C tam olarak budur — one-hot çarpımı israfı olmadan, id ile doğrudan satır seçimi.
- **MLP = Ders 1.** Gizli `tanh` katmanı + çıktı katmanı, micrograd'da kurduğumuz Neuron/Layer/MLP'nin tensör hâli; `tanh` türevi ($1 - \tanh^2$) aynı.
- **Cross-entropy = Ders 2 NLL.** `F.cross_entropy`, Ders 2'de elle yazdığımız NLL'in sayısal-kararlı ve hızlı PyTorch hâli (Stat 110 MLE).
- **Minibatch / overfitting = Ders 1.** Ders 1 §9 mini-batch SGD (varyans $\propto 1/B$) ve §10 overfitting/regularization burada gerçek veride uygulanır.

İleriye: Bu MLP, serinin geri kalanının iskeleti. Ders 4'te tam **bu ağın** aktivasyon/gradyan istatistiklerini düzeltip BatchNorm ekleyeceğiz; Ders 5'te aynı ağın backward'ını elle yazacağız. Embedding fikri ise doğrudan GPT'nin token embedding tablosuna gider.
Tek cümleyle: Bağlamı sayım tablosunda tutmak üstel patlar; bunun yerine her karakteri öğrenilen bir vektöre gömüp (embedding) bir MLP'den geçirmek, aynı "sonraki karakteri tahmin et" problemini ölçeklenebilir kılar.

10.2 Bigram'ın Sınırı ve Bağlam Patlaması

Ders 2'nin bigram modeli yalnızca **bir önceki** karaktere bakardı. Daha iyi tahmin için daha çok bağlam gerek — örneğin son **3** karaktere bakmak. Ama sayım yaklaşımıyla bu felaket: 3 karakter bağlamı için tablo $27 \times 27 \times 27 = 19\,683$ satır; 10 karakter için 27^{10} (≈ 200 trilyon) — imkânsız. Her ek karakter, tabloyu 27 kat büyütür ve çoğu hücre hiç görülmez (sıfır sayım).

Çözüm: sayım yerine **öğrenme**. Karakterleri ayrık tablo hücreleri olarak değil, sürekli bir uzayda **vektörler** olarak temsil edersek, model benzer karakterleri (örn. sesli harfler) birbirine yakın yerleştirip genelleyebilir. İşte Bengio 2003'ün fikri.

Builder Notu — Ayrık Tablo → Sürekli Uzay

Geriye (Ders 2): Bu, Ders 2'nin son egzersizinin (27^n patlaması) cevabı. Sayım, parametreyi veriye "ezberletir"; öğrenilen embedding ise genelleme yapar.
İleriye: "Ayrık tablo → sürekli vektör uzayı" geçişi, tüm modern NLP'nin temelidir (word2vec, GPT embedding'leri). Bağlamı büyütmenin maliyeti artık üstel değil, doğrusal/parametrik.

10.3 Bengio 2003 Makalesi

Karpathy makaleyi ekranda açar. Bu, sinir ağlarıyla dil modellemeyi öneren ilk makale değil ama en etkililerinden:

> *"So I have the paper pulled up here. Now this isn't the very first paper that proposed the use of MLPs or neural networks to predict the next character or token in a sequence, but it's definitely one that was very influential around that time."* — Karpathy, 2:01

Makalenin mimarisi: her kelime (Bengio'da kelime, bizde karakter) bir **arama tablosundan** (lookup table, C) küçük bir vektöre çevrilir. Bağlamdaki tüm vektörler birleştirilip bir **gizli katmana**, oradan da tüm sözlük üzerinde bir **softmax çıktısına** gider. Karpathy bu vektörlerin (embedding) eğitim sırasında backprop ile öğrenildiğini vurgular: benzer anlamlı öğeler uzayda birbirine yaklaşır.

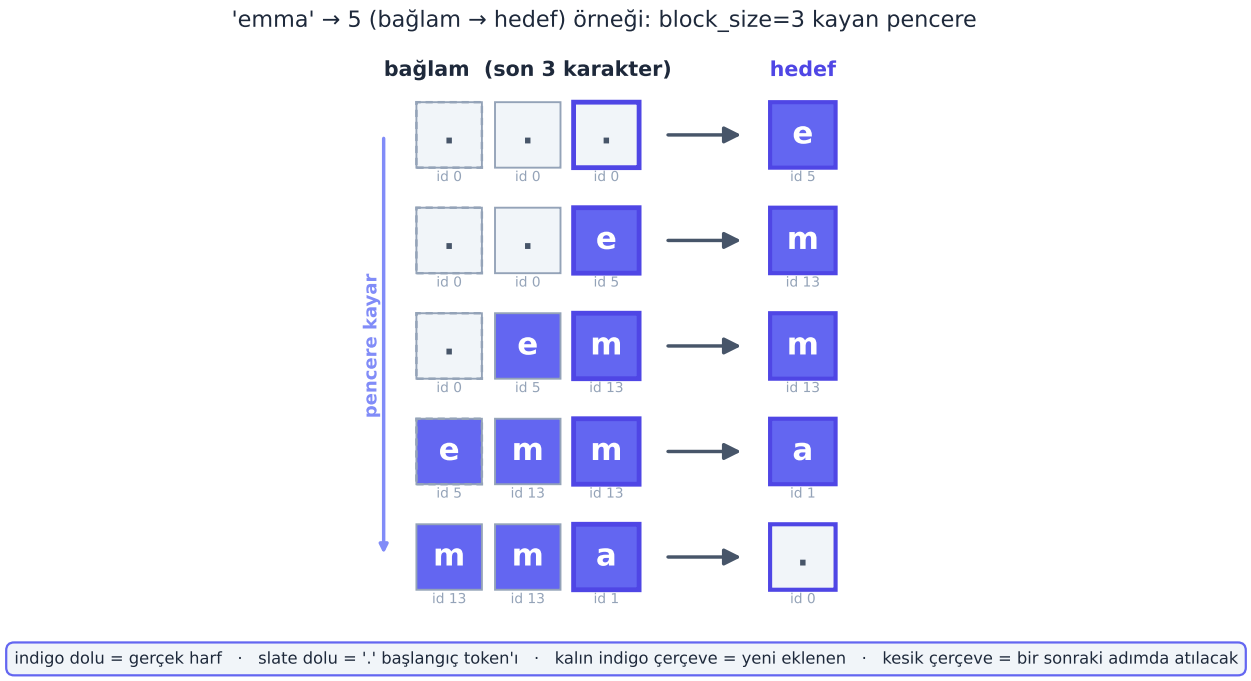
> Builder Notu — Bengio = Transformer'ın İskeleti
>
> **İleriye:** Bengio 2003 mimarisi — embedding + gizli katman + softmax — 20 yıl sonra transformer'ın da iskeletidir; transformer yalnızca "bağlamı birleştirme" kısmını (concatenation) **dikkat mekanizmasıyla** (Ders 7) değiştirir. Temel fikir aynı kalır.

10.4 Eğitim Veri Setini Kurmak (block_size)

Önce veriyi MLP'ye uygun hâle getiririz. **block_size**, kaç önceki karaktere bakacağımız (bağlam uzunluğu). Karpathy 3 ile başlar: "3 karakter gör, 4.'yü tahmin et." Her isim, kayan bir pencereyle birçok (bağlam → hedef) örneğine bölünür.

```
block_size = 3   # kac karaktere bakarak tahmin ediyoruz

def build_dataset(words):
    X, Y = [], []
    for w in words:
        context = [0] * block_size        # '...' ile basla (hepsi nokta)
        for ch in w + '.':
            ix = s2i[ch]
            X.append(context)             # girdi: son 3 karakter id
            Y.append(ix)                  # hedef: sonraki karakter
            context = context[1:] + [ix]  # pencereyi kaydir
    return torch.tensor(X), torch.tensor(Y)

X, Y = build_dataset(words)   # X: (n, 3), Y: (n,)
```

`context = context[1:] + [ix]` satırı pencereyi kaydırır: en eski karakteri atar, yeni karakteri ekler. Örneğin "emma" için: `... -> e`, `..e -> m`, `.em -> m`, `emm -> a`, `mma -> .` — yani **5 örnek**. `X`'in her satırı 3 tamsayı id, `Y` ise hedef id. Tüm isimlere uygulandığında veri seti **228.146** örneğe ulaşır (X şekli $(228146, 3)$, Y şekli $(228146,)$).



Şekil 10.2: `block_size` kayan pencere: ‘emma’ kelimesi `build_dataset(block_size=3)` ile 5 (bağlam → hedef) örneğine bölünür. Her satır 3 hücreli bağlamı (`s2i`: `.`= 0, a= 1, e= 5, m= 13) ve hedef karakteri gösterir: `...`→e, `..e`→m, `.em`→m, `emm`→a, `mma`→. — yani 5 örnek. Pencere her adımda sağa kayar: en sağdaki **yeni** hücre (indigo) eklenir, en soldaki **atılan** karakter (soluk slate) düşer. `.` başlangıç token’ları slate-100 dolgulu, gerçek harfler indigo dolguludur.

Builder Notu — Kayan Pencere = Veri Yükleyicinin Çekirdeği

İleriye: Bu kayan bağlam penceresi, her dil modelinin veri yükleyicisinin çekirdeği. GPT'de `block_size` yüzlerce/binlerce token olur (Ders 7'de context length). Bağlamı sabit tutmak (fixed-context) MLP'nin sınırı; transformer bunu uzun bağlama taşır.

10.5 Embedding Arama Tablosu C

Şimdi Ders 2'nin ödülü. Her karakter id'sini küçük bir vektöre gömeriz. `C` bir matris: 27 satır (her karakter), her satır küçük bir vektör (örn. 2 boyutlu). Bir id'nin embedding'i = `C`'nin o satırı.

```
C = torch.randn((27, 2))   # 27 karakter, her biri 2-boyutlu vektor
emb = C[X]                 # (n, 3, 2): her ornek icin 3 karakterin embedding'i
```

`C[X]` PyTorch'un güzel bir özelliği: `X` bir tamsayı tensörü $(n, 3)$, `C[X]` her id'yi `C`'nin ilgili satırıyla değiştirir → sonuç $(n, 3, 2)$. Bu, Ders 2'deki "one-hot @ W = satır seçimi"nin doğrudan, verimli hâli — one-hot çarpımı yapmadan id ile indeksleme.

> *"There's also a lookup table that they call C. This lookup table is a matrix…"* — Karpathy, 6:17

Builder Notu — C[X] = Verimli Satır Seçimi

Geriye (Ders 2 + 18.06): `C[X]`, Ders 2'deki `xenc @ W`'nin (one-hot ile satır seçimi) verimli karşılığı — aynı işlem, çarpım masrafı yok (18.06: baz vektörüyle çarpım = satır seçimi). `C`'nin satırları eğitim sırasında **backprop ile öğrenilir**.
İleriye: Embedding tablosu, GPT'nin `wte` (word token embedding) tablosunun ta kendisi (Ders 10). Öğrenilen embedding uzayında benzer token'lar kümelenir — §13'te bunu gözle göreceğiz.

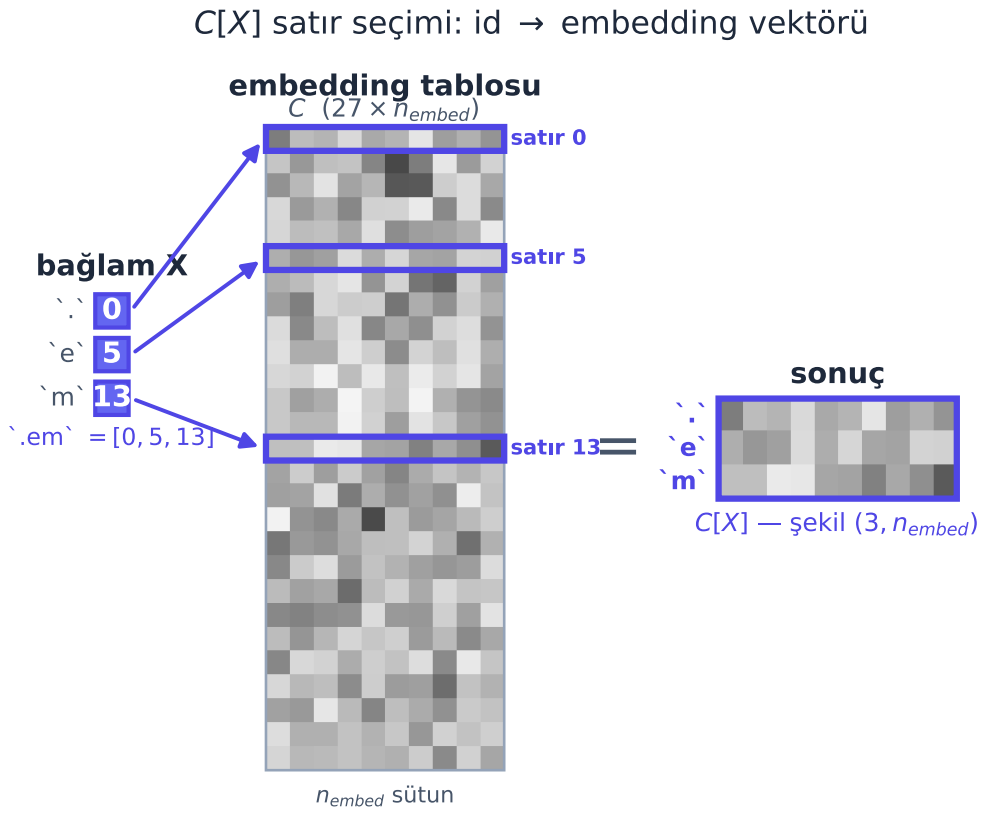
10.6 Gizli Katman ve tensör .view()

Embedding'leri $(n, 3, 2)$ gizli katmana vermeden önce **düzleştirmek** gerekir: 3 karakterin 2'şer boyutlu vektörünü tek bir 6-boyutlu vektörde birleştir → $(n, 6)$. Karpathy bunun için `.view()` kullanır ve tensör iç yapısına (storage/strides) kısa bir gezinti yapar.

```
W1 = torch.randn((6, 100))   # giris 6 (=3*2), gizli 100 noron
b1 = torch.randn(100)
h = torch.tanh(emb.view(-1, 6) @ W1 + b1)   # (n, 100), gizli aktivasyon
```

`emb.view(-1, 6)` embedding'i $(n, 3, 2) \to (n, 6)$ yeniden şekillendirir. `.view()` veriyi **kopyalamaz**, yalnızca aynı bellek üzerindeki "görünümü" değiştirir (storage + strides) — bu yüzden çok ucuzdur. Sonra `@ W1 + b1` (lineer katman) ve `tanh` (Ders 1'in aktivasyonu) ile gizli katman aktivasyonu `h` elde edilir.

> *"There are usually many ways of implementing what you'd like to do in torch, and some of them will be faster, better, shorter, etc."* — Karpathy, 19:54



Şekil 10.3: Embedding arama tablosu $C[X]$: bir bağlam örneğinin 3 tamsayı id'si (`.em` $= [0, 5, 13]$) doğrudan C'nin $(27 \times n_{embed})$ o satırlarını **indeksleyerek** çeker — sonuç $(3, n_{embed})$. Bu, Ders 2'deki one-hot $@\, W$ satır seçiminin verimli hâlidir (18.06 baz vektörü): one-hot vektörü kurup çarpmak yok, çarpım masrafı yok. Tüm batch için $C[X]$ şekli $(n, 3, n_{embed})$ olur.

Builder Notu — .view() = Bellek Düzeni Farkındalığı

Geriye (Ders 1 + 18.06): `emb.view(-1,6) @ W1 + b1`, Ders 1'in $Wx + b$ lineer katmanı (18.06 matris çarpımı); `tanh` ise Ders 1'in aktivasyonu (türevi $1 - \tanh^2$). `.view()`'in kopyalamadan çalışması, tensör bellek düzeni (contiguous storage + strides) bilgisidir.
İleriye: `.view()` / `.reshape()`, tüm tensör kodunda her gün kullanılır (batch boyutlarını ayarlamak, attention'da head'leri ayırmak — Ders 7/10). Bellek-düzeni farkındalığı (contiguous vs view) production performansında önemlidir.

10.7 Çıktı Katmanı ve NLL

Gizli katmandan (100 boyut) çıktıya: 27 karakterin her biri için bir logit üretmeliyiz. İkinci bir lineer katman:

```
W2 = torch.randn((100, 27))
b2 = torch.randn(27)
logits = h @ W2 + b2            # (n, 27)
```

Sonra Ders 2'deki softmax + NLL'in birebir aynısı: logitleri üstel al (counts), normalize et (probs), doğru hedeflerin olasılığının ortalama negatif log'unu al:

```
counts = logits.exp()
probs = counts / counts.sum(1, keepdim=True)   # softmax
loss = -probs[torch.arange(len(Y)), Y].log().mean()   # ortalama NLL
```

Bu, Ders 2'deki tek-katmanlı ağın kaybıyla aynı yapı — fark, arada bir gizli `tanh` katmanı olması (yani artık derin bir ağ). Aşağıdaki şema tüm forward akışını uçtan uca toplar.

Builder Notu — Çıktı + NLL, Ders 2'nin Tekrarı

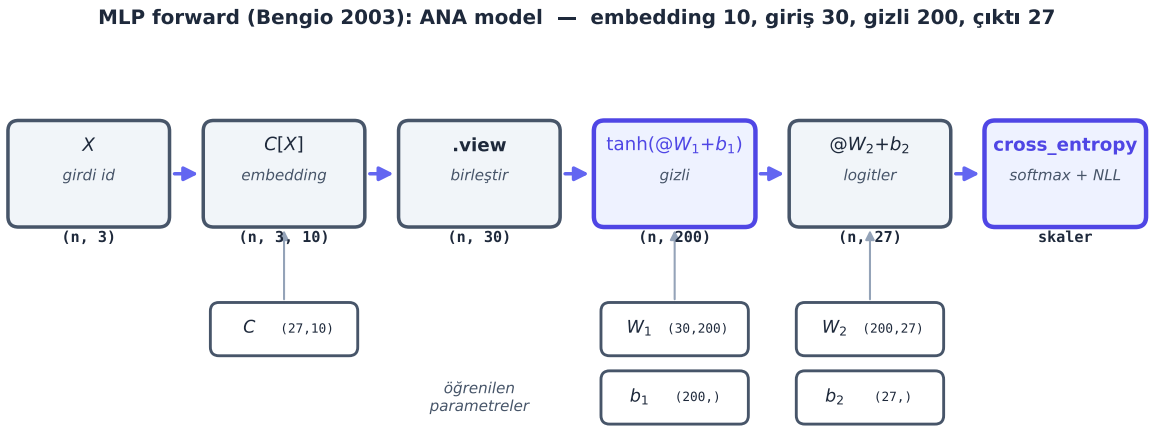
Geriye (Ders 1-2): Çıktı katmanı + softmax + NLL, Ders 2'nin birebir tekrarı; tek fark girdinin embedding + gizli katmandan gelmesi. `h @ W2 + b2` yine Ders 1'in lineer katmanı.

10.8 F.cross_entropy ve Neden Kullanılır

Elle yazdığımız `counts = logits.exp(); probs = ...; loss = -...log().mean()` zinciri çalışır ama PyTorch bunu tek bir fonksiyonda toplar: `F.cross_entropy`. Karpathy bunun **sadece kısalık için değil**, iki ciddi nedenle tercih edildiğini vurgular:

> *"That's why there is a functional.cross_entropy function in PyTorch to calculate this much more [efficiently]."* — Karpathy, 33:03

```
loss = F.cross_entropy(logits, Y)   # tek satir, daha hizli + kararli
```



Şekil 10.4: MLP forward şeması (Bengio 2003): $X \to C[X] \to$ view $\to$ tanh $\to$ logits $\to$ NLL. Soldan sağa veri akışı (indigo oklar); ANA modelin gerçek şekilleri her kutu altında: girdi X (n,3 id) $\to$ embedding $C[X]$ (n,3,10) $\to$ `.view` ile birleştir (n,30) $\to$ gizli tanh$(@W_1 + b_1)$ (n,200) $\to$ çıktı $@W_2 + b_2$ logitler (n,27) $\to$ `F.cross_entropy` ile NLL. Parametre kutuları (C, W_1, b_1, W_2, b_2) slate çerçeveyle ilgili katmana bağlanır; tanh ve cross-entropy düğümleri indigo vurgulu (Ders 1 köprüsü: aktivasyon + NLL).

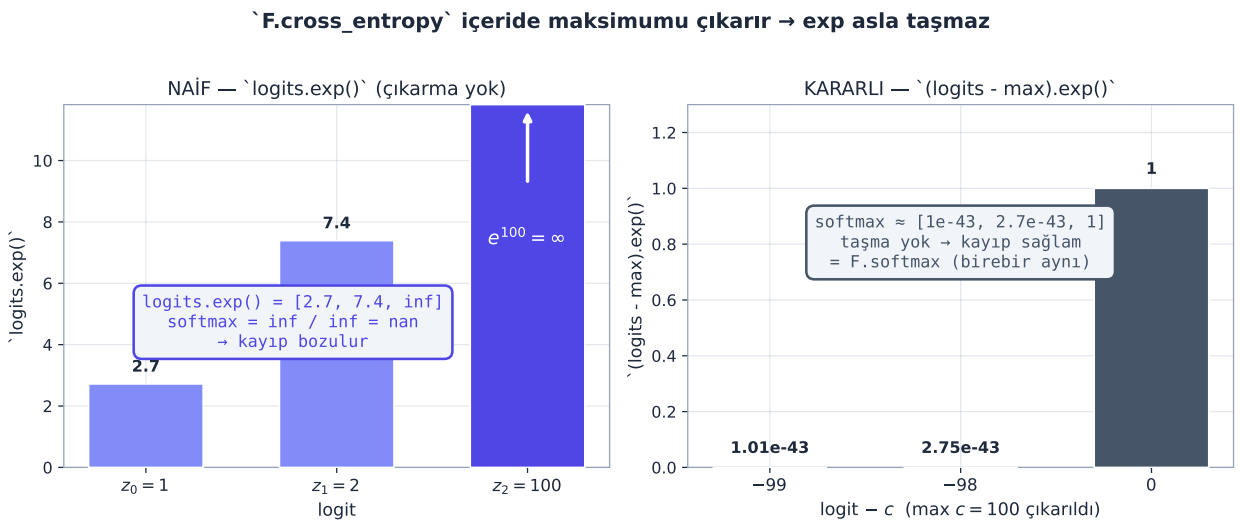
İki kazanç: **(1) Verimlilik** — PyTorch ara tensörleri (counts, probs) bellekte oluşturmaz, işlemleri tek bir füzyonlu çekirdekte yapar, backward'ı daha basit/hızlıdır. **(2) Sayısal kararlılık** — büyük bir logit (örn. 100) için `logits.exp()` taşar (inf $\to$ nan). `F.cross_entropy` içeride logitlerden **maksimumu çıkarır** (softmax bu kaydırmaya karşı değişmezdir), böylece üstel hiçbir zaman taşmaz.

> Builder Notu — Füzyon + Sayısal Kararlılık
>
> **Geriye (Ders 2 + Stat 110):** `F.cross_entropy`, Ders 2'de elle yazdığımız NLL'in ta kendisi (Stat 110 MLE), yalnızca sayısal-kararlı ve füzyonlu. "Softmax max çıkarmaya karşı değişmez" özelliği: $\text{softmax}(z) = \text{softmax}(z - c)$.
> **İleriye:** Bu "ara değerleri materyalize etme, füzyonlu kernel kullan" fikri, production'da **operator fusion**'ın (Ders 10: torch.compile, FlashAttention) çekirdeğidir. Sayısal kararlılık (max çıkarma) her ciddi softmax implementasyonunda vardır.

10.9 Eğitim Döngüsü ve Tek Batch'e Overfit

Şimdi eğitiriz — Ders 1-2'deki aynı döngü (forward $\to$ zero_grad $\to$ backward $\to$ update), ama artık birden çok parametre (`C, W1, b1, W2, b2`).

```
parameters = [C, W1, b1, W2, b2]
for p in parameters:
    p.requires_grad = True
```



Şekil 10.5: `F.cross_entropy` sayısal kararlılık: **sol (indigo uyarı):** büyük logitler $[1, 2, 100]$ için naif `logits.exp()` → en büyük üs taşar ($e^{100} = \infty$), ardından ∞/∞ = nan; softmax çöker. **Sağ (slate güvenli):** aynı logitlerden maksimum $c = 100$ çıkarılınca $[-99, -98, 0]$ olur; `exp()` artık $[\approx 0,\ \approx 0,\ 1]$ verir, taşma yok, softmax $\approx [0,\ 0,\ 1]$. Softmax kaydırmaya karşı değişmez olduğundan ($\text{softmax}(z) = \text{softmax}(z - c)$) sonuç matematiksel olarak aynıdır. `F.cross_entropy` bu çıkarmayı içeride yapar — bu yüzden elle yazılan zincirin aksine asla taşmaz.

```
for _ in range(200):
    # forward
    emb = C[X]
    h = torch.tanh(emb.view(-1, 6) @ W1 + b1)
    logits = h @ W2 + b2
    loss = F.cross_entropy(logits, Y)
    # backward
    for p in parameters:
        p.grad = None        # zero_grad
    loss.backward()
    # update
    for p in parameters:
        p.data += -0.1 * p.grad
```

Karpathy önce bir **akıl-sağlığı kontrolü** yapar: çok küçük bir veri parçasına (örn. tek bir batch, 32 örnek) modeli **kasten aşırı uydurur** (overfit). Loss neredeyse 0'a inerse, ağ öğrenebiliyor demektir — mimaride/kodda temel bir hata yok. (Loss tam 0 olmaz çünkü aynı bağlamın farklı hedefleri olabilir.)

> **Builder Notu — Önce Küçük Veriye Overfit Et**
>
> **Geriye (Ders 1):** Döngü Ders 1 micrograd'ın aynısı; tek fark parametre listesinin uzaması ve `p.grad = None` ile zero_grad. Tek-batch'e overfit, Ders 1'deki "ağ öğreniyor mu?" kontrolünün pratiği.

İleriye: "Önce küçük bir veriye overfit et" her ML mühendisinin ilk sanity-check'idir: öğrenemiyorsa veri/loss/mimaride bug var demektir. Production debug'ın standart ilk adımı.

10.10 Minibatch SGD

Tüm veri seti (228 binden fazla örnek) üzerinde her adımda forward/backward yapmak yavaş. Çözüm Ders 1'den tanıdık: **minibatch** — her adımda rastgele küçük bir örnek kümesi (örn. 32) seç, gradyanı onun üzerinde hesapla.

```
ix = torch.randint(0, X.shape[0], (32,))   # rastgele 32 ornek indeksi
emb = C[X[ix]]                              # sadece bu batch
# ... forward/backward/update yalnizca X[ix], Y[ix] uzerinde
```

Minibatch gradyanı, tam gradyanın **gürültülü ama tarafsız** bir tahminidir. Adımlar biraz "yanlış yönde" olabilir ama çok daha hızlı atılır — pratikte yaklaşık doğru yönde çok sayıda hızlı adım, az sayıda mükemmel adımdan iyidir.

Builder Notu — Gürültülü Ama Hızlı

Geriye (Ders 1 + Stat 110): Bu, Ders 1 §9'un birebir uygulaması: minibatch gradyanı = tam gradyanın Monte Carlo tahmini; varyans $\propto 1/B$ (Stat 110 örneklem ortalaması). `torch.randint` ile rastgele örnekleme.
İleriye: Tüm büyük-ölçek eğitim minibatch'le yapılır; batch boyutu throughput/bellek dengesidir (Ders 10: gradient accumulation, DDP — minibatch'in dağıtık hâli).

10.11 İyi Bir Öğrenme Oranı Bulmak

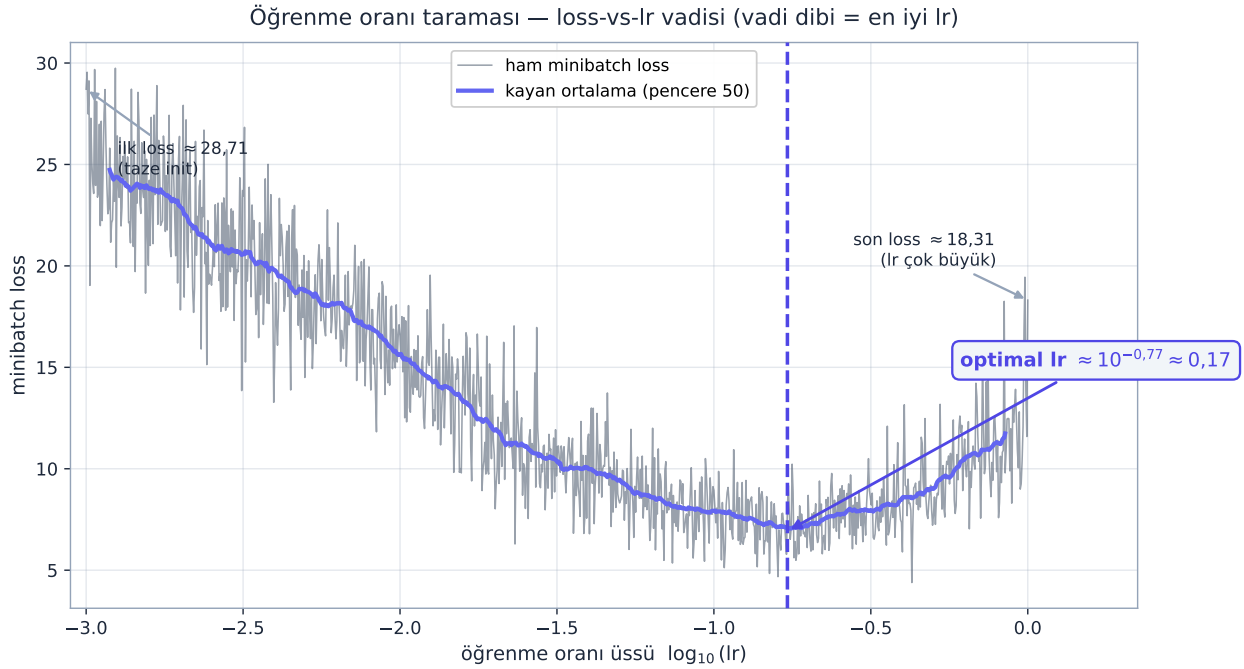
Öğrenme oranını (lr) elle tahmin etmek yerine Karpathy prensipli bir yöntem gösterir: bir **tarama** (sweep). lr'yi geniş bir aralıkta (örn. 10^{-3} ile 10^{0} arası) üstel olarak dene, her biri için loss'u kaydet, loss-vs-lr grafiğini çiz, **vadinin dibini** (en hızlı düşüş) seç.

```
lre = torch.linspace(-3, 0, 1000)   # ussel aralik
lrs = 10**lre                       # lr = 10^exp (0.001 .. 1.0)
# her adimda bir lr dene, loss'u kaydet, sonra loss-vs-exp ciz
```

Bizim taramamızda vadinin dibi gerçekte üs $\approx -0{,}77$'de, yani lr $\approx 10^{-0{,}77} \approx 0{,}17$ çıkar. İyi lr bulunduktan sonra, eğitimin sonlarına doğru lr **azaltılır** (decay) — büyük adımlarla yaklaş, küçük adımlarla yerleş.

Builder Notu — lr Taraması → LR Schedule

Geriye (Ders 1): lr, Ders 1 §8'in η'sı — çok büyük ıraksar, çok küçük yavaş. Tarama, "çok küçük/çok büyük arası dengeyi" deneysel bulma yöntemi.
İleriye: lr taraması ve **lr decay** (sonlara doğru azaltma), production'da **learning rate schedule**'a dönüşür:



Şekil 10.6: Karpathy'nin imza yöntemi — **öğrenme oranı taraması**: tek bir ağı 1000 adım eğitirken her adımda $10^{-3} \ldots 10^{0}$ aralığından üstel olarak artan bir lr dener; x = lr üssü, y = minibatch loss. Slate çizgi gürültülü ham loss, indigo eğri kayan-ortalama (pencere 50). Vadinin dibi (en düşük 50-pencere ortalaması) GERÇEK olarak üs $\approx -0{,}77$'de, yani lr $\approx 10^{-0{,}77} \approx 0{,}17$ — indigo dikey çizgi. İlk loss $\approx 28{,}71$ (taze rastgele init, hokey-sopası) küçük lr'lerle yavaş düşer; lr büyüyüp $\approx 0{,}17$'yi aşınca eğitim ıraksamaya başlar ve loss son adımda $\approx 18{,}31$'e tırmanır. Vadinin dibi = en hızlı kararlı düşüş bölgesi.

warmup + cosine/linear decay (Ders 10). Hyperparameter araması (Optuna, ablation) bunun otomatik hâli.

10.12 Train / Dev / Test Bölmesi

Şimdiye dek tek bir veri seti kullandık — ama bu tehlikeli: model veriyi **ezberleyebilir** (overfitting) ve yeni veride başarısız olur. Çözüm, veriyi üçe bölmek:

> *"We have the training split, the dev/validation split, and the test split."* — Karpathy, 54:35

```
import random
random.shuffle(words)
n1 = int(0.8 * len(words))   # %80 egitim
n2 = int(0.9 * len(words))   # %10 dev, %10 test
Xtr, Ytr = build_dataset(words[:n1])    # train: ogren
Xdev, Ydev = build_dataset(words[n1:n2])  # dev: hyperparametre ayarla
Xte, Yte = build_dataset(words[n2:])    # test: yalnizca en sonda, bir kez
```

- **Train (%80):** Parametreleri öğrenmek için. (Bizde 182.546 örnek.)
- **Dev/validation (%10):** Hyperparametreleri (ağ boyutu, lr, embedding boyutu) ayarlamak için. (22.840 örnek.)
- **Test (%10):** Yalnızca en sonda, **bir kez** — gerçek genelleme performansı. Sık bakarsan ona da overfit edersin. (22.760 örnek.)

Tanı: train loss ≈ dev loss ise model **yetersiz** (underfitting — daha büyük ağ gerek); train loss dev loss ise **aşırı öğrenme** (overfitting — düzenlileştirme/daha çok veri gerek).

Builder Notu — Üçlü Bölme = Standart Protokol

Geriye (Ders 1 + Stat 110): Train/dev/test ve under/overfitting, Ders 1 §10'un pratiği; bias-variance dengesi (Stat 110 Ders 34). "Test setine sık bakma" = ona bilgi sızdırma (data leakage).
İleriye: Bu üçlü bölme, tüm ML'in standart protokolüdür. Production'da dev set üzerinde model seçimi + test set ile final değerlendirme; LLM'lerde ayrıca held-out benchmark'lar (Ders 10: HellaSwag).

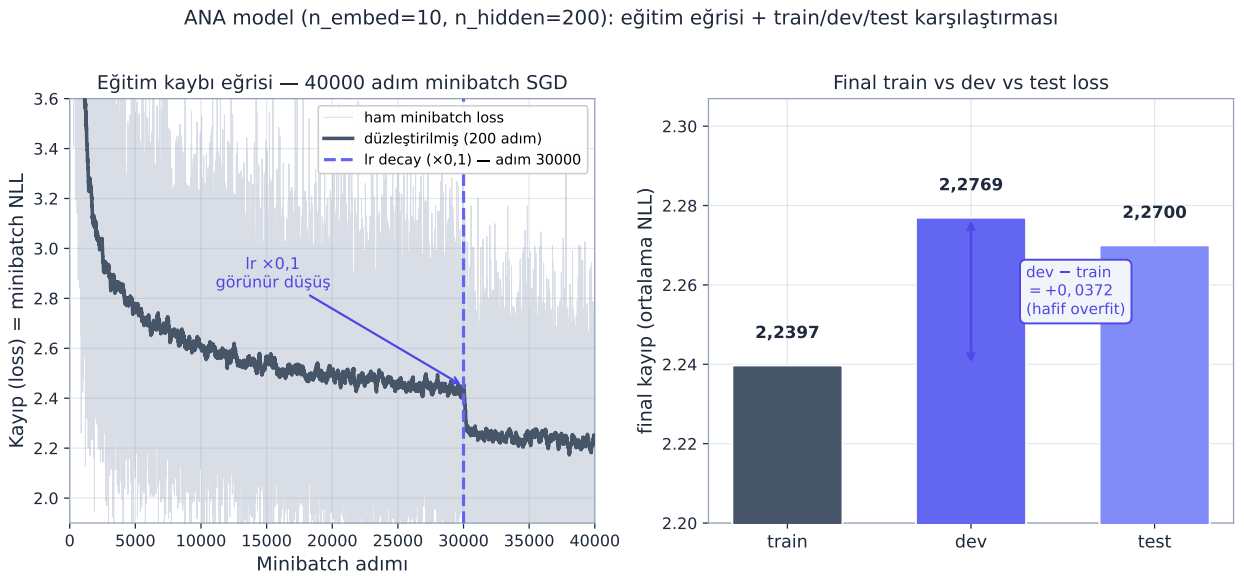
10.13 Ölçeklendirme: Daha Büyük Ağ

Loss'u daha da düşürmek için ağı büyütürüz: daha çok gizli nöron, daha yüksek boyutlu embedding. Karpathy gizli katmanı ve embedding boyutunu artırıp yeniden eğitir.

Önemli gözlem: bazen darboğaz **embedding boyutudur**. 2 boyutlu embedding, 27 karakterin ilişkilerini sıkıştırmak için fazla küçük olabilir; boyutu artırmak (örn. 10) loss'u düşürür. Ama her büyütme bir denge: daha çok parametre = daha iyi uyum ama daha yavaş eğitim + overfitting riski (dev loss'u izle).

Bizim ANA modelimiz ($n_{embed} = 10$, $n_{hidden} = 200$, 40.000 adım, lr $= 0{,}1$, 30.000. adımda decay) şu final değerleri verir: **train** $\approx 2{,}24$, **dev** $\approx 2{,}28$, **test** $\approx 2{,}27$. Dev, train'den yalnızca $+0{,}0372$ yüksek — bu,

hafif bir overfit ama sağlıklı bir denge. Aşağıdaki figür hem eğitim eğrisini (lr decay düşüşü görünür) hem de üç bölmenin final karşılaştırmasını gösterir.



Şekil 10.7: ANA model (n_embed=10, n_hidden=200) eğitimi: train vs dev loss. **Sol (slate):** 40000 minibatch adımı boyunca kayıp eğrisi (hafif düzleştirilmiş); 30000. adımdaki lr decay (lr $\times 0{,}1$) indigo dikey çizgiyle işaretli — orada görünür bir düşüş olur (büyük adımlarla yaklaş, küçük adımlarla yerleş). **Sağ (bar):** final TRAIN $\approx 2{,}24$ vs DEV $\approx 2{,}28$ vs TEST $\approx 2{,}27$; dev train'den $+0{,}0372$ yüksek (hafif overfit, indigo anot). Slate çizgi + indigo dev/işaretçi; deterministik (SEED).

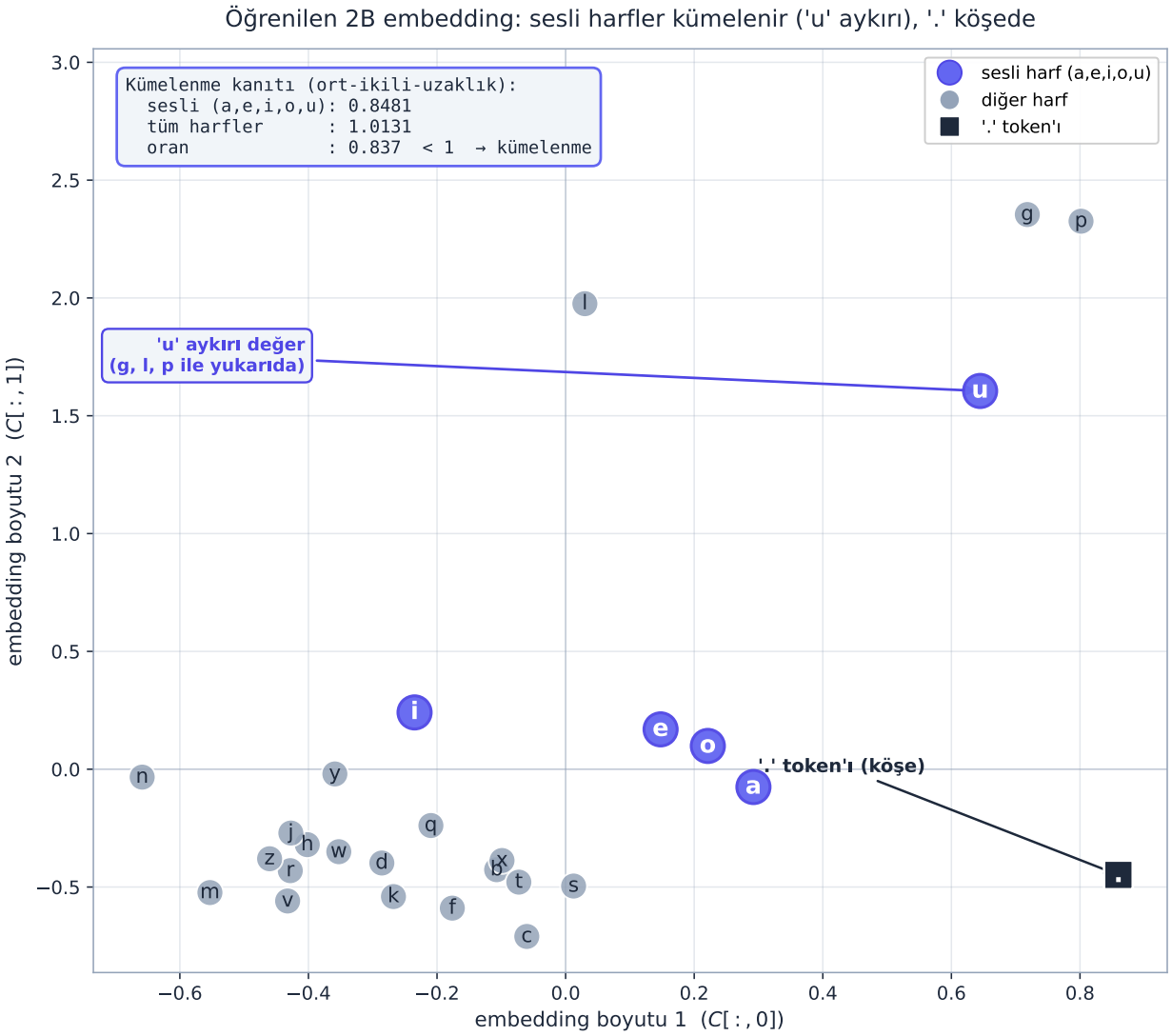
> Builder Notu — Darboğaz Nerede?
>
> **İleriye:** "Darboğaz nerede?" sorusu (embedding mi, gizli katman mı, veri mi) production model tasarımının özüdür. Parametre sayısını ölçeklemek (genişlik/derinlik/embedding) doğrudan **scaling laws**'a (Ders 6/10) bağlanır: model boyutu vs veri vs compute dengesi.

10.14 Embedding'leri Görselleştirme

2 boyutlu embedding'lerin güzel yanı: doğrudan çizilebilirler. Karpathy 27 karakteri bir scatter plot'ta gösterir — ve model kimse söylemeden anlamlı yapı öğrenmiştir: **sesli harfler** (a, e, i, o, u) birbirine yakın kümelenir, **.** token'ı kendi başına bir köşede durur.

Bunu kendi GÖRSEL modelimizde ($n_{embed} = 2$) ölçtük: sesli harflerin ortalama ikili uzaklığı $0{,}8481$, tüm harflerin ortalaması ise $1{,}0131$ (oran $\approx 0{,}837 < 1$) — yani sesli harfler birbirine, rastgele bir harf çiftine göre belirgin biçimde daha yakın. Dürüst not: bu tohumda 'u' bir aykırı değerdir (sesli ailesinden kopup yukarıda g/l/p ile durur); figür bunu olduğu gibi gösterir, sahte bir "mükemmel 5'li küme" çizmez. Yine de kümelenme gerçektir — model a/e/i/o'yu kimse söylemeden birbirine yaklaştırmıştır.

Bu, embedding'in neden güçlü olduğunun görsel kanıtı: model, "benzer davranan karakterleri" sürekli bir uzayda birbirine yaklaştırarak **genelleme** yapar — say-tablosunun asla yapamayacağı bir şey.



Şekil 10.8: GÖRSEL modelin (n_embed=2) öğrendiği 2B embedding uzayı: 27 karakterin gerçek C konumları. Sesli harfler (a, e, i, o) indigo ile vurgulanır ve orijin yakınında **sıkı** kümelenir; geri kalan harfler slate. **.** token'ı sağ-alt köşede $(+0{,}86, -0{,}45)$ tek başına durur. DÜRÜSTLÜK: bu tohumda **'u' bir aykırı değerdir** $(+0{,}64, +1{,}61)$ — sesli ailesinden kopup yukarıda g/l/p ile durur; figür bunu olduğu gibi gösterir, sahte 'mükemmel 5'li küme' çizmez. Yine de kümelenme gerçektir: sesli ortalama-ikili-uzaklık $0{,}8481$ < tüm-harf ortalaması $1{,}0131$ (oran $\approx 0{,}837$) — model kimse söylemeden a/e/i/o'yu birbirine yaklaştırmıştır.

> Builder Notu — Embedding Geometrisi = Anlamsal Yapı
>
> **İleriye:** Öğrenilen embedding uzayında anlamsal yapı, tüm modern NLP'nin temel gözlemidir (word2vec'in "king − man + woman ≈ queen"i). GPT'nin token embedding'leri de aynı şekilde yapı öğrenir, yalnızca çok daha yüksek boyutta. Embedding görselleştirme (t-SNE, PCA) production'da model anlamanın standart aracı.

10.15 Sonuç ve Örnekleme

Eğitilen MLP'den isim örneklemek, Ders 2'deki döngünün aynısı — yalnızca bir satır olasılık tablosu yerine MLP'nin forward'ı kullanılır:

```
context = [0] * block_size
while True:
    emb = C[torch.tensor([context])]
    h = torch.tanh(emb.view(1, -1) @ W1 + b1)
    logits = h @ W2 + b2
    probs = F.softmax(logits, dim=1)
    ix = torch.multinomial(probs, num_samples=1, generator=g).item()
    context = context[1:] + [ix]   # pencereyi kaydir
    if ix == 0:
        break
```

Üretilen isimler bigram'dan **belirgin biçimde daha iyidir** — 3 karakterlik bağlam, tek karaktere göre çok daha gerçekçi diziler verir. Karpathy çalıştırılabilir Colab notebook'unu paylaşır.

> Builder Notu — Örnekleme = GPT'nin Üretim İskeleti
>
> **İleriye:** Örnekleme döngüsü (bağlamı kaydır → forward → softmax → multinomial → tekrarla), GPT'nin metin üretiminin birebir iskeleti (autoregressive generation). Tek fark: MLP sabit bağlam, GPT uzun bağlam + dikkat.

10.16 Bu Dersin Özeti

1. Bigram'ın **bağlam patlaması** (27^n) çözümü: say-tablosu yerine **öğrenilen embedding + MLP** (Bengio 2003).
2. **block_size** bağlam uzunluğunu belirler; `build_dataset` kayan pencereyle (bağlam → hedef) örnekleri üretir ("emma" → 5 örnek; tüm veri → 228.146 örnek).
3. **Embedding tablosu C**: `C[X]` ile id → vektör (Ders 2'nin "one-hot @ W = satır seçimi"nin verimli hâli; backprop ile öğrenilir).
4. **MLP**: embedding'i `.view()` ile düzleştir → gizli `tanh` katmanı (W1, b1) → çıktı logitleri (W2, b2) → softmax.
5. **F.cross_entropy**: Ders 2'nin NLL'i; daha hızlı (füzyonlu) + sayısal kararlı (max çıkarma, exp taşmasını önler).

6. **Eğitim döngüsü** Ders 1'in aynısı (forward → zero_grad → backward → update); önce tek batch'e **overfit** ederek sanity-check.
7. **Minibatch SGD** (`torch.randint`): gürültülü ama hızlı gradyan tahmini (varyans $\propto 1/B$).
8. **Öğrenme oranı taraması** (`torch.linspace`, loss-vs-lr): vadinin dibi gerçekte üs $\approx -0{,}77$ (lr $\approx 0{,}17$); sonra decay.
9. **Train/dev/test** bölmesi (80/10/10); test'e yalnızca bir kez bak. ANA model final train $\approx 2{,}24$, dev $\approx 2{,}28$ (hafif overfit). Embedding'ler anlamlı yapı öğrenir (sesli harfler kümelenir: $0{,}8481 < 1{,}0131$).

! Tek Bir Cümle

Bağlamı say-tablosunda tutmak üstel patlar; her karakteri öğrenilen bir vektöre (embedding) gömüp bir MLP'den geçirmek, aynı "sonraki karakteri tahmin et" problemini ölçeklenebilir kılar — ve model, kimse söylemeden benzer karakterleri (sesli harfler) embedding uzayında birbirine yaklaştırarak genelleme öğrenir.

10.17 Kontrol Soruları

i Soru 1: block_size = 3 ile, 'an' kelimesi build_dataset'te hangi (bağlam → hedef) örneklerine bölünür? (s2i: . = 0, a = 1, n = 14)

Bağlam `[0, 0, 0]` (üç nokta) ile başlar, her adımda kayar. "an" + bitiş `.`:

- `[0, 0, 0]` → `1` (`...` → a)
- `[0, 0, 1]` → `14` (`..a` → n)
- `[0, 1, 14]` → `0` (`.an` → .)

Cevap: 3 örnek. Her satır 3 tamsayı id (girdi bağlam), hedef ise bir sonraki karakterin id'sidir. `context = context[1:] + [ix]` her adımda en eski karakteri atıp yenisini ekler (kayan pencere).

i Soru 2: Elle yazılan counts = logits.exp(); probs = …; loss = -…log().mean() yerine neden F.cross_entropy(logits, Y) tercih edilir? İki neden.

Cevap: (1) Verimlilik: `F.cross_entropy` ara tensörleri (counts, probs) bellekte oluşturmaz; işlemleri tek bir füzyonlu çekirdekte yapar ve backward'ı daha basit/hızlıdır. **(2) Sayısal kararlılık:** Büyük bir logit (örn. 100) için `logits.exp()` taşar (inf → nan, loss bozulur). `F.cross_entropy` içeride logitlerden **maksimumu çıkarır** — softmax bu kaydırmaya karşı değişmez olduğu için (softmax$(z) =$ softmax$(z - c)$) sonuç aynı kalır ama üstel asla taşmaz. Matematiksel olarak elle yazılanla aynı, ama pratikte güvenli ve hızlı.

i Soru 3: C şekli (27, 2), X şekli (n, 3) ise C[X] hangi şekli verir? Gizli katmana vermeden önce neden .view(-1, 6) gerekir?

Cevap: `C[X]` şekli $(n, 3, 2)$ — her örnek (n), 3 karakter, her karakter 2-boyutlu embedding. Ama gizli katman düz bir vektör bekler: 3 karakterin 2'şer boyutunu birleştirip **6-boyutlu** tek vektör. `emb.view(-1, 6)` bunu yapar → $(n, 6)$. `.view()` veriyi **kopyalamaz**, yalnızca aynı bellekteki görünümü (storage

+ strides) değiştirir — bu yüzden ucuzdur. `-1`, "bu boyutu otomatik hesapla" demek (burada n).

ℹ Soru 4: (Builder) C[X] (embedding arama) ile Ders 2'deki xenc @ W (one-hot çarpımı) neden aynı işlemdir? 18.06 ile bağla.

Cevap: Ders 2'de bir id'yi one-hot vektöre çevirip `W` ile çarpıyorduk; bir one-hot (baz vektörü e_i) ile matris çarpımı, `W`'nin i'inci satırını seçer (18.06 Ders 30). `C[X]` ise aynı satırı **doğrudan indeksleyerek** seçer — one-hot vektörü oluşturup çarpma masrafına girmeden. İkisi de "id → matrisin o satırı" yapar; embedding sadece verimli (büyük sözlüklerde 50.000-boyutlu one-hot çarpımı israfından kaçınır). Bu yüzden embedding tablosu = öğrenilen bir satır-arama matrisidir; GPT'nin `wte` tablosu da budur.

10.18 Egzersizler

Egzersiz 1 (Veri setini kur). `build_dataset`'i block_size = 3 ile yaz, tüm isimlere uygula. `X.shape` ve `Y.shape` yazdır (X: $(n, 3)$, Y: $(n,)$; bizde $n = 228146$). Birkaç örneği `i2s` ile geri çevirip "bağlam → hedef" olarak elle doğrula.

Egzersiz 2 (MLP'yi kur, tek batch'e overfit). C (27×2), W1/b1, W2/b2 ile MLP'yi kur. Çok küçük bir veri parçasına (örn. ilk 32 örnek) kasten **overfit** et — loss'un neredeyse 0'a indiğini gözlemle. Bu, ağın öğrenebildiğinin (kodun doğru olduğunun) sanity-check'idir.

Egzersiz 3 (Öğrenme oranı taraması). `torch.linspace(-3, 0, 1000)` ile lr üslerini, `10**lre` ile lr'leri üret. Her adımda bir lr dene, loss'u kaydet, loss-vs-üs grafiğini çiz. "Vadinin dibini" (en hızlı düşüş) bul — bizde üs $\approx -0{,}77$, yani lr $\approx 0{,}17$. Bu lr ile tam eğitim yap.

Egzersiz 4 (Train/dev/test). Veriyi 80/10/10 böl (`build_dataset` ile üç ayrı set). Eğit, sonra **train loss** ve **dev loss**'u karşılaştır (bizde train $\approx 2{,}24$, dev $\approx 2{,}28$). İkisi yakınsa underfitting (ağı büyüt), train dev ise overfitting. Gizli katmanı/embedding'i büyütüp dev loss'un nasıl değiştiğine bak.

Egzersiz 5 (Sonraki dersin habercisi). Ağı eğitmeye başladığında **ilk iterasyonun loss'u** kaçtır? (a) Model başlangıçta hiçbir şey bilmiyorsa, 27 karaktere eşit (uniform) olasılık vermeli — bu durumda loss $\approx -\log(1/27) \approx 3{,}30$ olmalı. (b) Ama rastgele başlatılmış W2 ile ilk loss çok daha yüksek çıkar (bizim ölçümümüzde $\approx 26{,}0$); neden? (İpucu: rastgele logitler aşırı uçlarda → model "kendinden emin ama yanlış".) İlk birkaç adım sadece bu aşırı logitleri ezmekle geçer ("hokey-sopası" loss eğrisi). Bu israfı nasıl önlersin? Bu soru, Ders 4'te (**makemore 3: aktivasyonlar, gradyanlar, BatchNorm**) başlangıç (initialization) ve aktivasyon istatistiklerini düzeltmeyi motive eder.

10.19 Sonraki Ders İçin Hazırlık

Ders 4: makemore 3 — Aktivasyonlar, Gradyanlar ve BatchNorm — Andrej Karpathy

Bu derste MLP'yi kurduk ve eğittik — ama "çalışıyor" ile "iyi çalışıyor" arasında fark var. Ders 4'te tam **bu ağın** içine bakacağız: başlangıçtaki loss neden çok yüksek (Egzersiz 5), `tanh` aktivasyonları neden "doyuyor" (saturation) ve ölü nöronlar oluşuyor, gradyanlar katmanlar arası nasıl akıyor. **Kaiming (He) başlatması** ile ağırlıkları doğru ölçekleyip, **Batch Normalization**'ı sıfırdan kuracağız.

Ana konular:

- Başlangıç loss'unu düzeltme (logitleri 0'a yakın başlatma).
- Doymuş tanh, ölü nöron, Kaiming init (fan-in ölçeklemesi).
- Batch Normalization: aktivasyonları normalize etme + running istatistikler.

⚠ Ders 4 Öncesi Yapılacak

- Egzersizleri çöz — özellikle 5 (ilk loss neden yüksek?).
- Bu dersteki MLP kodunu hazır tut; Ders 4 doğrudan bunun üstüne kurar.
- "Embedding = öğrenilen satır-arama" ve "F.cross_entropy = kararlı NLL" cümlelerini hatırla.

10.20 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Karpathy'de
Bağlam patlaması	Say-tablosu bağlamla üstel büyür (27^n); çözüm öğrenilen embedding	0m00
Bengio 2003	MLP dil modeli makalesi: embedding + gizli katman + softmax	1m48
block_size	Bağlam uzunluğu; kayan pencereyle (bağlam → hedef) örnek üretir	9m01
Embedding tablosu C	C[X] ile id → vektör; Ders 2 one-hot @ W satır seçiminin verimli hâli	12m19
.view() reshape	Tensörü kopyalamadan yeniden şekillendirme (storage + strides görünümü)	18m35
Gizli tanh katmanı	emb.view(-1,6) @ W1 + b1, sonra tanh; Ders 1'in lineer katmanı + aktivasyon	18m35
F.cross_entropy	Ders 2 NLL'in füzyonlu + sayısal-kararlı hâli (max çıkarır, exp taşmaz)	32m50
Tek batch'e overfit	Küçük veriye loss≈0; ağın öğrenebildiğinin sanity-check'i	37m56
Minibatch SGD	torch.randint ile rastgele batch; gürültülü ama hızlı gradyan (varyans $\propto 1/B$)	41m28
Öğrenme oranı taraması	torch.linspace ile lr üs taraması; loss-vs-lr vadisi (üs ≈ −0,77); sonra decay	45m39

Kavram	Tanım	Karpathy'de
Train / dev / test	80/10/10; train=öğren, dev=ayarla, test=bir kez; train ≈2,24 dev ≈2,28	53m20
Embedding görselleştirme	2B scatter; model sesli harfleri kümeler (0,8481 < 1,0131)	1h05m

10.21 ML Builder Bağlantıları

9 köprü

1. **Embedding C[X]** → Ders 2 "one-hot @ W = satır seçimi" + 18.06 baz vektörü. İleriye: GPT'nin `wte` token embedding tablosu (Ders 10).
2. **MLP (gizli tanh)** → Ders 1 Neuron/Layer/MLP + tanh türevi ($1 - \tanh^2$). İleriye: transformer'ın feed-forward bloğu (Ders 7).
3. **F.cross_entropy** → Ders 2 NLL + Stat 110 MLE; füzyon + sayısal kararlılık. İleriye: operator fusion (Ders 10 torch.compile).
4. **.view() / reshape** → tensör storage/strides. İleriye: attention'da head ayırma/birleştirme (Ders 7/10).
5. **Minibatch SGD** → Ders 1 §9 + Stat 110 varyans $\propto 1/B$. İleriye: DDP, gradient accumulation (Ders 10).
6. **lr taraması + decay** → Ders 1 §8 η. İleriye: learning rate schedule, warmup+cosine (Ders 10).
7. **Train/dev/test** → Ders 1 §10 overfitting + Stat 110 bias-variance. İleriye: held-out benchmark (Ders 10 HellaSwag).
8. **Embedding'de anlamsal yapı** → öğrenilen temsil (sesli harf kümesi). İleriye: word2vec, GPT embedding geometrisi.
9. **block_size sabit bağlam** → MLP'nin sınırı. İleriye: transformer'ın uzun bağlamı + dikkat (Ders 7).

10.22 Karpathy'nin Önerdiği Kaynaklar

Karpathy'nin bu ders için verdiği kaynaklar:

- **Ders notebook'u:** makemore_part2_mlp.ipynb — dersin adım adım kodu.
- **cs231n NumPy/Python rehberi:** cs231n.github.io/python-numpy-tutorial — tensör/indeksleme temelleri.
- **PyTorch başlangıç:** Tensors + Deep Learning with PyTorch.
- **Ders Colab notebook'u:** Google Colab — çalıştırılabilir ortam.

! Bu dersten tek bir şey alıp gideceksen

Bağlamı say-tablosunda tutmak üstel patlar (27^n); bunun yerine her karakteri öğrenilen bir vektöre (embedding) gömüp bir MLP'den geçirmek, aynı "sonraki karakteri tahmin et" problemini ölçeklenebilir kılar. Embedding tablosu C, Ders 2'deki "one-hot @ W = satır seçimi"nin ta kendisidir — ve model, kimse söylemeden benzer karakterleri embedding uzayında kümeleyerek genelleme öğrenir.

11 makemore 3 — Aktivasyonlar, Gradyanlar ve BatchNorm

Bir ağı eğitilebilir kılmak yalnızca mimari değildir; aktivasyon ve gradyan istatistiklerini de doğru ayarlamaktır — doğru başlatma (Kaiming) sinyali katmanlar boyunca korur, Batch Normalization onu zorla dayatır

> **Bölüm bilgisi**
>
> - **Karpathy'nin videosu:** YouTube — Building makemore Part 3: Activations & Gradients, BatchNorm (≈116 dk)
> - **Seri:** Neural Networks: Zero to Hero — Ders 4
> - **Hoca:** Andrej Karpathy
> - **Kaynak repo:** github.com/karpathy/makemore
> - **Okuma süresi:** ≈34 dk

11.1 Bu Derste Ne Var?

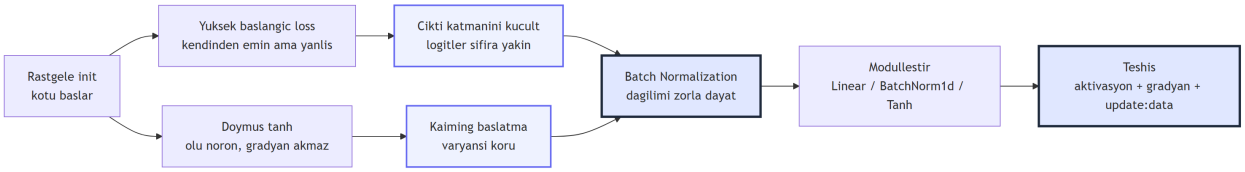
Ders 3'te MLP'yi kurduk ve çalıştırdık. Ama "çalışıyor" ile "iyi çalışıyor" aynı şey değil. Bu derste **tam o ağın** içine bakarız: ağırlıklar nasıl başlatılmalı, aktivasyonlar ve gradyanlar katmanlar arasında nasıl akmalı, ve neden derin ağları "sağlıklı" tutmak başlı başına bir mühendislik.

> *"In the last lecture we implemented the multilayer perceptron along the lines of Bengio 2003 for character-level language modeling."* — Karpathy, 0:00

Büyük fikir: rastgele başlatılan bir ağ genelde **kötü başlar** — ya aşırı kendinden emin (yanlış), ya aktivasyonlar doymuş (gradyan akmıyor). Bunu önce elle (Kaiming başlatma), sonra otomatik bir mekanizmayla (**Batch Normalization**) düzeltiriz. Son olarak kodu PyTorch-tarzı modüllere dönüştürüp ağı **teşhis araçlarıyla** (histogramlar) izleriz.

Dersin üç büyük fikri:

1. **Başlatma (initialization)** — başlangıç loss'unu düzelt (logitleri 0'a yakın başlat), doymuş `tanh`'ı önle, ağırlıkları **Kaiming** ile ölçekle.
2. **Batch Normalization** — aktivasyonları batch üzerinden normalize ederek eğitimi sağlamlaştıran katman; running istatistikler, ölçekle-kaydır, epsilon.
3. **Teşhis ve PyTorch-laştırma** — kodu modüllere (Linear/BatchNorm1d/Tanh) böl, aktivasyon/gradyan histogramlarıyla ağın sağlığını izle.



Şekil 11.1: Ders 4'ün kavram haritası: rastgele init kötü başlar (yüksek loss + doymuş tanh), önce elle (başlangıç-loss fix + Kaiming) sonra otomatik (BatchNorm) düzeltilir, ardından kod modüllere bölünüp teşhis araçlarıyla izlenir. Slate akış + indigo dönüm noktaları (Kaiming ve BatchNorm).

Builder Notu — Ders 3'ün TAM Ağını Mercek Altına Almak

Geriye (Ders 1-3 + Stat 110):

- **Bu, Ders 3'ün TAM ağı.** Yeni model yok; aynı MLP'yi (embedding + gizli `tanh` + softmax) mercek altına alıyoruz.
- **Başlangıç loss'u = Ders 3 Egzersiz 5.** "Neden ilk loss 26, oysa $\log(27) \approx 3{,}3$ olmalı?" sorusunun cevabı burada (hokey-sopası eğrisi).
- **Doymuş `tanh` = Ders 1.** `tanh`'ın türevi $1 - \tanh^2$; çıktı ± 1'e yapışınca türev 0 olur → gradyan akmaz (ölü nöron).
- **Kaiming + BatchNorm = Stat 110.** Başlatma, aktivasyon **varyansını** katmanlar arası korumakla ilgili (Stat 110 varyans/Normal); BatchNorm ise **standardizasyon** (z-skoru: $(x - \mu)/\sigma$).

İleriye: Başlatma ve normalizasyon, derin ağları eğitilebilir kılan iki temel. BatchNorm 2015'te derin ağ devrimini (ResNet) mümkün kıldı; Ders 7'de transformer'ın tercih ettiği **LayerNorm**'u göreceğiz. "Aktivasyon/gradyan istatistiklerini izle" alışkanlığı production debug'ının çekirdeği.
Tek cümleyle: Bir ağı eğitilebilir kılmak, yalnızca mimariyi değil **aktivasyon ve gradyan istatistiklerini** de doğru ayarlamaktır — doğru başlatma (Kaiming) ve normalizasyon (BatchNorm) bunu sağlar.

11.2 Neden Aktivasyon ve Gradyan İstatistiği?

Ders 3'ün MLP'si çalışıyordu ama Karpathy daha derine iner: rastgele başlatılan ağırlıklar, ağın içindeki sayıları (aktivasyonlar) ve onların gradyanlarını kötü bir hâle sokabilir. Çok büyük/küçük aktivasyonlar, doygunluk, akıp giden veya patlayan gradyanlar — bunlar eğitimi yavaşlatır veya tamamen durdurur.

> *"If you're well-versed in the dark arts of backpropagation and have an intuitive sense of how these gradients flow through a neural net, [you'll understand why this matters]."* — Karpathy, 15:04

Bu ders, "ağ çalışıyor" ile "ağ sağlıklı öğreniyor" arasındaki farkı görmeyi ve düzeltmeyi öğretir.

Builder Notu — İç İstatistikleri İzle

İleriye: "İç istatistikleri izle" yaklaşımı — aktivasyon dağılımları, gradyan normları, update:data oranı — her ciddi eğitim hattında (W&B, TensorBoard) standarttır. Modelin "neden öğrenmiyor" sorusunun cevabı genelde bu istatistiklerdedir.

11.3 Başlangıç Loss'unu Düzeltme

Ders 3 Egzersiz 5'in cevabı: eğitimin ilk iterasyonunda loss çok yüksek (bizim ölçümümüzde $\approx 26{,}01$) çıkıyordu, oysa "hiçbir şey bilmeyen" bir model 27 karaktere eşit olasılık verip $-\log(1/27) \approx 3{,}30$ loss almalı. Sorun: rastgele başlatılan `W2/b2`, aşırı uç logitler üretiyor — ağ "kendinden emin ama yanlış".

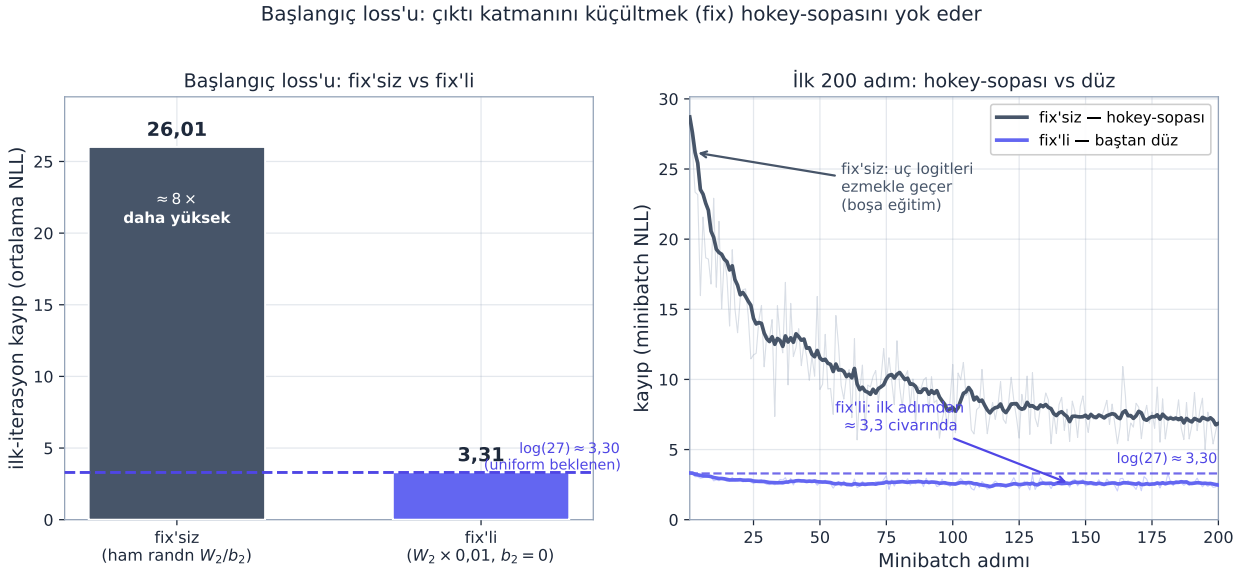
> *"The network is very confidently wrong."* — Karpathy, 5:57

Bu, **hokey-sopası** loss eğrisine yol açar: ilk birkaç iterasyon yalnızca bu aşırı logitleri ezmekle geçer (boşa giden eğitim).

> *"In the hockey stick, the very first few iterations of the loss… the optimization is just squashing down the logits and then rearranging."* — Karpathy, 11:48

Çözüm: logitleri başlangıçta **0'a yakın** yap. Çıktı katmanını küçült: `W2 *= 0.01`, `b2 *= 0`. Böylece başlangıç logitleri $\approx 0 \rightarrow$ softmax $\approx$ uniform $\rightarrow$ ilk loss $\approx 3{,}3$ (beklenen). Hokey-sopası kaybolur, eğitim baştan verimli.

```
# Karpathy §2: cikti katmanini kucult -> logitler ~0 -> ilk loss ~log(27)
W2 = torch.randn((n_hidden, vocab_size)) * 0.01   # ham randn yerine kucuk
b2 = torch.zeros(vocab_size)                       # b2 = 0 (uc logit yok)
# ilk loss artik ~3.3 (uniform beklenen), hokey-sopasi kaybolur
```



Şekil 11.2: Başlangıç loss'u: fix'siz (ham randn W_2/b_2) vs fix'li ($W_2 \times 0{,}01$, $b_2 = 0$). **Sol (bar):** ilk-iterasyon loss — fix'siz $= 26{,}01$ (slate, çok yüksek: ağ kendinden emin ama yanlış) vs fix'li $= 3{,}31$ (indigo); uniform beklenen $-\log(1/27) = \log(27) \approx 3{,}30$ indigo kesik referans çizgisiyle işaretli, fix'li tam onun üstünde. **Sağ (eğri):** ilk 200 minibatch adımı — fix'siz **hokey-sopası** (slate, ≈ 29'dan keskin düşüş; ilk iterasyonlar yalnızca uç logitleri ezmekle geçer, boşa eğitim) vs fix'li (indigo, baştan $\approx 3{,}3$ civarında düz; eğitim ilk adımdan verimli). Deterministik (SEED).

> Builder Notu — Başlangıç Loss'u = İlk Sanity-Check
>
> **Geriye (Stat 110 + Ders 3):** Beklenen başlangıç loss'u $-\log(1/n) = \log(n)$ (uniform dağılımın NLL'i, Stat 110). Bu, "modelin sağlıklı başladığını" doğrulayan ilk göstergedir — Ders 3 Egzersiz 5'in tam cevabı.
> **İleriye:** "Başlangıç loss'u beklenen değere yakın mı?" production'da modelin doğru kurulduğunu gösteren ilk sanity-check'tir. Yanlışsa, eğitimin ilk %5'i boşa gider.

11.4 Doymuş tanh ve Ölü Nöron

Logit sorununu çözdük; şimdi gizli katmana bakalım. `tanh`, girdisini -1 ile $+1$ arasına sıkıştırır. Eğer aktivasyon-öncesi değerler (pre-activation) çok büyük/küçükse, `tanh` çıktısı ± 1'e **yapışır** — buna **doygunluk** (saturation) denir.

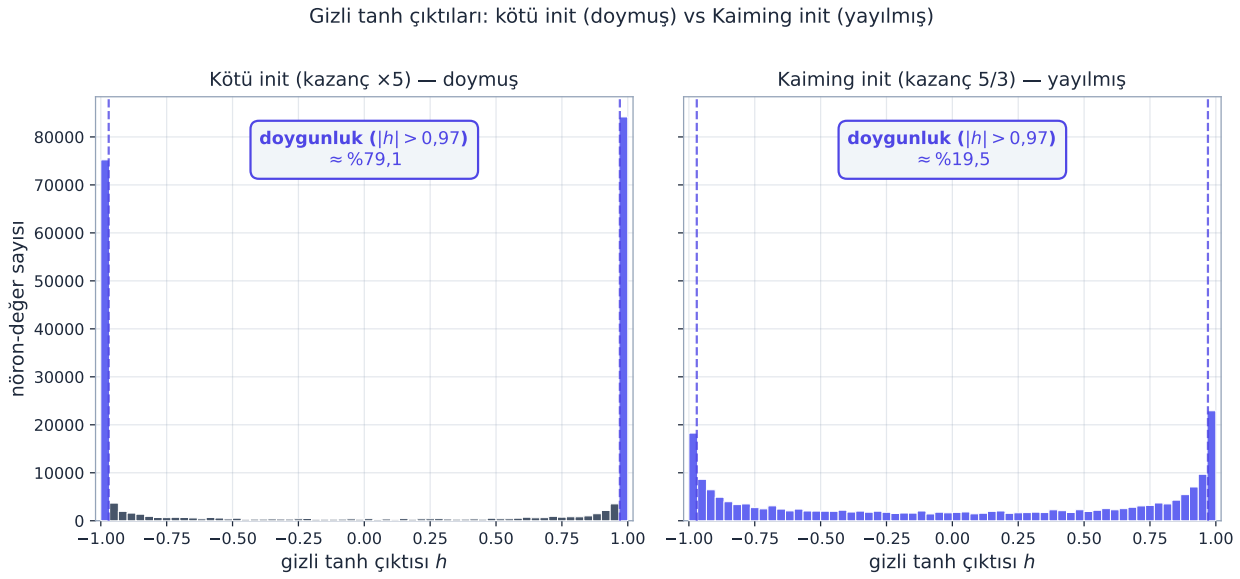
Sorun şu: `tanh`'ın türevi $1 - \tanh^2$'dir (Ders 1). Çıktı ± 1'e yapışınca türev ≈ 0 olur → o nörondan **gradyan akmaz** → o nöron öğrenemez. Karpathy bunu bir histogramla gösterir: rastgele büyük başlatmada gizli aktivasyonların çoğu ± 1'de toplanır (doymuş). Aşağıdaki figürde kötü init (gizli `Linear` kazancı $\times 5$) ile doygunluk oranı $\approx \%79{,}1$, Kaiming init ile yalnızca $\approx \%19{,}5$.

```
# Bir batch ileri gecir, gizli tanh ciktilarinin histogramini incele
h = torch.tanh(emb @ W1 + b1)             # (batch, n_hidden) aktivasyon
saturation = (h.abs() > 0.97).float().mean()   # |h|>0.97 doygunluk orani
# kotu init -> ~%79 doymus; Kaiming -> ~%20 (saglikli)
```

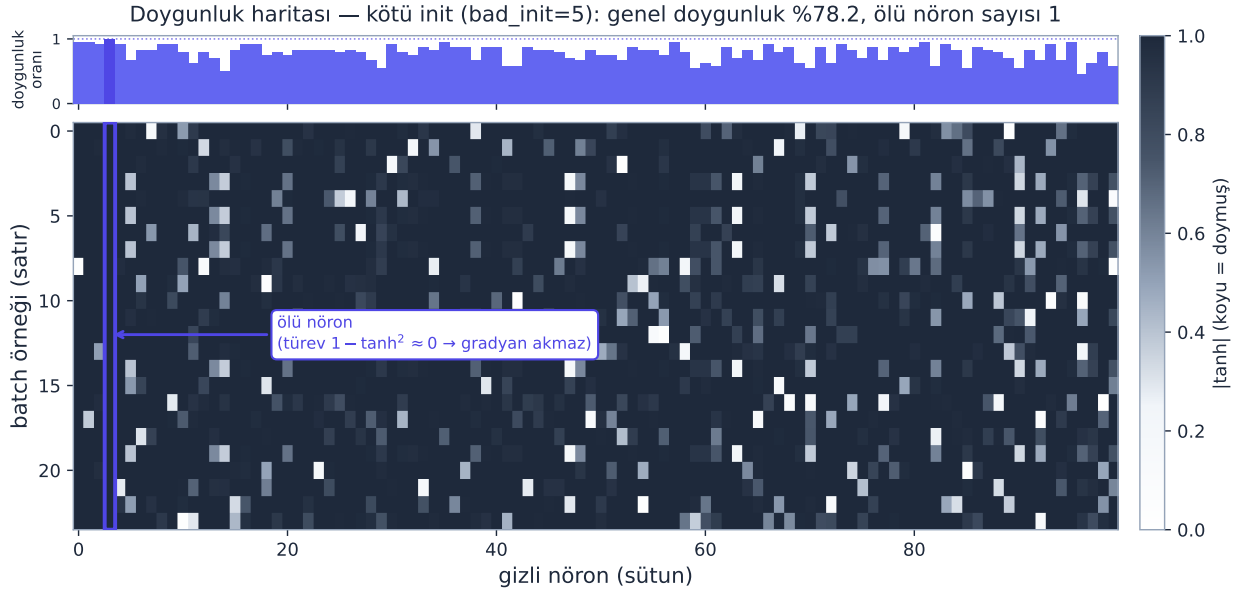
> *"We have what's called a dead neuron… it's kind of like a permanent brain damage in the mind of a network."* — Karpathy, 19:26

Bir nöron tüm batch boyunca hep doymuşsa (hep ± 1), hiç gradyan almaz, asla güncellenmez — kalıcı olarak ölüdür. Aşağıdaki harita bunu somutlaştırır: bir sütun (nöron) tüm batch boyunca koyu (doymuş) ise o nöron ölüdür. Çözüm: aktivasyon-öncesini 0 civarında (`tanh`'ın aktif, eğimli bölgesinde) tutmak. Bunu da `W1` ağırlıklarını küçülterek yaparız (örn. `W1 *= 0.2`). Ama "ne kadar küçük?" sorusunun prensipli cevabı bir sonraki bölümde: Kaiming init.

> Builder Notu — Ölü Nöron Ders 1'in tanh Türevidir
>
> **Geriye (Ders 1):** Ölü nöron, doğrudan Ders 1'in `tanh` türevinden $(1 - o^2)$ çıkar: $o = \pm 1 \rightarrow$ türev $0 \rightarrow$ zincir kuralında o nöronun gradyanı sıfırlanır. ReLU'da karşılığı: girdi hep negatifse çıktı hep 0, türev hep $0 \rightarrow$ ölü ReLU.
> **İleriye:** Doygunluk/ölü nöron, derin ağ eğitiminin klasik sorunudur. Çözümler: dikkatli init (Kaiming), normalizasyon (BatchNorm/LayerNorm), residual bağlantılar (Ders 7). Modern aktivasyonlar (GELU, SiLU) da bu yüzden tercih edilir.



Şekil 11.3: Gizli `tanh` aktivasyonlarının histogramı (bir batch, $n_{hidden} = 200$ nöron). **Sol (slate):** kötü init (gizli `Linear` kazancı $\times 5$) — değerler ± 1'de yığılmış (doymuş); doygunluk ($|h| > 0{,}97$) oranı $\approx \%79{,}1$ (± 1 uçları indigo vurgulu). Doymuş `tanh`'ın türevi $1 - \tanh^2 \approx 0$, gradyan akmaz. **Sağ (indigo):** Kaiming init (`tanh` kazancı $5/3$) — dağılım 0 civarında yayılmış (aktif, eğimli bölge); doygunluk oranı $\approx \%19{,}5$. Aynı tohum (SEED), aynı batch; deterministik.



Şekil 11.4: Doygunluk haritası: satır = batch örneği, sütun = gizli nöron; her hücre kötü-init (bad_init=5) ağında ilk tanh katmanının $|\tanh|$ değerini gösterir — koyu = doymuş ($|h| > 0{,}97$). Bu batch'te genel doygunluk $\approx \%78{,}2$. Bir nöron TÜM batch boyunca doymuşsa (sütun tamamen koyu) **ölü nörondur**: indigo çerçeveyle vurgulanmıştır. Ölü nöron asla güncellenmez (kalıcı brain damage) çünkü türev $1 - \tanh^2 \approx 0 \rightarrow$ gradyan akmaz. Üstteki indigo çubuk nöron-başı doygunluk oranıdır. Slate ısı + indigo ölü-nöron işareti; deterministik (SEED).

11.5 Kaiming (He) Başlatması

Ağırlıkları "ne kadar küçük" başlatmalı? Rastgele tahmin yerine prensipli bir formül var: **Kaiming (He) başlatması** (He ve ark. 2015, "Delving Deep into Rectifiers").

> *"Calculating the init scale: Kaiming init."* — Karpathy, 27:53

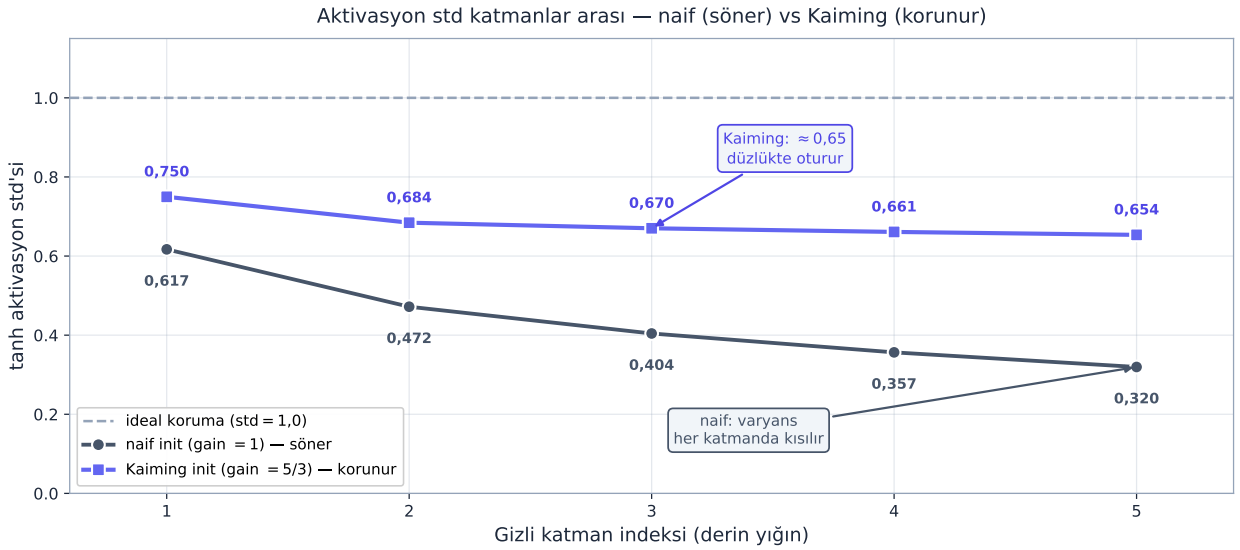
Fikir: her katmanın çıktısının **varyansı**, girdisinin varyansıyla aynı kalmalı — böylece sinyal katmanlar boyunca ne patlar ne söner. Bunu sağlayan ölçek, ağırlık standart sapmasını **fan-in** (girdi bağlantı sayısı) ve aktivasyona özgü bir **kazanç** (gain) ile ayarlar:

$$\text{std} = \frac{\text{gain}}{\sqrt{n_{in}}}$$

Burada n_{in} = fan-in (katmana giren bağlantı sayısı). Kazanç aktivasyona bağlıdır: `tanh` için $5/3$, ReLU için $\sqrt{2}$ (ReLU negatifleri kestiği için varyansı telafi eder).

```
W1 = torch.randn((n_in, n_hidden)) * (5/3) / (n_in**0.5)   # Kaiming (tanh)
```

Bu ölçekle başlatınca aktivasyonlar 0 civarında, iyi yayılmış (ne doymuş ne çökmüş) kalır — gradyanlar sağlıklı akar. Aşağıdaki figür 6 katmanlı derin bir yığında bunu kanıtlar: naif init (gain $= 1$) katman-katman söner ($0{,}617 \to 0{,}320$), Kaiming (gain $= 5/3$) korunur ($0{,}750 \to 0{,}654$, $\approx 0{,}65$ düzlükte oturur).



Şekil 11.5: Aktivasyon std katmanlar arası — **6 katmanlı derin tanh yığını**: naif init (gain $= 1$, slate) katman-katman SÖNER ($0{,}617 \to 0{,}472 \to 0{,}404 \to 0{,}357 \to 0{,}320$); Kaiming init (gain $= 5/3$, indigo) KORUNUR ($0{,}750 \to 0{,}684 \to 0{,}670 \to 0{,}661 \to 0{,}654$, $\approx 0{,}65$ düzlükte oturur). Slate kesik çizgi $y = 1{,}0$ ideal-koruma referansı. tanh kontraktiftir; naif init varyansı her katmanda biraz daha kısar, Kaiming gain $= 5/3$ tam bu daralmayı telafi eder. Deterministik (SEED); gerçek std değerleri (`activation_std_per_layer`).

Builder Notu — fast.ai L17 Köprüsü: Varyans Korunumu → Kaiming → BatchNorm

Çapraz-ders (Phase 2): fast.ai L17 (Temeller B: aktivasyonlar/init) ile **AYNI** kavram: varyans korunumu → Kaiming gain (`tanh` $5/3$, ReLU $\sqrt{2}$) → BatchNorm. Howard ConvNet üzerinden (Xavier/Kaiming + BatchNorm running_mean/register_buffer), Karpathy MLP üzerinden — aynı matematik, farklı çatı. İki ders aynı zinciri kurar: "aktivasyon varyansını katman boyunca koru, init yetmezse normalizasyonla zorla dayat."
Geriye (Stat 110): Kaiming, tamamen **varyans** muhasebesidir (Stat 110): bir lineer kombinasyonun varyansı, terim sayısı (fan-in) ile büyür; bunu $1/\sqrt{n_{in}}$ ile telafi ederiz. `gain` ise aktivasyonun varyansı ne kadar daralttığını (`tanh` kontraktiftir) dengeler.
İleriye: Kaiming/Xavier init, her derin ağ kütüphanesinin varsayılanıdır (PyTorch `nn.init.kaiming_normal_`). Ama BatchNorm/LayerNorm geldikten sonra init'e hassasiyet azalır — bir sonraki konu tam da bu.

11.6 Batch Normalization: Fikir

Kaiming init işe yarar ama kırılgandır: ağ derinleştikçe, her katmanın aktivasyon dağılımını doğru tutmak giderek zorlaşır. 2015'te çıkan radikal fikir: **init'le uğraşma — istediğin dağılımı zorla**.

> *"Normalization layers like batch normalization, layer normalization, group normalization…"* — Karpathy, 36:41

Batch Normalization'ın sezgisi: gizli katmanın aktivasyon-öncesi değerlerinin (pre-activation) **kabaca Gaussian** (ortalama 0, standart sapma 1) olmasını istiyoruz — o zaman doğrudan **öyle yap**. Her eğitim adımında, batch üzerinden istatistikleri hesapla ve aktivasyonları normalize et. Bu, "doğru init'i bulma" problemini büyük ölçüde ortadan kaldırır.

Builder Notu — BatchNorm = Stat 110 Standardizasyonu

Geriye (Stat 110): "Bir dağılımı ortalama 0, std 1 yap" tam olarak **standardizasyondur** (z-skoru: $z = (x - \mu)/\sigma$, Stat 110). BatchNorm bunu ağın içinde, her katmanda yapar.
İleriye: Normalizasyon katmanları (Batch/Layer/Group/RMSNorm) modern derin öğrenmenin vazgeçilmezi. Transformer LayerNorm kullanır (Ders 7); fikir hep aynı: aktivasyon dağılımını kontrol altında tut.

11.7 Batch Normalization: İmplementasyon

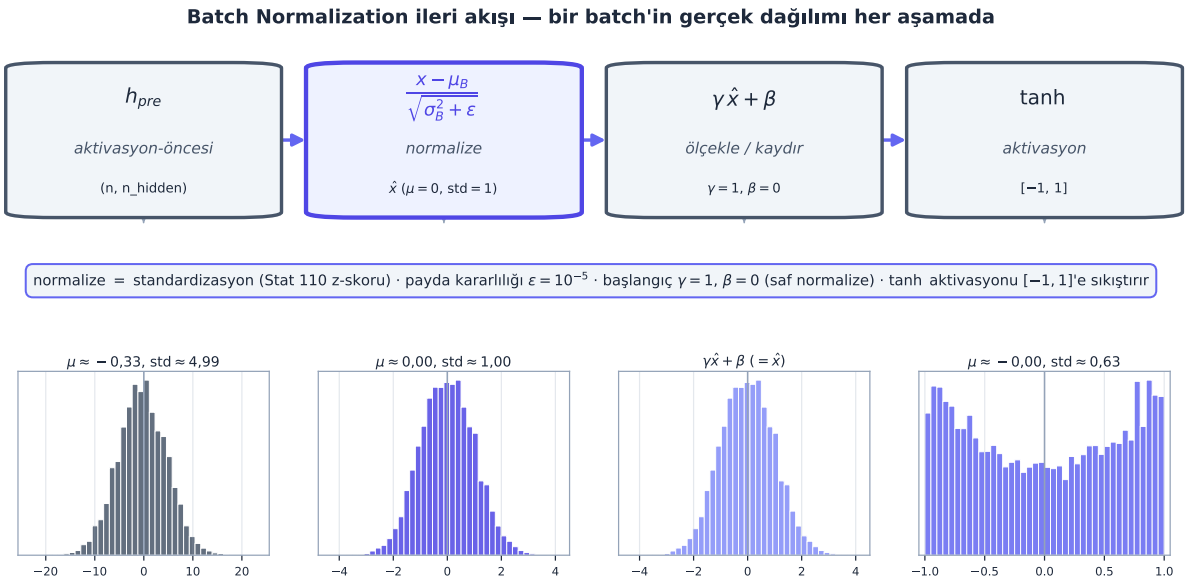
BatchNorm, gizli aktivasyon-öncesi `hpreact`'i batch boyutu üzerinden normalize eder. Her nöron için batch'in ortalaması μ_B ve varyansı σ_B^2 hesaplanır; sonra normalize edilir:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \qquad y_i = \gamma\,\hat{x}_i + \beta$$

Önemli ayrıntı: normalize edip bırakmayız. **Ölçekle ($ = $ bngain) ve kaydır ($ = $ bnbias)** ekleriz — bunlar öğrenilen parametrelerdir. Neden? Çünkü ağ bazen normalize-edilmemiş bir dağılım isteyebilir; γ/β ona normalizasyonu "geri alma" özgürlüğü verir. Başlangıçta $\gamma = 1$, $\beta = 0$ (saf normalize).

```
# hpreact: (batch, n_hidden) aktivasyon-oncesi
bnmean = hpreact.mean(0, keepdim=True)            # batch ortalamasi
bnstd  = hpreact.std(0, keepdim=True)             # batch std
hpreact = bngain * (hpreact - bnmean) / bnstd + bnbias   # normalize + olcekle/kaydir
h = torch.tanh(hpreact)                            # sonra aktivasyon
```

`bngain` (γ, başlangıçta 1'ler) ve `bnbias` (β, başlangıçta 0'lar) öğrenilen parametrelerdir. Normalize edilen `hpreact` artık 0 civarında, iyi yayılmış → `tanh` doymaz, gradyanlar akar. Aşağıdaki şema akışın her aşamasındaki gerçek dağılımı gösterir: önce geniş/kayık ($\mu \approx -0{,}33$, std $\approx 4{,}99$), normalize sonrası $\mu \approx 0$ std ≈ 1, `tanh` sonrası $[-1, 1]$ aralığına yayılmış (std $\approx 0{,}63$).



Şekil 11.6: BatchNorm ileri akış şeması: gizli aktivasyon-öncesi h_{pre} (n, n_hidden) → normalize $\hat{x} = (x - \mu_B)/\sqrt{\sigma_B^2 + \epsilon}$ (indigo vurgu) → ölçekle/kaydır $\gamma\hat{x} + \beta$ (γ=1, β=0 başlangıç) → tanh. Her aşamanın altında bir batch'in GERÇEK dağılımı: önce geniş/kayık ($\mu \approx -0{,}33$, std$\approx 4{,}99$), normalize sonrası $\mu \approx 0$ std≈ 1, tanh sonrası $[-1, 1]$ aralığına yayılmış (std$\approx 0{,}63$). $\epsilon = 10^{-5}$ payda kararlılığı; normalize adımı Stat 110 standardizasyonudur (z-skoru).

> **Builder Notu — Normalize + Öğrenilen Scale/Shift Deseni**
>
> **Geriye (Stat 110 + Ders 1):** Normalize formülü $(x - \mu)/\sigma$ doğrudan Stat 110 standardizasyonu; γ/β ise konum-ölçek dönüşümü ($X = \mu + \sigma Z$). $\gamma = 1/\beta = 0$ başlatmak, BatchNorm'un başta saf-normalize, sonra gerekirse öğrenip uyarlamasını sağlar.
> **İleriye:** γ (scale) ve β (shift), her normalizasyon katmanında (LayerNorm dahil) vardır. "Normalize et, sonra öğrenilen scale/shift ile geri-alma özgürlüğü ver" deseni tüm modern mimarilerde tekrarlanır.

11.8 BatchNorm Test Zamanı ve Running İstatistikler

BatchNorm'un bir sorunu var: eğitimde batch'in ortalama/std'sini kullanır. Ama **test zamanında** tek bir örnek tahmin etmek isteyebilirsin — batch yok, ortalama/std hesaplanamaz. Çözüm: eğitim boyunca istatistikleri **yürüyen (running)** bir ortalamayla biriktir, test zamanında onları kullan.

> *"[We estimate] the mean and standard deviation in a running manner during training of the neural net."* — Karpathy, 56:11

```
with torch.no_grad():   # bu istatistikler gradyan TASIMAZ
    bnmean_running = 0.999 * bnmean_running + 0.001 * bnmean
    bnstd_running  = 0.999 * bnstd_running  + 0.001 * bnstd
```

Bu bir **üstel hareketli ortalamadır** (EMA): her adımda mevcut tahmini biraz günceller. Test zamanında bu sabit `bnmean_running`/`bnstd_running` kullanılır (batch'e ihtiyaç kalmaz). Ayrıca: BatchNorm, bir örneğin çıktısını batch'teki **diğer örneklere** bağlar (istatistikler ortak) — bu küçük bir "jitter" (gürültü) katar ve hafif bir **regularization** etkisi yapar.

> Builder Notu — Buffer + Train/Eval Modu
>
> **Geriye (Stat 110 + Calculus):** Running mean/std, Stat 110'daki çevrimiçi ortalama tahmini; EMA katsayısı $(0{,}999)$ Calculus'taki üstel sönümün ayrık hâli. `torch.no_grad`: bu istatistikler öğrenilen parametre değil, **tampon** (buffer) — gradyan taşımaz.
> **İleriye:** Running stats / buffer kavramı tüm normalizasyon katmanlarında var. Train/eval modu ayrımı (`model.train()` / `model.eval()`) tam da bu yüzden kritik: BatchNorm eval'de running stats kullanır, dropout eval'de kapanır.

11.9 BatchNorm Detayları: epsilon ve Spurious Bias

İki ince nokta:

epsilon (ϵ): Bölme paydasında varyansa küçük bir ϵ eklenir: $\sqrt{\sigma^2 + \epsilon}$. Neden? Bir batch'te bir nöronun varyansı sıfıra çok yakınsa (tüm örnekler aynı), $\sqrt{\sigma} \approx 0$ olur ve bölme patlar (inf/nan). ϵ (örn. 10^{-5}) bunu önler.

Spurious (gereksiz) bias: BatchNorm aktivasyondan ortalamayı çıkarır. Ama bir önceki lineer katmanın bias'ı `b1` de aktivasyona sabit bir kayma ekler — ve bu kayma, BatchNorm'un ortalama-çıkarma adımında **tam olarak iptal olur**. Yani `b1` hiçbir işe yaramaz (gradyanı bile garip davranır). Çözüm: BatchNorm'dan önceki katmanda bias'ı **kaldır** (zaten $ = $ bnbias onun işini görür).

> Builder Notu — bias=False + epsilon = nan-Önleyici Ayrıntılar
>
> **İleriye:** "BatchNorm'dan önceki Linear'da bias=False" kuralı, PyTorch'ta da standarttır (`nn.Linear(..., bias=False) + nn.BatchNorm1d`). ϵ ise her normalizasyon katmanının sayısal-kararlılık parametresidir. Bu küçük ayrıntılar, "neden gradyanım nan oldu" hatalarının sık kaynağıdır.

11.10 BatchNorm Nereye Oturur + ResNet Örneği

BatchNorm tipik olarak şu sırayla kullanılır: **Lineer katman → BatchNorm → doğrusal-olmama** (örn. Linear → BatchNorm1d → Tanh). Karpathy gerçek bir örnek olarak **ResNet-50**'yi gösterir: derin görü ağlarında bu Linear/Conv → BatchNorm → ReLU bloğu tekrar tekrar istiflenir, üstüne **residual (artık) bağlantılar** eklenir.

Karpathy BatchNorm hakkında dürüst bir uyarı da yapar:

> *"No one likes this layer. It causes a huge amount of bugs, and intuitively it's because it is coupling examples in the forward pass of a neural net."* — Karpathy, 1:17:13

Yani BatchNorm güçlü ama tuzaklı: örnekleri batch içinde birbirine bağladığı için (bir örneğin çıktısı diğerlerine bağlı) çok sayıda ince bug'a yol açar. Bu yüzden modern mimariler (özellikle transformer) çoğunlukla **LayerNorm**'u tercih eder — o, her örneği bağımsız normalize eder.

> Builder Notu — Conv→BN→ReLU + Residual
>
> **İleriye:** BatchNorm → LayerNorm geçişi, Ders 7'de transformer'da somutlaşır. ResNet'in "Conv→BN→ReLU + residual" bloğu, derin öğrenmenin en etkili tasarım desenlerinden; residual bağlantıları Ders 7'de (transformer) yeniden göreceğiz. "BatchNorm örnekleri bağlar" sorunu, neden inference'ta running stats'a geçildiğinin de sebebi.

11.11 PyTorch Katman İçleri (Linear, BatchNorm1d)

Karpathy elle kurduğumuz katmanları PyTorch'un gerçek katmanlarıyla karşılaştırır: `torch.nn.Linear` ve `torch.nn.BatchNorm1d`. PyTorch'un dokümantasyonunu açıp, bizim yazdığımızla **birebir aynı** olduğunu gösterir — sadece daha çok ayar (kwarg) ile.

`nn.Linear(fan_in, fan_out, bias=True)`: tam olarak `x @ W + b`; ağırlıkları varsayılan olarak Kaiming-benzeri bir dağılımla başlatır.

`nn.BatchNorm1d(dim, eps=1e-5, momentum=0.1, affine=True, track_running_stats=True)`:

- `eps`: payda kararlılığı (bkz. BatchNorm Detayları).
- `momentum`: running istatistiklerin EMA katsayısı (bkz. Running İstatistikler).
- `affine`: γ (scale) ve β (shift) öğrenilsin mi (bkz. İmplementasyon).
- `track_running_stats`: running mean/std tutulsun mu (bkz. Running İstatistikler).

Yani kurduğumuz her parça PyTorch'ta bir kwarg'a karşılık geliyor — "kara kutu" değil, anladığımız mekanizma.

> Builder Notu — Önce Sıfırdan Kur, Sonra Kütüphaneyle Eşleştir
>
> **Geriye (Ders 1):** Bu, Ders 1'in "micrograd'ı PyTorch ile karşılaştır" anının tekrarı: önce sıfırdan kur, sonra kütüphaneyle eşleştir. `nn.Linear` = Ders 1'in $Wx + b$'si; `nn.BatchNorm1d` = bu derste kurduğumuz normalize+scale+shift+running.
> **İleriye:** Kütüphane katmanlarının kwarg'larını "anlayarak" kullanmak, production'da doğru tasarım

kararları (bias var/yok, eval modu, momentum) vermeni sağlar — kopyala-yapıştır yerine.

11.12 Kodu PyTorch-laştırma (Modüller)

Karpathy kodu, PyTorch'un `torch.nn` tarzında **Lego blokları**na (modüller) böler. Her modül bir `__call__` (forward) ve `parameters()` metoduna sahip — Ders 1'deki Neuron/Layer/MLP arabiriminin aynısı.

```
class Linear:
    def __init__(self, fan_in, fan_out, bias=True):
        self.weight = torch.randn((fan_in, fan_out)) / fan_in**0.5   # Kaiming
        self.bias = torch.zeros(fan_out) if bias else None
    def __call__(self, x):
        self.out = x @ self.weight
        if self.bias is not None:
            self.out = self.out + self.bias
        return self.out
    def parameters(self):
        return [self.weight] + ([] if self.bias is None else [self.bias])

class BatchNorm1d:
    def __init__(self, dim, eps=1e-5, momentum=0.1):
        self.eps = eps; self.momentum = momentum; self.training = True
        self.gamma = torch.ones(dim); self.beta = torch.zeros(dim)
        self.running_mean = torch.zeros(dim); self.running_var = torch.ones(dim)
    def __call__(self, x):
        if self.training:
            xmean = x.mean(0, keepdim=True)
            xvar = x.var(0, keepdim=True)
        else:
            xmean = self.running_mean; xvar = self.running_var
        xhat = (x - xmean) / torch.sqrt(xvar + self.eps)   # normalize
        self.out = self.gamma * xhat + self.beta           # olcekle/kaydir
        if self.training:
            with torch.no_grad():
                self.running_mean = (1-self.momentum)*self.running_mean + self.momentum*xmean
                self.running_var  = (1-self.momentum)*self.running_var  + self.momentum*xvar
        return self.out
    def parameters(self):
        return [self.gamma, self.beta]

class Tanh:
    def __call__(self, x):
        self.out = torch.tanh(x)
        return self.out
    def parameters(self):
```

```
        return []

# katmanlari istifle (torch.nn.Sequential gibi)
layers = [Linear(n_embd*block_size, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh(),
          Linear(n_hidden, vocab_size)]
```

Her modül `self.out`'u saklar (birazdan teşhis için), `training` bayrağı BatchNorm'un train/eval davranışını ayırır. Bu yapı, `torch.nn.Module` API'sinin birebir taklididir.

> Builder Notu — **call** + parameters() = Ders 1'in Arabirimi
>
> **Geriye (Ders 1):** `__call__` + `parameters()` arabirimi, Ders 1'in Neuron/Layer/MLP'sinin ta kendisi; artık katmanlar (Linear/BatchNorm1d/Tanh) ayrı bloklar. `bias=False` (BatchNorm'dan önce) §spurious-bias dersidir.
> **İleriye:** Bu modülerlik, `torch.nn.Module`'un tasarım felsefesi: her katman `forward` + `parameters` sunar, istiflenebilir. Tüm modern ağlar (transformer dahil) bu Lego-blok yaklaşımıyla kurulur.

11.13 Diagnostik: Aktivasyon ve Gradyan Histogramları

Modüller `self.out` sakladığı için artık ağın sağlığını **görselleştirebiliriz**. Karpathy iki temel histogram çizer:

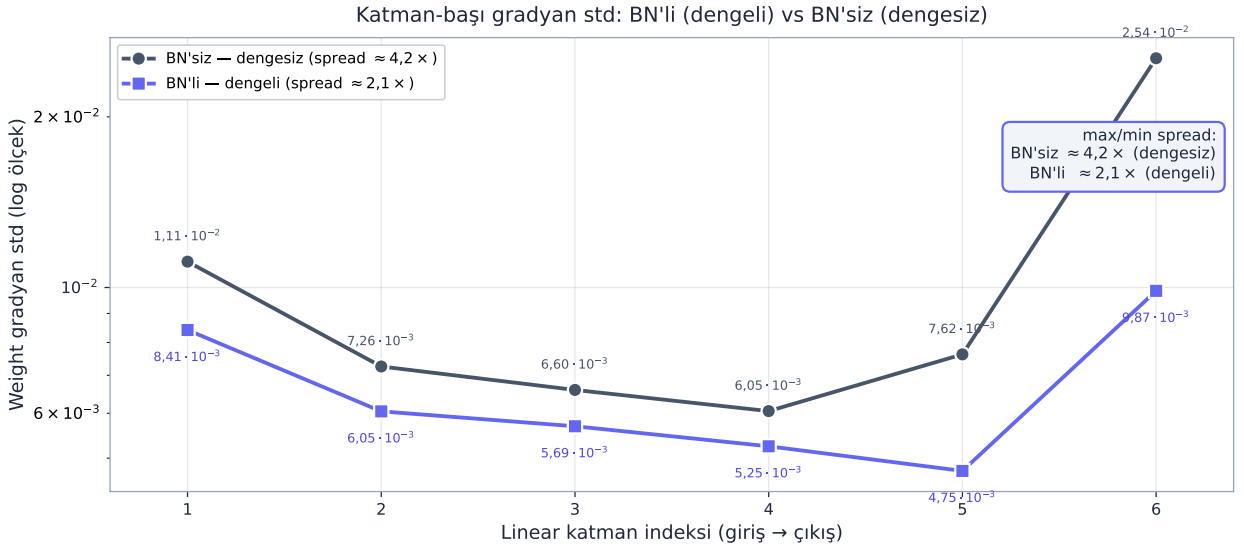
Aktivasyon histogramları (ileri geçiş): Her `tanh` katmanının çıktı dağılımını çiz. İyi bir ağda dağılım 0 civarında, makul yayılmış olmalı; ± 1'de yığılma (doygunluk) veya hep-0 (ölü) kötüdür. "Doygunluk yüzdesi" ($|\text{aktivasyon}| > 0{,}97$ olanların oranı) bir sağlık metriğidir.

Gradyan histogramları (geri geçiş): Her katmanın gradyan dağılımını çiz. Katmanlar arası gradyan büyüklükleri **birbirine yakın** olmalı; derinleştikçe sönüyorsa (vanishing) veya büyüyorsa (exploding) ağ dengesizdir. Kaiming init + BatchNorm tam da bunu dengeler. Aşağıdaki figür bunu ölçer: BN'siz ağda katman-başı gradyan std uçlarda yüksek-ortada düşük (dengesiz, max/min spread $\approx 4{,}2\times$), BN'li ağda çok daha düz (dengeli, spread $\approx 2{,}1\times$).

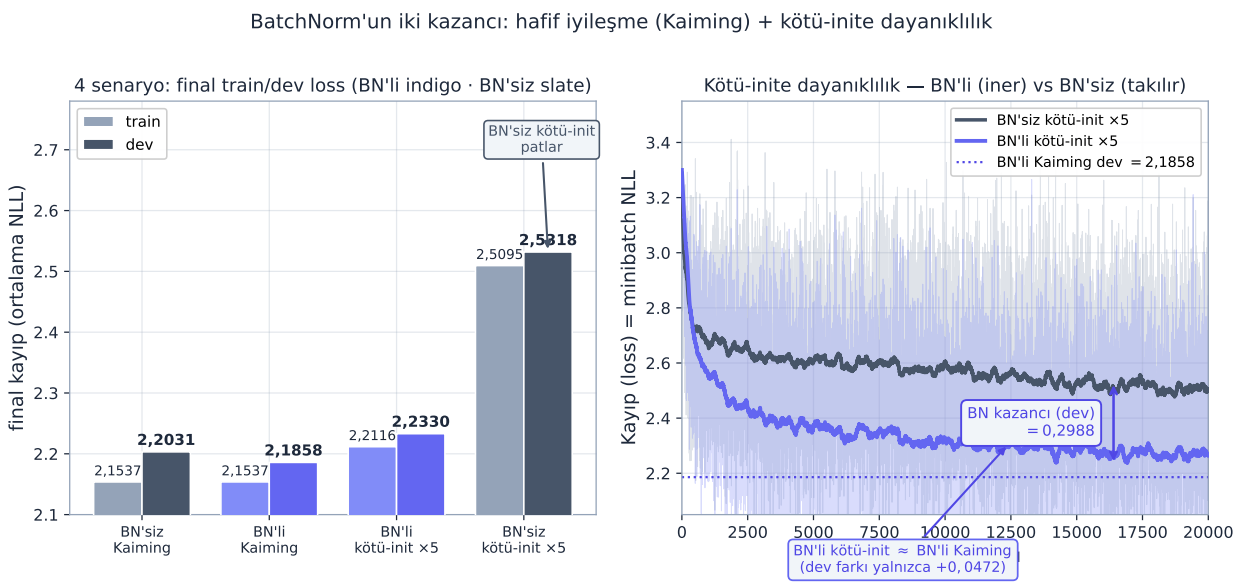
> *"If you're well-versed in the dark arts of backpropagation and have an intuitive sense of how these gradients flow through a neural net…"* — Karpathy, 15:04

Aynı 6 katmanlı ağ üzerinde, BatchNorm'un iki kazancını birlikte de görebiliriz: makul init'te hafif iyileşme + kötü init'e karşı dayanıklılık.

> Builder Notu — Gradyan Akışı = Ders 1 Zincir Kuralı
>
> **Geriye (Ders 1):** Gradyan akışı, Ders 1'in backprop zincir kuralıdır; her katmanda gradyan yerel türevle çarpılır, bu yüzden kötü init/aktivasyon gradyanı katlanarak söndürür/patlatır. Histogram, bunu gözle görmenin yolu.
> **İleriye:** Aktivasyon/gradyan histogramları, W&B/TensorBoard'da standart izleme panelleridir. "Gradyan normu katmanlar arası dengeli mi" sorusu, derin ağ debug'ının ilk adımı.



Şekil 11.7: Katman-başı gradyan std (6 Linear katmanı, log ölçek): **BN'siz** (slate) uçlarda yüksek, ortada düşük — dengesiz (max/min spread $\approx 4{,}2\times$); **BN'li** (indigo) çok daha düz — dengeli (spread $\approx 2{,}1\times$). BN'siz std'ler $1{,}11{\cdot}10^{-2} \to 7{,}26{\cdot}10^{-3} \to 6{,}60{\cdot}10^{-3} \to 6{,}05{\cdot}10^{-3} \to 7{,}62{\cdot}10^{-3} \to 2{,}54{\cdot}10^{-2}$; BN'li $8{,}41\cdot10^{-3} \to 6{,}05\cdot10^{-3} \to 5{,}69\cdot10^{-3} \to 5{,}25\cdot10^{-3} \to 4{,}75\cdot10^{-3} \to 9{,}87\cdot10^{-3}$. BatchNorm gradyanı katmanlar arası dengeler — sağlıklı akış. Deterministik (SEED); eğitilmiş ağlarda tek backward (`gradient_per_layer`, _verify_L4 §e).



Şekil 11.8: BatchNorm'un iki kazancı — **6 katmanlı derin ağ**, 20000 adım, lr $= 0{,}1$. **Sol (bar):** 4 senaryonun final train/dev loss'u. BN'siz Kaiming (slate) dev 2,2031; BN'li Kaiming (indigo) dev 2,1858 — biraz daha iyi. Asıl fark KÖTÜ init'te ($\times 5$): BN'siz kötü-init dev 2,5318 patlar, BN'li kötü-init dev 2,2330 — BN kötü init'i kurtarır. **Sağ (eğri):** BN'li vs BN'siz kötü-init eğitim loss eğrisi. BN'li (indigo) düşük platoya iner, BN'siz (slate) yüksekte takılır — kötü-inite dayanıklılık kanıtı, BN kazancı dev'de $+0{,}2988$. BN'li kötü-init, BN'li Kaiming'e neredeyse eşit (dev farkı yalnızca $+0{,}0472$): BN init'e büyük ölçüde duyarsız. Deterministik (SEED); gerçek değerler (`evaluate/train`).

11.14 Diagnostik: update:data Oranı

Üçüncü ve en kullanışlı metrik: **update:data oranı**. Her parametre için, bir adımdaki güncellemenin büyüklüğünü (lr$\times$ grad'ın std'si) parametrenin kendi std'sine böl, $\log_{10}$'unu al.

Kaba kural: bu oran ≈ -3 (yani 10^{-3}) olmalı — her adımda parametreler kendi büyüklüklerinin binde biri kadar değişir. Çok yüksekse (örn. -1) öğrenme oranı fazla büyük (kararsız); çok düşükse (örn. -5) ağ neredeyse öğrenmiyor. Bizim ölçümümüzde medyan $\approx -2{,}14$ (aralık $[-2{,}99,\ -1{,}89]$) — sağlıklı -3 bandının üst sınırında. Bu oran, öğrenme oranını ayarlamanın Ders 3'teki taramadan daha prensipli bir yoludur.

```
with torch.no_grad():   # olcum; gradyan tasimaz
    ud = (lr * p.grad.std() / p.data.std()).log10()   # update:data orani (log10)
# kaba kural: ud ~ -3 (binde bir). cok yuksek -> lr buyuk; cok dusuk -> ogrenmiyor
```

Builder Notu — update:data = lr'nin İnce Ayarı

Geriye (Ders 3): Bu, Ders 3'teki lr taramasının daha ince hâli: lr'yi loss eğrisine değil, parametre-güncelleme oranına göre ayarla. `torch.no_grad` ile ölçülür (gradyan taşımaz).
İleriye: update:data oranı (ve gradyan normu izleme), büyük model eğitiminde standart sağlık göstergesi — lr schedule ve gradient clipping kararları buna dayanır (Ders 10).

11.15 Linear-Collapse Notu ve Sonuç

Karpathy bir kenar not düşer: eğer katmanlar arasına **doğrusal-olmama koymazsan** (sadece Linear'lar), tüm derin ağ tek bir lineer katmana **çöker** (Ders 1'deki aynı gözlem: lineer $\circ$ lineer = lineer). Doğrusal-olmama (`tanh`/ReLU) derinliği anlamlı kılan şeydir.

Sonuç olarak: doğru **init** (Kaiming) + **normalizasyon** (BatchNorm), derin bir ağı "eğitilebilir" yapar. Aktivasyon/gradyan/update istatistiklerini izlemek ise ağın sağlığını teşhis etmenin yoludur — "çalışıyor" ile "iyi öğreniyor" arasındaki fark budur.

Builder Notu — Linear-Collapse = Ders 1 Kontrol Sorusu 2

Geriye (Ders 1): Linear-collapse, Ders 1 Kontrol Sorusu 2'nin tekrarı: aktivasyon olmadan $W_2(W_1 x) = (W_2 W_1)x = $ tek lineer katman (18.06 bileşke).
İleriye: Init + normalizasyon + residual bağlantılar (Ders 7), 100+ katmanlı ağları (ResNet, transformer) eğitilebilir kılan üçlü. Modern mimari tasarımı büyük ölçüde "gradyanı sağlıklı akıt" mühendisliğidir.

11.16 Bu Dersin Özeti

1. **Aktivasyon ve gradyan istatistikleri** önemlidir: kötü init, doygunluk veya patlayan/sönen gradyan eğitimi durdurur.

2. **Başlangıç loss'unu düzelt**: çıktı katmanını küçült (`W2 *= 0.01`, `b2 = 0`) → logitler ≈ 0 → ilk loss $\approx \log(27) \approx 3{,}3$ (bizde fix'siz 26,01, fix'li 3,31; hokey-sopası kaybolur).
3. **Doymuş `tanh` / ölü nöron**: çıktı ± 1'e yapışınca türev $(1 - \tanh^2) \approx 0$ → gradyan akmaz (kötü init doygunluğu $\approx \%79{,}1$ vs Kaiming $\approx \%19{,}5$). Aktivasyon-öncesini 0 civarında tut.
4. **Kaiming init**: std $=$ gain$/\sqrt{n_{in}}$; gain `tanh` için $5/3$, ReLU için $\sqrt{2}$. Aktivasyon varyansını katmanlar arası korur (Kaiming son katman std $0{,}654$ vs naif $0{,}320$).
5. **BatchNorm**: aktivasyonları batch üzerinden normalize ($\hat{x} = (x - \mu)/\sqrt{\sigma^2 + \epsilon}$) + öğrenilen ölçekle/kaydır (γ, β).
6. **Running stats**: test zamanı için EMA ile mean/std biriktir (`torch.no_grad`, buffer); train/eval modu ayrımı.
7. **epsilon** (payda kararlılığı) + **spurious bias** (BatchNorm öncesi Linear'da bias=False).
8. Kodu **modüllere** böl (Linear/BatchNorm1d/Tanh, `torch.nn` gibi); `self.out` ile **teşhis** (gradyan std BN'li spread $\approx 2{,}1\times$ vs BN'siz $\approx 4{,}2\times$; update:data $\approx 10^{-2{,}1}$).
9. BatchNorm güçlü ama tuzaklı (örnekleri bağlar; kötü init'te bile dev 2,2330 vs BN'siz 2,5318); transformer **LayerNorm** tercih eder (Ders 7).

> **!** Tek Bir Cümle
>
> Bir ağı eğitilebilir kılmak yalnızca mimariyi değil, aktivasyon ve gradyan istatistiklerini de doğru ayarlamaktır: doğru başlatma (Kaiming) ve normalizasyon (BatchNorm) sinyali katmanlar boyunca sağlıklı akıtır — ve bu istatistikleri izlemek (histogramlar, update:data oranı), "çalışıyor" ile "iyi öğreniyor" arasındaki farkı görmenin yoludur.

11.17 Kontrol Soruları

> **i** Soru 1: 27 karakterli bir dil modeli, eğitimin başında 'hiçbir şey bilmiyorsa' ilk loss kaç olmalı? Neden? Pratikte 26 gibi çok daha yüksek çıkarsa sebebi nedir?

Hiçbir şey bilmeyen model, 27 karaktere **eşit (uniform)** olasılık verir: her birine $1/27$. Loss = ortalama NLL:

$$\text{loss} = -\log\left(\frac{1}{27}\right) = \log(27) \approx 3{,}30$$

Cevap: $\approx 3{,}3$ olmalı. Pratikte bizim ölçümümüzde $\approx 26{,}01$ çıktı; çünkü rastgele başlatılan `W2/b2` aşırı uç logitler üretiyor — model "kendinden emin ama yanlış", bazı karakterlere çok yüksek/düşük olasılık atıyor. Bu, ilk birkaç iterasyonu boşa harcayan **hokey-sopası** loss eğrisine yol açar. Çözüm: `W2 *= 0.01`, `b2 = 0` ile logitleri 0'a yakın başlat → ilk loss $\approx 3{,}31$ (uniform beklenenin tam üstünde).

> **i** Soru 2: Kaiming init ile BatchNorm, ikisi de aktivasyon istatistiklerini düzeltir. Aralarındaki temel fark nedir?

Cevap: Kaiming init yalnızca **başlangıçta** ağırlıkları doğru ölçekleyerek aktivasyon varyansını 1 civarında tutmaya çalışır — ama eğitim ilerledikçe ağırlıklar değişir ve istatistik kayabilir; derin ağda doğru ayarlamak kırılgandır. **BatchNorm** ise her adımda, her ileri geçişte aktivasyonları **aktif olarak normalize eder** — yani "doğru dağılımı umut etmek" yerine "zorla dayatmak". BatchNorm init'e hassasiyeti büyük ölçüde ortadan kaldırır (bu yüzden çok kullanışlı): bizim ölçümümüzde kasten kötü

init ($\times 5$) ile bile BN'li ağ dev $2{,}2330$ verirken (BN'li Kaiming $2{,}1858$'e neredeyse eşit), BN'siz kötü-init dev $2{,}5318$'e patlar. Ama BatchNorm örnekleri batch içinde birbirine bağlar (bug kaynağı, jitter). İkisi birlikte de kullanılır: makul init + BatchNorm.

i Soru 3: 'Ölü nöron' ne zaman oluşur ve neden 'kalıcı' olabilir? tanh türevi ile bağla.

Cevap: `tanh`'ın türevi $1 - \tanh^2$'dir (Ders 1). Bir nöronun aktivasyon-öncesi değeri çok büyük/küçükse, `tanh` çıktısı ± 1'e yapışır (doygunluk) → türev ≈ 0. Eğer bir nöron **tüm batch boyunca** hep doymuşsa, geri yayılımda hep ≈ 0 gradyan alır → ağırlıkları asla güncellenmez → öğrenemez. "Kalıcı" çünkü güncellenmeyince doygunluktan çıkamaz; bir kısır döngü (Karpathy'nin deyişiyle "permanent brain damage"). Önlem: dikkatli init (Kaiming) veya BatchNorm ile aktivasyon-öncesini 0 civarında tutmak. (ReLU'da karşılığı: girdi hep negatif → çıktı hep 0 → ölü ReLU.)

i Soru 4: (Builder) BatchNorm'un normalize adımı (x – μ)/√(σ²+ε) Stat 110'daki hangi kavramdır? Neden ek olarak öğrenilen γ (scale) ve β (shift) eklenir?

Cevap: Normalize adımı tam olarak **standardizasyondur** (Stat 110 z-skoru): $z = (x-\mu)/\sigma$, ortalamayı 0, std'yi 1 yapar. BatchNorm bunu batch istatistikleriyle, ağın içinde yapar. γ **ve** β **neden?** Çünkü her zaman ortalama-0-std-1 istemeyebiliriz — ağ bazen farklı bir ölçek/konum isteyebilir (örn. `tanh`'ı belli bir bölgede kullanmak). γ (scale) ve β (shift), öğrenilen parametrelerdir ve modele normalizasyonu **kısmen veya tamamen geri alma** özgürlüğü verir (Stat 110 konum-ölçek dönüşümü: $X = \mu + \sigma Z$). Başlangıçta $\gamma = 1$, $\beta = 0$ (saf normalize); eğitimle ağ ne istediğini öğrenir. Yani BatchNorm "0-1 yap, ama gerekirse değiştirmene izin ver" der.

11.18 Egzersizler

Egzersiz 1 (Başlangıç loss'unu düzelt). Ders 3 MLP'sini al, ilk iterasyonun loss'unu yazdır (bizde $\approx 26{,}01 \gg 3{,}3$). Sonra `W2 *= 0.01` ve `b2 = 0` (veya `b2 *= 0`) yap, ilk loss'u tekrar ölç — $\approx \log(27) \approx 3{,}31$ çıkmalı. Loss eğrilerini karşılaştır: hokey-sopası kayboldu mu?

Egzersiz 2 (tanh doygunluğunu gör). Gizli katman aktivasyonlarının (`h = tanh(...)`) histogramını çiz. (a) Büyük `W1` ile: çoğu değer ± 1'de mi (doymuş)? (b) `W1`'i küçült (örn. Kaiming), histogramın 0 civarında yayıldığını gözlemle. $|h| > 0{,}97$ olanların yüzdesini (doygunluk oranı) hesapla (bizde kötü init $\approx \%79{,}1$, Kaiming $\approx \%19{,}5$).

Egzersiz 3 (Kaiming init). `std = gain / fan_in**0.5` (`tanh` için gain $= 5/3$) ile `W1`'i başlat. Ağ derinleştikçe (birkaç gizli katman) her katmanın aktivasyon std'sinin ≈ 1 civarında kaldığını doğrula — naif (Kaiming'siz) init ile karşılaştır (std katmanlar boyunca söner/patlar; bizde Kaiming son katman $0{,}654$, naif $0{,}320$).

Egzersiz 4 (BatchNorm kur). Modüller bölümündeki `BatchNorm1d` modülünü kur, gizli katmandan sonra ekle (Linear → BatchNorm1d → Tanh, Linear'da bias=False). Eğit, loss'u Kaiming-only ile karşılaştır. Sonra **kasten kötü bir init** dene (`W1`'i çok büyük) — BatchNorm'un ağı yine de kurtardığını (init'e duyarsızlık; bizde BN'li kötü-init dev $2{,}2330$ vs BN'siz kötü-init $2{,}5318$) gözlemle. `model.eval()` ile running stats'ı test et.

Egzersiz 5 (Sonraki dersin habercisi). Ders 2'den beri `loss.backward()` çağırıp gradyanları PyTorch'un autograd'ına bırakıyoruz. Ama **bu ağın** (embedding + Linear + BatchNorm + tanh + cross_entropy) her parametresinin gradyanını PyTorch tam olarak nasıl hesaplıyor, gerçekten biliyor musun? (a) cross_entropy'nin logitlere göre gradyanını elle türetmeyi dene (ipucu: softmax − one-hot). (b) BatchNorm'un backward'ının neden zor olduğunu düşün (μ ve σ tüm batch'e bağlı). Bu sorular, Ders 5'te (makemore 4: backprop ninja) `loss.backward()`'ı **kaldırıp** her gradyanı tensör düzeyinde **elle** yazmayı motive eder — backprop'u gerçekten anlamak için.

11.19 Sonraki Ders İçin Hazırlık

Ders 5: makemore 4 — Backprop Ninjası (Elle Geri Yayılım) — Andrej Karpathy

Bu derste `loss.backward()` bizim için tüm gradyanları hesaplıyordu — bir kara kutu gibi. Ders 5'te o kutuyu açıyoruz: **bu MLP+BatchNorm ağının** backward'ını PyTorch autograd olmadan, tensör düzeyinde **elle** yazacağız. Karpathy'nin deyişiyle backprop "sızdıran bir soyutlama" (leaky abstraction) — körü körüne `.backward()` çağırmak, ince bug'ları gözden kaçırmana yol açabilir.

Ana konular:

- "Backprop is a leaky abstraction": neden elle backward yazmak öğretici.
- Atomik hesaplama grafiği boyunca elle backward (cross_entropy, tanh, BatchNorm, matmul, embedding).
- Analitik cross-entropy ve BatchNorm gradyanları (tek satırlık formüller).

> ⚠ Ders 5 Öncesi Yapılacak
>
> - Egzersizleri çöz — özellikle 5 (cross_entropy gradyanını elle türet: softmax − one-hot).
> - Ders 1'in micrograd backward'ını hatırla: her işlemin yerel türevi, zincir kuralıyla geriye taşınır.
> - "Aktivasyon/gradyan istatistiklerini izle" ve "Kaiming + BatchNorm" cümlelerini hatırla.

11.20 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Karpathy'de
Başlangıç loss'u	Beklenen $\approx \log(\text{vocab})$; logitleri 0'a yakın başlat (W2 küçült, b2=0)	4m16
Hokey-sopası eğrisi	İlk iterasyonlar aşırı logitleri ezmekle geçer; kötü init belirtisi	11m48
Doymuş tanh	Çıktı ± 1'e yapışır; türev $(1 - \tanh^2) \approx 0$; gradyan akmaz	13m04
Ölü nöron	Tüm batch'te doymuş nöron hiç güncellenmez (kalıcı)	19m26

Kavram	Tanım	Karpathy'de
Kaiming (He) init	std $=$ gain $/\sqrt{n_{in}}$; gain tanh$\rightarrow 5/3$, ReLU$\rightarrow \sqrt{2}$; varyansı korur	27m53
Batch Normalization	Aktivasyonu batch'te normalize: $(x-\mu)/\sqrt{\sigma^2+\epsilon}$, sonra $\gamma\hat{x}+\beta$	40m43
Running stats (EMA)	Test için mean/std biriktir (buffer, torch.no_grad); train/eval modu	54m02
epsilon + spurious bias	ϵ payda kararlılığı; BatchNorm öncesi Linear'da bias gereksiz (=False)	1h00m
γ **(scale) /** β **(shift)**	Öğrenilen ölçekle-kaydır; normalizasyonu geri-alma özgürlüğü	40m43
Modülleştirme	Linear/BatchNorm1d/Tanh blokları (call, parameters); torch.nn gibi	1h18m
Aktivasyon/gradyan histogramı	Doygunluk % ve katmanlar arası gradyan dengesi teşhisi	1h26m
update:data oranı	$\log_{10}(\text{lr}\cdot \text{grad\_std}/\text{param\_std}) \approx -3$ olmalı; lr sağlık metriği	1h26m

11.21 ML Builder Bağlantıları

9 köprü — BatchNorm

1. **Başlangıç loss'u** → Stat 110 uniform NLL $=\log(n)$ + Ders 3 Egzersiz 5. İleriye: model sanity-check.
2. **Doymuş tanh / ölü nöron** → Ders 1 tanh türevi $(1-\tanh^2)=0$. İleriye: ReLU/GELU, residual bağlantılar.
3. **Kaiming init** → Stat 110 varyans/Normal dağılım + **fast.ai L17 varyans korunumu (aynı matematik, farklı çatı)**. İleriye: PyTorch nn.init, derin ağ eğitilebilirliği.
4. **BatchNorm = standardizasyon** → Stat 110 z-skoru $(x-\mu)/\sigma$. İleriye: LayerNorm (Ders 7), RMSNorm.
5. **Running stats (EMA)** → Stat 110 çevrimiçi ortalama + Calculus üstel sönüm. İleriye: train/eval

modu, model.eval().

6. **Modülleştirme** → Ders 1 Neuron/Layer/MLP arabirimi. İleriye: torch.nn.Module, tüm modern mimariler.
7. **Gradyan akışı/histogram** → Ders 1 backprop zincir kuralı. İleriye: vanishing/exploding gradient, gradient clipping (Ders 10).
8. **update:data oranı** → Ders 3 lr taraması. İleriye: learning rate schedule, gradient norm izleme (Ders 10).
9. **Linear-collapse** → Ders 1 Kontrol Sorusu 2 (lineer ∘ lineer = lineer, 18.06). İleriye: derinliğin neden doğrusal-olmama gerektirdiği.

11.22 Karpathy'nin Önerdiği Kaynaklar

Karpathy'nin bu ders için verdiği kaynaklar:

- **Kaiming (He) init makalesi:** arxiv 1502.01852 — "Delving Deep into Rectifiers" (He ve ark. 2015).
- **Batch Normalization makalesi:** arxiv 1502.03167 — Ioffe & Szegedy 2015.
- **Ek kaynak:** arxiv 2105.07576.
- **Ders Colab notebook'u:** Google Colab — çalıştırılabilir ortam.

! Bu dersten tek bir şey alıp gideceksen

Bir ağı eğitilebilir kılmak yalnızca mimariyi değil, **aktivasyon ve gradyan istatistiklerini** de doğru ayarlamaktır. Doğru başlatma (Kaiming) sinyali katmanlar boyunca korur; Batch Normalization onu zorla dayatır; ve aktivasyon/gradyan/update istatistiklerini izlemek, "ağ çalışıyor" ile "ağ sağlıklı öğreniyor" arasındaki farkı görmenin tek yoludur.

12 makemore 4 — Backprop Ninjası (Elle Geri Yayılım)

loss.backward() sihir değil; zincir kuralının bir hesaplama grafiği boyunca tensör düzeyinde uygulanmasıdır — onu bir kez elle yazmak, backprop'u kullanan biriyle anlayan biri arasındaki farkı koyar

> Bölüm bilgisi
>
> - **Karpathy'nin videosu:** YouTube — Building makemore Part 4: Becoming a Backprop Ninja (≈115 dk)
> - **Seri:** Neural Networks: Zero to Hero — Ders 5
> - **Hoca:** Andrej Karpathy
> - **Kaynak repo:** github.com/karpathy/makemore
> - **Okuma süresi:** ≈34 dk

12.1 Bu Derste Ne Var?

Ders 2'den beri `loss.backward()` çağırıp tüm gradyanları PyTorch'un autograd'ına bırakıyorduk — bir kara kutu gibi. Bu derste o kutuyu **açıyoruz**: Ders 4'ün MLP+BatchNorm ağının backward'ını PyTorch autograd olmadan, **tensör düzeyinde elle** yazıyoruz. Bu, Ders 1'deki micrograd'ın sırrının gerçek-ölçek hâli.

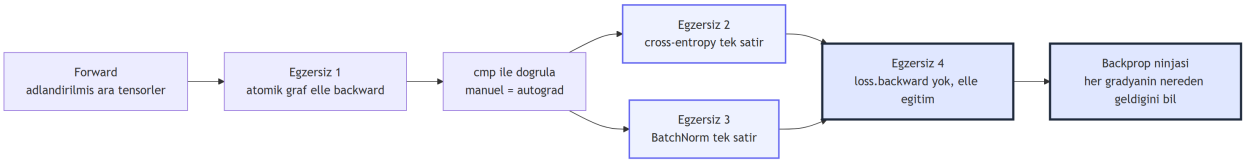
> *"I'd like to call backpropagation a leaky abstraction."* — Karpathy, 1:00

Karpathy'nin tezi: backprop **sızdıran bir soyutlamadır** (leaky abstraction). `loss.backward()`'ı körü körüne çağırmak, gradyanlarda ince hataları (ölü nöron, doygunluk, patlayan gradyan) gözden kaçırmanıza yol açar. Gradyanları bir kez elle yazarsan, bir daha asla onlara kara kutu gibi bakmazsın.

> *"The line of code here that I take an issue with is loss.backward()."* — Karpathy, 0:34

Ders dört egzersiz olarak kurulu:

1. **Egzersiz 1** — atomik hesaplama grafiği boyunca (logprobs'tan C'ye) her ara tensörün gradyanını elle yaz.
2. **Egzersiz 2** — cross-entropy'nin tek-satırlık analitik backward'ı (softmax − one-hot).
3. **Egzersiz 3** — BatchNorm'un tek-satırlık füzyonlu backward'ı.
4. **Egzersiz 4** — hepsini birleştir: `loss.backward()` olmadan tam eğitim.



Şekil 12.1: Ders 5'in kavram haritası: adlandırılmış ara tensörlerle forward'tan başlar, Egzersiz 1 atomik grafı elle geri yürür, Egzersiz 2/3 onu tek-satırlık füzyonlu formüllere indirir, Egzersiz 4 ise loss.backward()'ı tamamen kaldırıp elle gradyanlarla eğitir. Slate akış + indigo dönüm noktaları (füzyonlu formüller ve autograd'sız eğitim).

Builder Notu — Ders 1'in Sırrı, Tensör Ölçeğinde

Geriye (Ders 1-4 + Calculus):

- **Bu, Ders 1'in sırrı.** micrograd'da her `_backward` yereldi; burada aynı zincir kuralını tensör düzeyinde (matris/batch) uyguluyoruz — ama elle, autograd olmadan.
- **Ağ = Ders 4.** Yeni mimari yok; Ders 4'ün embedding + Linear + BatchNorm + `tanh` + `cross_entropy` ağının backward'ı.
- **Tek araç = Calculus zincir kuralı.** Her adım, Calculus'un zincir kuralının tensör/matris hâli; matris türevleri (transpose'lar) 18.06 ile gelir.

İleriye: Gradyanları elle türetebilmek, herhangi bir framework'ü derinden anlamanın ve gradyan bug'larını teşhis etmenin temelidir. Özel bir katman/loss yazdığında (production'da sık) backward'ı da yazman gerekir. "Sezgisel olarak gradyanın nasıl aktığını bilmek", Ders 4'ün aktivasyon/gradyan teşhisini de mümkün kılan şey.
Tek cümleyle: `loss.backward()` sihir değil; zincir kuralının bir hesaplama grafiği boyunca tensör düzeyinde uygulanmasıdır — ve onu bir kez elle yazmak, backprop'u "kullanan" biriyle "anlayan" biri arasındaki farkı koyar.

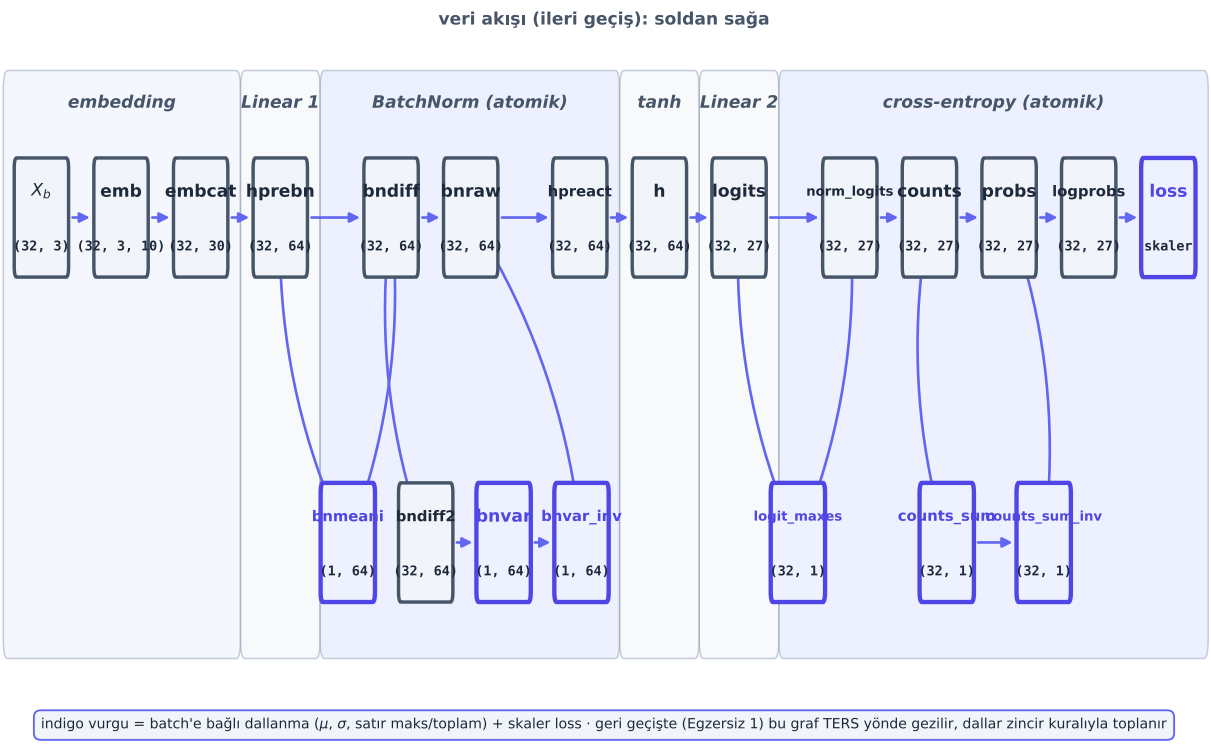
12.2 Neden Elle Backprop? Sızdıran Soyutlama

Bir soyutlama, altındaki detayları gizlemeli. Ama backprop **sızdırır**: `loss.backward()`'ı çağırırsın ama gradyanların nasıl aktığını anlamazsan, neden öğrenmediğini (ölü nöron, vanishing gradient, kötü init) teşhis edemezsin.

> *"I think people sort of understood how these neural networks work on a very intuitive level, so I think it's a good exercise [to write the backward pass by hand]."* — Karpathy, 6:58

Karpathy'nin amacı: gradyanları bir kez elle yazarak, autograd'ın ne yaptığını sezgisel bir düzeyde anlamak. Bundan sonra `loss.backward()`'a baktığında, arkasında ne olduğunu görürsün.

Bu dersin bütün omurgası, Ders 4 ağının ileri geçişini bir **hesaplama grafiğine** açmaktır. Forward'ı bir sürü adlandırılmış ara tensöre (logprobs, probs, counts, bnraw, …) böleriz; bu graf, geri geçişte tersten yürüyeceğimiz harita olur. Aşağıdaki figür o ileri grafı gösterir.



Şekil 12.2: İleri geçiş hesaplama grafiği (adlandırılmış ara tensörler, $n = 32$): soldan sağa veri akışı. Aşamalar — **embedding** ($C[X_b] \to$ emb $\to$ embcat) $\to$ **Linear 1** (hprebn $=$ embcat $W_1 + b_1$) $\to$ **BatchNorm (atomik)** (bnmeani $\to$ bndiff $\to$ bndiff2 $\to$ bnvar $\to$ bnvar_inv $\to$ bnraw $\to$ hpreact) $\to$ **tanh** (h) $\to$ **Linear 2** (logits) $\to$ **cross-entropy (atomik)** (logit_maxes $\to$ norm_logits $\to$ counts $\to$ counts_sum $\to$ counts_sum_inv $\to$ probs $\to$ logprobs $\to$ loss). İndigo vurgu = batch'e bağlı dallanma düğümleri (BatchNorm μ/σ, cross-entropy satır maks/toplam) ve skaler loss; bu noktalarda akış birden çok yola ayrılır, geri geçişte zincir kuralı bu yolları toplar. Her düğümde tensör adı + GERÇEK şekil (`forward_named` cache'inden). Bu, Notion §3'teki adlandırılmış ara tensör listesinin tensör hâli — Ders 1 micrograd Value ağacının batch sürümü.

> Builder Notu — Sızdıran Soyutlama: Aleti Anlayarak Kullan
>
> **İleriye:** "Sızdıran soyutlama" kavramı (Joel Spolsky) yazılımın her yerinde geçerli: bir aleti güvenle kullanmak için altında ne olduğunu yeterince bilmen gerekir. ML'de bu, gradyan bug'larını (NaN, vanishing, ölü nöron) teşhis edebilmek demek — production'da paha biçilmez.

12.3 Tarihsel Not ve cmp Yardımcısı

Karpathy hatırlatır: autograd'dan önce ('10'ların başı), herkes backward pass'i **elle yazıyordu** — kağıt-kalem, calculus ile her katmanın gradyanını türetmek standarttı.

> *"[We used] pen and paper and mathematics and calculus to derive the gradient through the batchnorm layer."* — Karpathy, 11:18

Elle yazdığımız gradyanları doğrulamak için bir **`cmp`** (compare) yardımcısı kullanırız: bizim manuel gradyanımızı, PyTorch'un `.grad`'ıyla karşılaştırır (tam eşit mi, yaklaşık mı, maksimum fark ne).

```
def cmp(s, dt, t):
    ex = torch.all(dt == t.grad).item()          # tam esit mi
    app = torch.allclose(dt, t.grad)             # yaklasik mi
    maxdiff = (dt - t.grad).abs().max().item()   # maksimum fark
    print(f'{s:15s} | exact: {str(ex):5s} | approx: {str(app):5s} | maxdiff: {maxdiff}')
```

Forward pass, gradyanları elle hesaplayabilmek için **bir sürü adlandırılmış ara tensöre** (logprobs, probs, counts, norm_logits, logit_maxes, hpreact, bnraw, bndiff, …) bölünür. Ara tensörlerin gradyanını saklamak için `retain_grad()` kullanılır (yalnızca yaprak tensörler varsayılan olarak `.grad` tutar).

> Builder Notu — cmp = Tensör Düzeyinde Gradient Check
>
> **Geriye (Ders 1):** `cmp`, Ders 1'in **gradient check**'inin (sayısal vs analitik) tensör düzeyindeki hâli — ama burada "analitik elle" ile "PyTorch autograd" karşılaştırılıyor. `retain_grad`, ara tensörlerin gradyanını saklatmak için (verimlilik için PyTorch bunları normalde atar).
> **İleriye:** Gradient checking, özel bir katman/loss yazdığında backward'ını doğrulamanın standart yoludur (PyTorch `torch.autograd.gradcheck`). Yeni bir CUDA kernel'ı veya custom `autograd.Function` yazan herkes bunu kullanır.

12.4 Egzersiz 1: Atomik Graf Boyunca Elle Backward

İlk egzersiz: forward pass'i parçaladığımız her ara tensörün gradyanını, çıktıdan (loss) girişe (C) doğru **elle** yaz. Bu, micrograd'ın `backward()`'ının tensör düzeyinde, autograd olmadan yapılması.

Örnekle başlayalım. Loss, doğru hedeflerin log-olasılıklarının ortalamasının negatifi:

```
loss = -logprobs[range(n), Yb].mean()
```

Bunun `logprobs`'a göre gradyanı: yalnızca seçilen (range(n), Yb) konumlarında $-1/n$, diğer her yerde 0:

$$\frac{\partial L}{\partial \text{logprobs}[i,j]} = \begin{cases} -\dfrac{1}{n} & j = y_i \\ 0 & \text{aksi halde} \end{cases}$$

```
dlogprobs = torch.zeros_like(logprobs)
dlogprobs[range(n), Yb] = -1.0 / n   # sadece hedef konumlar
cmp('logprobs', dlogprobs, logprobs)   # PyTorch ile karsilastir: exact True
```

Sonra zincir kuralıyla geriye yürürüz: `logprobs = probs.log()` olduğundan `dprobs = (1/probs) * dlogprobs`; sonra `counts_sum_inv`, `counts_sum`, `counts`, `norm_logits`, `logit_maxes`, `logits`, … her biri için yerel türev × gelen gradyan. Her adımda `cmp` ile PyTorch'a karşı doğrularız (exact True olmalı). 22'den fazla ara gradyan; sıkıcı ama her biri Ders 1'in tek kuralı: **yerel türev, zincir kuralıyla geriye taşınır**. Bu graf, fig-forward-dag'ın TERS yönüdür; aşağıda gradyan akışı kökten (loss) yapraklara (C) çizilmiştir.

Şekil 12.3: Geri geçiş: gradyan **kökten** (loss) **yapraklara** (C) akar — fig-forward-dag'ın AYNI grafının ters yönü. Köşeden $d$logprobs kıvılcımı başlar: seçilen $(\text{range}(n), Y_b)$ konumlarında $-\frac{1}{n} = -0{,}03125$, gerisi 0. Her düğümün üstündeki indigo etiket o düğümün **yerel türevi**dir; zincir kuralıyla gelen gradyanla çarpılarak geriye taşınır: log $\to \frac{1}{\text{probs}}$, exp $\to$ counts, max $\to$ onehot, matmul $\to @W^\top$ / $@h^\top$, tanh $\to (1-h^2)$, BN ölçek $\to$ bngain·, sum $\to$ dağıt, broadcast $\to$ topla. Yapraktaki dC bir **scatter-add**'tir: aynı satıra gelen gradyanlar toplanır. Bu grafın TÜM ara gradyanları (26 tensör) PyTorch autograd ile **birebir eşleşir** — gerçek cmp: exact True, maxdiff 0,0.

Bizim ölçümümüzde bu batch'in forward loss'u 3,408828; Egzersiz 1'in TÜM ana gradyanları (dlogprobs $\to$ dC/dW1/db1/dW2/db2/dbngain/dbnbias) PyTorch autograd ile **birebir** (exact True, maxdiff 0,0) eşleşir. Bu, dersin sayısal-dürüstlük çekirdeği: elle yazdığımız her gradyan, kara-kutu autograd'ın hesapladığıyla aynıdır.

Builder Notu — micrograd'ın Topolojik backward'ı, Tensör Hâli

Geriye (Ders 1): Bu tam olarak micrograd'ın topolojik-sıralı `backward()`'ı — ama elle, her tensör için. Toplama gradyanı dağıtır, çarpma diğerini taşır, exp/log kendi yerel türevini uygular (Ders 1'in kuralları, batch boyutunda).
İleriye: Bir gradyanı doğru türetip `cmp` ile "exact True" almak, gradient checking'in özüdür — özel bir işlemin backward'ını yazdığında doğruluğunu böyle kanıtlarsın.

12.5 Lineer Katman (matmul) Backward'ı

En önemli ve en sık karşılaşılan: matris çarpımının backward'ı. `C = A @ B` (artı bias) ileri geçişi için, gradyanları **ilk ilkelerden** (boyut uyumu + küçük bir örnek) türetiriz. Karpathy anahtar sezgiyi verir: gradyan formülünde **transpoze** belirir.

$$C = AB \;\Rightarrow\; \frac{\partial L}{\partial A} = \frac{\partial L}{\partial C}\,B^{\top}, \qquad \frac{\partial L}{\partial B} = A^{\top}\frac{\partial L}{\partial C}$$

> *"…multiplying B, but B transpose actually. You see that B21 and B12 have changed [places]."*
> — Karpathy, 47:30

Bias için (her satıra eklenen, broadcast olan `b`), gradyan batch boyutu üzerinden **toplanır**: `db = dC.sum(0)`.

```
# C = A @ B + b  ileri gecis icin backward:
dA = dC @ B.T          # (n,p) = (n,m) @ (m,p)  -> dC @ B^T, A ile ayni sekil
dB = A.T @ dC          # (p,m) = (p,n) @ (n,m)  -> A^T @ dC, B ile ayni sekil
db = dC.sum(0)         # bias broadcast oldugu icin toplanir
```

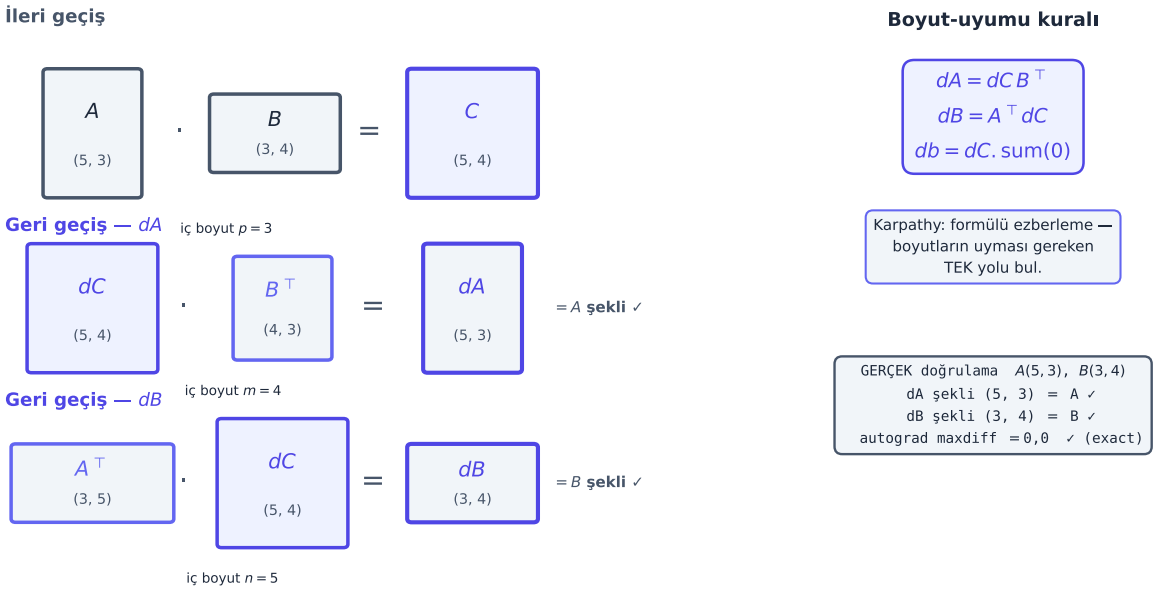
Hangi transpoze nereye? Karpathy'nin pratik kuralı: **boyutların uyması gereken tek bir yol vardır**. dA, A ile aynı şekilde olmalı; dC (n, m) ve B (p, m) verildiğinde, $dA = dC\,B^{\top}$ tek tutarlı çarpımdır. Bu "boyut uyumu" sezgisi, matris gradyanlarını ezberlemeden türetmeni sağlar.

Builder Notu — Transpoze: 18.06 + Ders 1 'Diğerini Geçir'

Geriye (18.06 + Ders 1): Transpoze'lar 18.06'dan (matris çarpımının nasıl çalıştığı); "diğerini geçir" kuralının (Ders 1 çarpma backward'ı) matris hâli. Bias toplama, broadcast'in tersi: ileri geçişte yayılan, geri geçişte toplanır (broadcasting/sum dualitesi).
İleriye: $dA = dC\,B^{\top}$, $dB = A^{\top}dC$ — bu iki satır, her sinir ağının her lineer katmanının backward'ı. GPU'da bu çarpımlar (GEMM) eğitimin FLOP'larının çoğu; forward bir matmul, backward iki matmul.

12.6 tanh ve BatchNorm scale/shift Backward

tanh backward. Ders 1'den tanıdık: `h = tanh(hpreact)` için yerel türev $1 - h^2$. Gelen gradyanla çarp:

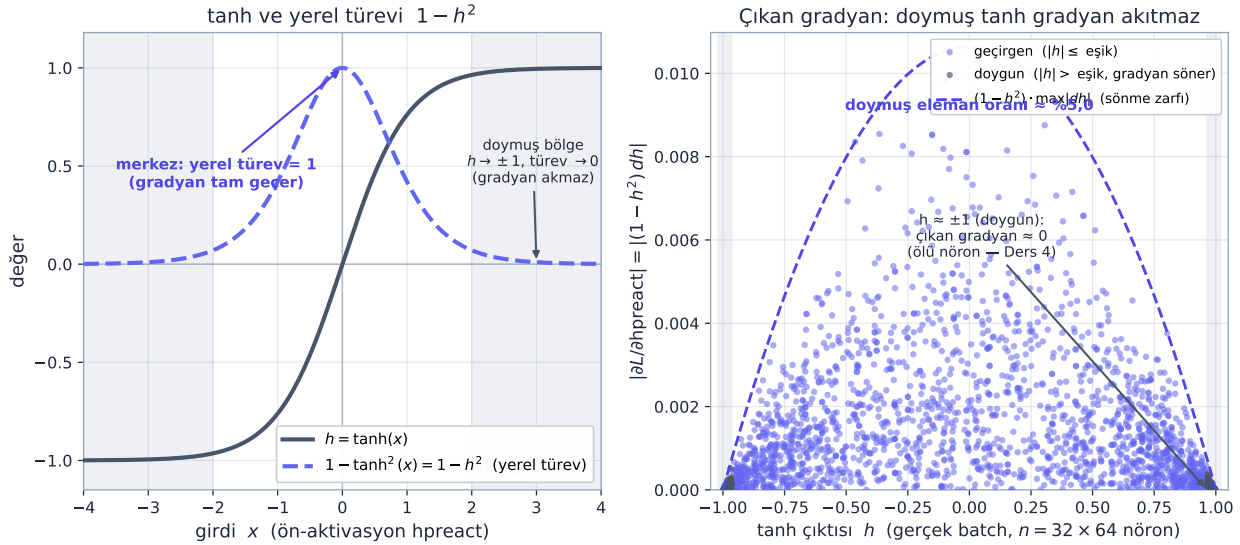


Şekil 12.4: matmul backward, boyut-uyumu kuralı. $C = AB$ için (A şekli (n,p), B şekli (p,m), dolayısıyla C ve dC şekli (n,m)) gradyanlar TEK tutarlı çarpımdan çıkar: $dA = dC\,B^\top = (n,m)(m,p) = (n,p)$ (A ile aynı şekil), $dB = A^\top dC = (p,n)(n,m) = (p,m)$ (B ile aynı şekil). Transpoze, iç boyutların uyması için zorunludur — Karpathy'nin kuralı: formülü ezberleme, boyutların uyması gereken tek yolu bul. Bias broadcast olduğu için geri geçişte toplanır: $db = dC.\text{sum}(0)$, şekil $(m,)$. GERÇEK doğrulama ($A(5,3)$, $B(3,4)$): dA şekli A ile, dB şekli B ile aynı; autograd ile maxdiff $= 0{,}0$ (exact).

$$\frac{\partial L}{\partial \text{hpreact}} = (1 - h^2) \odot \frac{\partial L}{\partial h}$$

```
dhpreact = (1.0 - h**2) * dh   # tanh backward (Ders 1'in 1-tanh^2'si)
```

Yerel türev $1 - h^2$, gelen gradyanı bir "süzgeç" gibi geçirir: $h \approx 0$ olan (aktif, eğimli) nöronlarda türev ≈ 1 (gradyan tam geçer), $h \approx \pm 1$ olan (doymuş) nöronlarda türev ≈ 0 (gradyan akmaz). Bu, Ders 4'ün ölü-nöron köprüsünün backward tarafıdır.



Şekil 12.5: tanh backward: yerel türev $1-h^2$ ile gelen gradyanı süzer — $\partial L/\partial \text{hpreact} = (1-h^2) \odot \partial L/\partial h$. **Sol:** $h = \tanh(x)$ (slate, düz) ve yerel türevi $1 - \tanh^2(x) = 1 - h^2$ (indigo, kesik). Merkezde türev en büyük ($= 1$, geçirgen); $|x| \gtrsim 2$ olan **doymuş bölge** gölgeli — orada $h \to \pm 1$, türev $\to 0$, gradyan akmaz. **Sağ:** GERÇEK batch'ten h değerleri ($n = 32$ örnek $\times$ 64 nöron) için gelen gradyan $\partial L/\partial h$ yerel türev $1 - h^2$ ile çarpılıp $\partial L/\partial$hpreact olur; $h \approx \pm 1$ (doygun) noktalarda çıkan gradyan ≈ 0 (indigo, sönen) — bu, Ders 4'ün **ölü-nöron** köprüsü: doymuş tanh gradyan geçirmez.

BatchNorm scale/shift backward. İleri geçişte `hpreact = bngain * bnraw + bnbias` ($\gamma \cdot \hat{x} + \beta$). Üç gradyan:

- `bnbias` (β) tüm batch'e broadcast eklendiği için gradyan **toplanır**: `dbnbias = dhpreact.sum(0)`.
- `bngain` (γ) `bnraw` ile çarpıldığı için: `dbngain = (bnraw * dhpreact).sum(0)`.
- `bnraw` ($\hat{x}$): `dbnraw = bngain * dhpreact`.

```
dbngain = (bnraw * dhpreact).sum(0, keepdim=True)   # carpma + broadcast toplami
dbnbias = dhpreact.sum(0, keepdim=True)             # broadcast toplami
dbnraw  = bngain * dhpreact                         # digerini gecir
```

Desen tekrar ediyor: **broadcast olan terim (γ, β) geri geçişte toplanır; çarpım diğer operandı geçirir** (Ders 1).

Builder Notu — Hep Aynı Zincir Kuralı

Geriye (Ders 1): `tanh` türevi $1 - h^2$ doğrudan Ders 1; γ/β gradyanları "çarpma diğerini geçirir + broadcast toplanır" kurallarının (Ders 1 + matmul backward) BatchNorm'a uygulanması. Hiç yeni kural yok, hep aynı zincir kuralı.
İleriye: Doymuş `tanh`'ın gradyanı geçirmemesi (yerel türev $\to 0$), Ders 4'ün ölü-nöron teşhisinin tam matematiğidir — backward'ı elle yazınca neden öyle olduğunu görürsün.

12.7 Bessel Düzeltmesi (BatchNorm Varyansı)

Karpathy bir kenar notu açar: BatchNorm'da varyansı hangi formülle hesaplamalı?

> *"Brief digression: Bessel's correction in BatchNorm."* — Karpathy, 1:05:14

İki seçenek var:

$$\sigma^2_{\text{biased}} = \frac{1}{n}\sum_i (x_i - \mu)^2, \qquad \sigma^2_{\text{unbiased}} = \frac{1}{n-1}\sum_i (x_i - \mu)^2$$

$n-1$'e bölen **yansız (unbiased)** tahmindir (Bessel düzeltmesi); n'e bölen **yanlı (biased)**. İncelik: PyTorch BatchNorm, eğitim sırasında normalizasyonda **biased** $(1/n)$ kullanır, ama **running varyansı** tutarken **unbiased** $(1/(n-1)$, Bessel) kullanır. Küçük batch'lerde fark önemlidir; bu, manuel backward'ı PyTorch'a tam eşitlemek için bilinmesi gereken bir ayrıntı. Füzyonlu BatchNorm formülünde (Egzersiz 3) Bessel düzeltmesi $\frac{n}{n-1}$ olarak belirir — $n = 32$ batch'inde bu $\approx 1{,}03$'tür.

Builder Notu — Yansız Varyans: Stat 110

Geriye (Stat 110): Yansız vs yanlı varyans tahmini, Stat 110'un klasik konusu: örneklem varyansında n yerine $n-1$ (Bessel), tahmincinin yansızlığını sağlar. BatchNorm bağlamında bu, batch istatistiğinin popülasyonu tahmin etme şeklidir.
İleriye: Bu tür "kütüphane hangi formülü kullanıyor" ayrıntıları, manuel implementasyonu referansla eşleştirirken (veya iki framework arası taşırken) NaN/uyuşmazlık hatalarının kaynağıdır. Küçük batch + yanlış varyans = kararsız BatchNorm.

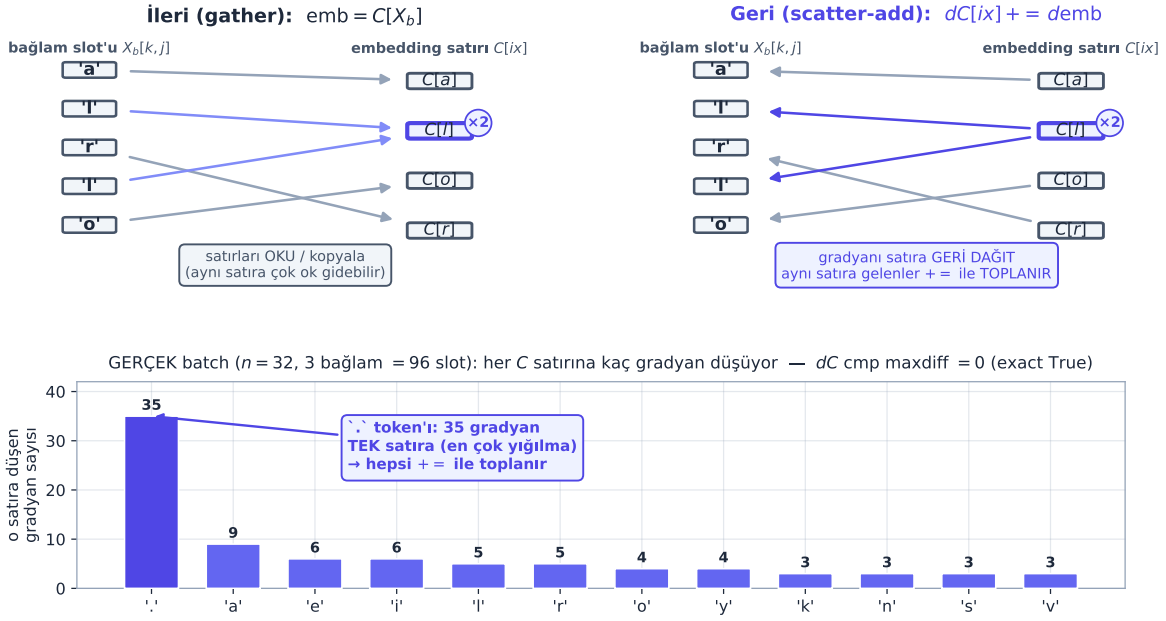
12.8 BatchNorm İçleri ve Embedding Backward

Egzersiz 1'i tamamlamak için BatchNorm'un iç istatistiklerinden (bndiff, bnvar, bnmeani, hprebn) ve embedding'den geçen gradyanları da yazarız. Bunların çoğu yine "çarpma/toplama/broadcast" kurallarının uygulanması — ama iki incelik var.

Broadcast/toplam dualitesi. Bir tensör ileri geçişte broadcast olduysa (örn. batch ortalaması tüm satırlara yayıldı), geri geçişte o eksende **toplanır**. Bu, manuel backward'ın en sık hata kaynağı: şekil uymazsa bir `.sum(0, keepdim=True)` eksik demektir.

Embedding backward (scatter-add). İleri geçişte `emb = C[Xb]` ile `C`'nin satırlarını **topladık** (gather). Geri geçişte gradyanları `C`'ye geri **dağıtırız** (scatter). Kritik nokta: aynı karakter farklı bağlamlarda birçok kez kullanıldığı için, aynı `C` satırına birden çok gradyan gelir — bunlar **toplanmalı** (Ders 1'in `+=` dersi!).

```
dC = torch.zeros_like(C)
for k in range(Xb.shape[0]):
    for j in range(Xb.shape[1]):
        ix = Xb[k, j]
        dC[ix] += demb[k, j]   # AYNI satira gelenler TOPLANIR (+=)
```



Şekil 12.6: Embedding backward = scatter-add: aynı C satırına gelen gradyanlar TOPLANIR. **Sol (ileri = gather):** emb $= C[X_b]$ — batch bağlamındaki her karakter id'si C'nin bir satırını OKUR (kopyalar). Aynı karakter farklı bağlamlarda tekrar ettiği için (örn. `r`, `l`, `o` birden çok kez) AYNI C satırına birden çok ok gider. **Sağ (geri = scatter-add):** $dC[ix] \mathrel{+}= demb$ — oklar tersine döner, her kullanımın gradyanı kendi C satırına geri akar; aynı satıra gelenler `+=` ile **biriktirilir** (Ders 1'in $+=$ kuralı). Eğer `=` (atama) kullanılsaydı son kullanım öncekileri EZER, o karakterin tüm bağlamlarından yalnızca sonuncusu sayılırdı → yanlış gradyan. **Sayaç (alt):** GERÇEK batch'te ($n = 32$, 3 bağlam $= 96$ slot) her satıra kaç gradyan düştüğü: `.` token'ı 35 kez (en yüksek yığılma), `a` 9, `e/i` 6. PyTorch'un `index_add_` / `scatter_add_`'i tam bunu verimli yapar; GPT'nin token embedding backward'ı da budur. Manuel scatter-add autograd ile birebir: dC cmp maxdiff $= 0$ (exact True).

Builder Notu — Scatter-Add = Ders 1'in += Hatası

Geriye (Ders 1): Embedding'in scatter-add'i, doğrudan Ders 1'in **gradyan biriktirme (+=) hatası**: bir değişken (C'nin satırı) birden çok yola besleniyorsa gradyanlar toplanır. `=` yazsan son kullanım öncekileri ezerdi — yanlış gradyan. Broadcast/sum dualitesi de aynı kuralın matris hâli.
İleriye: Gather/scatter (toplama/dağıtma), embedding katmanlarının ve seyrek (sparse) gradyanların

temelidir; PyTorch `index_add_` / `scatter_add_` bunu verimli yapar. GPT'nin token embedding'inin backward'ı da budur.

12.9 Egzersiz 2: Analitik cross-entropy Backward

Egzersiz 1'de cross-entropy'nin gradyanını atomik graf boyunca (logprobs → probs → counts → … → logits) adım adım hesapladık — uzun ve dolambaçlı. Egzersiz 2: aynı gradyanı **tek satırda**, analitik olarak türet.

Cross-entropy'nin logitlere göre gradyanı şaşırtıcı derecede zariftir:

$$\frac{\partial L}{\partial \text{logits}} = \frac{\text{softmax}(\text{logits}) - \text{onehot}(y)}{n}$$

Yani: softmax olasılıklarını al, doğru hedef konumlarından 1 çıkar, n'e böl. Sezgi: model doğru karaktere atadığı olasılığı 1'e, diğerlerini 0'a itmek ister; gradyan tam bu "fark"tır.

> *"I came up with one line of code that does that. Let me just erase a bunch of stuff here."* — Karpathy, 39:56

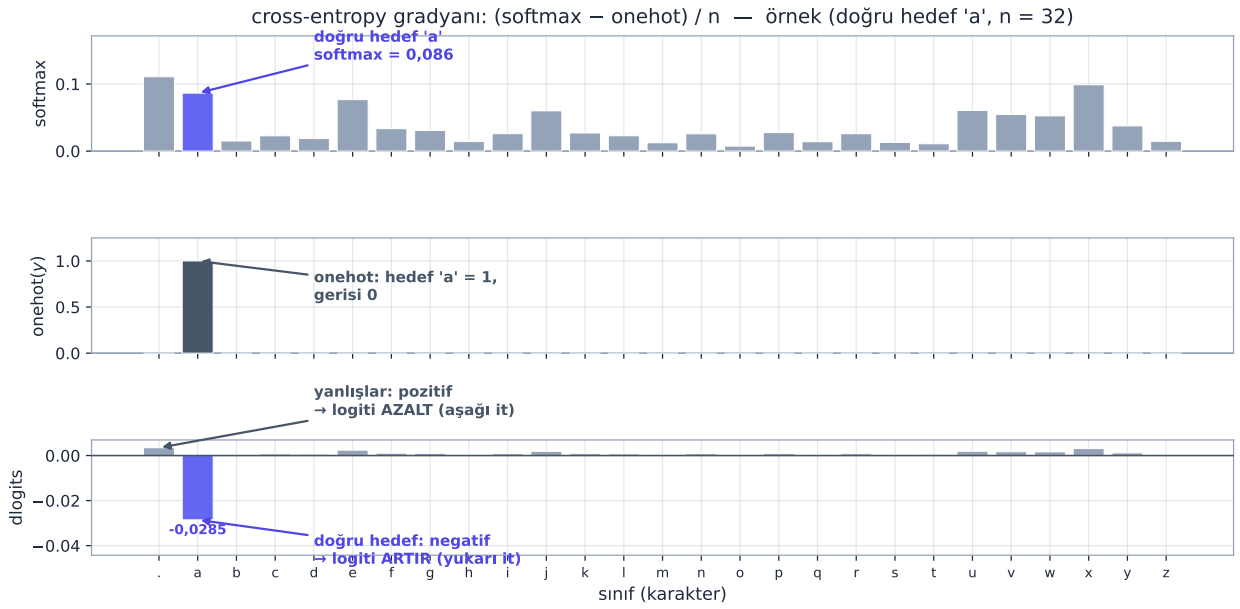
```
dlogits = F.softmax(logits, 1)   # softmax olasiliklari
dlogits[range(n), Yb] -= 1       # dogru hedeflerden 1 cikar
dlogits /= n                     # batch ortalamasi
cmp('logits', dlogits, logits)   # Egzersiz 1'le AYNI sonuc, tek satirda
```

Bu tek satır, Egzersiz 1'in onlarca ara adımıyla **birebir aynı** sonucu verir — bizim ölçümümüzde füzyonlu ile atomik dlogits arasındaki fark yalnızca maxdiff $\approx 9{,}31 \times 10^{-9}$ (float yuvarlama, 10^{-8} eşiğinin altında), PyTorch autograd ile de aynı yakınlıkta (approx True). Hem çok daha kısa hem de sayısal olarak kararlı. İşte analitik türevin gücü: zincirin tamamını sadeleştirip kapalı-form bir ifadeye indirir.

> Builder Notu — softmax – onehot: Tüm Üstel-Aile Modellerinin Gradyanı
>
> **Geriye (Ders 1-2 + Stat 110):** `softmax - onehot`, Ders 1'deki cross-entropy/sigmoid gradyanının $(\hat{y} - y)$ çok-sınıflı genellemesi; Stat 110 multinomial MLE'nin gradyanı. "Modelin tahmini eksi gerçek" deseni, tüm üstel-aile (exponential family) modellerinin gradyanıdır.
> **İleriye:** $d\text{logits} = (\text{softmax} - \text{onehot})/n$, her dil modelinin (GPT dahil) çıkış katmanı backward'ı. `F.cross_entropy` bunu içeride yapar (Ders 3'teki "füzyonlu + kararlı" sebebi). Analitik sadeleştirme, FlashAttention gibi optimizasyonların da ruhu (Ders 10).

12.10 Egzersiz 3: Analitik BatchNorm Backward

cross-entropy gibi, BatchNorm'un da atomik-graf backward'ını (bndiff, bnvar, bnmeani, …) tek bir **füzyonlu** ifadeye indirebiliriz. Ama BatchNorm daha zorludur: ortalama (μ) ve varyans (σ) **tüm batch'e bağlı** olduğu için, bir örneğin gradyanı diğerlerine sızar (örnekler kuple).



Şekil 12.7: cross-entropy gradyanı tek bir batch örneği için GERÇEK değerlerle: $\partial L/\partial \text{logits} = (\text{softmax} - \text{onehot})/n$. **Üst (indigo):** softmax olasılıkları (27 sınıf) — eğitilmemiş ağ neredeyse uniform, doğru hedef `a` yalnızca $0{,}086$ olasılık alır (argmax `.`'da, $0{,}111$). **Orta (slate):** $\text{onehot}(y)$ — doğru hedef `a` konumunda tek 1 çubuğu, gerisi 0. **Alt (fark = gradyan):** $(\text{softmax} - \text{onehot})/n$ ($n = 32$). Doğru hedef `a` TEK **negatif** gradyanı alır ($-0{,}0285 = (0{,}086 - 1)/32$, indigo, yukarı ok: gradient descent bu logiti *artırır*); 26 yanlış sınıf küçük **pozitif** gradyan alır (slate, aşağı ok: logitleri *azaltılır*); en büyük itiş argmax `.`'da ($+0{,}0035$). Satır toplamı ≈ 0 (gradyan korunur). Kontrol Sorusu 1 sezgisinin gerçek-veri hâli: "doğruyu yukarı it, yanlışları aşağı it".

> *"[We use] pen and paper and mathematics and calculus to derive the gradient through the batchnorm layer."* — Karpathy, 11:18

Karpathy kağıt-kalem ile türetip tek satıra indirir. `hpreact = bngain * bnraw + bnbias` ve `bnraw = (hprebn - μ)/√(σ²+ε)` için, `dhprebn` (BatchNorm'a giren gradyan):

$$\frac{\partial L}{\partial \text{hprebn}} = \frac{\gamma\,\sigma_{inv}}{n}\left(n\frac{\partial L}{\partial \text{hpreact}} - \sum_i \frac{\partial L}{\partial \text{hpreact}_i} - \frac{n}{n-1}\hat{x}\sum_i \frac{\partial L}{\partial \text{hpreact}_i}\hat{x}_i\right)$$

```
dhprebn = bngain * bnvar_inv / n * (
    n * dhpreact
    - dhpreact.sum(0)
    - n/(n-1) * bnraw * (dhpreact * bnraw).sum(0)
)
cmp('hprebn', dhprebn, hprebn)   # atomik grafla AYNI, tek satirda
```

Üç terim sezgisel: birincisi doğrudan gradyan, ikincisi ortalama-çıkarmanın etkisi (tüm batch toplamı), üçüncüsü varyans-normalizasyonunun etkisi (Bessel'in $n/(n-1)$'i burada belirir). Atomik grafla birebir aynı — bizim ölçümümüzde füzyonlu ile atomik dhprebn arasındaki fark yalnızca maxdiff $\approx 9{,}31 \times 10^{-10}$ (approx True) — ama tek satır.

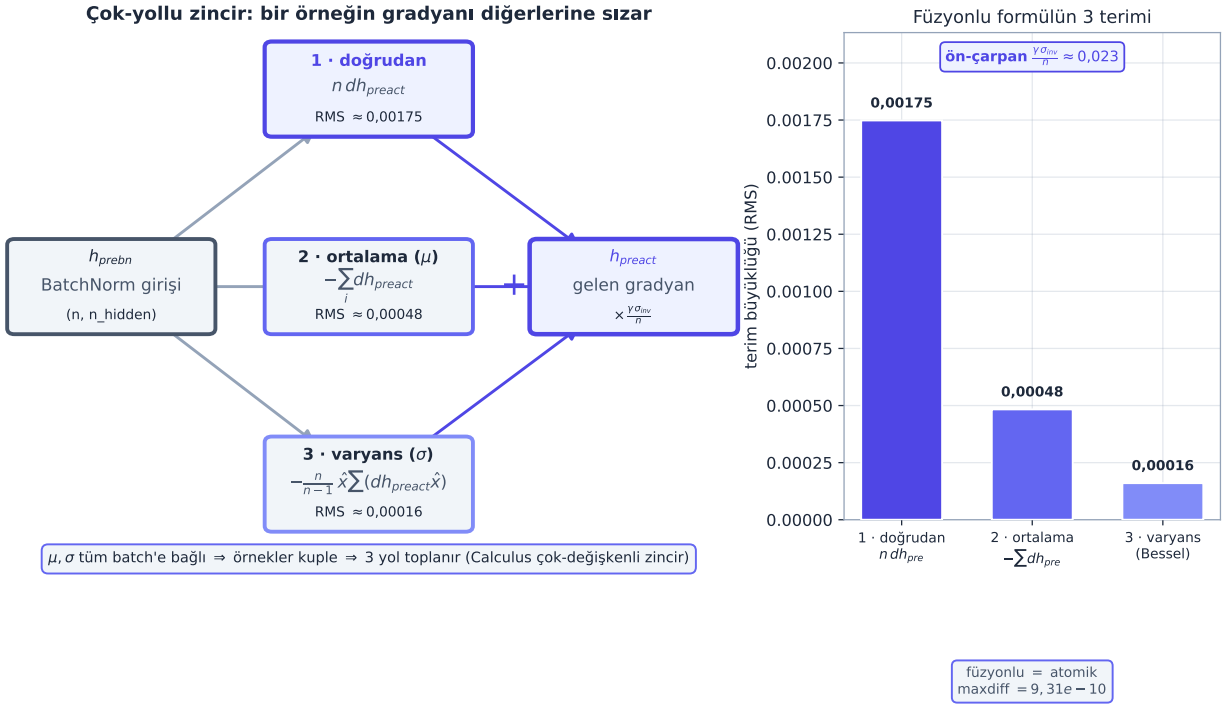
> Builder Notu — Çok-Yollu Zincir: μ ve σ Batch'e Bağlı
>
> **Geriye (Stat 110 + Calculus):** Bu formül, μ ve σ'nın hprebn'in **fonksiyonu** olmasından gelen çok-yollu zincir kuralı (Calculus): hprebn hem doğrudan, hem μ üzerinden, hem σ üzerinden hpreact'i etkiler — üç yol toplanır (Ders 1'in çok-değişkenli += kuralı). $n/(n-1)$ Bessel'den (Stat 110).
> **İleriye:** Bu fused backward, BatchNorm/LayerNorm'un production implementasyonudur — atomik grafı çalıştırmak yerine tek kernel. FlashAttention da aynı ruhla attention'ı füzyonlar (Ders 10).

12.11 Egzersiz 4: Hepsini Birleştir — Elle Backprop'la Eğitim

Son egzersiz: `loss.backward()`'ı **tamamen kaldır** ve tüm manuel gradyanları (dC, dW1, db1, dW2, db2, dbngain, dbnbias) eğitim döngüsünde kullan. Artık autograd yok — gradyanları biz hesaplıyoruz, `torch.no_grad()` ile sarıp parametreleri güncelliyoruz.

```
with torch.no_grad():
    # ... tum gradyanlari elle hesapla (Egzersiz 1-3) ...
    grads = [dC, dW1, db1, dW2, db2, dbngain, dbnbias]
    for p, grad in zip(parameters, grads):
        p.data += -lr * grad   # loss.backward() YOK!
```

Ağ, elle yazılan gradyanlarla **tam olarak öğrenir** — loss düşer, PyTorch autograd ile aynı sonuç. Bizim doğrulamamızda 2000 adım, lr $= 0{,}1$ ile manuel eğitim ile autograd eğitim AYNI loss eğrisini izler: ikisi de ilk loss $3{,}408827$, son loss $2{,}333302$; tüm eğitim boyunca en büyük loss farkı yalnızca $\approx 7{,}15 \times 10^{-7}$ (birebir aynı). Karpathy bunu zaferle ilan eder:



Şekil 12.8: BatchNorm füzyonlu backward, üç terimli çok-yollu zincir. **Sol (şema):** h_{prebn}, μ ve σ üzerinden tüm batch'e bağlandığı için h_{preact}'i ÜÇ yoldan etkiler (örnekler kuple): (1) doğrudan gradyan $n\,dh_{preact}$, (2) ortalama-çıkarma yolu $-\sum dh_{preact}$ (μ tüm batch'e bağlı), (3) varyans-normalize yolu $-\frac{n}{n-1}\,\hat{x}\sum(dh_{preact}\hat{x})$ (Bessel $\frac{n}{n-1} = 1{,}03$, n=32). Üç yol toplanır, ortak ön-çarpan $\frac{\gamma\,\sigma_{inv}}{n} \approx 0{,}023$ ile çarpılır. **Sağ (bar):** üç terimin GERÇEK büyüklüğü (RMS): doğrudan $0{,}00175$ baskın, ortalama-çıkarma $0{,}00048$, varyans-normalize $0{,}00016$. Füzyonlu tek-satır formül, atomik graf ile birebir aynı: maxdiff $= 9{,}31 \times 10^{-10}$ (approx True).

"So we don't need it anymore. It feels amazing to say that." — Karpathy, 1:52:05

Artık `loss.backward()` bir kara kutu değil; her gradyanın nereden geldiğini biliyorsun. Bütün doğrulamanın özeti aşağıdaki cmp tablosunda: atomik graf gradyanları exact, füzyonlu formüller approx — manuel = autograd.

cmp Doğrulama Tablosu — Manuel Gradyan = Autograd Kanıtı

elle hesaplanan gradyan vs PyTorch .grad (batch $n = 32$, forward loss = 3,4088)

gradyan	exact (==)	approx	maxdiff	sonuç
Egzersiz 1 — atomik graf elle backward (16 ana gradyan, hepsi exact)				
dlogprobs	✓ True	✓ True	0,0	tam eşit
dprobs	✓ True	✓ True	0,0	tam eşit
dnorm_logits	✓ True	✓ True	0,0	tam eşit
dlogits	✓ True	✓ True	0,0	tam eşit
dh	✓ True	✓ True	0,0	tam eşit
dhpreact	✓ True	✓ True	0,0	tam eşit
dbnraw	✓ True	✓ True	0,0	tam eşit
dhprebn	✓ True	✓ True	0,0	tam eşit
dembcat	✓ True	✓ True	0,0	tam eşit
dC	✓ True	✓ True	0,0	tam eşit
dW1	✓ True	✓ True	0,0	tam eşit
db1	✓ True	✓ True	0,0	tam eşit
dW2	✓ True	✓ True	0,0	tam eşit
db2	✓ True	✓ True	0,0	tam eşit
dbngain	✓ True	✓ True	0,0	tam eşit
dbnbias	✓ True	✓ True	0,0	tam eşit
Egzersiz 2 + 3 — füzyonlu (tek satır), approx ✓				
dlogits (fused)	False	✓ True	9,31e-09	çok yakın
dhprebn (fused)	False	✓ True	9,31e-10	çok yakın

Egzersiz 1: 16/16 exact (maxdiff = 0) · Egzersiz 2/3: approx ✓ (maxdiff $< 10^{-8}$) · TÜM gradyanlar geçti: True

Şekil 12.9: cmp doğrulama tablosu — manuel gradyanlar autograd ile birebir/çok-yakın eşleşir. Her satır, elle hesaplanan gradyanın PyTorch `.grad` ile GERÇEK cmp çıktısıdır (uydurma YOK; $n = 32$ batch). **Üst blok (Egzersiz 1, atomik graf):** 16 ana gradyan (`dlogprobs` → `dC/dW1/db1/dW2/db2/dbngain/dbnbias`) hepsi **exact** ✓ (maxdiff = 0,0, tam eşit) — yeşil-indigo vurgu. **Alt blok (füzyonlu):** Egzersiz 2 `dlogits` (softmax−onehot)/n **approx** ✓ (maxdiff $\approx 9{,}31 \times 10^{-9}$) ve Egzersiz 3 `dhprebn` füzyonlu BatchNorm **approx** ✓ (maxdiff $\approx 9{,}31 \times 10^{-10}$) — float yuvarlama, eşik 10^{-8} altında. Bu tablo dersin sayısal-dürüstlük kanıtıdır: "manuel = autograd".

Builder Notu — Ders 1'in Tam Dairesi

Geriye (Ders 1): Bu, Ders 1'in tam dairesi: micrograd'da elle backward yazmıştık (skaler), şimdi gerçek bir MLP+BatchNorm için elle backward yazdık (tensör) ve autograd'ı tamamen değiştirdik. Ders 1'in "her şey efficiency" iddiası kanıtlandı.

İleriye: Çoğu zaman autograd kullanırsın (pratiklik), ama gradyanın nasıl aktığını bilmek, gradyan bug'larını (NaN, vanishing) teşhis etmeni ve özel katman/loss yazmanı sağlar — bu, "kullanıcı" ile

12.12 Sonuç ve Önizleme

Bu dersle bir **backprop ninjası** oldun: gradyanların bir hesaplama grafiği boyunca tensör düzeyinde nasıl aktığını elle yazabiliyorsun. `loss.backward()` artık şeffaf — arkasında zincir kuralının matris hâlinin çalıştığını biliyorsun.

Ders 6'da mimariye geri dönüyoruz. Not: **Ders 4 (BatchNorm) ve Ders 5 (manuel backprop) birer "aside"di** — ağı derinleştirmek/anlamak için. Ders 6'nın başlangıç kodu doğrudan **Ders 3'ün MLP'sinden** gelir; oradan bağlamı büyütüp hiyerarşik bir **WaveNet** mimarisine geçeceğiz.

Builder Notu — 'Backprop'u Anla' Yatırımı

İleriye: "Backprop'u anla" yatırımı serinin geri kalanında geri döner: Ders 7'de attention'ın, Ders 10'da FlashAttention'ın neden öyle tasarlandığını anlamak, gradyan akışını görmekten geçer. Ninja olmak = soyutlamayı güvenle kullanmak.

Builder Notu — Strang D2: Uzaktan Akraba (Gradient ~ En Küçük Adım)

Yatay köprü (Strang Matrix Methods, Ders 2 — uzaktan akraba): Backprop'un gradyanı, kaybı en hızlı azaltan **en küçük yerel adım** yönüdür; gradient descent her adımda bu en-iyi yerel hamleyi seçer. Strang D2'nin Eckart-Young teoremi de aynı epistemolojinin lineer-cebir hâlidir: bir matrisi rank-k ile en iyi yaklaşan şey, en büyük k tekil değeri tutup gerisini atmaktır — yani belirli bir kısıt altında **en küçük hatayı** veren çözüm. İkisi de "bir hedef fonksiyonu, bir kısıt altında en iyi nasıl optimize ederim" sorusunun cevabı; biri sonsuz-küçük adım (türev), öbürü kapalı-form en-iyi düşük-rank. Bağlantı uzak ama epistemolojik: optimizasyon = en iyi yerel/global hamleyi türetmek.

Builder Notu — fast.ai L13/L17: Aynı Epistemoloji, Farklı Çatı

Yatay köprü (fast.ai L13 + L17): Howard, fast.ai L13'te "fully matmul'd backward" ile her katmanın backward'ını (Lin, ReLU, MSE) elle yazar; L17'de ise "her adımın türevini" tek tek izler. Bu, Karpathy'nin bu dersteki "her katmanın backward'ı" (matmul, tanh, BatchNorm, cross-entropy, embedding) yaklaşımıyla **aynı epistemoloji**: framework'e güvenmeden önce, gradyanın her atomda nasıl aktığını elle çıkar. Howard konuyu bir ConvNet/MLP üstünden, Karpathy bir MLP+BatchNorm üstünden işler — aynı matematik (zincir kuralı + matris türevleri), farklı çatı. İki ders birlikte: "autograd'ı kullanmadan önce bir kez elle yaz" disiplini, iki ayrı kursta bağımsızca aynı sonuca varır.

12.13 Bu Dersin Özeti

1. **Backprop "sızdıran soyutlama"dır**: `loss.backward()`'ı körü körüne çağırmak gradyan bug'larını gizler. Bir kez elle yaz, bir daha kara kutu olmasın.
2. **cmp yardımcısı**: manuel gradyanı PyTorch `.grad` ile karşılaştır (exact/approx/maxdiff); forward bir sürü adlandırılmış ara tensöre bölünür (`retain_grad`).
3. **Egzersiz 1**: atomik graf boyunca her ara gradyanı elle yaz — Ders 1 micrograd'ın tensör hâli (yerel türev $\times$ zincir kuralı); bizde tüm ana gradyanlar exact (maxdiff $0{,}0$).

4. **matmul backward**: $dA = dC\,B^\top$, $dB = A^\top dC$, $db = dC.\text{sum}(0)$; transpoze'lar boyut uyumundan çıkar (18.06).
5. **tanh backward** $(1 - h^2) \cdot dh$; **BatchNorm scale/shift**: broadcast olan (γ, β) toplanır, çarpım diğerini geçirir.
6. **Bessel düzeltmesi**: yansız $(1/(n-1))$ vs yanlı $(1/n)$ varyans; PyTorch normalize'da biased, running var'da Bessel kullanır.
7. **embedding backward = scatter-add**: aynı C satırına gelen gradyanlar **toplanır** (Ders 1'in += kuralı).
8. **Egzersiz 2**: cross-entropy backward tek satır $\rightarrow (\text{softmax} - \text{onehot})/n$ ("model tahmini − gerçek"); füzyonlu vs atomik maxdiff $\approx 9{,}31 \times 10^{-9}$.
9. **Egzersiz 3**: BatchNorm fused backward (tek formül, üç terim; maxdiff $\approx 9{,}31 \times 10^{-10}$); **Egzersiz 4**: `loss.backward()` olmadan tam eğitim (manuel = autograd, son loss 2,333302).

> **!** Tek Bir Cümle
>
> `loss.backward()` sihir değil, zincir kuralının bir hesaplama grafiği boyunca tensör düzeyinde uygulanmasıdır; gradyanları bir kez elle yazıp (matmul'da transpoze, cross-entropy'de softmax−onehot, BatchNorm'da füzyonlu formül) `loss.backward()` olmadan eğitebilmek, backprop'u "kullanan" biriyle "anlayan" biri arasındaki farkı koyar.

12.14 Kontrol Soruları

> **i** Soru 1: Tek bir örnek için softmax çıktısı [0,1; 0,7; 0,2] ve doğru hedef indeks 0 (yani onehot = [1, 0, 0]). cross-entropy'nin logitlere göre gradyanı (dlogits) nedir? İşaretlerini yorumla.

Formül: $d\text{logits} = (\text{softmax} - \text{onehot})/n$. Tek örnek ($n = 1$):

$$\frac{\partial L}{\partial \text{logits}} = \text{softmax} - \text{onehot} = [0{,}1,\ 0{,}7,\ 0{,}2] - [1, 0, 0] = [-0{,}9,\ 0{,}7,\ 0{,}2]$$

Cevap: $d\text{logits} = [-0{,}9,\ 0{,}7,\ 0{,}2]$. **Yorum:** Doğru hedef (indeks 0) **negatif** gradyan alır — gradient descent bu logiti *artıracak* (modelin doğru karaktere daha çok olasılık vermesini sağlar). Yanlış hedefler (1, 2) **pozitif** gradyan alır → logitleri *azaltılır*. Yani gradyan, "doğruyu yukarı it, yanlışları aşağı it" der; büyüklük, modelin ne kadar yanıldığıyla orantılı (doğru hedefe yalnızca $0{,}1$ vermiş, fark $0{,}9$ büyük).

> **i** Soru 2: Egzersiz 1 (atomik graf, onlarca adım) ile Egzersiz 2/3 (tek satırlık füzyonlu formül) aynı gradyanı veriyor. O hâlde neden füzyonlu hâli öğreniriz?

Cevap: Üç sebep. **(1) Hız/bellek:** Atomik graf onlarca ara tensör (counts, probs, bndiff, …) oluşturur ve bunların hepsini bellekte tutup geri geçer; füzyonlu formül tek bir ifadede sonuca varır — daha az bellek, daha hızlı. **(2) Sayısal kararlılık:** Ara adımlar (exp, bölme) taşma/sıfıra-bölme riski taşır; füzyonlu formül bunları sadeleştirir. **(3) Anlama:** $(\text{softmax} - \text{onehot})/n$ gibi kapalı-form, gradyanın *ne anlama geldiğini* (model − gerçek) gösterir. Bu yüzden gerçek kütüphaneler (`F.cross_entropy`, BatchNorm) füzyonlu backward kullanır — atomik graf öğretici, füzyonlu hâli production. (Bizim ölçümümüzde ikisi maxdiff $< 10^{-8}$ ile birebir aynı.)

ℹ Soru 3: Embedding backward'ında (dC[ix] += demb) neden = değil += kullanılır? = kullansak ne olurdu?

Cevap: Aynı karakter (örn. `a`), veri setinde birçok farklı bağlamda geçer — yani `C`'nin **aynı satırı** ileri geçişte birçok kez kullanılır. Çok-değişkenli zincir kuralına göre, bir değer birden çok yola katkı veriyorsa gradyanları **toplanmalıdır** (Ders 1'in += dersi). `dC[ix] += demb` her kullanımın katkısını biriktirir. `=` (atama) kullansaydık, son kullanım öncekilerin gradyanını **ezerdi** → yanlış gradyan (o karakterin tüm bağlamlarından yalnızca sonuncusu sayılırdı). Bu, scatter-add'in (`index_add_`) neden toplama yaptığının sebebi. (Bizim batch'imizde `.` token'ı tek satıra 35 gradyan toplar; `=` olsaydı 34'ü kaybolurdu.)

ℹ Soru 4: (Builder) C = A @ B için dA = dC @ B^T (B transpoze). Neden transpoze ve neden bu sıra? 18.06 / boyut uyumu ile bağla.

Cevap: `dA`, `A` ile **aynı şekilde** olmalı (gradyan, değişkeniyle aynı boyut). A şekli (n, p), C şekli (n, m), B şekli (p, m). dC şekli C ile aynı: (n, m). dA'yı (n, p) üretmek için tek tutarlı çarpım: $dC\,(n, m) @ B^{\top}(m, p) = (n, p)$. ✓ $B^{\top}$ olmasının sebebi: çarpımın iç boyutları uymalı (m ile m). Karpathy'nin pratik kuralı: **transpoze ve sırayı ezberleme — boyutların uyması gereken tek yolu bul**. Bu, 18.06'nın matris-çarpım boyut kurallarının doğrudan sonucu; "çarpma diğerini geçirir" (Ders 1) kuralının matris hâlidir (diğer operand, doğru transpoze ile).

12.15 Egzersizler

Egzersiz 1 (İlk backward adımı). `loss = -logprobs[range(n), Yb].mean()` için `dlogprobs`'u elle yaz: yalnızca (range(n), Yb) konumlarında $-1/n$, gerisi 0. `cmp('logprobs', dlogprobs, logprobs)` ile PyTorch'a karşı "exact: True" aldığını doğrula.

Egzersiz 2 (cmp + atomik gradyanlar). `cmp` yardımcısını kur. Forward'ı adlandırılmış ara tensörlere böl (`retain_grad` ile). Birkaç atomik gradyanı (`dh`, `dW2`, `db2`) elle hesaplayıp `cmp` ile doğrula — her biri "exact" veya "approx: True" olmalı.

Egzersiz 3 (cross-entropy tek satır). Atomik graf yerine analitik backward yaz: `dlogits = F.softmax(logits, 1); dlogits[range(n), Yb] -= 1; dlogits /= n`. Bunun, Egzersiz 1'deki onlarca-adımlı `dlogits` ile birebir aynı çıktığını `cmp` ile doğrula.

Egzersiz 4 (loss.backward() olmadan eğitim). Egzersiz 1-3'teki tüm manuel gradyanları topla, `loss.backward()`'ı kaldır, `torch.no_grad()` içinde parametreleri elle güncelle (`p.data += -lr * grad`). Ağı eğit, loss'un PyTorch autograd ile aynı şekilde düştüğünü gözlemle. Artık autograd'a ihtiyacın yok.

Egzersiz 5 (Sonraki dersin habercisi). Ders 3'ün MLP'si, bağlamdaki tüm karakterleri **bir kerede** düzleştirip (`.view(-1, 6)`) tek bir gizli katmana veriyordu. Şimdi bağlamı 8 karaktere çıkardığını düşün. (a) 8 karakteri tek seferde düzleştirip tek katmana vermek yerine, onları **kademeli** (önce 2'şer, sonra o çiftleri 2'şer, ağaç gibi) füzyonlamak neden daha iyi olabilir? (İpucu: yerel desenleri önce, küresel yapıyı sonra öğrenmek.) (b) Bu hiyerarşik füzyon fikri **WaveNet** mimarisidir. Bu soru, Ders 6'da (**makemore 5: WaveNet**) MLP'yi hiyerarşik bir yapıya dönüştürmeyi motive eder.

12.16 Sonraki Ders İçin Hazırlık

Ders 6: makemore 5 — WaveNet (Hiyerarşik Mimari) — Andrej Karpathy

Bu derste (ve Ders 4'te) ağı *anlamaya* odaklandık — Ders 4 BatchNorm, Ders 5 manuel backprop, ikisi de birer "aside". Ders 6'da **mimariye** geri dönüyoruz ve başlangıç kodu doğrudan **Ders 3'ün MLP'sinden** gelir. MLP'yi, bağlamı tek seferde düzleştirmek yerine **kademeli/hiyerarşik** birleştiren bir **WaveNet** yapısına dönüştüreceğiz; bu arada kodu daha da `torch.nn`-benzeri modüllere (Embedding, Flatten, Sequential) çekeceğiz.

Ana konular:

- Kodu PyTorch-tarzı konteynerlere çekme (Embedding, FlattenConsecutive, Sequential).
- Bağlamı 3'ten 8 karaktere çıkarma; ardışık çiftleri kademeli füzyonlama.
- WaveNet'in hiyerarşik (ağaç-benzeri) yapısı ve BatchNorm1d'in 3B girdi düzeltmesi.

⚠ Ders 6 Öncesi Yapılacak

- Egzersizleri çöz — özellikle 4 (`loss.backward()` olmadan eğitim) ve 5 (hiyerarşik füzyon sezgisi).
- "Backprop = zincir kuralının tensör hâli" ve "softmax — onehot = cross-entropy gradyanı" cümlelerini hatırla.
- Ders 3'ün MLP kodunu hazır tut (Ders 6 onun üstüne kurar; Ders 4-5 birer aside'dı).

12.17 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Karpathy'de
Sızdıran soyutlama	`loss.backward()`'ı anlamadan çağırmak gradyan bug'larını gizler	1m00
cmp / gradient check	Manuel gradyanı PyTorch `.grad` ile karşılaştır (exact/approx/maxdiff)	7m25
Atomik graf backward	Her ara tensörün gradyanı, yerel türev × zincir kuralı (Ders 1, tensör)	12m57
matmul backward	C=A@B için $dA = dC\,B^\top$, $dB = A^\top dC$, $db = dC.\text{sum}(0)$	41m44
tanh backward	$d\text{hpreact} = (1-h^2)\cdot dh$; Ders 1'in $1-\tanh^2$ türevi	53m33

Kavram	Tanım	Karpathy'de
BatchNorm scale/shift	γ, β broadcast → geri geçişte toplanır; $d$bnraw = bngain · $d$hpreact	53m33
Bessel düzeltmesi	Yansız varyans $1/(n-1)$ vs yanlı $1/n$; PyTorch BN ikisini farklı yerde kullanır	1h05m
embedding scatter-add	$dC[ix]$ += $d$emb; aynı satıra gelen gradyanlar toplanır (Ders 1 +=)	1h08m
cross-entropy backward	$d$logits = $(\text{softmax} - \text{onehot})/n$; tek satır, "model − gerçek"	1h26m
BatchNorm fused backward	Tek formül (3 terim); μ ve σ batch'e bağlı → çok-yollu zincir	1h36m
Elle eğitim (no backward)	`loss.backward()` kaldır, manuel gradyanlarla `torch.no_grad()` güncelle	1h50m

12.18 ML Builder Bağlantıları

9 köprü — Backprop Ninja

1. **Backprop = zincir kuralı (tensör)** → Ders 1 micrograd + Calculus zincir kuralı. İleriye: her framework'ün autograd'ı.
2. **matmul backward (transpoze)** → 18.06 boyut uyumu + Ders 1 "diğerini geçir". İleriye: GPU GEMM, forward 1 / backward 2 matmul.
3. **cross-entropy backward (softmax − onehot)** → Ders 1 ($\hat{y} - y$) + Stat 110 multinomial MLE. İleriye: GPT çıkış katmanı backward.
4. **embedding scatter-add** → Ders 1 += biriktirme (çok-yollu zincir). İleriye: seyrek gradyan, `index_add_`, token embedding backward.
5. **BatchNorm fused backward** → Stat 110 Bessel + Calculus çok-yollu zincir. İleriye: production norm kernel'ları, FlashAttention ruhu.
6. **broadcast/sum dualitesi** → tensör mekaniği. İleriye: şekil-uyumsuzluğu bug'larından kaçınma.
7. **cmp / gradient check** → Ders 1 sayısal gradyan. İleriye: `torch.autograd.gradcheck`, custom Function doğrulama.
8. **Sızdıran soyutlama** → aleti anlayarak kullan. İleriye: NaN/vanishing gradient teşhisi (Ders 4 ile birlikte).
9. **Elle eğitim (autograd'sız)** → Ders 1'in tam dairesi. İleriye: özel `autograd.Function` yazmak

(production'da yeni katman/loss).

12.19 Karpathy'nin Önerdiği Kaynaklar

Karpathy'nin bu ders için verdiği kaynaklar:

- **Bessel düzeltmesi açıklaması:** Oxford/Emory math117 — Bessel's correction — yansız varyans tahmini.
- **Ders Colab notebook'u:** Google Colab — egzersizlerin (boş) şablonu; backward'ları sen doldurursun.

! Bu dersten tek bir şey alıp gideceksen

`loss.backward()` sihir değil — zincir kuralının bir hesaplama grafiği boyunca tensör düzeyinde uygulanmasıdır. Gradyanları bir kez elle yazıp (matmul'da transpoze, cross-entropy'de softmax−onehot, BatchNorm'da füzyonlu formül, embedding'de scatter-add) `loss.backward()` olmadan eğitebilmek, seni backprop'u "kullanan" biriden "anlayan" birine — bir **backprop ninjasına** — dönüştürür.

13 makemore 5 — WaveNet (Hiyerarşik Mimari)

Bağlamı tek hamlede ezmek yerine ardışık öğeleri kademeli (ağaç gibi: 8→4→2→1) füzyonlamak — WaveNet'in hiyerarşik fikri — daha derin ve güçlü bir dil modeli verir; ve bu, aslında elle yazılmış bir convolution'dır

> Bölüm bilgisi
>
> - **Karpathy'nin videosu:** YouTube — Building makemore Part 5: Building a WaveNet (≈56 dk)
> - **Seri:** Neural Networks: Zero to Hero — Ders 6
> - **Hoca:** Andrej Karpathy
> - **Kaynak repo:** github.com/karpathy/makemore
> - **Okuma süresi:** ≈26 dk

13.1 Bu Derste Ne Var?

Ders 3'ün MLP'si, bağlamdaki tüm karakterleri **tek seferde** düzleştirip (`.view`) tek bir gizli katmana veriyordu. Bu derste daha derin, **hiyerarşik** bir yapıya geçiyoruz: bilgiyi kademeli olarak (önce 2'şer, sonra o çiftleri 2'şer, ağaç gibi) füzyonlayan bir **WaveNet**.

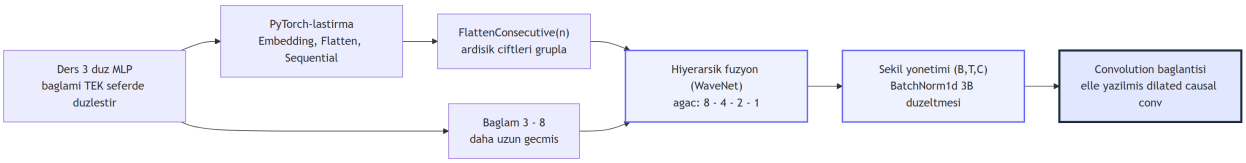
> *"You'll notice the background behind me is different — that's because I am in Kyoto and it is awesome."* — Karpathy, 0:06

> *"We would like to make a deeper model that progressively fuses this information to make its guess about the next character, with this tree-like structure."* — Karpathy, 0:46

Büyük fikir iki katmanlı: **(1)** Kodu daha da `torch.nn`-benzeri modüllere (Embedding, Flatten, Sequential) çekmek. **(2)** MLP'yi, bağlamı (3'ten 8 karaktere çıkarılmış) tek hamlede değil **kademeli** birleştiren bir ağaç yapısına dönüştürmek — bu, 2016 WaveNet makalesinin mimarisi.

Dersin üç büyük fikri:

1. **PyTorch-laştırma** — Embedding, FlattenConsecutive, Sequential modülleri; `torch.nn` API'sinin birebir taklidi.
2. **Hiyerarşik füzyon (WaveNet)** — bağlamı tek seferde değil, ardışık çiftler hâlinde kademeli birleştirme (ağaç / dilated causal convolution).
3. **Şekil (shape) yönetimi** — 3 boyutlu tensörler, BatchNorm1d'in 3B düzeltmesi; "şekilleri kollamak" mühendisliği.



Şekil 13.1: Ders 6'nın kavram haritası: Ders 3'ün düz MLP'si bağlamı tek hamlede ezerken, kodu önce torch.nn-benzeri modüllere (Embedding, Flatten, Sequential, FlattenConsecutive) çekeriz; sonra bağlamı 3'ten 8 karaktere büyütüp ardışık çiftleri kademeli füzyonlayan hiyerarşik bir ağaç (8→4→2→1, derinlik 3 = log2(8)) kurarız — bu ise tensör şekillerini (B, T, C) kollamayı (BatchNorm1d 3B düzeltmesi) ve nihayetinde tüm yapının elle yazılmış bir convolution olduğunu görmeyi getirir. Slate akış + indigo dönüm noktaları (hiyerarşik ağaç ve convolution bağlantısı).

Builder Notu — Geriye Ders 3-5, İleriye Ders 7

Geriye (Ders 3-5):

- **Başlangıç kodu = Ders 3.** WaveNet, doğrudan Ders 3'ün MLP'sinin üstüne kurulur. Karpathy net söyler: Ders 4 (BatchNorm) ve Ders 5 (manuel backprop) birer **"aside"** idi; ana hat Ders 3'ten devam eder.
- **Modüller = Ders 4.** Ders 4'te başlattığımız Linear/BatchNorm1d/Tanh bloklarını burada Embedding/Flatten/Sequential ile tamamlıyoruz — `torch.nn.Module` taklidi olgunlaşıyor.
- **Autograd zemini = Ders 1+5.** Hiyerarşik yapının gradyanı yine `loss.backward()` ile akar; Ders 5'te backward'ı bir kez elle yazdığımız için arkada ne olduğunu biliyoruz — şimdi autograd'a güvenle dönebiliyoruz.

İleriye (Ders 7): WaveNet'in ağacı **sabittir** — hangi karakterin hangisiyle birleşeceği önceden bellidir. Ders 7'nin transformer'ı bunu yeniden organize eder: sabit ağaç yerine **her token her tokena** öğrenilen ağırlıklarla bakar (attention). Aynı "uzun bağlamı verimli işle" probleminin daha esnek (ama daha pahalı) çözümü.
Tek cümleyle: Bağlamı tek hamlede ezmek yerine, ardışık öğeleri kademeli (ağaç gibi) füzyonlamak — WaveNet'in hiyerarşik fikri — daha derin ve daha güçlü bir dil modeli verir; kod ise olgun `torch.nn`-tarzı modüllere kavuşur.

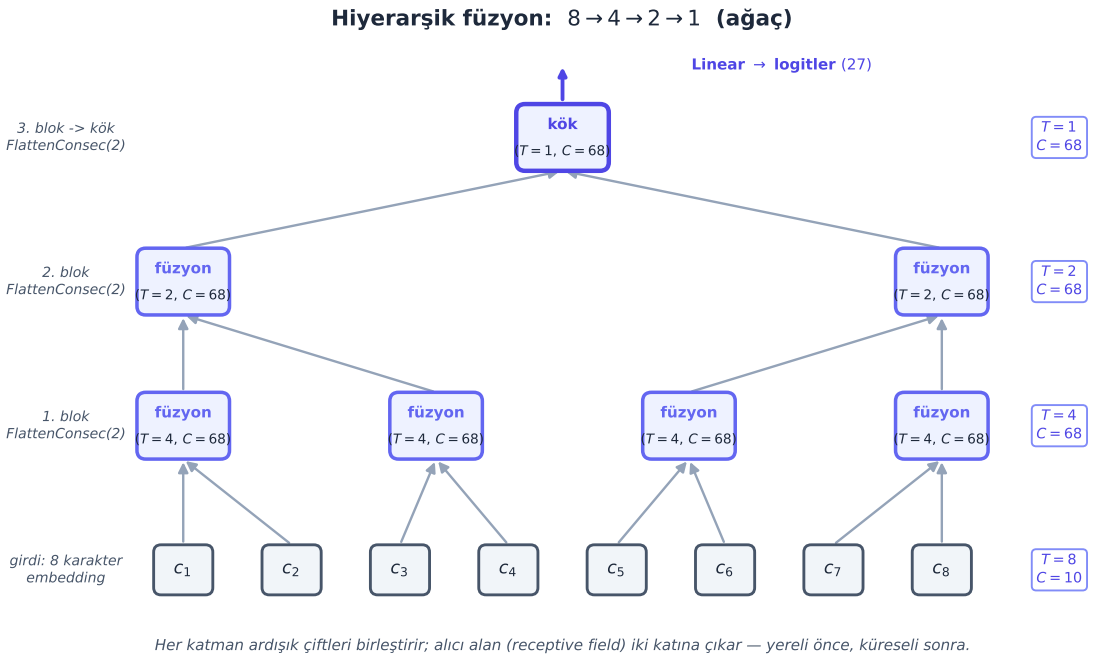
13.2 Motivasyon

Ders 3'ün MLP'sinin bir sınırı var: bağlamdaki tüm karakterlerin embedding'lerini **tek seferde** düzleştirip (`emb.view(-1, ...)`) bir gizli katmana "ezdiriyor". Yani 8 karakterlik bir bağlamda bile, model tüm bilgiyi ilk katmanda birden yutuyor — yerel yapıyı kademeli kuramaz.

> *"This architecture takes this interesting hierarchical, tree-like structure to predicting the next character."* — Karpathy, 1:25

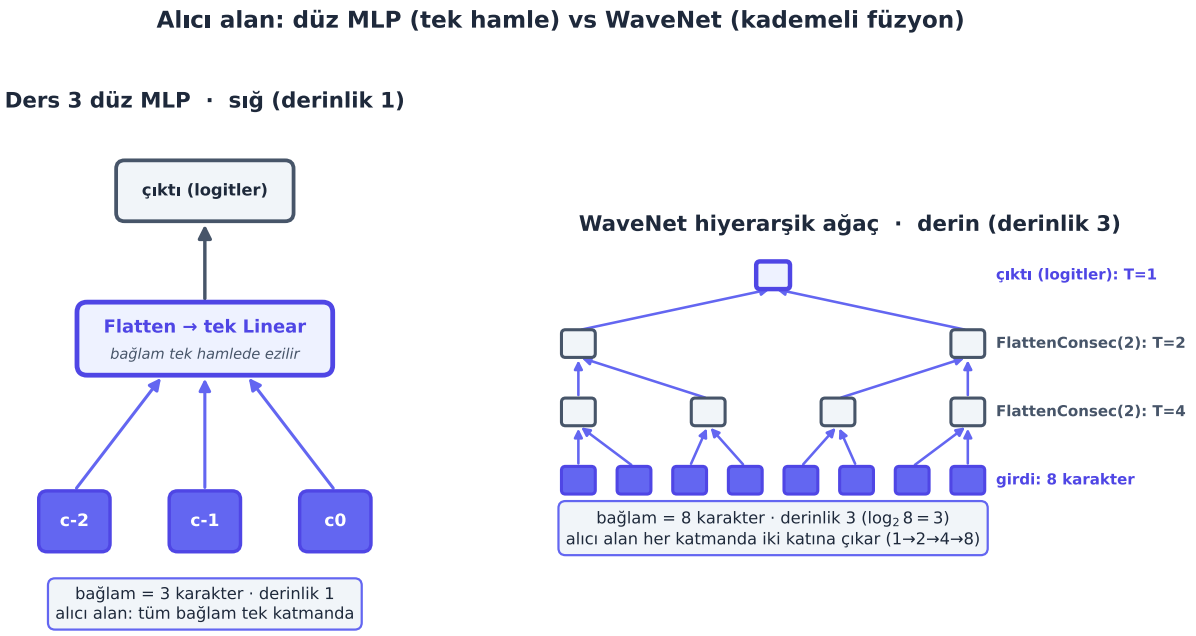
WaveNet'in fikri: bilgiyi **kademeli** birleştir. Önce komşu karakter çiftlerini füzyonla (2'şer), sonra bu füzyonları yine 2'şer birleştir, ta ki tüm bağlam tek bir temsile inene dek — bir **ağaç** gibi. Böylece model önce yerel desenleri (bigram-benzeri), sonra daha geniş yapıları öğrenir.

Aşağıdaki figür bu fikrin pedagojik imzasıdır: 8 karakterlik bağlam, ardışık çiftler hâlinde $8 \to 4 \to 2 \to 1$ kademeli olarak tek temsile iner. Gösterilen şekiller uydurma değil — ANA WaveNet'in GERÇEK forward ara şekilleridir.



Şekil 13.2: WaveNet'in hiyerarşik ağaç füzyonu (Karpathy 1m25). 8 karakterlik bağlam tek hamlede ezilmez; ardışık çiftler kademeli birleşir: $8 \to 4 \to 2 \to 1$. Her seviye, bir `FlattenConsecutive(2)` $\to$ `Linear` $\to$ `BatchNorm1d` $\to$ `Tanh` bloğudur; alıcı alan (receptive field) her katmanda iki katına çıkar, $\log_2 8 = 3$ katmanda tüm bağlam tek temsile iner. Zaman ekseni T (yaprak sayısı) yarıya inerken kanal ekseni C çiftlenir; GERÇEK forward şekilleri (ANA WaveNet, B çıkarılmış): yapraklar embedding $(T{=}8,\ C{=}10) \to (4,\ 68) \to (2,\ 68) \to (1,\ 68) \to$ logitler (27). Bu kademeli füzyon, elle yazılmış bir dilated causal convolution'dır (van den Oord 2016).

Bu hiyerarşinin Ders 3'ün düz MLP'siyle farkı, **alıcı alan** (receptive field) açısından nettir: düz MLP tüm bağlamı tek katmanda (sığ) ezer; WaveNet kademeli derinleşir.



Şekil 13.3: Alıcı alan (receptive field) karşılaştırması: **solda** Ders 3 düz MLP — bağlamdaki 3 karakter (block_size $= 3$) **tek** katmanda (`Flatten` $\rightarrow$ tek `Linear`) hep birden ezilir; yerel yapı ile uzak yapı ayrımsız işlenir, derinlik 1. **Sağda** WaveNet — block_size $= 8$ karakter, $\log_2 8 = 3$ katmanda kademeli (ağaç gibi $8 \rightarrow 4 \rightarrow 2 \rightarrow 1$) füzyonlanır: her `FlattenConsecutive(2)` adımı alıcı alanı iki katına çıkarır (1 katman 2 karakter görür, 2 katman 4, 3 katman 8). İndigo çizgiler, çıktının hangi girdilere bağlı olduğunu (alıcı alan) gösterir; düz MLP tüm bağlamı sığ tek hamlede, WaveNet derin + hiyerarşik birleştirir.

13.3 Başlangıç Kodu

Karpathy net bir şey söyler: bu dersin başlangıç kodu **Ders 3'ün MLP'sidir** — Ders 4 ve 5 birer kenar duraktı.

> *"The starter code for part five is very similar to where we ended up in part three. Recall that part four was the manual backprop exercise — that is kind of an aside."* — Karpathy, 1:43

Yani ana hat: Ders 3 (embedding + MLP) → Ders 6 (aynı modeli hiyerarşik + PyTorch-laştırılmış). Ders 4'te kurduğumuz modüller (Linear/BatchNorm1d/Tanh) başlangıç noktası; bu derste Embedding/Flatten/Sequential ekleyip yapıyı tamamlayacağız.

> Builder Notu — Ana Hat ve Aside'lar
>
> **Geriye (Ders 1-5):** "Hangi ders neyin üstüne kurulur" netliği önemli: ana hat Ders 1 (autograd) → Ders 2 (bigram) → Ders 3 (MLP) → **Ders 6 (WaveNet)**; Ders 4 (BatchNorm) ve Ders 5 (manuel backprop) derinleştirici aside'lar. Bu, serinin kasıtlı pedagojik yapısı.
> **İleriye:** Başlangıç kodunu "hazır modüllerle temiz kurmak", production'da yeni bir modele başlarken iyi bir alışkanlık — sıfırdan değil, denenmiş bloklardan.

13.4 Loss Düzleştirme

Karpathy önce küçük bir temizlik yapar: ham loss grafiği gürültülü ve okunaksız (her minibatch loss'u zıplıyor).

> *"Okay first let's fix this graph because it is daggers in my eyes and I just can't take it anymore."* — Karpathy, 6:58

Çözüm: loss kayıtlarını (`lossi`) 1000'lik gruplara böl, her grubun **ortalamasını** al. Bu, gürültüyü düzleştirip eğilimi (trend) görünür kılar.

```
# lossi: her adimin loss'u (cok gurultulu)
plt.plot(torch.tensor(lossi).view(-1, 1000).mean(1))   # 1000'lik gruplarin ortalamasi
```

`.view(-1, 1000)` loss listesini 1000-sütunlu satırlara böler, `.mean(1)` her satırın ortalamasını alır → düzgün bir trend eğrisi. Bizim çekirdeğimizde bu, `smooth_loss(lossi, k=1000)` fonksiyonudur; düzleştirilmiş eğriyi **WaveNet Eğitme** bölümündeki eğitim figüründe kullanırız.

> Builder Notu — .view + .mean Düzleştirme
>
> **Geriye (Ders 3 + Stat 110):** `.view` (Ders 3) ile yeniden şekillendirip `.mean` ile gruplama; minibatch gürültüsünün ortalama ile azalması (Stat 110: varyans $\propto 1/B$).
> **İleriye:** Loss eğrisini düzleştirme (moving average / smoothing), her eğitim panelinin (W&B, TensorBoard) standart görselleştirmesi — ham gürültü trendi gizler.

13.5 PyTorch Modülleri

Ders 4'te Linear/BatchNorm1d/Tanh modüllerini kurmuştuk. Şimdi eksik parçaları ekleyip yapıyı `torch.nn` gibi tamamlarız: **Embedding** (id → vektör arama), **Flatten** (boyut düzleştirme), ve hepsini saran **Sequential** konteyner.

> *"There's a Sequential [container]..."* — Karpathy, 13:13

```
class Embedding:
    def __init__(self, num_embeddings, embedding_dim):
        self.weight = torch.randn((num_embeddings, embedding_dim))
    def __call__(self, IX):
        self.out = self.weight[IX]   # id -> satir (Ders 2-3: lookup)
        return self.out
    def parameters(self):
        return [self.weight]

class Flatten:
    def __call__(self, x):
        self.out = x.view(x.shape[0], -1)   # (B, ...) -> (B, duz)
        return self.out
    def parameters(self):
        return []

class Sequential:
    def __init__(self, layers):
        self.layers = layers
    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        return x
    def parameters(self):
        return [p for layer in self.layers for p in layer.parameters()]
```

Artık tüm model tek bir `Sequential([...])` olarak kurulur — `model(x)` ileri geçişi, `model.parameters()` optimizer'ı besler. Bu, `torch.nn.Module` + `nn.Sequential` API'sinin birebir taklidi (Ders 1'in Neuron/Layer/MLP arabiriminin olgunlaşmış hâli).

> Builder Notu — call + parameters Arabirimi
>
> **Geriye (Ders 1-4):** `Embedding`, Ders 2-3'ün id→satır lookup'ı; `Sequential`, Ders 1'in MLP istifleme mantığı. `__call__` + `parameters()` arabirimi seri boyunca sabit.
> **İleriye:** `nn.Embedding`, `nn.Flatten`, `nn.Sequential` — bunlar gerçek PyTorch katmanları; bu derste neden öyle tasarlandıklarını anlayarak kullanırsın. Transformer da (Ders 7) bu modüler bloklardan kurulur.

13.6 Hiyerarşik Fikir

WaveNet'in mimarisi, kulağa korkutucu gelen ama aslında basit bir fikre dayanır:

> *"In the WaveNet's case, this is a visualization of a stack of dilated causal convolution layers — and this makes it sound very scary, but actually the idea is very simple."* — Karpathy, 19:04

Sezgi: tüm bağlamı tek katmanda yutmak yerine, bilgiyi bir **ağaç** gibi kademeli birleştir. 8 karakterlik bağlamda: 1. katman komşu çiftleri (4 çift) füzyonlar, 2. katman bu 4 sonucu 2'şer füzyonlar (2 sonuç), 3. katman son 2'yi birleştirir (1 temsil). Her katman, alıcı alanı (receptive field) iki katına çıkarır — logaritmik derinlikte ($\log_2 8 = 3$) tüm bağlama ulaşılır.

Bu, "yereli önce, küreseli sonra" prensibidir: alt katmanlar bigram-benzeri yerel desenleri, üst katmanlar geniş bağlamı yakalar.

> Builder Notu — Sabit Ağaç vs Öğrenilen Dikkat
>
> **İleriye (Ders 7):** Hiyerarşik/ağaç yapısı, **dilated causal convolution** (genişletilmiş nedensel evrişim) olarak da bilinir — "nedensel" çünkü yalnızca geçmişe bakar (geleceğe değil), "dilated" çünkü her katman daha geniş bir aralığı kapsar. Bu, ses üretiminden (WaveNet) zaman serilerine kadar geniş kullanım alanı bulur. Ders 7'nin transformer'ı bunu **yeniden organize eder**: sabit ağaç yerine **her token her tokena** öğrenilen ağırlıklarla bakar (attention). WaveNet hangi öğenin hangisiyle birleşeceğini önceden sabitler; attention bunu veriye bırakır — bu yüzden daha esnek, ama $O(T^2)$ maliyetiyle daha pahalı.

13.7 Bağlam 3-8

Hiyerarşik yapının değerli olması için daha uzun bağlam gerekir. Karpathy `block_size`'ı 3'ten **8**'e çıkarır. MLP'de 8 karakteri tek seferde düzleştirmek mümkün ama verimsiz; WaveNet'in ağaç yapısı 8 karakteri 3 katmanda ($8 \to 4 \to 2 \to 1$) kademeli işler.

Bağlam büyüdükçe model daha uzak geçmişi görebilir — "emma"da 'e' ile 'a' arasındaki ilişki gibi. Ama bu, tensör şekillerini (B, T, C) yönetmeyi zorlaştırır; dersin geri kalanı büyük ölçüde **doğru şekli korumakla** ilgili.

Bağlamı tek başına 3'ten 8'e çıkarmak bile kaybı düşürür: bizim ölçümümüzde düz MLP-flat-3'ün final dev kaybı $2{,}2484$ iken, aynı düz mimari bağlam 8'e çıkarılınca (MLP-flat-8) $2{,}1872$'ye iner. WaveNet'in hiyerarşik ağacı bunu daha da öteye götürür ($2{,}1324$) — bunu **WaveNet Eğitme** bölümünde ölçeceğiz.

> Builder Notu — block_size = Bağlam Uzunluğu
>
> **Geriye (Ders 3):** `block_size`, Ders 3'te tanımladığımız bağlam uzunluğu; burada 3→8 büyütülüyor. Veri seti (`build_dataset`) aynı kayan-pencere mantığıyla çalışır, yalnızca pencere geniş.
> **İleriye:** Bağlam uzunluğu (context length), her dil modelinin temel kapasite eksenidir — GPT-2'de 1024, modern modellerde yüz binlerce token (Ders 10). Uzun bağlam = daha çok bellek + hesap.

13.8 Batched matmul

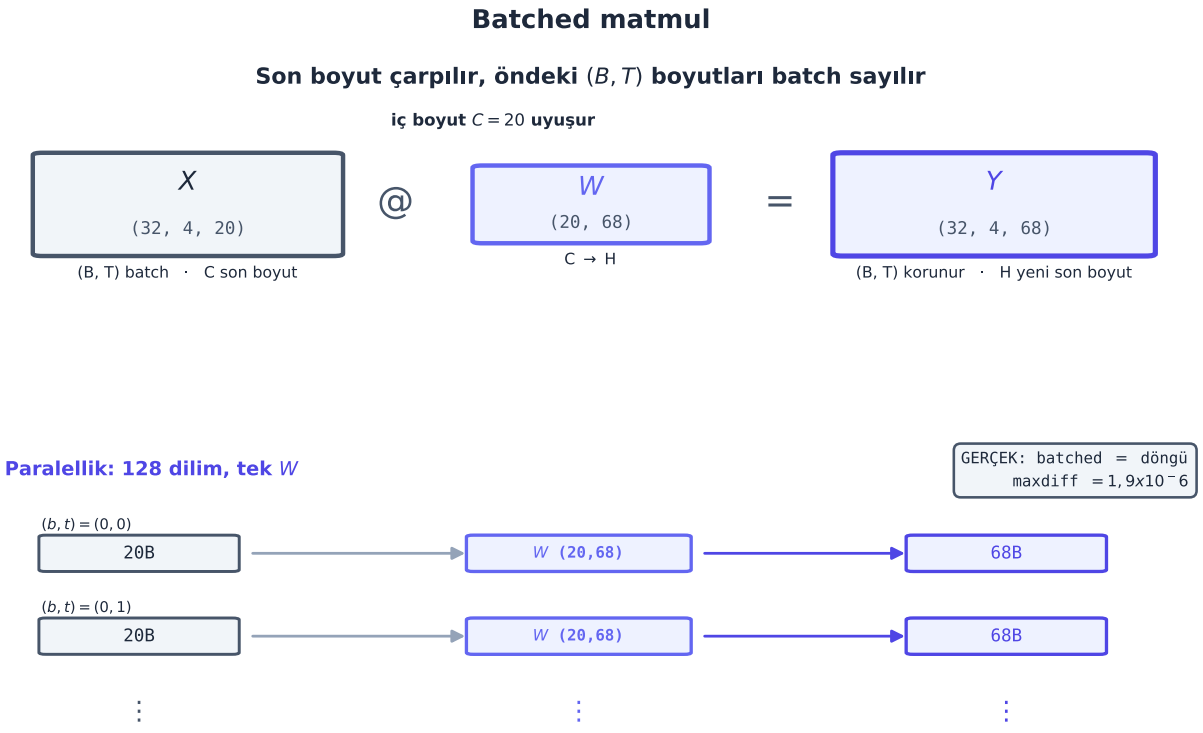
Hiyerarşik füzyonu nasıl koda dökeriz? Karpathy şaşırtıcı ama güçlü bir PyTorch özelliğini kullanır: **matris çarpımı son boyut üzerinde çalışır, öndeki boyutları batch sayar.**

> *"The surprising thing that I'd like to show you, that you may not expect, is that this input that is being multiplied doesn't actually have to be 2-dimensional."* — Karpathy, 24:44

Yani şekli (B, T, C) olan bir tensörü W (C, H) ile çarparsan, sonuç (B, T, H) olur — tüm (B, T) çiftleri **paralel** işlenir. Bu, hiyerarşik füzyonu mümkün kılar: bağlamı (B, T, C) gibi 3 boyutlu tut, ardışık çiftleri grupla, her grubu aynı lineer katmandan paralel geçir.

```
# (B, T, C) @ W(C, H) -> (B, T, H): B ve T batch gibi, C son boyutta carpilir
# WaveNet: ardisik ciftleri grupla -> (B, T/2, 2*C) -> Linear -> (B, T/2, H)
```

Anahtar: lineer katman değişmedi (Ders 1'in $Wx + b$'si); değişen, girdiyi nasıl **gruplayıp** ona beslediğimiz. Bu gruplama işini bir sonraki bölümdeki `FlattenConsecutive` yapar.



Şekil 13.4: Batched matmul: `(B, T, C)` @ W son boyutu C çarpar, öndeki `(B, T)` boyutlarını **batch** sayar — sonuç `(B, T, H)`. GERÇEK ($B = 32, T = 4, C = 20, H = 68$): WaveNet'in ilk bloğunda $(32, 4, 20)@(20, 68) \rightarrow (32, 4, 68)$. Aynı W (Linear katmanı), $B \cdot T = 128$ dilime **paralel** uygulanır: her (b, t) konumundaki 20-boyutlu vektör tek başına W ile çarpılır. Doğrulama: tüm tensörü tek hamlede çarpmak ile T ekseni üzerinde tek tek dolaşıp çarpmak aynı sonucu verir (maxdiff $= 1{,}9 \times 10^{-6}$, kayan-nokta gürültüsü). Anahtar: Linear değişmedi (Ders 1'in Wx'i); değişen, girdiyi nasıl gruplayıp ona beslediğimiz.

> Builder Notu — Batched matmul = Linear'ın Çok-Boyutlu Hâli
>
> **Geriye (Ders 1 + 18.06):** Batched matmul, Ders 1'in lineer katmanının çok-boyutlu hâli (18.06 matris çarpımı, son eksende). "Öndeki boyutlar batch" kuralı, tensör hesabının her yerinde geçerli.
> **İleriye:** Bu batched-matmul deseni, transformer'da (Ders 7) attention'ın çok başlı (multi-head) hesabının da temeli: (B, head, T, d) şekilli tensörlerde son boyutlarda matmul, öndekiler batch. GPU bu paralelliği sever (yüksek throughput).

13.9 FlattenConsecutive

Hiyerarşik füzyonun çekirdeği: **ardışık öğeleri gruplama**. Karpathy `FlattenConsecutive(n)` adlı bir modül yazar — şekli (B, T, C) olan bir tensörü alıp ardışık n öğeyi birleştirir: (B, T, C) → (B, T/n, C·n).

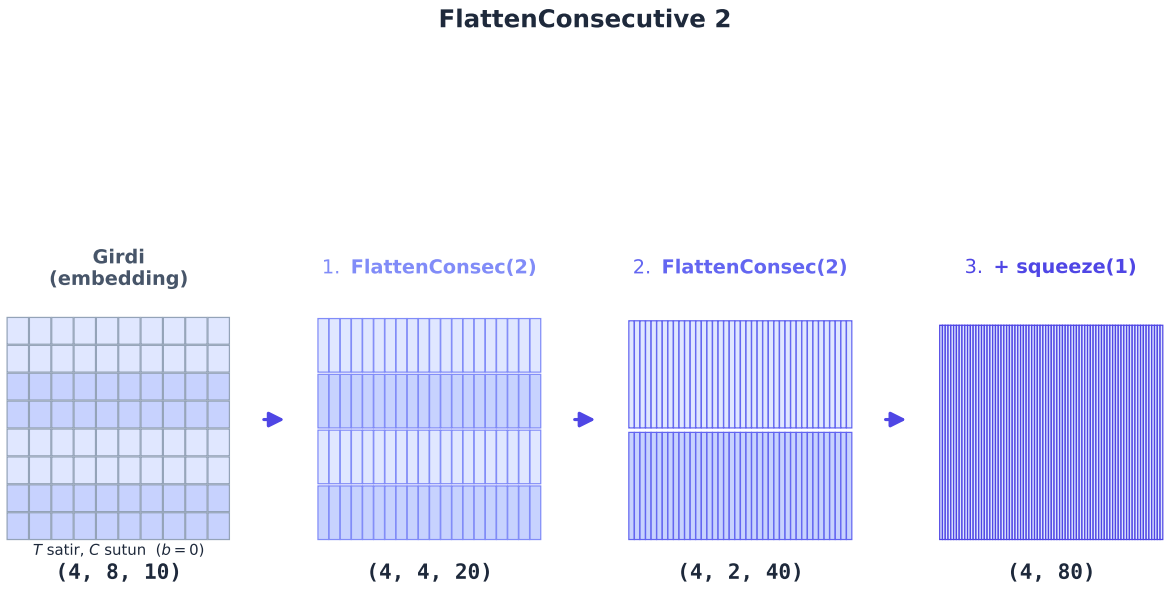
```
class FlattenConsecutive:
    def __init__(self, n):
        self.n = n
    def __call__(self, x):
        B, T, C = x.shape
        x = x.view(B, T // self.n, C * self.n)   # ardisik n ogeyi birlestir
        if x.shape[1] == 1:
            x = x.squeeze(1)   # T/n == 1 ise o boyutu kaldir
        self.out = x
        return self.out
    def parameters(self):
        return []
```

Örneğin $n = 2$ ile (4, 8, 10) → (4, 4, 20): 8 zaman-adımı 4 çifte iner, her çiftin 10-boyutlu embedding'leri yan yana gelip 20-boyut olur. Sonra bir Linear bu 20'yi işler. Bunu üst üste istifleyince (FlattenConsecutive(2) → Linear → BatchNorm → Tanh blokları) ağaç yapısı oluşur: $8 \to 4 \to 2 \to 1$. `squeeze(1)`, son katmanda $T/n = 1$ olunca gereksiz boyutu temizler.

> Builder Notu — FlattenConsecutive = Elle Yazılmış Convolution
>
> **Geriye (Ders 3):** `.view` ile yeniden gruplama, Ders 3'ün `.view(-1, 6)`'sının genellemesi — orada tüm bağlamı tek seferde, burada kademeli. `squeeze` ise 1-boyutlu ekseni atar (broadcasting'in tersi gibi).
> **İleriye:** FlattenConsecutive, aslında elle yazılmış bir **convolution** (kayan grup) — `torch.nn.Conv1d` bunu daha genel yapar. Bu, "katmanı kendin yazınca kütüphane katmanını anlarsın" prensibinin tekrarı.

13.10 WaveNet Eğitme

Tüm blokları bir Sequential'da istifleriz: Embedding → (FlattenConsecutive(2) → Linear → BatchNorm1d → Tanh) ×3 → Linear (çıkış). Eğitim döngüsü Ders 3-4'ün aynısı (forward → cross_entropy → backward → update).



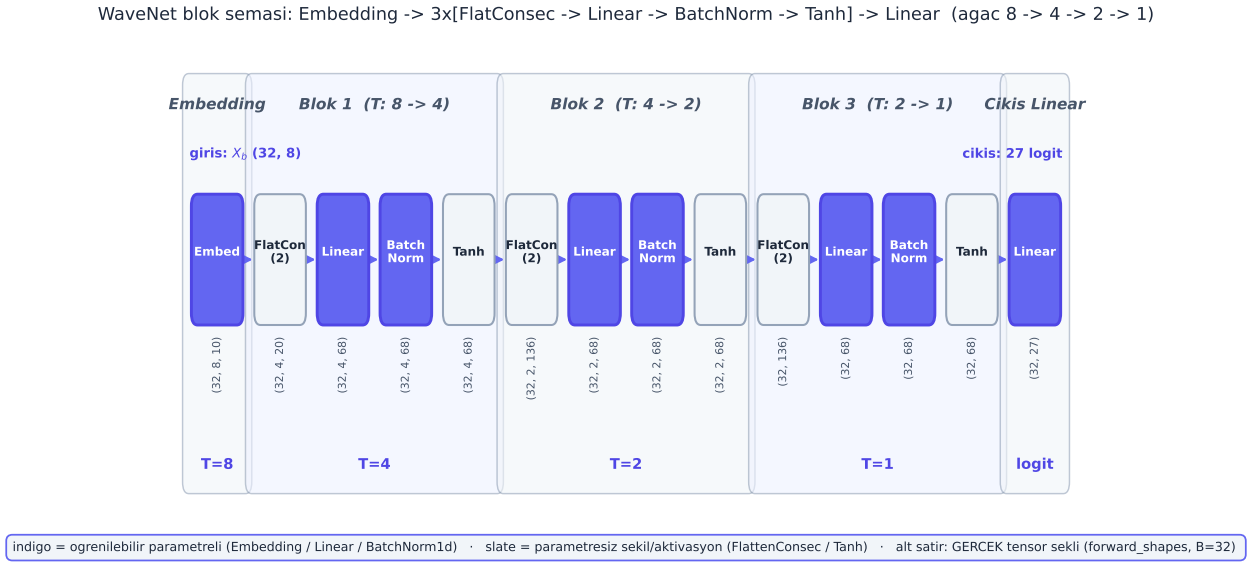
Şekil 13.5: `FlattenConsecutive(2)`, ardışık çiftleri yan yana birleştirir: `(B, T, C)` → `(B, T/2, 2C)`. Aynı katman üst üste istiflenince WaveNet ağacı çıkar: $(4, 8, 10) \to (4, 4, 20) \to (4, 2, 40) \to (4, 80)$ — zaman ekseni T her adımda yarıya iner ($8 \to 4 \to 2 \to 1$), kanal ekseni C iki katına çıkar ($10 \to 20 \to 40 \to 80$); öğe sayısı (izgara hücresi) B hariç sabit kalır (80). Son adımda $T/2 = 1$ olunca `squeeze(1)` o ekseni atar: $(4, 1, 80) \to (4, 80)$. Bu, elle yazılmış bir convolution adımıdır — yereli (komşu çiftleri) önce füzyonlar.

```
model = Sequential([
    Embedding(vocab_size, n_embd),
    FlattenConsecutive(2), Linear(n_embd*2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh(),
    FlattenConsecutive(2), Linear(n_hidden*2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh()
    FlattenConsecutive(2), Linear(n_hidden*2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh()
    Linear(n_hidden, vocab_size),
])
```

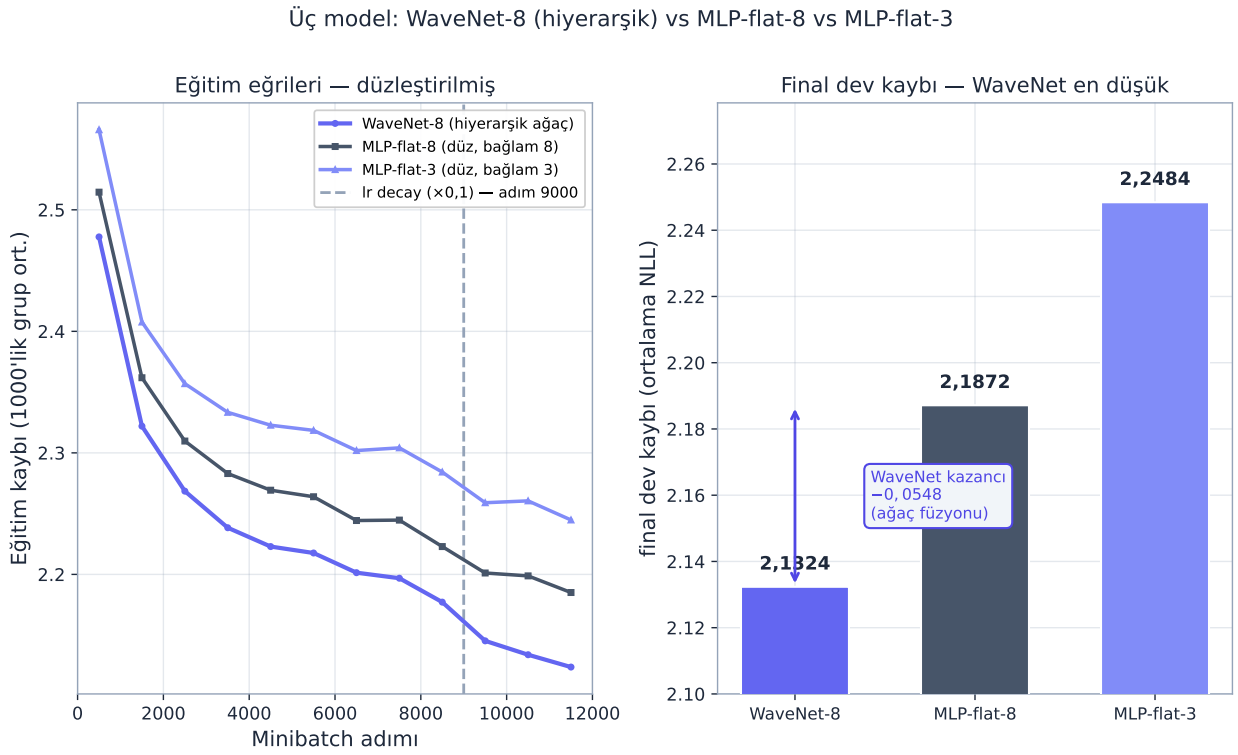
Aşağıdaki blok şeması tam yapıyı GERÇEK tensör şekilleriyle gösterir: T ekseni ağaç gibi $8 \to 4 \to 2 \to 1$ inerken kanal ekseni C çift FlattenConsec ile geçici olarak ikiye katlanır ($10 \to 20$, $68 \to 136$) ve Linear onu tekrar 68'e indirir.

İlk eğitimde loss biraz iyileşir ama Karpathy bir bug fark eder: BatchNorm1d 3 boyutlu girdiyle (B, T, C) yanlış çalışıyor. Sıradaki bölümde düzeltilir. Düzeltmeden sonra üç modeli (12000 adım, 9000. adımda lr $\times 0{,}1$ decay, aynı tohum) kıyaslarız:

GERÇEK ölçüm net bir sıralama verir: **WaveNet-8** $= 2{,}1324 <$ **MLP-flat-8** $= 2{,}1872 <$ **MLP-flat-3** $= 2{,}2484$ (ortalama dev NLL). Hiyerarşik ağaç füzyonu, düz MLP'nin tek-seferlik düzleştirmesini geçer; WaveNet'in MLP-flat-8'e göre kazancı $0{,}0548$'dir.



Şekil 13.6: Tam WaveNet blok şeması (GERÇEK batch $B = 32$): soldan sağa **Embedding** $\rightarrow$ $3\times$[**FlattenConsec(2)** $\rightarrow$ **Linear** (bias=False) $\rightarrow$ **BatchNorm1d** $\rightarrow$ **Tanh**] $\rightarrow$ **Linear** (çıkış). Her aşamanın altında `forward_shapes` cache'inden okunan GERÇEK tensör şekli yazılır; zaman ekseni T ağaç gibi $8 \rightarrow 4 \rightarrow 2 \rightarrow 1$ inerken kanal ekseni C çift FlattenConsec ile geçici olarak ikiye katlanır ($10 \rightarrow 20$, $68 \rightarrow 136$) ve Linear onu tekrar 68'e indirir. Üç FlattenConsec(2) katmanı $= \log_2 8 = 3$: blok başına alıcı alan iki katına çıkar. Son FlattenConsec(2)'de $T/n = 1$ olunca `squeeze(1)` ile zaman ekseni düşer (şekil $(32, 136)$, 3B'den 2B'ye); ardından çıkış Linear'ı 27 logit verir. İndigo dolgu = öğrenilebilir parametreli katmanlar (Embedding/Linear/BatchNorm1d — Embedding'in $(27, 10) = 270$ elemanlı arama tablosu da öğrenilir), slate dolgu = parametresiz şekil/aktivasyon katmanları (FlattenConsec/Tanh).



Şekil 13.7: Eğitim eğrileri — üç model, 12000 adım minibatch SGD (9000. adımda lr $\times 0{,}1$ decay), aynı tohum (SEED), GERÇEK ölçüm. **Sol:** üç modelin düzleştirilmiş (1000'lik grup ortalaması) eğitim kaybı eğrisi — WaveNet-8 (indigo), MLP-flat-8 (slate), MLP-flat-3 (açık indigo). **Sağ:** final dev kaybı (ortalama NLL): **WaveNet-8** $= 2{,}1324 <$ **MLP-flat-8** $= 2{,}1872 <$ **MLP-flat-3** $= 2{,}2484$. Hiyerarşik ağaç füzyonu (8→4→2→1) + uzun bağlam (8 karakter), düz MLP'nin tek-seferlik düzleştirmesini geçer; bağlamı 3'ten 8'e çıkarmak tek başına da kaybı düşürür (MLP-flat-3 → MLP-flat-8). Slate+Indigo; deterministik.

Builder Notu — BatchNorm WaveNet Bloğunda da, Şimdi 3B-fix

Geriye (Ders 3-4): Eğitim döngüsü hiç değişmedi (Ders 1'den beri aynı); değişen yalnızca model mimarisi (düz MLP → hiyerarşik). `bias=False` (BatchNorm öncesi) Ders 4'ün spurious-bias dersi. **Ders 4'te kurduğumuz BatchNorm1d, WaveNet bloğunda da kullanılır** — ama girdi artık 3 boyutlu (B, T, C); bu yüzden bir sonraki bölümde 3B-fix gerekecek.
İleriye: "İlk geçişte bug çıkar, düzelt" Karpathy'nin kasıtlı pedagojisi (Ders 1 `+=` / `zero_grad` gibi); gerçek geliştirme süreci de böyle ilerler.

13.11 BatchNorm 3B Bug

Bug şu: Ders 4'te `BatchNorm1d`'i 2 boyutlu girdi (B, C) için yazmıştık — istatistikleri yalnızca 0. eksen (batch) üzerinden alıyordu. Ama WaveNet'te girdi 3 boyutlu (B, T, C); bu durumda ortalama/varyans **hem batch hem zaman** ekseni üzerinden alınmalı, yani $(0, 1)$ boyutları.

```
def __call__(self, x):
    if x.ndim == 2:
        dim = 0          # (B, C): batch ekseni
    elif x.ndim == 3:
        dim = (0, 1)     # (B, T, C): batch VE zaman ekseni
    xmean = x.mean(dim, keepdim=True)
    xvar = x.var(dim, keepdim=True)
    # ... normalize + scale/shift (Ders 4)
```

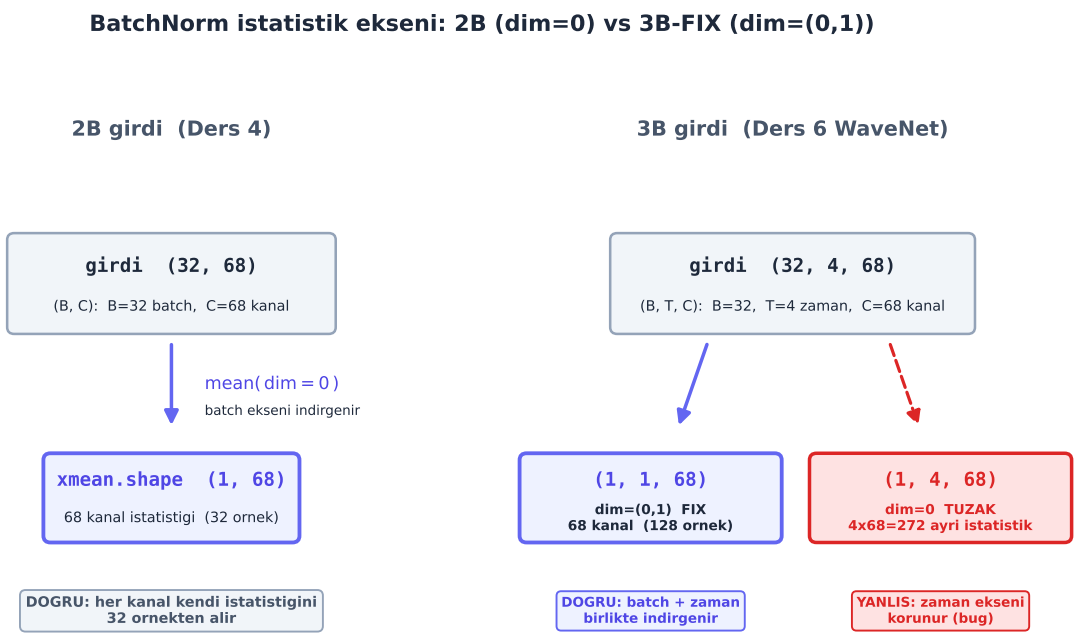
GERÇEK ölçümle: 2B girdi $(32, 68)$'de `dim=0` → `xmean.shape` $(1, 68)$ (kanal başına 32 örnek). 3B girdi $(32, 4, 68)$'de doğru `dim=(0,1)` → $(1, 1, 68)$ (kanal başına $B \cdot T = 128$ örnek). Eğer 2B-only kod körü körüne `dim=0` uygularsa şekil $(1, 4, 68)$ olur — zaman ekseni yanlışlıkla korunur, $4 \times 68 = 272$ ayrı istatistik kalır.

Karpathy bu dersteki işin çoğunun **şekil yönetimi** olduğunu itiraf eder:

> *"I'm spending a lot of time basically babysitting the shapes and making sure everything is correct."* — Karpathy, 53:44

Builder Notu — Hangi Eksende Normalize? (Ders 5 Köprüsü)

Geriye (Ders 4-5): Bu bug, Ders 4'ün BatchNorm'unun ve Ders 5'in "broadcast/sum dualitesi"nin doğrudan sonucu: hangi eksende normalize/topla sorusu. 2B vs 3B ayrımı tam da bir "şekil tuzağı". **Ders 5'te backward'ı elle yazdığımız için** hangi eksenin neyi taşıdığını sezgisel biliyoruz; o yüzden artık autograd'a güvenle dönebiliyoruz — `loss.backward()` 3B'de de doğru aksar, çünkü forward'daki `dim` seçimini doğru yaptık.
İleriye: "Şekilleri kollamak" (shape babysitting), tensör programlamanın günlük gerçeği — production'da NaN/yanlış-sonuç bug'larının büyük kısmı şekil/eksen hatalarıdır. `print(x.shape)` (Ders 1) ve dikkatli eksen seçimi şart.



Şekil 13.8: BatchNorm istatistik ekseni: 2B vs 3B (GERÇEK şekiller, $C = 68$ kanal). **Sol (Ders 4, 2B):** girdi `(B, C)` $= (32, 68)$; ortalama/varyans yalnızca batch ekseni üzerinden alınır (`dim=0`), sonuç xmean.shape $= (1, 68)$ — kanal başına 32 örneğin tek istatistiği. **Sağ (Ders 6, 3B-FIX):** WaveNet girdisi `(B, T, C)` $= (32, 4, 68)$; istatistik hem batch hem zaman ekseni üzerinden alınmalı (`dim=(0,1)`), sonuç $(1, 1, 68)$ — kanal başına $B \cdot T = 128$ örneğin tek istatistiği. **Tuzak (kırmızı):** Ders 4'ün 2B-only kodu 3B girdide körü körüne `dim=0` uygularsa şekil $(1, 4, 68)$ olur — zaman ekseni yanlışlıkla korunur, $4 \times 68 = 272$ ayrı istatistik kalır (her zaman-adımı kendi istatistiğini sanır). 3B-FIX, normalize edilmeyen tüm eksenleri (batch + zaman) tek seferde indirir.

13.12 WaveNet Büyütme

Bug düzeldikten sonra Karpathy modeli büyütür: daha çok kanal (channel), daha geniş katmanlar. Bizim son modelimiz 68 gizli kanal genişliğinde, 22 397 parametreli; loss, Ders 3'ün düz MLP'sinden (2,2484) belirgin biçimde daha iyiye (2,1324) iner — hiyerarşik yapı + uzun bağlam işe yaradı. Bağlam 8 karakter + ağaç füzyonu, modele daha zengin desenler öğrenme imkânı verdi.

Karpathy ayrıca eğitim sürecinin gerçekçi yanını gösterir: deney kaydı tutmak (performans günlüğü), hiperparametre denemeleri, "babysitting" — gerçek ML geliştirme sürecinin doğal parçaları.

> Builder Notu — Scaling Laws'ın Küçük-Ölçek Önizlemesi
>
> **İleriye:** Model büyütmenin loss'u düşürmesi, **scaling laws**'ın küçük-ölçek önizlemesidir (Ders 10): parametre/bağlam arttıkça performans artar (bir noktaya kadar). Düz MLP-flat-3 (4 309 param) → MLP-flat-8 (7 709 param) → WaveNet-8 (22 397 param) zincirinde loss tek yönde düşer. Deney kaydı + hiperparametre arama, MLOps'un (W&B, Optuna) çekirdeği.

13.13 Notlar

Karpathy kapanışta birkaç bağlantı kurar:

Convolutions. Kurduğumuz hiyerarşik yapı, aslında elle yazılmış bir **convolution** ağıdır. Gerçek WaveNet, `torch.nn.Conv1d` ile kayan filtreler (dilated causal) kullanır — ama mekanizma aynı: yerel pencereleri paylaşılan ağırlıklarla işle, hiyerarşik istifle.

Neden torch.nn değil? Kendi modüllerimizi (Linear/BatchNorm1d/Embedding/Flatten/Sequential) yazdık — `torch.nn`'i doğrudan kullanmak yerine. Sebep pedagojik: her parçanın içini görmek. Artık `torch.nn`'e baktığında, arkasında ne olduğunu biliyorsun.

Geliştirme süreci. Karpathy, gerçek araştırma/geliştirmenin nasıl göründüğünü gösterir: dokümantasyonla boğuşmak, şekil hatalarını ayıklamak, deney kaydı tutmak, kademeli iyileştirme. "Temiz nihai kod" değil, **süreç**.

> Builder Notu — makemore Kapanışı, Ders 7'de Attention
>
> **İleriye (Ders 7):** Bu ders, makemore serisinin kapanışı. Bundan sonra (Ders 7) **transformer**'a geçiyoruz — hiyerarşik convolution yerine **dikkat (attention)** ile bağlam işleme. WaveNet "sabit ağaç" kurar; transformer her token'ın her tokena bakmasına izin verir (daha esnek, daha pahalı). Başka deyişle attention, WaveNet'in **yeniden organize edilmiş** hâli: sabit ağaç füzyonu yerine, hangi token'ın hangisine bakacağını model kendisi öğrenir. İkisi de "uzun bağlamı verimli işleme" probleminin farklı çözümleri.

13.14 Bu Dersin Özeti

1. Ders 3 MLP'si bağlamı **tek seferde** düzleştiriyordu; WaveNet bunu **kademeli/hiyerarşik** (ağaç) füzyona çevirir.
2. **Başlangıç kodu Ders 3'ten** gelir (Ders 4 BatchNorm + Ders 5 manuel backprop birer **aside**'dı).

3. Kod **PyTorch-laştırılır**: Embedding (id→vektör), FlattenConsecutive (ardışık gruplama), Sequential (konteyner) — `torch.nn` taklidi tamamlanır.
4. **WaveNet fikri**: bilgiyi ağaç gibi kademeli füzyonla ($8 \to 4 \to 2 \to 1$); alıcı alan her katmanda iki katına çıkar ($\log_2 8 = 3$ katman, dilated causal convolution).
5. **Bağlam 3 → 8** karaktere çıkarılır; daha uzun geçmiş = daha zengin desen (tek başına bile dev loss $2{,}2484 \to 2{,}1872$).
6. **Batched matmul**: (B, T, C) @ W son boyutta çarpar, (B, T) batch sayılır — hiyerarşik füzyonu paralel kılar ($B \cdot T = 128$ dilim, tek W).
7. **FlattenConsecutive(n)**: (B, T, C) → (B, T/n, C·n) ardışık gruplama; ağaç katmanlarının çekirdeği (squeeze, $T/n = 1$ olunca).
8. **BatchNorm1d 3B bug'ı**: 2B (B,C) için yazılmıştı (xmean $(1, 68)$); 3B (B,T,C) için istatistik ekseni $(0, 1)$ olmalı (xmean $(1, 1, 68)$, 128 örnek/kanal).
9. WaveNet, elle yazılmış bir **convolution**'dır; final dev loss WaveNet-8 $= 2{,}1324$ (22 397 param) düz MLP'leri geçer. Transformer (Ders 7) aynı "uzun bağlam" problemini **dikkat** ile çözer.

! Tek Bir Cümle

Bağlamı tek hamlede düzleştirip ezmek yerine, ardışık öğeleri kademeli (ağaç gibi) füzyonlamak — WaveNet'in hiyerarşik fikri — daha derin ve güçlü bir dil modeli verir; ve bu, aslında elle yazılmış bir convolution'dır (yerel desenleri önce, küresel yapıyı sonra öğrenen).

13.15 Kontrol Soruları

i Soru 1: Şekli (4, 8, 10) olan bir tensör FlattenConsecutive(2)'ye verilirse çıktı şekli ne olur? Adımları açıkla.

Cevap: Çıktı **(4, 4, 20)**. Açıklama: (B, T, C) = (4, 8, 10). FlattenConsecutive(2), ardışık $n = 2$ öğeyi birleştirir: T 8'den $8/2 = 4$'e iner, C 10'dan $10 \cdot 2 = 20$'ye çıkar. Yani `x.view(B, T//n, C*n) = x.view(4, 4, 20)`. 8 zaman-adımı 4 komşu çifte gruplandı; her çiftin 10-boyutlu embedding'leri yan yana gelip 20-boyut oldu. Bunu üst üste istifleyince (her aşamada bir Linear araya girse de): $(4, 8, 10) \to (4, 4, 20) \to (4, 2, 40) \to (4, 1, 80) \to$ squeeze $\to (4, 80)$. Ağaç: $8 \to 4 \to 2 \to 1$ (GERÇEK ölçüm).

i Soru 2: WaveNet'in hiyerarşik (kademeli) füzyonu, Ders 3 MLP'sinin 'hepsini tek seferde düzleştir' yaklaşımından neden daha iyi olabilir?

Cevap: MLP, 8 karakterin tüm embedding'lerini **ilk katmanda birden** karıştırır — yerel yapı (komşu karakterler arası ilişki) ile uzak yapı aynı anda, ayrımsız işlenir. WaveNet ise **kademeli** işler: alt katmanlar yerel desenleri (komşu çiftler, bigram-benzeri), üst katmanlar giderek daha geniş bağlamı yakalar. Bu "yereli önce, küreseli sonra" yapısı (a) hiyerarşik dil yapısına (harfler→heceler→kelimeler) daha uygun bir tümevarımsal önyargı (inductive bias) taşır, (b) derin temsil kurar. Sonuç: bizim ölçümümüzde aynı bağlamda (8 karakter) WaveNet dev loss $2{,}1324$ < düz MLP-flat-8 $2{,}1872$.

Soru 3: Ders 4'ün BatchNorm1d'i 2 boyutlu (B, C) girdi için yazılmıştı. WaveNet'in 3 boyutlu (B, T, C) girdisinde neden bug verdi, nasıl düzeltilir?

Cevap: BatchNorm, istatistikleri (ortalama, varyans) **normalize edilmeyen** eksenler üzerinden almalı. 2B (B, C)'de bu yalnızca batch ekseni (`dim=0`) — her kanal (C) kendi istatistiğini batch'ten alır (`xmean.shape` $(1, 68)$). Ama 3B (B, T, C)'de hem batch (B) hem zaman (T) ekseni üzerinden alınmalı (`dim=(0,1)`, `xmean.shape` $(1, 1, 68)$) — yoksa BatchNorm yalnızca batch'i hesaba katıp zaman boyutunu yanlış işler ($(1, 4, 68) \to 4 \times 68 = 272$ ayrı istatistik). Düzeltme: `x.ndim`'e göre `dim`'i seç (2B $\to$ 0, 3B $\to$ (0,1)). Bu, Ders 5'in "hangi eksende topla/normalize" (broadcast/sum dualitesi) sorusunun BatchNorm'a uygulanması.

Soru 4: (Builder) Hem WaveNet hem transformer (Ders 7) uzun bağlamı işler. İkisi bu problemi nasıl farklı çözer?

Cevap: WaveNet sabit, **hiyerarşik** bir yapı kurar: bağlamı ağaç gibi kademeli füzyonlar (dilated causal convolution); alıcı alan katman sayısıyla logaritmik büyür ($\log_2 8 = 3$), ama her token yalnızca belirli komşularına (ağacın yapısına göre) bakar. **Transformer** ise **dikkat (attention)** kullanır: her token, bağlamdaki **her** tokena doğrudan bakabilir (öğrenilen ağırlıklarla), yapı sabit değil veriye-bağlı. Transformer daha esnek (uzak bağımlılıkları tek adımda yakalar) ama daha pahalı ($O(T^2)$ — her çift token). WaveNet daha ucuz ($O(T)$) ama daha katı. Yani attention, WaveNet'in yeniden organize edilmiş hâlidir: sabit ağaç yerine öğrenilen bakış. İkisi de "sabit-bağlam MLP'nin sınırını aşma" probleminin farklı çözümleri; modern LLM'ler transformer'ı seçti.

13.16 Egzersizler

Egzersiz 1 (FlattenConsecutive). `FlattenConsecutive(n)` modülünü yaz. (4, 8, 10) şekilli bir tensöre $n = 2$ ile uygula, çıktının (4, 4, 20) olduğunu doğrula. $n = 2$'yi tekrar uygula $\to$ (4, 2, 40). `squeeze` mantığını da test et ($T/n = 1$ olunca $\to$ (4, 80)).

Egzersiz 2 (Modülleri kur). Embedding, Flatten, Sequential modüllerini (**PyTorch Modülleri** bölümü) ve FlattenConsecutive'i (**FlattenConsecutive** bölümü) yaz. WaveNet'i Sequential olarak istifle (Embedding $\to$ (FlattenConsecutive(2)$\to$Linear$\to$BatchNorm1d$\to$Tanh)×3 $\to$ Linear). Tek bir forward'ın çalıştığını (`model(Xb)` doğru şekil — (32, 27) logit — verdiğini) doğrula.

Egzersiz 3 (BatchNorm1d 3B düzeltmesi). BatchNorm1d'i hem 2B (B,C) hem 3B (B,T,C) girdiyi destekleyecek şekilde düzelt (`dim` seçimi: 0 vs (0,1)). 3B girdiyle istatistiklerin doğru eksende alındığını (`xmean.shape` $= (1, 1, 68)$) doğrula.

Egzersiz 4 (WaveNet'i eğit). Bağlamı 8 yap, WaveNet'i eğit. Loss'u Ders 3'ün düz MLP'siyle (bağlam 3) karşılaştır — hiyerarşik + uzun bağlamın daha düşük loss verdiğini gözlemle (bizde 2,1324 vs 2,2484). Loss grafiğini 1000'lik gruplarla düzleştirip (**Loss Düzleştirme** bölümü) çiz.

Egzersiz 5 (Sonraki dersin habercisi). WaveNet, bağlamı **sabit bir ağaç** yapısıyla füzyonlar — hangi karakterin hangisiyle birleşeceği önceden belli. Şimdi farklı bir fikir düşün: her karakterin, bağlamdaki **diğer karakterlerden hangilerine bakacağını kendisinin öğrenmesi** (sabit ağaç değil, veriye-bağlı). (a) Bu neden daha esnek olurdu? (b) Maliyeti ne olurdu (her token her tokena bakarsa)? Bu fikir **dikkat (attention)**

mekanizmasıdır ve Ders 7'de (**Sıfırdan GPT İnşa Etmek**) transformer'ı kuracağız — makemore serisini bırakıp doğrudan GPT'ye geçiyoruz.

13.17 Sonraki Ders İçin Hazırlık

Ders 7: Sıfırdan GPT İnşa Etmek (Transformer) — Andrej Karpathy

makemore serisi bitti. Ders 7'de doğrudan **transformer**'a — ChatGPT'yi çalıştıran mimariye — geçiyoruz. tiny Shakespeare veri setinde, karakter-düzeyli bir GPT'yi sıfırdan kuracağız: WaveNet'in sabit hiyerarşisi yerine **self-attention** (her token'ın diğerlerine öğrenilen ağırlıklarla bakması).

Ana konular:

- tiny Shakespeare + karakter tokenizer + bigram baseline (seriye bağlanış).
- Self-attention'ın kalbi: query/key/value, ölçekli nokta-çarpım dikkati.
- Multi-head attention, feed-forward, residual bağlantılar, pre-norm LayerNorm, dropout.

> ⚠ Ders 7 Öncesi Yapılacak
>
> - Egzersizleri çöz — özellikle 4 (WaveNet eğitimi) ve 5 (dikkat sezgisi).
> - "Modüller + Sequential = `torch.nn` taklidi" ve "uzun bağlamı verimli işleme" fikirlerini hatırla — transformer aynı problemi farklı çözer (sabit ağaç → öğrenilen dikkat).
> - Ders 4 (LayerNorm vs BatchNorm) ve Ders 5 (backprop ninja) — transformer'ın LayerNorm + residual'ını derin anlamak için faydalı.

13.18 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Karpathy'de
WaveNet / hiyerarşik füzyon	Bağlamı ağaç gibi kademeli birleştir ($8 \to 4 \to 2 \to 1$); MLP'nin tek-seferlik düzleştirmesine alternatif	1m25
Başlangıç kodu	Ders 3 MLP'sinden gelir; Ders 4-5 birer aside	1m43
Loss grafiği düzleştirme	`lossi.view(-1,1000).mean(1)`; gürültüyü silip trendi gösterir	6m58
Embedding modülü	`weight[IX]` ile id→vektör; Ders 2-3 lookup'ın modül hâli	9m19

Kavram	Tanım	Karpathy'de
Flatten / Sequential	Boyut düzleştirme + katman konteyneri; `torch.nn` taklidi tamamlanır	9m19
dilated causal convolution	WaveNet'in yapısı; nedensel (geçmişe bakar) + genişleyen alıcı alan	19m04
Bağlam 3 → 8	`block_size` büyütülür; ağaç yapısı 8 karakteri 3 katmanda işler	19m35
Batched matmul	(B,T,C) @ W son boyutta çarpar, (B,T) batch; hiyerarşik füzyonu paralel kılar	24m44
FlattenConsecutive(n)	(B,T,C) → (B, T/n, C·n); ardışık öğeleri gruplar (ağaç katmanı)	31m32
BatchNorm1d 3B düzeltmesi	2B'de `dim=0`, 3B'de `dim=(0,1)`; istatistik ekseni şekle göre	38m55
Şekil yönetimi	(B,T,C) tensörleri kollamak; bug'ların çoğu eksen/şekil hatası	53m44
Convolution bağlantısı	Kurduğumuz hiyerarşi = elle yazılmış convolution (`torch.nn.Conv1d`)	46m59

13.19 ML Builder Bağlantıları

9 köprü — WaveNet

1. **WaveNet / hiyerarşik füzyon** → Ders 3 MLP'nin derinleştirilmişi. İleriye: dilated causal conv (van den Oord 2016), CNN hiyerarşisi.
2. **Embedding / Flatten / Sequential** → Ders 1-4 modül arabirimi (`__call__` + `parameters`). İleriye: `torch.nn.Module`, `nn.Sequential`.
3. **Batched matmul (son boyut)** → Ders 1 lineer katman + 18.06. İleriye: multi-head attention'ın paralel hesabı (Ders 7).
4. **FlattenConsecutive = convolution** → Ders 3 `.view` genellemesi. İleriye: `torch.nn.Conv1d`, paylaşılan-ağırlık filtreler.
5. **BatchNorm1d 3B** → Ders 4 BatchNorm + Ders 5 broadcast/sum dualitesi. İleriye: dikkatli eksen seçimi (shape babysitting).
6. **Bağlam uzunluğu (3→8)** → Ders 3 `block_size`. İleriye: context length (GPT-2: 1024, Ders 10), uzun-bağlam modelleri.

7. **Loss düzleştirme** → Ders 3 minibatch gürültüsü + Stat 110 ortalama. İleriye: W&B/TensorBoard smoothing.
8. **Model büyütme → loss düşer** → scaling laws önizlemesi ($4309 \to 7709 \to 22397$ param, loss tek yönde düşer). İleriye: parametre/veri/compute dengesi (Ders 10).
9. **Hiyerarşik conv vs dikkat** → uzun-bağlamın iki çözümü. İleriye: transformer attention (Ders 7) — sabit ağaç yerine öğrenilen, esnek (WaveNet'in yeniden organize hâli).

13.20 Karpathy'nin Önerdiği Kaynaklar

Karpathy'nin bu ders için verdiği kaynaklar:

- **WaveNet makalesi:** arxiv 1609.03499 — van den Oord ve ark. 2016, dilated causal convolution ile ses üretimi.
- **Ders Colab notebook'u:** Google Colab.
- (Genel referans: github.com/karpathy/makemore — makemore serisinin tamamı.)

! Bu dersten tek bir şey alıp gideceksen

Bağlamı tek hamlede düzleştirip ezmek yerine, ardışık öğeleri kademeli (ağaç gibi) füzyonlamak — WaveNet'in hiyerarşik fikri — daha derin ve güçlü bir model verir. Ve bu yapı, aslında elle yazılmış bir convolution'dır. makemore serisi burada kapanır; Ders 7'de aynı "uzun bağlamı işleme" problemini bu kez **dikkat (attention)** ile, transformer kurarak çözeceğiz.

14 Sıfırdan GPT — Self-Attention ve Transformer

WaveNet'in sabit ağacı yerine öğrenilen iletişim: her token bir sorgu (query) ve anahtar (key) yayınlar, yakınlıklar softmax'lanır, değerler (value) ağırlıklı toplanır — bunun üzerine feed-forward, residual ve LayerNorm eklenince ChatGPT'yi çalıştıran GPT mimarisi ortaya çıkar

Bölüm bilgisi

- **Karpathy'nin videosu:** YouTube — Let's build GPT: from scratch, in code, spelled out (≈116 dk)
- **Seri:** Neural Networks: Zero to Hero — Ders 7 (**serinin doruğu**)
- **Hoca:** Andrej Karpathy
- **Kaynak repo:** github.com/karpathy/nanoGPT · ng-video-lecture
- **Okuma süresi:** ≈34 dk

14.1 Bu Derste Ne Var?

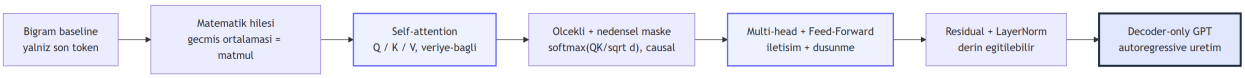
makemore serisi bitti. Bu derste doğrudan **transformer**'a — ChatGPT'yi çalıştıran mimariye — geçiyoruz. tiny Shakespeare veri setinde, karakter-düzeyli bir **GPT**'yi sıfırdan kuracağız. Bu, serinin doruğu: Ders 9 (tokenizer) ve Ders 10 (GPT-2) buna dayanır.

Büyük fikir, WaveNet'in **sabit ağacı** yerine **öğrenilen iletişim**: her token, bağlamdaki diğer token'lardan hangilerine ne kadar bakacağını **kendisi öğrenir** (self-attention). Sabit hiyerarşi yok; her token kendi "sorgusunu" yayınlar, diğerlerinin "anahtar"larıyla eşleşir, ve bulduğu "değer"leri ağırlıklı toplar.

> *"Now we get the crux of self-attention — this is probably the most important part of this video to understand."* — Karpathy, 1:01:58

Dersin üç büyük fikri:

1. **Bigram'dan transformer'a** — tiny Shakespeare, karakter tokenizer, bigram baseline (seriye bağlanış), sonra dikkat eklemek.
2. **Self-attention** — query/key/value, ölçekli nokta-çarpım dikkati: token'ların öğrenilen, veriye-bağlı iletişimi.
3. **Tam transformer bloğu** — multi-head attention + feed-forward + residual bağlantılar + pre-norm LayerNorm + dropout.



Şekil 14.1: Ders 7'nin kavram haritası: bigram baseline'dan başlayıp, geçmişin ortalamasını matris çarpımı olarak yazma hilesiyle self-attention'a (query/key/value) geçeriz; ölçekleme ve nedensel maske eklenir; çok başlı dikkat + feed-forward bir transformer bloğu kurar; residual ve LayerNorm derinliği mümkün kılar; sonuç decoder-only bir GPT'dir. Slate akış + indigo dönüm noktaları (self-attention çekirdeği ve tam blok).

Builder Notu — Geriye Ders 1-6, İleriye Ders 8-10

Geriye (Ders 1-6):

- **Bigram baseline = Ders 2.** Transformer'a, Ders 2'nin bigram'ından başlarız (`nn.Embedding` + cross-entropy + `generate`) — sonra üzerine dikkat ekleriz.
- **Modüller = Ders 6.** `nn.Module`/`Sequential` deseni (Ders 6'da olgunlaşan) ile blokları istifleriz.
- **LayerNorm = Ders 4.** Ders 4'te BatchNorm'u kurmuş, transformer'ın neden **LayerNorm** (her örneği bağımsız normalize) tercih ettiğini görmüştük.
- **Residual + FFN gradyanı = Ders 5.** backprop ninja sayesinde residual bağlantının gradyanı neden kolay akıttığını derinden anlıyoruz.

İleriye (Ders 8-10): Bu char-GPT, GPT-2'nin (Ders 10) küçük kardeşi — aynı mimari, daha küçük ölçek + karakter tokenizer (Ders 9'da gerçek BPE). Self-attention, tüm modern LLM'lerin (GPT, Claude, Llama) çekirdeği. Ders 8 (State of GPT) bu modelin nasıl asistana çevrildiğini anlatır.
Tek cümleyle: Transformer, token'ların birbirine **öğrenilen ağırlıklarla baktığı** (self-attention) bir iletişim mekanizmasıdır; bunun üzerine feed-forward, residual ve LayerNorm eklenince ChatGPT'yi çalıştıran GPT mimarisi ortaya çıkar.

14.2 Giriş: ChatGPT, Transformer ve nanoGPT

Karpathy ChatGPT'yle başlar: dil modelleri dünyayı salladı. Altındaki mimari **transformer** (2017, "Attention is All You Need"). Hedef: bu mimariyi sıfırdan, karakter-düzeyli bir dil modeli olarak kurmak — **nanoGPT**'nin eğitim versiyonu.

GPT'nin açılımı: **Generatively Pretrained Transformer**. "Transformer" mimari; "pretrained" devasa metin üzerinde "bir sonraki token'ı tahmin et" ile eğitilmiş (Ders 2-6'nın aynı çerçevesi, dev ölçekte). Bu derste mimariyi (transformer) kuruyoruz; ChatGPT'yi asistana çeviren ek aşamalar (SFT/RLHF) Ders 8'in konusu.

Builder Notu — En Etkili Makale

İleriye: "Attention is All You Need" (Vaswani 2017), modern yapay zekânın en etkili makalesi. Bu derste kuracağımız mimari, GPT-2/3/4, Claude, Llama'nın hepsinin temelidir — yalnızca ölçek (parametre, veri, bağlam) değişir.

14.3 Veri ve Karakter Tokenizer

Veri: **tiny Shakespeare** (Shakespeare metinlerinin tek bir dosyası, ~1 MB). Önce metni modele verilebilir sayılara çevririz — **tokenization**. Bu derste en basit yol: **karakter-düzeyli** tokenizer. Metindeki benzersiz karakterleri (65 tane) sıralayıp her birine bir tamsayı atarız (Ders 2'nin s2i/i2s'i).

```
chars = sorted(list(set(text)))      # 65 benzersiz karakter
stoi = {ch: i for i, ch in enumerate(chars)}
itos = {i: ch for i, ch in enumerate(chars)}
encode = lambda s: [stoi[c] for c in s]        # metin -> id listesi
decode = lambda l: ''.join(itos[i] for i in l)  # id listesi -> metin
data = torch.tensor(encode(text), dtype=torch.long)
```

Karpathy bunun **naif** bir tokenizer olduğunu vurgular: 65 karakter = küçük sözlük, uzun diziler. Gerçek GPT'ler **byte-pair encoding** (BPE) kullanır — bu Ders 9'un konusu. Şimdilik karakter yeter. Bizim veri setimizde 1.115.394 karakter var; %90 eğitim (1.003.854), %10 doğrulama (111.540).

> Builder Notu — s2i/i2s Buradan Geliyor
>
> **Geriye (Ders 2):** Karakter tokenizer, Ders 2'nin s2i/i2s arama tablolarının ta kendisi. "Metin → tamsayı → model" zinciri her dil modelinin ilk adımı.
> **İleriye:** Naif karakter tokenizer → Ders 9'da gerçek BPE (tiktoken/sentencepiece): daha büyük sözlük, daha kısa diziler, daha verimli. Tokenizer seçimi modelin verimliliğini doğrudan etkiler.

14.4 Veri Yükleyici: Bloklar, Zaman Boyutu, Batch'ler

Modeli sabit uzunlukta parçalarla (block) eğitiriz. `block_size` (bağlam uzunluğu) kadar karakter al; her konum, bir sonrakini tahmin eder. Yani tek bir blok bile `block_size` adet (bağlam → hedef) örneği taşır — bu, modele küçük ve büyük bağlamları aynı anda görmeyi öğretir.

```
block_size = 8
batch_size = 32

def get_batch(split):
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))   # rastgele baslangic
    x = torch.stack([data[i:i+block_size] for i in ix])         # (B, T) girdi
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])     # (B, T) hedef (1 kaydirilmis)
    return x, y
```

İki boyut: **B** (batch — paralel diziler) ve **T** (time/zaman — bloktaki konumlar). `x[b, t]`'nin hedefi `y[b, t] = x[b, t+1]`. Bu `(B, T)` yapısı, transformer'ın çalışacağı zemin.

Builder Notu — Zaman Boyutu (T) Dikkatin Kalbi

Geriye (Ders 3-6): Bağlam penceresi (`block_size`), Ders 3'ten beri tanıdık; `(B, T)` tensör yapısı, Ders 6'nın `(B, T, C)`'sinin embedding'siz hâli. `get_batch`, Ders 3'ün minibatch örneklemesinin (`torch.randint`) dizi versiyonu.
İleriye: Bu "zaman boyutu" (T), transformer'ın kalbi: dikkat, T boyutundaki token'lar arası iletişimdir. GPT-2'de T=1024, modern modellerde yüz binlerce (Ders 10).

14.5 Bigram Baseline

Transformer'a sıçramadan önce, Ders 2'nin **bigram** modelini PyTorch'ta yeniden kurarız — baseline (kıyas noktası). `nn.Embedding` (token → logits doğrudan), cross-entropy loss, ve metin üreten `generate()`.

```
class BigramLanguageModel(nn.Module):
    def __init__(self, vocab_size):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)  # her token -> logits
    def forward(self, idx, targets=None):
        logits = self.token_embedding_table(idx)   # (B, T, vocab_size)
        loss = None
        if targets is not None:
            B, T, C = logits.shape
            loss = F.cross_entropy(logits.view(B*T, C), targets.view(B*T))
        return logits, loss
    def generate(self, idx, max_new_tokens):
        for _ in range(max_new_tokens):
            logits, _ = self(idx)
            logits = logits[:, -1, :]                # son zaman adimi
            probs = F.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
            idx = torch.cat((idx, idx_next), dim=1)   # diziye ekle
        return idx
```

Bu, Ders 2'nin bigram'ı: yalnızca son token'a bakıp sonrakini tahmin eder (bağlamı kullanmaz). Üretilen metin Shakespeare'e benzemez — ama transformer'ın üzerine kurulacağı iskelet hazır. Bizim ölçümümüzde bigram'ın final doğrulama kaybı $2{,}4809$ (rastgele tahmin temeli $-\ln(1/65) = 4{,}1744$).

Builder Notu — Her Şey Ders 2'den Tanıdık

Geriye (Ders 2-3): `nn.Embedding` = Ders 2-3'ün lookup tablosu; cross-entropy = Ders 2 NLL; `generate` = Ders 2'nin örnekleme döngüsü (multinomial). Hepsi tanıdık, yalnızca dizi `(B, T)` boyutunda.
İleriye: `generate` döngüsü (forward → softmax → multinomial → ekle → tekrarla), her LLM'in **autoregressive** metin üretiminin birebir iskeleti — GPT-4 de böyle üretir, yalnızca model devasa.

14.6 Eğitim ve Script'e Taşıma

Bigram'ı eğitiriz: optimizer olarak **AdamW** (Ders 4'te bahsedilen adaptif optimizer), ve gürültüyü azaltmak için `estimate_loss` (birkaç batch'in ortalama train/val loss'u, `torch.no_grad` + eval modu).

```
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
for step in range(max_iters):
    xb, yb = get_batch('train')
    logits, loss = model(xb, yb)
    optimizer.zero_grad(set_to_none=True)   # Ders 1 zero_grad
    loss.backward()                          # Ders 5: arkasinda ne var biliyoruz
    optimizer.step()
```

Döngü Ders 1'den beri aynı (forward → zero_grad → backward → update); tek fark `optimizer` nesnesinin güncellemeyi yapması. Karpathy kodu bir Jupyter hücresinden bir Python **script**'ine taşır (gerçek geliştirme).

> Builder Notu — Döngü Ders 1'den Beri Aynı
>
> **Geriye (Ders 1-5):** Eğitim döngüsü Ders 1 micrograd'ın aynısı; `optimizer.zero_grad` = Ders 1 zero_grad, `loss.backward()` = Ders 5'te elle yazdığımız şey (artık arkasını biliyoruz), `AdamW` = Ders 4'te bahsedilen adaptif optimizer.
> **İleriye:** AdamW + estimate_loss + train/val izleme, production eğitiminin standart iskeleti; Ders 10'da buna learning rate schedule, gradient clipping, mixed precision eklenir.

14.7 Matematik Hilesi: Geçmişi Matris Çarpımıyla Ortalama

Self-attention'a girmeden önce, kalbindeki bir numarayı kurarız. Bir token, yalnızca **kendinden önceki** token'larla iletişebilmeli (geleceği göremez — buna **nedensel/causal** denir).

> *"A fifth token would like to communicate with [past] tokens… but not the 6th, 7th, 8th."* — Karpathy, 43:23

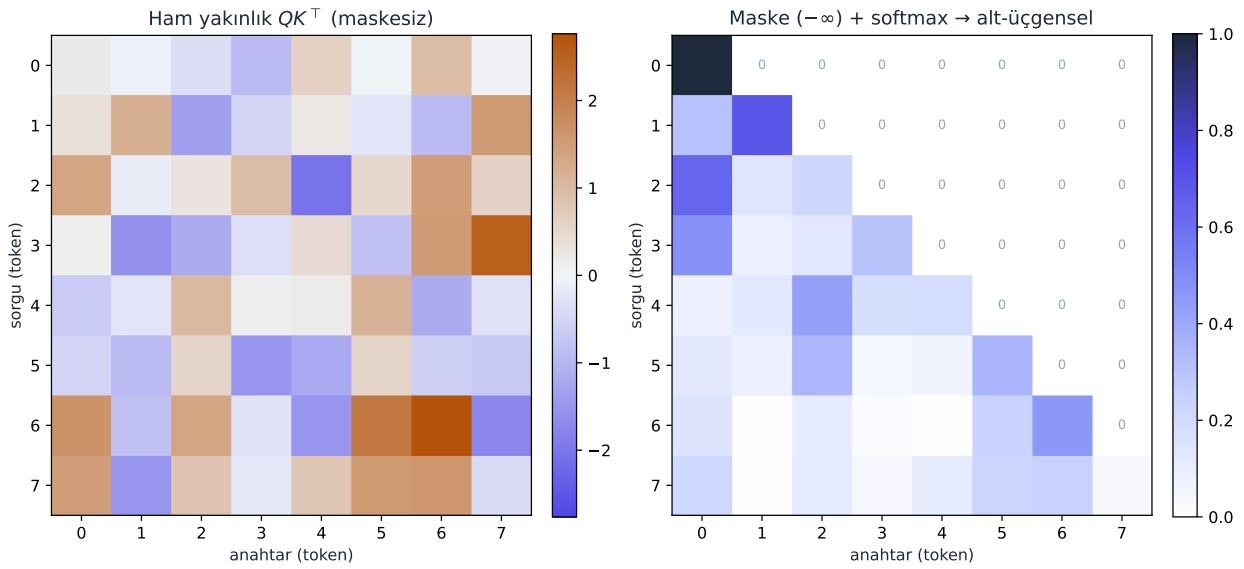
En basit iletişim: geçmiş token'ların **ortalamasını** al (bag-of-words). Bu ortalamayı bir `for` döngüsüyle yapmak yavaş; numara, onu bir **matris çarpımı** olarak yapmak:

> *"The trick in self-attention: matrix multiply as weighted aggregation."* — Karpathy, 47:07

Alt-üçgensel (lower-triangular) bir ağırlık matrisi `wei` kur (her satır, o konuma kadar olan token'lara eşit ağırlık, gerisi 0), normalize et; `xbow = wei @ x` her token için geçmişin ortalamasını verir. Daha esnek bir biçim: gelecek konumları $-\infty$ ile maskeleyip `softmax` uygula → otomatik olarak normalize alt-üçgensel ağırlıklar:

```
tril = torch.tril(torch.ones(T, T))
wei = torch.zeros((T, T))
wei = wei.masked_fill(tril == 0, float('-inf'))   # gelecek -> -sonsuz
wei = F.softmax(wei, dim=-1)                        # satir-normalize aglik
xbow = wei @ x                                       # gecmisin agirlikli toplami
```

`softmax(-∞) = 0` olduğundan, gelecek token'lar sıfır ağırlık alır (nedensellik). Şu an ağırlıklar sabit (eşit ortalama); birazdan onları **öğrenilen, veriye-bağlı** hâle getireceğiz — işte self-attention.



Şekil 14.2: Nedensel maske (causal mask), $T = 8$ sentetik örnekte. **Sol:** ham yakınlıklar $QK^\top$ — her token her tokenla (gelecek dahil) bir skor taşır. **Sağ:** üst-üçgeni $-\infty$ ile maskeleyip satır bazında softmax uygulayınca sonuç alt-üçgensel olur; $\text{softmax}(-\infty) = 0$ olduğundan gelecek token'lar sıfır ağırlık alır, her satır toplamı 1. Böylece token "gelecekten kopya çekemez" — GPT'nin metin üretebilmesinin şartı.

> **Builder Notu — Ortalama = Matris Çarpımı**
>
> **Geriye (18.06 + Ders 6):** "Ortalama = matris çarpımı" (normalize tril ile), Ders 6'nın batched matmul'unun bir uygulaması (18.06 matris çarpımı). softmax = Ders 2-3'ün üstel-normalize'ı. $-\infty$ maskeleme, nedenselliği zarif biçimde kodlar.
> **İleriye:** Bu causal mask (tril + $-\infty$ + softmax), her decoder transformer'ın (GPT) kalbinde vardır — model gelecekten "kopya çekemez". FlashAttention (Ders 10) bu maskelemeyi verimli yapar.

14.8 Çekirdek: Tek Self-Attention Başı (Q/K/V)

İşte videonun en önemli anı — sabit ortalamayı **öğrenilen, veriye-bağlı** iletişime çeviriyoruz. Her token iki vektör yayınlar: bir **query** (sorgu) ve bir **key** (anahtar).

> *"[Each token emits] a query and a key. The query roughly speaking is 'what am I looking for', and the key is 'what do I contain'."* — Karpathy, 1:04:06

İki token'ın **yakınlığı** (affinity), query'sinin diğerinin key'iyle nokta-çarpımıdır. Bu, sabit değil — token'ların içeriğine bağlı (veriye-bağlı).

> *"The affinities between tokens are not going to be constant zero, they're going to be data dependent."* — Karpathy, 57:25

Üç projeksiyon (Q, K, V), bir nedensel maske, bir softmax, ve değerlerin (V) ağırlıklı toplamı:

$$Q = xW_Q, \quad K = xW_K, \quad V = xW_V$$

$$\text{wei} = QK^\top \xrightarrow{\text{maskele+softmax}} \text{out} = \text{softmax}(\text{wei})\, V$$

```
head_size = 16
key   = nn.Linear(C, head_size, bias=False)
query = nn.Linear(C, head_size, bias=False)
value = nn.Linear(C, head_size, bias=False)

k = key(x)        # (B, T, head_size)  "ne iceriyorum"
q = query(x)      # (B, T, head_size)  "ne ariyorum"
wei = q @ k.transpose(-2, -1)                    # (B, T, T) yakinliklar
wei = wei.masked_fill(tril == 0, float('-inf'))   # nedensel maske
wei = F.softmax(wei, dim=-1)                       # agirlik dagilimi
v = value(x)      # (B, T, head_size)  "ne iletecegim"
out = wei @ v     # (B, T, head_size)  agirlikli toplam
```

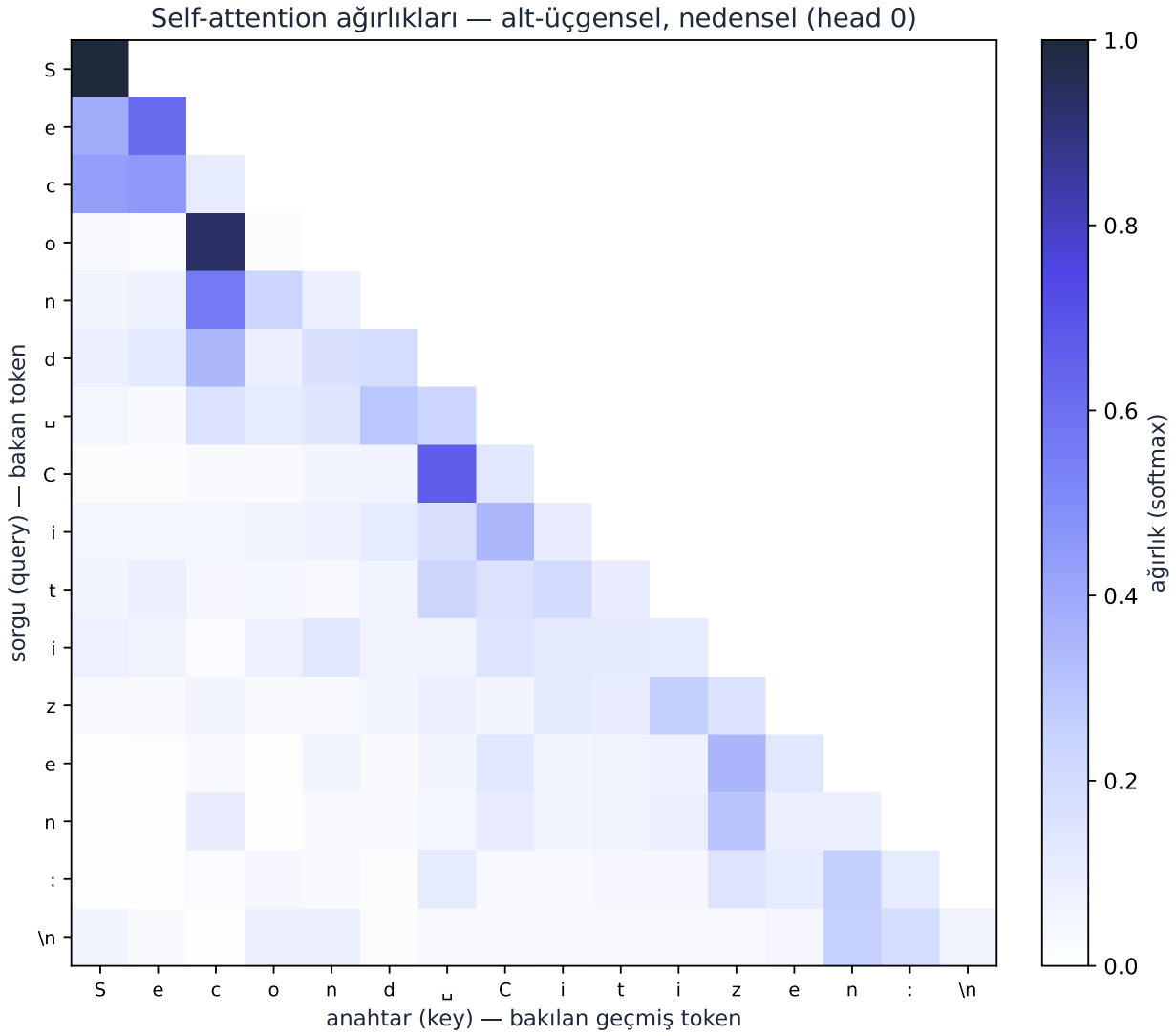
Sezgi: her token sorgusunu yayınlar; içeriği o sorguya uyan geçmiş token'lar yüksek ağırlık alır; o token'ların **değerleri** ağırlıklı toplanır. Artık iletişim sabit değil, içeriğe göre **öğreniliyor**. Aşağıda, eğitilmiş GPT'mizin ilk başının gerçek ağırlık matrisini görüyoruz — alt-üçgensel (nedensel) ve öğrenilmiş:

> Builder Notu — Q·K^T = Nokta-Çarpım Benzerliği
>
> **Geriye (18.06 + Stat 110 + Ders 6):** $QK^\top$ = nokta-çarpım benzerliği (18.06); softmax sonrası ağırlıklı toplam = **koşullu beklenti** sezgisi (Stat 110: ağırlıklar olasılık gibi). Üç projeksiyon ($W_Q/W_K/W_V$) = Ders 6'nın lineer katmanları.
> **İleriye:** Bu tek baş, GPT'nin temel taşı. Self-attention'ın $O(T^2)$ maliyeti (her token çifti) uzun bağlamda darboğaz; FlashAttention (Ders 10) bunu bellek-verimli yapar. Q/K/V çerçevesi, cross-attention (encoder-decoder) ve retrieval'a da genişler.

> Builder Notu — NYU H12 ile Köprü
>
> **Çapraz kurs (NYU Derin Öğrenme, H12 — Mike Lewis, NLP):** NYU dersi de self-attention'ı (self-attention matrisi, nedensel maske, konum kodlama) işler — ama **aynı kavramı** farklı bir çerçeveden: orada attention bir matris operasyonu olarak NLP bağlamında sunulur, burada Karpathy "her token bir sorgu yayınlar" sezgisiyle sıfırdan kurar. İkisi aynı softmax$(QK^\top/\sqrt{d_k})V$ formülüne iner — biri sezgi-önce, biri matris-önce.

14.9 Dikkat Üzerine Notlar ve Ölçekli Dikkat

Karpathy self-attention hakkında birkaç kritik not düşer:



Şekil 14.3: Eğitilmiş GPT'nin ilk bloğunun ilk dikkat başı: softmax$(QK^\top/\sqrt{d_k})$ ağırlık matrisi, 16 karakterlik gerçek bir Shakespeare bağlamında. **Alt-üçgensel** — her token (satır = sorgu) yalnızca kendinden önceki ve kendi konumundaki token'lara (sütun = anahtar) ağırlık verir; üst-üçgen tamamen 0 (nedensellik). Ağırlıklar sabit değil, **öğrenilen ve veriye-bağlı**: koyu hücreler, sorgunun o geçmiş token'a yoğunlaştığını gösterir.

- **Dikkatin konum kavramı yoktur.** Self-attention, token'ları bir küme gibi görür (permütasyona değişmez) — "5. token" ile "2. token" arasındaki sıra bilgisi yok. Bu yüzden token embedding'ine bir **konum embedding'i** (positional encoding) eklenir: `x = token_emb + pos_emb`.
- **Her örnek bağımsız.** Batch'teki diziler birbirine bakmaz (BatchNorm'un aksine — Ders 4).
- **Encoder vs decoder.** Nedensel maske (tril) varsa **decoder** (GPT — gelecek gizli); maske yoksa **encoder** (her token herkese bakar).
- **Self vs cross attention.** Q/K/V aynı kaynaktan gelirse self-attention; Q farklı bir kaynaktan gelirse cross-attention (çeviri gibi).

En önemli not: **ölçekleme**.

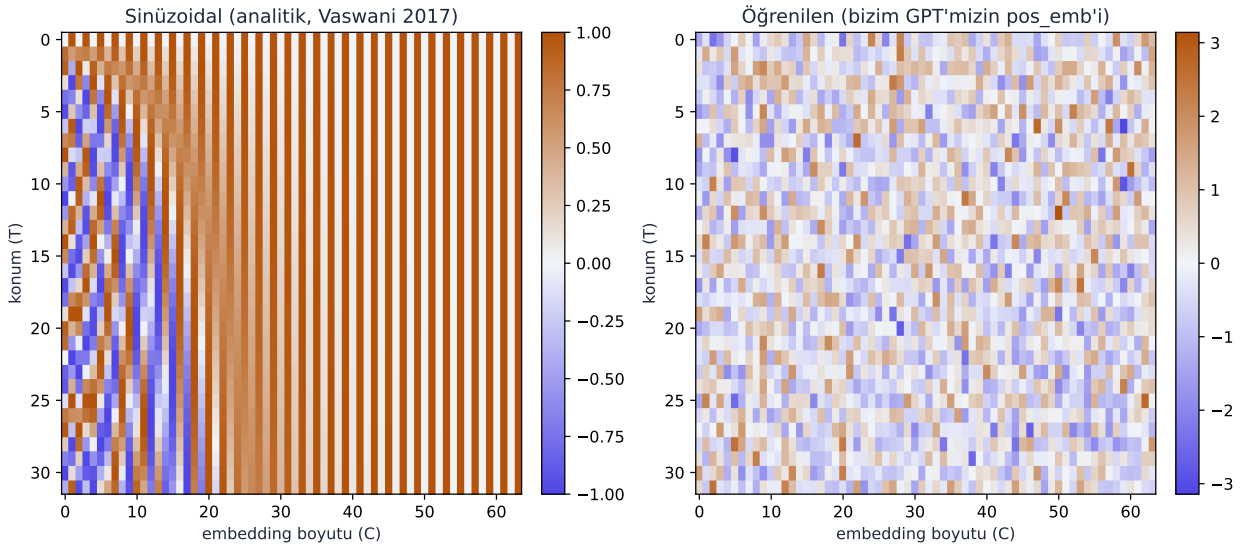
> *"Note 6: scaled self-attention — why divide by sqrt(head_size)?"* — Karpathy, 1:16:51

Yakınlıkları ($Q \cdot K^\top$) head boyutunun kareköküne böleriz:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

Neden? Head boyutu (d_k) büyüdükçe $Q \cdot K^\top$'nin varyansı büyür; softmax'a çok büyük değerler girerse softmax **doygunlaşır** (neredeyse one-hot, tek bir token'a tüm ağırlık) → gradyan akmaz (Ders 4 doygunluk!). $\sqrt{d_k}$'ye bölmek varyansı ≈ 1'de tutar, softmax yumuşak kalır. Ölçümümüzde: head_size $= 16$ için ham $Q \cdot K^\top$ std $\approx 4{,}295$ → ölçekli $\approx 1{,}074$; head_size $= 100$ için ham $\approx 10{,}093$ → ölçekli $\approx 1{,}009$.

Aşağıda konum embedding'ini — dikkatin "uzaysızlığını" telafi eden mekanizmayı — sinüzoidal (klasik) ve öğrenilen (bizim GPT) biçimleriyle görüyoruz:



Şekil 14.4: Konum (positional) embedding. Self-attention permütasyona değişmezdir — token'ları bir küme gibi görür, sıra bilgisi taşımaz. Bu yüzden token embedding'ine bir konum embedding'i eklenir ($x = \text{token\_emb} + \text{pos\_emb}$). **Sol:** klasik sinüzoidal kodlama (Vaswani 2017) — analitik sin/cos örüntüsü. **Sağ:** bizim GPT'mizin **öğrenilen** konum embedding tablosu (eğitilmiş ağırlıklar). İkisi de her konuma ayırt edici bir parmak izi verir; modern GPT'ler genelde öğrenilen biçimi kullanır.

Builder Notu — $\sqrt{d_k}$ = Varyans Kontrolü (Ders 4 Ruhu)

Geriye (Ders 4 + Stat 110): $\sqrt{d_k}$ ölçeklemesi, Stat 110 **varyans** muhasebesi (Ders 4 Kaiming'le aynı ruh: varyansı kontrol et); doygun softmax = Ders 4'ün doymuş aktivasyonu (gradyan ölür). Positional encoding, dikkatin "uzaysız" oluşunu telafi eder.
İleriye: Scaled dot-product attention, "Attention is All You Need"in tam formülü. Pozisyon kodlaması (öğrenilen, sinüzoidal, RoPE) modern modellerde aktif araştırma; decoder-only (GPT) tasarımı tüm üretken LLM'lerin standardı.

14.10 Multi-Head Attention ve Feed-Forward

Multi-head attention. Tek bir dikkat başı tek tür ilişki yakalar. Karpathy birden çok başı (head) **paralel** çalıştırır, çıktılarını birleştirir (concat). Her baş farklı bir ilişki türü öğrenir (biri sözdizimi, biri anlam, vb.) — tıpkı birden çok filtre gibi.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(num_heads * head_size, n_embd)   # residual'a geri projeksiyon
    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)    # baslari birlestir
        return self.proj(out)
```

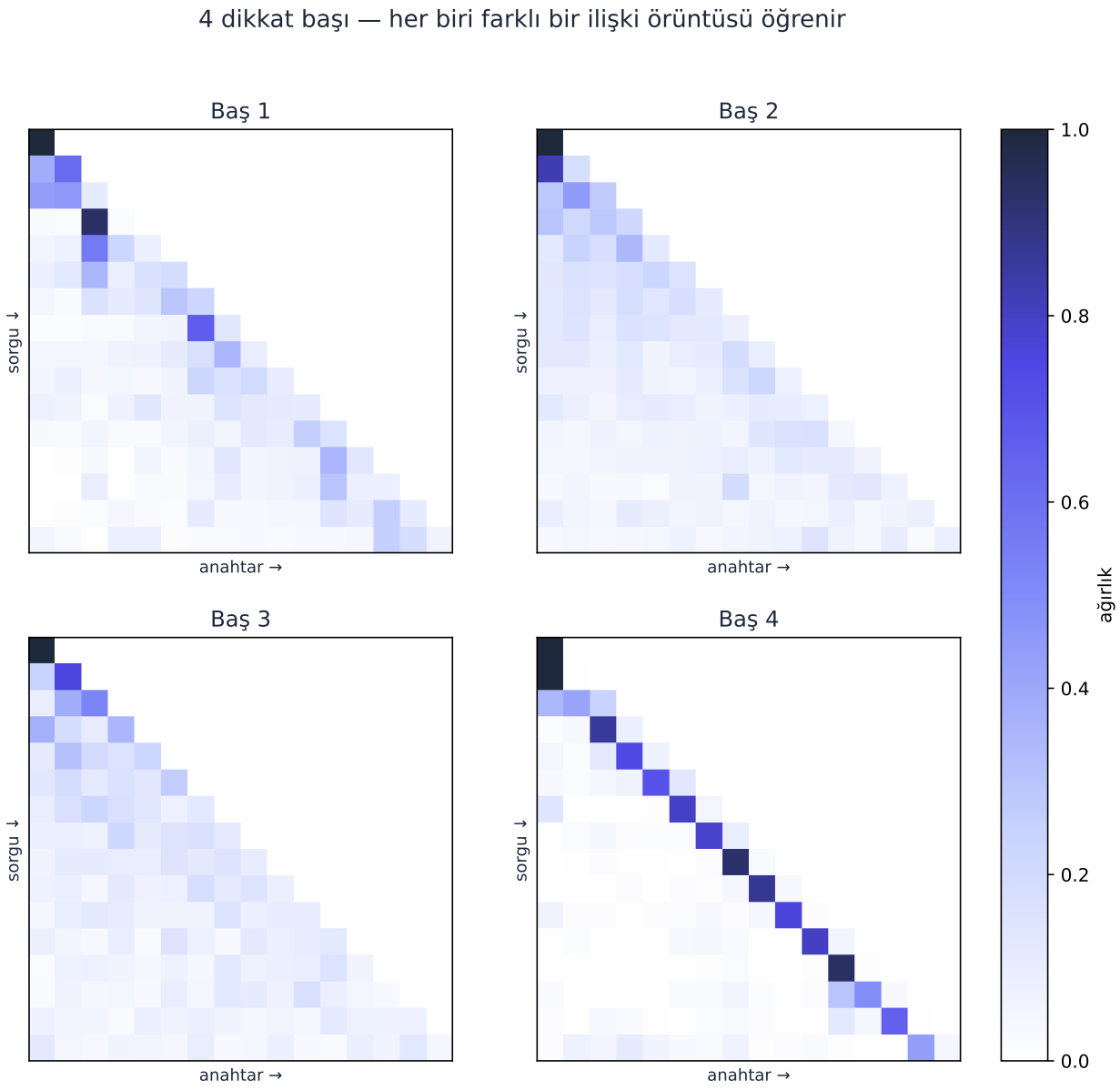
Aşağıda eğitilmiş GPT'mizin ilk bloğundaki dört başın gerçek ağırlık matrisleri — her biri farklı bir örüntü öğrenmiş:

Feed-forward. Dikkat **iletişimdir** (token'lar bilgi toplar); ama topladıkları bilgiyi tek tek **işlemeleri** de gerekir. Bu, her token'a ayrı ayrı uygulanan küçük bir MLP (genelde $4\times$ genişleme):

> *"Self-attention is the communication; then once they've gathered the data, they need to think on that data individually — and that's [the feed-forward]."* — Karpathy, 1:26:06

```
class FeedForward(nn.Module):
    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),   # 4x genisleme
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),   # residual'a geri
        )
    def forward(self, x):
        return self.net(x)
```

Yani transformer bloğu iki adımdır: **iletişim** (multi-head attention) + **düşünme** (feed-forward).



Şekil 14.5: Multi-head attention: eğitilmiş GPT'nin ilk bloğundaki **4 paralel dikkat başının** ağırlık matrisleri (aynı 16 karakterlik bağlam). Her baş, tamamen ayrı $W_Q/W_K/W_V$ projeksiyonlarıyla **farklı bir ilişki türü** öğrenir — bazıları yakın komşulara (köşegen yakını), bazıları daha uzak geçmişe yoğunlaşır. Hepsi alt-üçgensel (nedensel); çıktıları birleştirilip (concat) tek bir projeksiyondan geçirilir.

Builder Notu — FFN = Ders 1-6'nın MLP'si

Geriye (Ders 1-6): FeedForward = Ders 1-6'nın MLP'si (Linear → nonlin → Linear); multi-head = Ders 6'nın paralel-matmul fikri. "İletişim + düşünme" ayrımı, transformer bloğunun temel ritmidir.
İleriye: $4\times$ genişleme + GELU (Ders 10), her transformer FFN'inin standardı. Multi-head, GPT-2'de 12 baş, büyük modellerde onlarca; başların ne öğrendiği (interpretability) aktif araştırma. Modern modellerde FFN, parametrelerin çoğunu tutar (MoE bunu seyrekleştirir).

14.11 Bloklar ve Residual Bağlantılar

Bir transformer **bloğu** = multi-head attention + feed-forward. Gücü için bunları üst üste istifleriz (derinlik). Ama derin ağlarda gradyan akışı zorlaşır (Ders 4 vanishing gradient). Çözüm: **residual (artık) bağlantılar**. Her alt-katmanın çıktısını girdisine **ekleriz** (bypass): `x = x + attention(x)`, `x = x + ffn(x)`. Bu "artık yol" (residual pathway), gradyanın derin ağda doğrudan akmasını sağlar (toplama gradyanı aynen geçirir — Ders 1!).

$$x \leftarrow x + \text{MultiHead}(x), \qquad x \leftarrow x + \text{FeedForward}(x)$$

> *"Residual connections."* — Karpathy, 1:26:46

```python
class Block(nn.Module):
    def __init__(self, n_embd, n_head):
        super().__init__()
        self.sa = MultiHeadAttention(n_head, n_embd // n_head)
        self.ffwd = FeedForward(n_embd)
    def forward(self, x):
        x = x + self.sa(x)      # residual: iletisim
        x = x + self.ffwd(x)    # residual: dusunme
        return x
```

Başlangıçta bu katkılar küçük tutulur ki blok "kimlik fonksiyonuna yakın" başlasın — derin ağ kolay öğrensin.

Builder Notu — Residual = Ders 1 Toplama Gradyanı

Geriye (Ders 1 + Ders 4): Residual'ın gücü doğrudan Ders 1: **toplama gradyanı aynen geçirir** (yerel türev 1). Bu yüzden `x + f(x)`'in gradyanı, derin ağda bile sönmeden akar. Ders 4'te ResNet örneğinde görmüştük; transformer da aynı fikri kullanır. **Ders 5'te backward'ı elle yazdığımız için** bu akışın neden işlediğini sezgisel biliyoruz.
İleriye: Residual bağlantılar, 100+ katmanlı ağları (GPT-3: 96 katman) eğitilebilir kılan temel. "Residual stream" (Ders 10), modern interpretability'nin de merkezi kavramı.

14.12 LayerNorm (Pre-Norm) ve Dropout

Derin transformer'ı kararlı eğitmek için iki ek: **LayerNorm** ve **dropout**.

> *"…this Norm is referring to something called LayerNorm."* — Karpathy, 1:32:58

LayerNorm, Ders 4'ün BatchNorm'una benzer (normalize + scale/shift) ama **her örneği bağımsız** normalize eder — batch'teki diğer örneklere bakmaz (BatchNorm'un "örnekleri bağlama" sorunu yok, Ders 4). Özellik (feature) boyutunda normalize eder. Modern transformer'lar **pre-norm** kullanır: LayerNorm, alt-katmandan **önce** uygulanır.

```
def forward(self, x):
    x = x + self.sa(self.ln1(x))     # pre-norm: once normalize, sonra attention
    x = x + self.ffwd(self.ln2(x))   # pre-norm: once normalize, sonra ffn
    return x
```

Dropout (Ders 1), ölçeklerken (daha çok parametre) overfitting'i önler: eğitimde aktivasyonların bir kısmını rastgele kapatır. Aşağıda tam transformer bloğunun yapısı — iki pre-norm + iki residual yolu:

> Builder Notu — LayerNorm = BatchNorm'un Per-Örnek Hâli
>
> **Geriye (Ders 1 + Ders 4):** LayerNorm = Ders 4 BatchNorm'un per-örnek hâli (Ders 4'te "transformer LayerNorm tercih eder" demiştik — işte burada). Dropout = Ders 1'in regularization'ı. Pre-norm, residual yolunu temiz tutar (gradyan akışı için).
> **İleriye:** LayerNorm (ve türevi RMSNorm), her modern LLM'de var; pre-norm vs post-norm tasarım kararı eğitim kararlılığını etkiler (Ders 10). Dropout, büyük modellerde bazen kaldırılır (yeterli veri varsa).

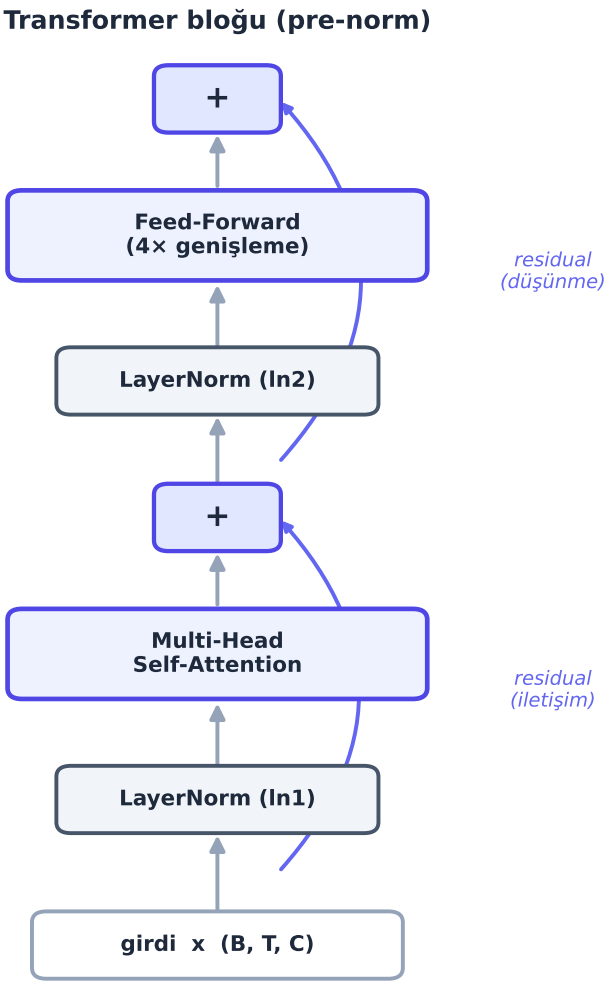
> Builder Notu — fast.ai L24 ile Köprü (gelecekte)
>
> **Çapraz kurs (fast.ai, L24 — Jeremy Howard):** [tahmin] Howard'ın transformer anlatımı self-attention'ı "matris çarpımı olarak dikkat" (matmul'lerin dizilişi) diliyle işler — Karpathy'nin "her token bir sorgu yayınlar" sezgisiyle aynı $\text{softmax}(QK^\top/\sqrt{d_k})V$'ye iner. fast.ai kursunun render'ı henüz L24'e ulaşmadı; bu köprü, fast.ai L24 hazır olduğunda detaylandırılacak.

14.13 Encoder, Decoder ve nanoGPT'ye Dönüş

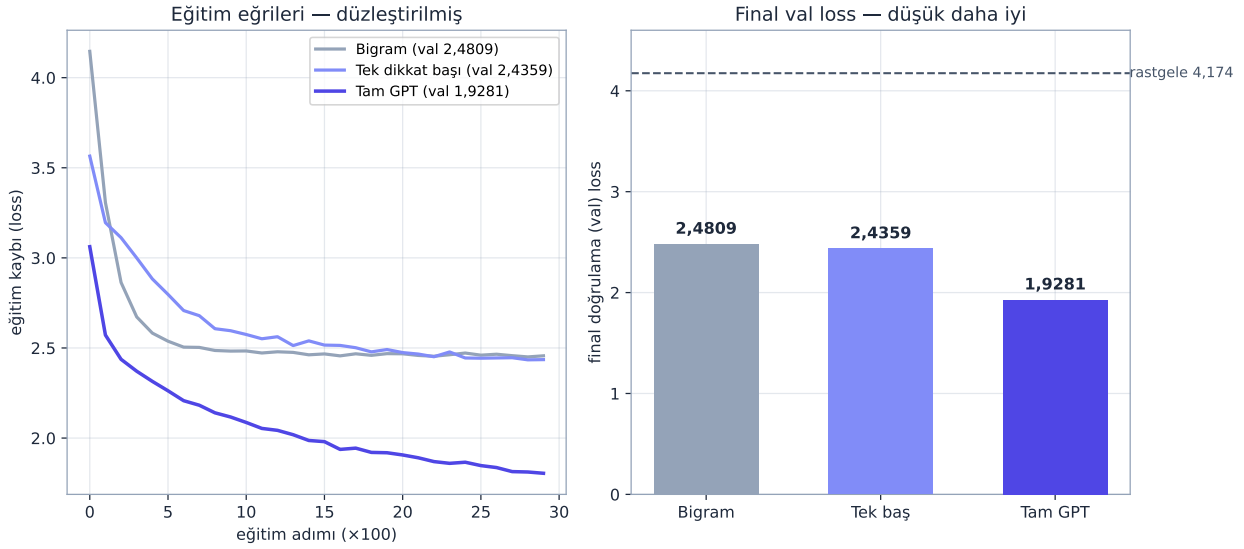
Kurduğumuz GPT, **decoder-only** bir transformer: nedensel maske (tril) sayesinde her token yalnızca geçmişe bakar — metin **üretmek** için (autoregressive). Orijinal "Attention is All You Need" hem encoder (maskesiz, çeviri girdisi için) hem decoder içeriyordu; GPT yalnızca decoder'ı kullanır.

Bizim tam GPT'miz (4 baş, 3 blok, $n_{embd} = 64$, 159.937 parametre) tiny Shakespeare'de eğitildiğinde, doğrulama kaybı bigram'ın 2,4809'undan 1,9281'e iner — tek dikkat başı ekleyince hafif (2,4359), tüm bileşenler (multi-head + feed-forward + residual + LayerNorm) istiflenince belirgin düşüş:

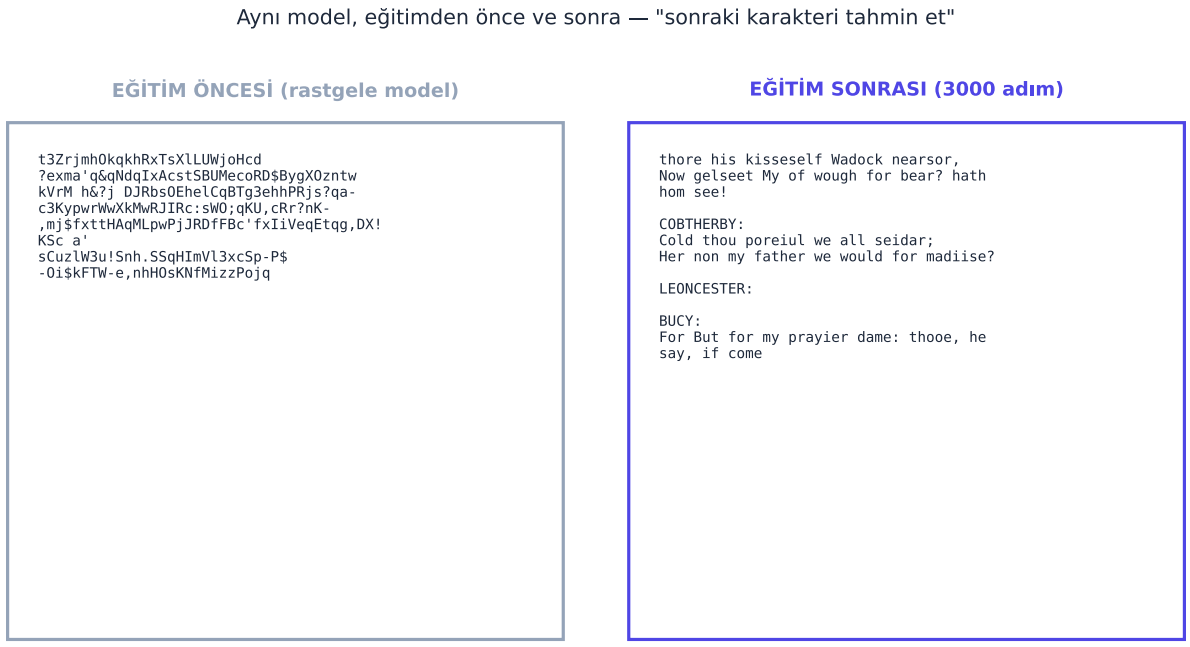
Asıl çarpıcı kanıt üretilen metinde: eğitimden önce saf gürültü, sonra Shakespeare-vari yapı (karakter adları, diyalog düzeni) — hiçbir kural verilmeden, yalnızca "bir sonraki karakteri tahmin et" hedefiyle:



Şekil 14.6: Tek bir transformer bloğu (pre-norm). İki alt-katman: **iletişim** (multi-head self-attention) ve **düşünme** (feed-forward, 4× genişleme). Her alt-katman önce LayerNorm'dan geçer (pre-norm), sonra çıktısı girdiye **eklenir** — residual (artık) bağlantı. Toplama, gradyanı aynen geçirdiği için (Ders 1) derin ağda bile akış sönmez: $x \leftarrow x + \text{MultiHead}(\text{LN}(x))$, ardından $x \leftarrow x + \text{FFN}(\text{LN}(x))$.



Şekil 14.7: Üç modelin GERÇEK eğitim sonucu (tiny Shakespeare, 3000 adım, AdamW, deterministik). **Sol:** düzleştirilmiş eğitim loss eğrileri. **Sağ:** final doğrulama (val) loss. Bigram (yalnız son token) 2,4809; tek dikkat başı eklenince hafif düşüş 2,4359; tam GPT (4 baş + 3 blok + feed-forward + residual + LayerNorm) belirgin sıçrama 1,9281. Rastgele tahmin temeli $-\ln(1/65) = 4{,}1744$. Büyük kazanç bileşenleri **istifleyince** gelir.



Şekil 14.8: Aynı GPT'nin ürettiği metin — eğitim **öncesi** (rastgele başlangıç ağırlıkları) ve **sonrası** (3000 adım). Öncesi: saf gürültü, rastgele karakter çorbası. Sonrası: Shakespeare-vari — büyük harfli karakter adları (`LEONCESTER:`, `BUCY:`), satır yapısı, kelime-benzeri parçalar. Model hiçbir kural verilmeden, yalnızca "bir sonraki karakteri tahmin et" hedefiyle dilin yapısını öğrendi. (Deterministik üretim.)

Bu char-GPT, **nanoGPT**'nin küçük kardeşi. Ölçeklendirip (daha çok blok/baş/embedding, daha uzun bağlam, gerçek tokenizer) GPT-2'ye ulaşırız — Ders 10'un konusu. Ve ChatGPT? O, böyle pretrain edilmiş bir GPT'nin üstüne **SFT + RLHF** eklenmiş hâli — Ders 8'in (State of GPT) konusu.

Builder Notu — WaveNet'in Sabit Ağacından Öğrenilen Dikkate

Geriye (Ders 6): WaveNet **sabit bir ağaç** kurar — hangi token'ın hangisiyle birleşeceği önceden bellidir. Transformer bunu **yeniden organize eder**: sabit ağaç yerine **her token her tokena** öğrenilen ağırlıklarla bakar (attention). Aynı "uzun bağlamı verimli işle" probleminin daha esnek (ama $O(T^2)$ ile daha pahalı) çözümü.
İleriye: Decoder-only GPT, tüm üretken LLM'lerin (GPT, Claude, Llama, Gemini) iskeleti. Encoder-only (BERT) sınıflandırma/anlama için; encoder-decoder (T5) çeviri için. Bu char-GPT'den GPT-2'ye (Ders 10) yol: aynı mimari, devasa ölçek + BPE tokenizer (Ders 9).

14.14 Bu Dersin Özeti

1. **GPT = decoder-only transformer**; tiny Shakespeare'de karakter-düzeyli kurarız (Ders 9'da gerçek BPE).
2. Veri **(B, T)** bloklar hâlinde; her konum sonrakini tahmin eder. **Bigram baseline** (Ders 2) başlangıç noktası (val $2{,}4809$).
3. **Matematik hilesi**: geçmişin ağırlıklı toplamı = normalize alt-üçgensel matrisle matmul; nedensellik $-\infty$ maske + softmax ile.
4. **Self-attention çekirdeği**: her token bir **query** (ne arıyorum) ve **key** (ne içeriyorum) yayınlar; yakınlık = $Q \cdot K^\top$ (veriye-bağlı); softmax sonrası **value**'ların ağırlıklı toplamı.
5. **Ölçekli dikkat**: $Q \cdot K^\top / \sqrt{d_k}$ — softmax doygunluğunu önler (varyans kontrolü, Ders 4).
6. **Multi-head**: paralel başlar (farklı ilişkiler), concat; **feed-forward**: iletişim sonrası bireysel düşünme ($4\times$ MLP).
7. **Residual bağlantılar**: $x = x +$ alt-katman(x); toplama gradyanı aynen geçirir (Ders 1) → derin ağda gradyan akar.
8. **LayerNorm (pre-norm)**: her örneği bağımsız normalize (Ders 4 BatchNorm'un per-örnek hâli); **dropout** ölçeklerken regularization.
9. Konum embedding'i (dikkatin uzaysızlığını telafi); GPT decoder-only; ölçekle → GPT-2 (Ders 10), + SFT/RLHF → ChatGPT (Ders 8). Bizim GPT'miz val $2{,}4809 \rightarrow 1{,}9281$.

Tek Bir Cümle

Transformer, token'ların birbirine **öğrenilen, veriye-bağlı ağırlıklarla baktığı** (query · key → softmax → value'ların ağırlıklı toplamı) bir iletişim mekanizmasıdır; bunun üzerine feed-forward (bireysel düşünme), residual bağlantılar (gradyan akışı) ve LayerNorm eklenince, ChatGPT'yi çalıştıran GPT mimarisi ortaya çıkar.

14.15 Kontrol Soruları

i Soru 1: Self-attention'da, softmax'tan önce yakınlık matrisinin (wei) üst-üçgen kısmı neden $-\infty$ ile maskelenir? softmax($-\infty$) ne verir ve bu neyi sağlar?

Cevap: softmax$(-\infty) = 0$ ($e^{-\infty} = 0$). Üst-üçgen, bir token'ın **gelecekteki** token'lara olan yakınlığıdır; bunları $-\infty$ yapıp softmax'tan geçirince o konumlar **sıfır ağırlık** alır. Böylece her token yalnızca kendinden önceki (ve kendisi) token'lardan bilgi toplar — **nedensellik** (causal). Bu, GPT'nin metin *üretebilmesi* için şart: model eğitimde "gelecekten kopya çekemez", yoksa bir sonraki karakteri tahmin etmek anlamsızlaşır (cevabı zaten görmüş olur). Maske olmasaydı (encoder), her token herkese bakardı — anlama görevleri için uygun ama üretim için değil.

i Soru 2: Matematik Hilesi bölümündeki bag-of-words ortalaması ile Self-Attention bölümündeki self-attention arasındaki temel fark nedir? Ne değişti?

Cevap: Bag-of-words'te ağırlıklar **sabit ve eşit**ti: her geçmiş token'a aynı ağırlık (basit ortalama, `wei` normalize tril). Self-attention'da ağırlıklar **öğrenilen ve veriye-bağlı**: her token bir query (ne arıyorum) ve key (ne içeriyorum) yayınlar; ağırlık = query·key benzerliği. Yani "şu an ararken hangi geçmiş token ilgimi çekiyor" sorusunun cevabı, token'ların **içeriğine** göre dinamik olarak hesaplanır. Ayrıca **value** projeksiyonu eklendi: token ne ilettiğini (value) de ayrı öğrenir. Sabit ortalama → öğrenilen, içeriğe-duyarlı ağırlıklı toplam.

i Soru 3: Yakınlıklar neden √(head_size)'a bölünür (ölçekli dikkat)? Bölünmezse ne olur?

Cevap: $Q \cdot K^\top$, head_size kadar terimin toplamıdır; head_size büyüdükçe bu çarpımın **varyansı** büyür (Stat 110: bağımsız terimlerin toplamının varyansı terim sayısıyla artar — ölçümümüzde head_size $16 \to 100$'de ham std $4{,}295 \to 10{,}093$). Çok büyük değerler softmax'a girerse, softmax **doygunlaşır** — neredeyse one-hot olur. Doygun softmax'ın gradyanı ≈ 0'dır (Ders 4 doymuş aktivasyon!), öğrenme durur. $\sqrt{\text{head\_size}}$'a bölmek, varyansı ≈ 1'de tutar → softmax yumuşak (ağırlıklar dağılmış) kalır, gradyanlar akar. Bu, Ders 4'teki Kaiming init'le aynı ruh: varyansı kontrol et.

i Soru 4: (Builder) Residual bağlantı (x = x + f(x)) derin transformer'ı neden eğitilebilir kılar? Ders 1 zincir kuralıyla bağla.

Cevap: `x = x + f(x)`'in `x`'e göre gradyanı, Ders 1'in **toplama kuralıdır**: toplama gradyanı her iki dala da **aynen** (yerel türev 1 ile) geçirir. Yani gradyan, `f` boyunca aktığı gibi, **doğrudan** (1 katsayısıyla) da geriye akar — bir "otoyol" (residual pathway). Derin ağda (100+ katman), normal yolda gradyan her katmanda yerel türevlerle çarpıla çarpıla sönebilir (vanishing, Ders 4); ama residual'ın $+1$ yolu gradyanı sönmeden en derine taşır. Bu yüzden GPT-3 gibi 96-katmanlı ağlar eğitilebilir. Başlangıçta f'nin katkısı küçük tutulur → blok "kimliğe yakın" başlar, ağ kademeli öğrenir.

14.16 Egzersizler

Egzersiz 1 (Nedensel maske). `torch.tril` ile alt-üçgensel matris kur. Bir yakınlık matrisini (rastgele) üst-üçgenden $-\infty$ ile maskele, `F.softmax(dim=-1)` uygula. Sonucun alt-üçgensel olduğunu ve her satırın toplamının 1 olduğunu (`wei.sum(-1)`) doğrula. `wei @ x`'in geçmişin ağırlıklı toplamını verdiğini gör.

Egzersiz 2 (Tek dikkat başı). Q/K/V projeksiyonlarını (`nn.Linear(C, head_size, bias=False)`) kur. `wei = q @ k.transpose(-2,-1)`, maskele, softmax, `out = wei @ v`. Çıktı şeklinin $(B, T, \text{head\_size})$ olduğunu doğrula. Maskeyi kaldırırsan (encoder) farkı gözlemle.

Egzersiz 3 (Ölçeklemenin etkisi). head_size'ı büyük seç (örn. 100). $Q \cdot K^\top$'nin (a) ölçeksiz ve (b) $/\sqrt{\text{head\_size}}$ ile std'sini ölç. Softmax çıktısının ölçeksizde doygunlaştığını (neredeyse one-hot), ölçeklide yumuşak kaldığını gözlemle.

Egzersiz 4 (Tam GPT). Block (multi-head + FFN + residual + pre-norm LayerNorm) kur, birkaçını istifle, token + positional embedding ekle. tiny Shakespeare'de eğit (AdamW), `generate` ile metin üret. Loss'un bigram baseline'dan ($\approx 2{,}5$) belirgin düştüğünü ve üretilen metnin Shakespeare-vari olduğunu gözlemle.

Egzersiz 5 (Sonraki dersin habercisi). Bu derste bir GPT'yi **pretrain** ettik: "bir sonraki token'ı tahmin et". Ama ChatGPT sadece metni *sürdürmez* — talimatları izler, yardımcı yanıtlar verir, "yapamam" der. (a) Ham bir pretrain edilmiş GPT (yalnızca internet metnini taklit eden) neden doğrudan iyi bir asistan değildir? (b) Onu yardımcı bir asistana çevirmek için hangi ek eğitim aşamaları gerekir (ipucu: insan örnekleri + insan tercihleri)? Bu sorular, Ders 8'de (**GPT'nin Hâli / State of GPT**) pretrain → SFT → ödül modeli → RLHF pipeline'ını motive eder.

14.17 Sonraki Ders İçin Hazırlık

Ders 8: GPT'nin Hâli (State of GPT) — Andrej Karpathy (Microsoft Build 2023)

Bu derste GPT mimarisini kurduk ve pretrain ettik. Ders 8 farklı bir ders: **canlı kodlama yok**, kavramsal bir konferans konuşması. Karpathy, ham bir pretrain edilmiş GPT'nin ChatGPT gibi bir **asistana** nasıl dönüştürüldüğünü (4-aşamalı pipeline) ve bu asistanların nasıl etkili kullanılacağını (prompt engineering) kuş bakışı anlatır.

Ana konular:

- 4-aşamalı eğitim hattı: pretraining → SFT (gözetimli ince ayar) → ödül modeli → RLHF.
- Base model vs assistant model; "token simülatörü" zihniyeti.
- Prompt mühendisliği: chain-of-thought, self-consistency, tool use, RAG.

> ⚠ Ders 8 Öncesi Yapılacak
>
> - Egzersizleri çöz — özellikle 4 (tam GPT'yi eğit) ve 5 (asistan sezgisi).
> - "GPT = pretrain edilmiş decoder transformer" cümlesini hatırla; Ders 8 bunun *üstüne* gelen aşamaları anlatır.
> - Not: Ders 8 seri akışının dışında bir "kuş bakışı" — kod yerine kavram.

14.18 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Karpathy'de
GPT / transformer	Decoder-only transformer; "bir sonraki token'ı tahmin et" ile pretrain	0m00
(B, T) bloklar	Batch × zaman; her konum sonrakini tahmin eder (`block_size` bağlam)	14m27
Nedensel maske (tril)	Üst-üçgen $-\infty$ + softmax $\rightarrow 0$; token yalnızca geçmişe bakar	42m13
query / key / value	Q: ne arıyorum, K: ne içeriyorum, V: ne iletiyorum	1h04m
Self-attention	softmax$(Q \cdot K^\top / \sqrt{d_k}) \cdot V$; öğrenilen, veriye-bağlı toplam	1h01m
Ölçekli dikkat ($\sqrt{d_k}$)	$Q \cdot K^\top$'yi $\sqrt{\text{head\_size}}$'a böl; softmax doygunluğunu önler	1h16m
Positional encoding	Dikkatin uzaysızlığını telafi; `token_emb + pos_emb`	1h11m
Multi-head attention	Paralel başlar (farklı ilişkiler), concat + projeksiyon	1h21m
Feed-forward ($4\times$)	İletişim sonrası bireysel düşünme; Linear→nonlin→Linear	1h24m
Residual bağlantı	$x = x +$ alt-katman(x); toplama gradyanı geçirir (Ders 1) → derin akış	1h26m
LayerNorm (pre-norm)	Her örneği bağımsız normalize (BatchNorm'un per-örnek hâli)	1h32m
Decoder-only	Nedensel maskeli GPT (üretim); encoder maskesiz (anlama)	1h42m

14.19 ML Builder Bağlantıları

9 köprü — Transformer

1. **Bigram baseline** → Ders 2 bigram (`nn.Embedding` + cross-entropy + generate). İleriye: her LLM'in autoregressive üretimi.
2. **Self-attention** ($Q \cdot K^\top$) → 18.06 nokta-çarpım + Stat 110 ağırlıklı/koşullu beklenti. İleriye: tüm LLM'lerin çekirdeği.
3. **Ölçekli dikkat** ($\sqrt{d_k}$) → Ders 4 varyans kontrolü/doygunluk. İleriye: FlashAttention (Ders 10).
4. **Nedensel maske** → Ders 6 tril + softmax. İleriye: KV cache, verimli decoding.
5. **Multi-head (paralel)** → Ders 6 batched matmul. İleriye: GPU paralelliği, throughput.
6. **Feed-forward** ($4\times$) → Ders 1-6 MLP. İleriye: parametrelerin çoğu burada; MoE seyrekleştirir.
7. **Residual bağlantı** → Ders 1 toplama gradyanı + Ders 4 ResNet. İleriye: residual stream (Ders 10), interpretability.
8. **LayerNorm** → Ders 4 BatchNorm'un per-örnek hâli. İleriye: RMSNorm, pre/post-norm tasarımı.
9. **Decoder-only GPT** → tüm üretken LLM iskeleti. İleriye: GPT-2 (Ders 10), + SFT/RLHF → ChatGPT (Ders 8). Çapraz: NYU H12 (Lewis) attention, fast.ai L24 (Howard) matris-dili.

14.20 Karpathy'nin Önerdiği Kaynaklar

Karpathy'nin bu ders için verdiği kaynaklar:

- **Ders deposu:** github.com/karpathy/ng-video-lecture — dersin tam kodu.
- **nanoGPT:** github.com/karpathy/nanoGPT — production-ölçek versiyon (Ders 10).
- **Attention is All You Need:** arxiv 1706.03762 — Vaswani ve ark. 2017, transformer makalesi.
- **GPT-3 makalesi:** arxiv 2005.14165 — Brown ve ark. 2020, "Language Models are Few-Shot Learners".

Bu dersten tek bir şey alıp gideceksen

Transformer, token'ların birbirine **öğrenilen, veriye-bağlı ağırlıklarla baktığı** bir iletişim mekanizmasıdır — her token bir query (ne arıyorum) ve key (ne içeriyorum) yayınlar, yakınlıklar softmax'lanır, value'lar ağırlıklı toplanır. Bunun üzerine feed-forward (bireysel düşünme), residual (gradyan akışı) ve LayerNorm eklenince ChatGPT'yi çalıştıran GPT mimarisi çıkar. makemore'un sabit ağacı yerine, artık iletişim öğreniliyor.

15 GPT'nin Hâli — Pretrain'den ChatGPT'ye (State of GPT)

Ham bir pretrain edilmiş GPT, internet metnini taklit eden bir token simülatörüdür; onu yardımcı bir asistana çevirmek insan örnekleri (SFT) ve insan tercihleri (ödül modeli + RLHF) ile ek eğitim, etkili kullanmak ise modele düşünmek için token verip istediğini açıkça istemek (prompt mühendisliği) gerektirir

> **i** Bölüm bilgisi
>
> - **Karpathy'nin videosu:** YouTube — State of GPT (≈41 dk)
> - **Seri:** Neural Networks: Zero to Hero — Ders 8
> - **Hoca:** Andrej Karpathy
> - **Kaynak:** Microsoft Build 2023 konferans konuşması
> - **Okuma süresi:** ≈24 dk

> **!** Bu ders farklı: kavramsal kuş bakışı, canlı kod yok
>
> Ders 8, serinin diğer derslerinden farklı olarak **canlı kodlama içermez**. Bir konferans konuşması: kavramsal, kuş bakışı. Bu yüzden "live coding adımları" yerine **kavramsal açıklama** bölümleri var; kod bloğu yok (yalnızca kavram, diyagram ve örnek prompt'lar). Ders 9 (tokenizer) ve Ders 10 (GPT-2) yine koda döner.

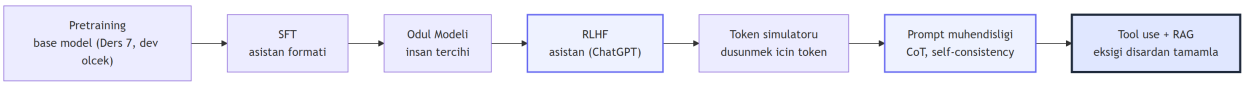
15.1 Bu Derste Ne Var?

Ders 7'de bir GPT'yi mimari olarak kurup **pretrain** ettik ("bir sonraki token'ı tahmin et"). Ama ChatGPT yalnızca metni sürdürmez — talimatları izler, yardımcı olur. Bu ders, ham bir pretrain edilmiş GPT'nin nasıl bir **asistana** dönüştürüldüğünü (4-aşamalı hat) ve bu asistanların nasıl **etkili kullanılacağını** (prompt mühendisliği) kuş bakışı anlatır.

Konuşma iki parça: **(1) GPT asistanları nasıl eğitilir** (pretraining → SFT → ödül modeli → RLHF) ve **(2) bu asistanlar nasıl kullanılır** (prompt mühendisliği, tool use, RAG).

Dersin üç büyük fikri:

1. **Dört aşamalı eğitim hattı** — her aşama bir öncekinin üstüne: pretraining (bilgi), SFT (asistan formatı), ödül modeli (insan tercihi), RLHF (tercihe göre optimize).
2. **"Token simülatörü" zihniyeti** — LLM başarmak değil **taklit** etmek ister; istediğin davranışı açıkça iste.
3. **Prompt mühendisliği** — "düşünmek için token" (chain-of-thought), self-consistency, tool use, RAG: modelin Sistem 2'sini dışarıdan kurmak.



Şekil 15.1: Ders 8'in kavram haritası: Ders 7'de kurduğumuz pretrain edilmiş GPT (base model) bir 'token simülatörü'dür; üç ek aşama (SFT → ödül modeli → RLHF) onu yardımcı bir asistana çevirir. İkinci yarı: bu asistanları etkili kullanmak — token simülatörü zihniyeti, prompt mühendisliği (chain-of-thought, self-consistency), tool use ve RAG. Slate akış + indigo dönüm noktaları (asistan modeli ve prompt mühendisliği).

Builder Notu — Geriye Ders 2-7, İleriye Ders 9-10

Geriye (Ders 2-7):

- **Pretraining = Ders 2-7.** Konuşmanın 1. aşaması (pretraining: "bir sonraki token'ı tahmin et"), tüm makemore + GPT serisinin yaptığı şeyin ta kendisi — yalnızca dev ölçekte (internet metni, milyarlarca parametre).
- **Base model = Ders 7 GPT.** Ders 7'de kurduğumuz pretrain edilmiş GPT, bu konuşmadaki "base model"; SFT/RLHF onun üstüne gelir.

Bağımsızlık notu: Bu ders, serinin kod-akışının **dışında** bir kuş bakışıdır (konferans konuşması). Ders 9 (tokenizer) ve Ders 10 (GPT-2) yine koda döner.
İleriye: Dört aşamalı hat, tüm modern LLM ürünlerinin (ChatGPT, Claude, Gemini) production pipeline'ı. Modern hizalama yöntemleri bunun üzerine kurulur: DPO (RLHF'nin basitleştirilmesi), Constitutional AI, RLAIF. Agent framework'leri (ReAct, tool use) bu dersin prompt fikirlerinin gelişmiş hâli.
Tek cümleyle: Ham bir pretrain edilmiş GPT, internet metnini taklit eden bir "token simülatörü"dür; onu yardımcı bir asistana çevirmek insan örnekleri (SFT) + insan tercihleri (ödül modeli + RLHF) ile ek eğitim ve doğru prompt mühendisliği gerektirir.

15.2 Bu Ne Tür Bir Ders? (Kuş Bakışı)

Karpathy iki parçalık bir harita çizer: önce GPT asistanlarının **nasıl eğitildiği**, sonra **nasıl kullanıldığı**. Bu, Ders 1-7'nin "sıfırdan kur" yaklaşımından farklı: artık mekanizmayı (transformer, Ders 7) biliyoruz; bu ders o mekanizmanın **production'da** nasıl ürüne dönüştüğünü anlatır.

Builder Notu — Mekanizma + Ürün

İleriye: "Mekanizmayı anla, sonra ürüne dönüştürmeyi anla" — bir builder'ın iki ayrı yetkinliği. Ders 7 birincisini, Ders 8 ikincisini verir. İkisi birlikte, LLM uygulaması geliştirmenin tam resmidir.

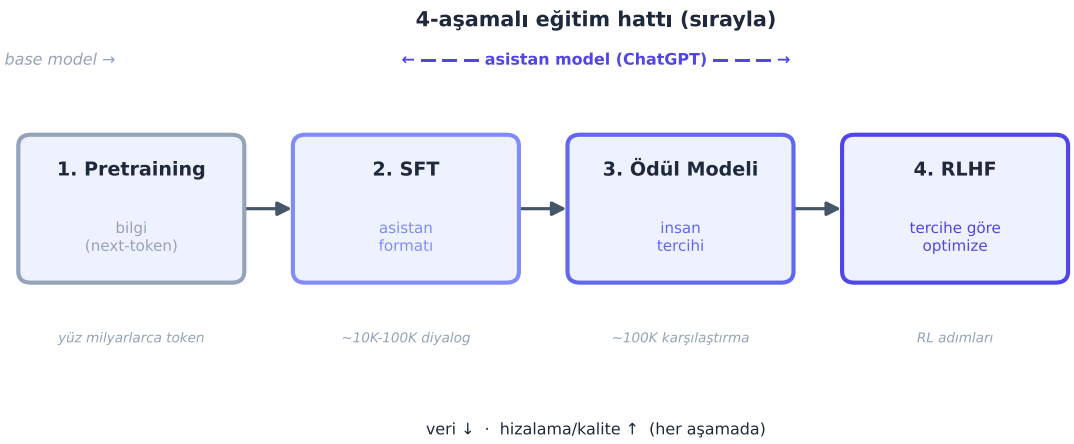
15.3 Dört Aşamalı Eğitim Hattı

GPT asistanı dört aşamada, **sırayla** eğitilir:

> *"Roughly speaking, we have four major stages: pretraining, supervised finetuning, reward modeling, reinforcement learning — and they follow each other serially."* — Karpathy, 0:59

1. **Pretraining** — devasa internet metninde "bir sonraki token'ı tahmin et" (Ders 2-7'nin yaptığı, dev ölçekte). Hesabın $\approx \%99$'u burada; aylar sürer. Çıktı: **base model** (genel bilgi ama asistan değil).
2. **SFT (gözetimli ince ayar)** — az sayıda, yüksek kaliteli (prompt, ideal yanıt) çiftiyle ince ayar. Çıktı: asistan formatını öğrenmiş model.
3. **Ödül modeli (reward modeling)** — insanların yanıtları sıralamasından, "iyi yanıt" skorlayan bir model öğren.
4. **RLHF** — ödül modeline göre, asistanı pekiştirmeli öğrenmeyle optimize et.

Her aşama daha az veri ama daha yüksek "kalite/hizalama" ekler.



Şekil 15.2: GPT asistanı eğitiminin dört aşaması, **sırayla**: Pretraining (devasa metinde "bir sonraki token'ı tahmin et" — bilgi) → SFT (az ama kaliteli prompt/yanıt çiftiyle asistan formatı) → Ödül Modeli (insan sıralamalarından "iyi yanıt" skoru) → RLHF (ödül modeline göre PPO ile optimize). Her aşama daha az veri ama daha yüksek hizalama ekler; ilki bilgiyi, sonraki üçü "yardımcı asistan" davranışını verir.

> Builder Notu — 1. Aşama = Ders 7
>
> **Geriye (Ders 7):** 1. aşama (pretraining), Ders 7'de kurduğumuz GPT'nin eğitimidir — yalnızca ölçek farklı. Geri kalan 3 aşama, o base model'i asistana çevirir.
> **İleriye:** Bu hat OpenAI'nin InstructGPT makalesinden gelir ve tüm asistan-LLM'lerin (Claude dahil) standardıdır; modern varyantlar (DPO, RLAIF, Constitutional AI) 3-4. aşamayı değiştirir.

15.4 Pretraining: Veri ve Tokenization

İlk ve en pahalı aşama. Devasa miktarda internet metni toplanır (terabaytlarca), tokenize edilir (Ders 9'un konusu — BPE), ve modele "bir sonraki token'ı tahmin et" hedefiyle verilir. Karpathy ölçekleri verir: yüz milyarlarca token, binlerce GPU, aylar süren eğitim, milyonlarca dolarlık maliyet.

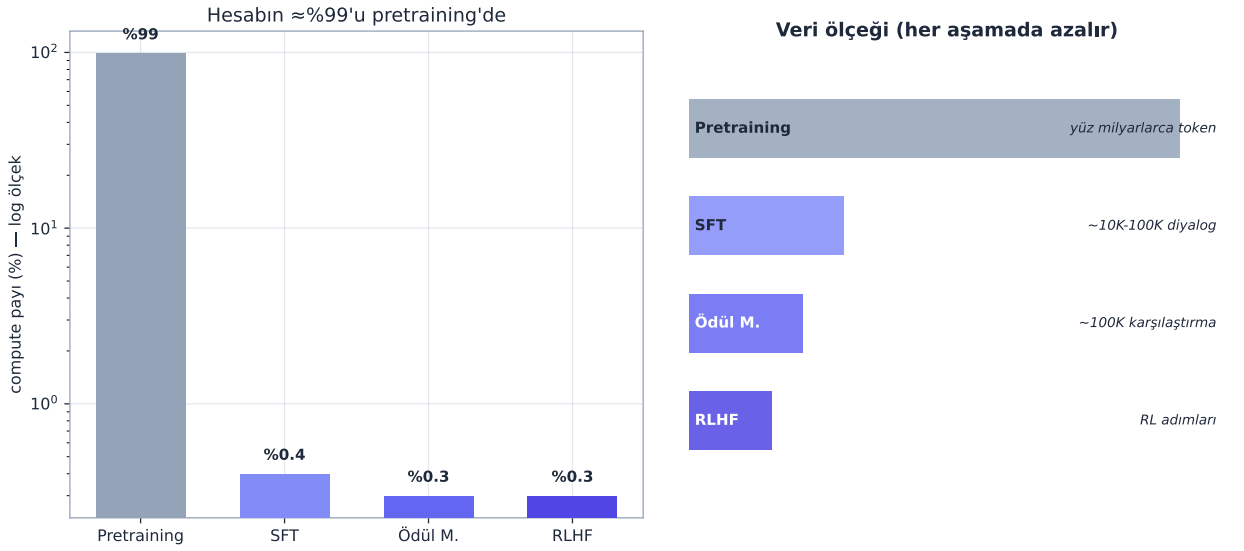
Veri farklı kaynaklardan (web, kitap, kod, …) belirli oranlarda karıştırılır. Token'lar, eğitim için $B \times T$'lik batch'lere dizilir (Ders 7'deki (B, T) yapısı) — metin parçaları arasına özel `<|endoftext|>` ayraç token'ı konur.

> **Builder Notu — Ders 7'nin Dev Ölçeği**
>
> **Geriye (Ders 7):** Bu, Ders 7'deki (B, T) batch yapısının + cross-entropy hedefinin dev ölçekli hâli. Tek fark: karakter yerine BPE token (Ders 9), tiny Shakespeare yerine internet.
> **İleriye:** Veri karışımı (data mixture), modern LLM'lerin gizli sosudur; veri kalitesi (FineWeb-EDU, Ders 10) model kalitesini doğrudan belirler.

15.5 Pretraining: Hiperparametreler ve Next-Token Hedefi

Pretraining'in çekirdeği, tüm seri boyunca gördüğümüz hedeftir: **bir sonraki token'ı tahmin et** (cross-entropy loss). Karpathy GPT'lerin kabaca büyüklük mertebelerini verir: parametre sayısı (milyarlar), token sayısı (yüz milyarlar), bağlam uzunluğu (binler).

Sonuç: **base model** — internet metnini taklit eden, devasa genel bilgi barındıran ama henüz "asistan" olmayan bir model. Base model bir soruyu yanıtlamaz; onu *sürdürür* (örneğin bir soru verirseniz, daha çok soru üretebilir — çünkü internette sorular soru listeleri içinde geçer).



Şekil 15.3: Aşamaların **compute payı** (Karpathy'nin verdiği büyüklük mertebesi). Pretraining hesabın $\approx$ %99'unu yutar (yüz milyarlarca token, binlerce GPU, aylar); diğer üç aşama (SFT + ödül modeli + RLHF) toplamda $\approx$ %1 — az veri, yüksek hizalama. Yani "asistan davranışı" görece ucuzdur; pahalı olan, base model'in genel bilgisini kazandıran pretraining'dir. (Sayılar konuşmadan, illüstratif.)

Builder Notu — Next-Token = Ders 2'den Ders 7'ye

Geriye (Ders 2-7): "Next-token prediction + cross-entropy" tam olarak Ders 2'den (bigram NLL) Ders 7'ye (GPT) kadar yaptığımız şey. Pretraining = bu hedefin dev ölçeği. **Ders 5'te bu cross-entropy'nin gradyanını elle yazmıştık** — RLHF (aşama 4) ise farklı bir hedef (ödül maksimizasyonu) kullanır. **Ders 4 köprüsü:** bu dev ölçekte (milyarlarca parametre, derin transformer) eğitimi **kararlı** kılan şey normalizasyondur — Ders 4'te küçük ölçekte aktivasyon/gradyan dağılımını ve BatchNorm'u incelemiştik; pretraining aynı stabilizasyonu transformer bloğunda **LayerNorm** (Ders 7) ile sağlar, yoksa derin ağ bu ölçekte eğitilemezdi.
İleriye: Bu büyüklük mertebeleri (parametre/token/compute), **scaling laws**'ın (Chinchilla) konusu — optimal eğitim için bu üçünün dengesi (Ders 10'da pratiği).

15.6 Base Model: Güçlü Genel Temsiller

Base model, "asistan" değil ama inanılmaz güçlü bir genel temsil taşır. 2020'de GPT-3 (base model) bir şey gösterdi: model, birkaç örnek (few-shot) verilince, ince ayar olmadan yeni görevleri **prompt** içinden öğrenebiliyor.

> *"This kicked off the era of prompting over finetuning, and seeing that this [works]."* — Karpathy, 9:45

Yani base model'i doğrudan kullanmanın iki yolu: **few-shot prompting** (prompt içine birkaç örnek koy) veya **finetuning** (ek eğitim). Karpathy örnek olarak GPT-3 base'i (API'de "Davinci") ve açık base modelleri (Meta'nın LLaMA serisi) anar. Base model güçlü ama "ham" — kullanışlı asistan için sonraki aşamalar gerek.

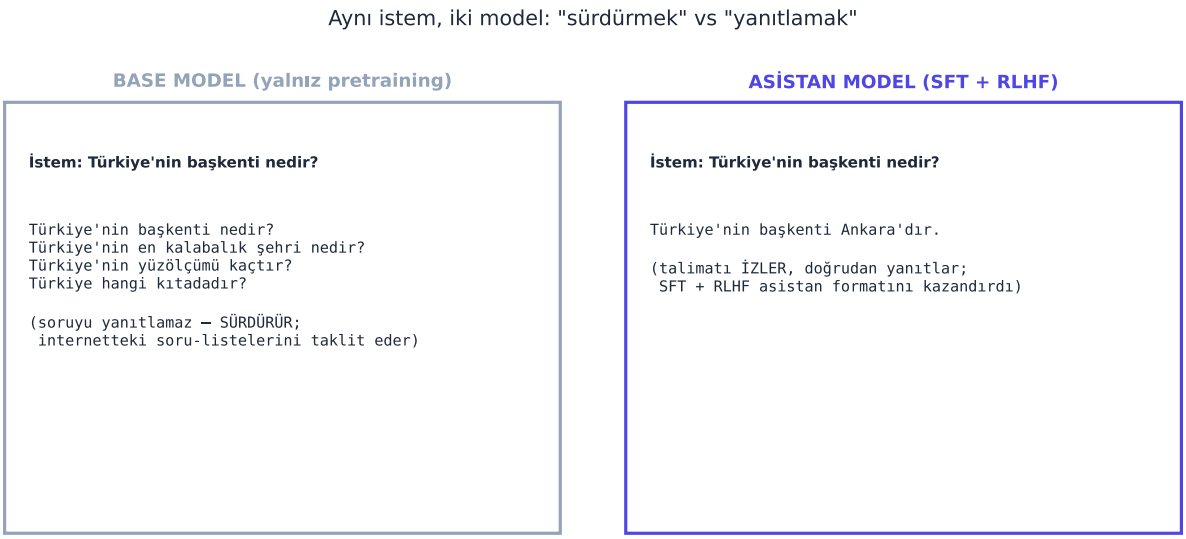
Builder Notu — Prompting over Finetuning

İleriye: "Prompting over finetuning" (few-shot, in-context learning), modern LLM kullanımının temelidir — modeli yeniden eğitmeden, prompt ile yönlendirmek. Base modeller (LLaMA, Mistral) açık kaynak ekosisteminin temeli; üstüne kendi SFT/LoRA'nı eklersin.

15.7 SFT (Gözetimli İnce Ayar)

İkinci aşama: base model'i **asistan formatına** sokmak. İnsan etiketleyiciler, az sayıda ama yüksek kaliteli **(prompt, ideal yanıt)** çifti yazar (binlerce, on binlerce). Base model bu veride ince ayarlanır (aynı next-token hedefi, ama artık asistan-vari yanıtlar üzerinde).

Sonuç: **SFT modeli** — soruları yanıtlayan, talimat izleyen bir model. Veri az ama kalite yüksek; model "böyle davran" formatını öğrenir. Pek çok pratik uygulama için SFT modeli yeterlidir.



Şekil 15.4: Base model vs asistan model. Aynı girdiye ("Türkiye'nin başkenti nedir?") **base model** (yalnız pretraining) yanıt vermez, metni *sürdürür* — internette sorular soru listelerinde geçtiği için daha çok soru üretebilir. **Asistan model** (SFT + RLHF sonrası) talimatı izler, doğrudan yanıtlar. Aradaki fark mimaride değil, base model'in üstüne eklenen 3 hizalama aşamasındadır (SFT + ödül modeli + RLHF).

> Builder Notu — Mekanizma Aynı, Veri Değişti
>
> **Geriye (Ders 7):** SFT, Ders 7'deki aynı eğitim (next-token + cross-entropy), yalnızca veri = elle yazılmış asistan diyalogları. Mekanizma değişmedi, veri değişti.
> **İleriye:** SFT, kendi asistanını kurmanın en erişilebilir adımı; LoRA/QLoRA (parametre-verimli ince ayar) ile küçük donanımda bile yapılabilir. Veri kalitesi (çeşitlilik + doğruluk) SFT'nin her şeyi.

15.8 Ödül Modeli (Reward Modeling)

Üçüncü aşama, RLHF'nin ilk yarısı. Fikir: bir prompt için modelin ürettiği **birden çok yanıtı** insanlara **sırala**t (hangisi daha iyi?). Bu sıralamalardan, bir yanıta "ne kadar iyi" skoru veren bir **ödül modeli** eğit.

Neden sıralama? Çünkü insanların "iyi yanıtı sıfırdan yazması" zor ama "iki yanıttan iyisini seçmesi" kolay — bu asimetri, ölçeklenebilir veri toplamayı sağlar. Ödül modeli, prompt + yanıt çiftlerini alıp tek bir skor (özel bir "ödül" token'ından okunan) üretir; insan tercih sıralamasıyla eğitilir.

> Builder Notu — Sıralamadan Skor (Stat 110)
>
> **Geriye (Stat 110):** Sıralamadan skor öğrenmek, Stat 110'daki Bradley-Terry tercih modeli (hangi öğenin tercih edileceğinin olasılığı). Ödül modeli = "insan tercihini taklit eden" bir sınıflandırıcı.
> **İleriye:** Ödül modeli, RLHF'nin kalbi; modern alternatifler (DPO) ödül modelini atlayıp tercihleri doğrudan optimize eder; RLAIF ise insan yerine AI tercihi kullanır (Constitutional AI).

15.9 RLHF (Pekiştirmeli Öğrenme) ve Mode Collapse

RLHF'nin ikinci yarısı: SFT modelini, **ödül modeline göre** pekiştirmeli öğrenmeyle (PPO) optimize et. Model yanıtlar üretir, ödül modeli skorlar, model yüksek-skorlu yanıtlar üretmeye yönlendirilir.

Neden işe yarar? Yine asimetri: **bir şeyi yargılamak, üretmekten kolaydır.** Ödül modeli (yargıç) iyi yanıtı tanıyabildiği için, üreticiyi o yöne itebilir. Sonuç: ChatGPT gibi PPO modelleri.

Bir yan etki var: **mode collapse**.

> *"Mode collapse."* — Karpathy, 18:46

RLHF modelleri, base modellere göre daha az **çeşitli** çıktı üretir — ödülü maksimize ederken belli kalıplara yapışırlar (entropi düşer). Bu yüzden çeşitlilik gereken görevlerde (örn. çok sayıda farklı fikir üretmek) base model bazen daha iyidir.

> Builder Notu — Hizalama-Yetenek Dengesi
>
> **İleriye:** RLHF (PPO), ChatGPT'yi mümkün kıldı ama karmaşık ve kararsız; bu yüzden **DPO** (Direct Preference Optimization) gibi basitleştirmeler popülerleşti. Mode collapse / çeşitlilik kaybı, hizalama-yetenek dengesinin (alignment tax) bir yüzü — aktif araştırma konusu.

Üç model tipinin davranış farkı:

Model	Nasıl eğitildi	Davranış	Ne zaman kullan
Base	yalnız pretraining	metni *sürdürür*, yanıtlamaz; en çeşitli	few-shot, ham tamamlama, kendi finetuning temelin
SFT	+ asistan diyalogları	talimat izler, yanıtlar	çoğu pratik asistan uygulaması
RLHF	+ ödül modeli + PPO	en hizalı/kibar; çeşitlilik düşük (mode collapse)	ChatGPT-vari ürün; çeşitlilik gerekmeyen işler

15.10 Token Simülatörü Zihniyeti

İkinci bölüm: bu asistanları nasıl etkili kullanırız? Karpathy kritik bir zihinsel model verir. İnsan bir metin yazarken zengin bir **iç monolog** çalıştırır (planlar, geri döner, düzeltir). LLM ise öyle değil:

> *"Basically these transformers are just like token simulators — they don't know what they don't know."* — Karpathy, 23:13

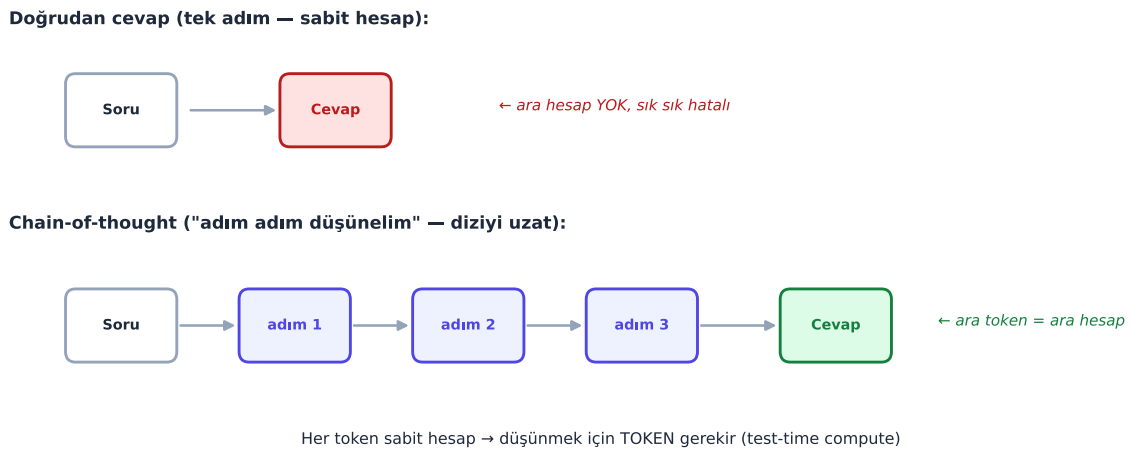
LLM her token için **sabit** miktarda hesap yapar; arada "durup düşünmez". O hâlde düşünmesini istiyorsak, ona **düşünmek için token vermeliyiz**:

> *"These transformers need tokens to think, I like to say sometimes."* — Karpathy, 24:57

Ve en pratik kural — LLM taklit eder, başarmayı hedeflemez:

> *"LLMs don't want to succeed, they want to imitate. You want to succeed, and you should ask for it."* — Karpathy, 30:16

Yani "uzman gibi, adım adım, dikkatli düşün" demek, modeli internetteki **iyi** örnekleri taklit etmeye yöneltir. İstediğin davranışı açıkça iste.



Şekil 15.5: "Düşünmek için token" (Ders 7 köprüsü). Transformer her token için **sabit** miktarda hesap yapar (sabit derinlik) — tek bir token'da karmaşık akıl yürütmeyi sıkıştıramaz. **Üst:** doğrudan cevap istenince model tek adımda yanıtlar, ara hesap için yer yok (sık sık hatalı). **Alt:** "adım adım düşünelim" (chain-of-thought) deyince model ara adımları token olarak üretir; her ara token sonraki adım için ek hesap + bağlam sağlar (kendi ürettiğine self-attention ile bakar). Sabit-derinlik kısıtı, **diziyi uzatarak** (test-time compute) aşılır.

> **Builder Notu — Sabit Hesap, Ders 7 Köprüsü**
>
> **Geriye (Ders 7):** Ders 7'de gördük: transformer her token için **sabit** miktarda hesap yapar (sabit derinlik). "Düşünmek için token" = bu sabit-derinlik kısıtını, diziyi uzatarak (kendi ürettiğine self-attention) aşmak. **İleriye:** "Token simülatörü" + "düşünmek için token" zihniyeti, tüm prompt mühendisliğinin temeli. Modern reasoning modelleri (o1, vb.) bu fikri içselleştirir: cevaptan önce uzun bir "düşünme" zinciri üretirler (test-time compute).

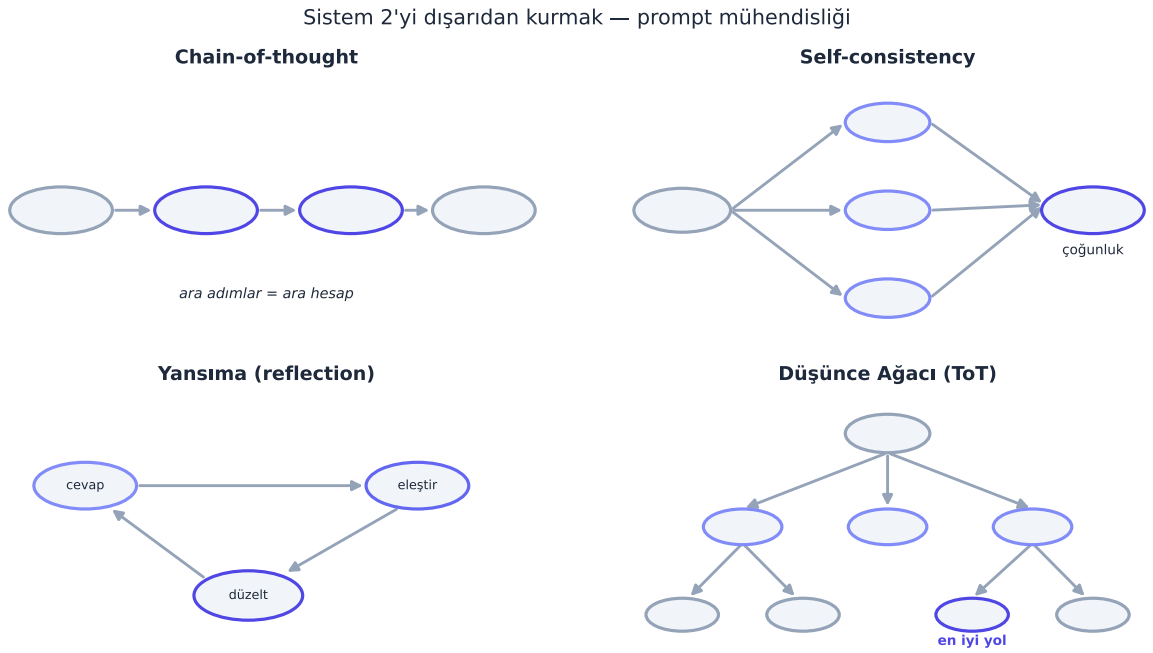
15.11 Prompt Mühendisliği: Sistem 2'yi Dışarıdan Kurmak

Karpathy bu tekniklerin ortak temasını verir: modelin eksik olan **Sistem 2'sini** (yavaş, kasıtlı düşünme) dışarıdan kurmak.

> *"A lot of these techniques fall into the bucket of recreating our System 2."* — Karpathy, 27:34

- **Chain-of-thought (CoT):** "Adım adım düşünelim" de — model ara adımları token olarak üretir, böylece "düşünme" için hesap kazanır.

- **Self-consistency:** Aynı soruyu birkaç kez çözdür, en sık çıkan cevabı al (çoğunluk oyu).
- **Yansıma (reflection):** Modele kendi cevabını eleştirmesini/düzeltmesini iste.
- **Düşünce Ağacı (Tree of Thought):** Birden çok düşünme yolunu dallandırıp en iyisini ara (AlphaGo'nun Monte Carlo ağaç aramasına koşut).



Şekil 15.6: Prompt mühendisliği teknikleri — ortak tema: modelin eksik **Sistem 2'sini** (yavaş, kasıtlı düşünme) dışarıdan kurmak. **Chain-of-thought:** ara adımları token olarak üret. **Self-consistency:** aynı soruyu birkaç kez çöz, çoğunluk oyunu al. **Yansıma:** modele kendi cevabını eleştirt/düzelt. **Düşünce Ağacı:** birden çok düşünme yolunu dallandır, en iyisini ara (AlphaGo'nun ağaç aramasına koşut).

Builder Notu — CoT = Ara Token = Ara Hesap

Geriye (Ders 5/7): CoT, "ara token'lar = ara hesap" — Ders 7'deki autoregressive üretimin, modele alan açacak biçimde kullanılması. Düşünce Ağacı, arama (search) + LLM birleşimi.
İleriye: CoT/self-consistency/ToT, modern reasoning'in (test-time compute, o1/o3) temeli; ReAct ve agent framework'leri bunları tool use ile birleştirir.

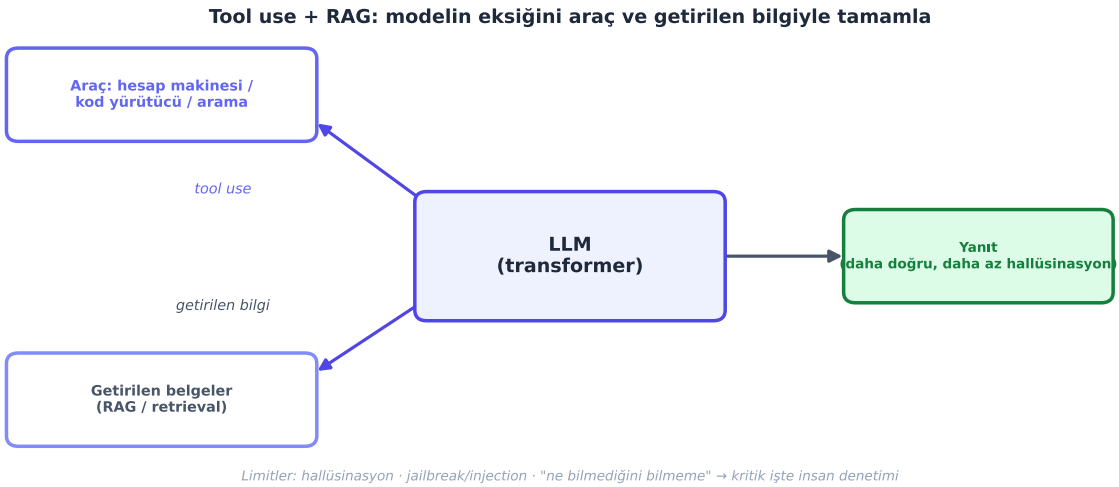
15.12 Tool Use, RAG ve Limitler

Karpathy birkaç pratik araç daha sıralar:

- **Tool use:** Modele hesap makinesi, kod yürütücü, arama gibi araçları kullanmayı öğret (Python "glue code" ile) — kendi yapamadığını araca yaptırır.
- **Retrieval-augmented generation (RAG):** İlgili belgeleri getirip prompt'a koy — model "ezberden" değil, getirilen bilgiden yanıtlar (LlamaIndex gibi araçlar). "Yalnızca-ezber"den "yalnızca-getirme"ye bir yelpaze; ortası en güçlü.

- **Kısıtlı istem (constraint prompting):** Çıktıyı belli bir formata (örn. JSON) zorla (Microsoft Guidance gibi).
- **Finetuning:** Gerekirse modeli kendi verinle ince ayarla (LoRA/QLoRA — parametre-verimli).

Limitler: Karpathy modellerin sınırlarını da hatırlatır — hallüsinasyon, prompt injection / jailbreak / veri zehirleme gibi güvenlik açıkları, "ne bilmediğini bilmeme". Bu yüzden kritik uygulamalarda insan denetimi şart.



Şekil 15.7: Tool use ve RAG — modelin tek başına yapamadığını dışarıdan tamamlama. **Tool use:** model bir hesap makinesi / kod yürütücü / arama çağırır (Python "glue code" ile), kendi yapamadığını araca yaptırır. **RAG (retrieval-augmented generation):** ilgili belgeler getirilip prompt'a konur; model "ezberden" değil, getirilen bilgiden yanıtlar (hallüsinasyon azalır). "Yalnızca-ezber" ile "yalnızca-getirme" arası bir yelpaze; ortası en güçlü.

Builder Notu — Modern LLM Uygulama Mimarisi

İleriye: Tool use + RAG + constraint prompting, modern LLM uygulama geliştirmenin (Claude/GPT uygulamaları) çekirdeği; MCP, LangChain, agent framework'leri bunları standartlaştırır. Güvenlik açıkları (jailbreak, injection) production'da ciddi bir konu — OWASP Top 10 for LLMs.

15.13 Bu Dersin Özeti

1. Ders 8, serinin **kavramsal kuş bakışı**: kod yok, GPT asistanlarının nasıl eğitildiği + nasıl kullanıldığı.
2. **Dört aşamalı hat** (sırayla): pretraining (bilgi, $\approx$ %99 compute) $\rightarrow$ SFT (asistan formatı) $\rightarrow$ ödül modeli (insan tercihi) $\rightarrow$ RLHF (tercihe göre optimize).
3. **Base model** (Ders 7 GPT'nin dev hâli) güçlü ama ham; soruyu yanıtlamaz, sürdürür. few-shot prompting veya finetuning ile kullanılır.
4. **SFT**: az ama kaliteli (prompt, yanıt) çiftiyle ince ayar $\rightarrow$ asistan formatı.
5. **Ödül modeli + RLHF**: insan sıralamasından skor öğren, PPO ile optimize; "yargılamak üretmekten kolay" asimetrisi. Yan etki: **mode collapse** (çeşitlilik kaybı).

6. **Token simülatörü** zihniyeti: LLM iç monolog çalıştırmaz, "düşünmek için token" gerekir; "başarmayı değil taklit etmeyi ister — istediğini açıkça iste".
7. **Prompt mühendisliği** (Sistem 2'yi dışarıdan kur): chain-of-thought, self-consistency, yansıma, Düşünce Ağacı.
8. **Tool use, RAG, kısıtlı istem**: modelin yapamadığını araçla/getirilen bilgiyle/formatla tamamla.
9. **Limitler**: hallüsinasyon, jailbreak/injection, "ne bilmediğini bilmeme" → kritik işte insan denetimi.

! Tek Bir Cümle

Ham bir pretrain edilmiş GPT (Ders 7), internet metnini taklit eden bir "token simülatörü"dür; onu yardımcı bir asistana çevirmek insan örnekleri (SFT) + insan tercihleri (ödül modeli + RLHF) ile ek eğitim gerektirir — ve onu etkili kullanmak, modele "düşünmek için token" verip istediğin davranışı açıkça istemekten (prompt mühendisliği) geçer.

15.14 Kontrol Soruları

i Soru 1: GPT asistanı eğitiminin 4 aşamasını sırayla say. Her aşama ne ekler ve hangisi en çok compute harcar?

Cevap: (1) Pretraining — devasa internet metninde "bir sonraki token'ı tahmin et"; genel bilgi ekler; **compute'un** $\approx \%99$**'u burada** (aylar, milyonlarca dolar). Çıktı: base model. **(2) SFT (gözetimli ince ayar)** — az ama kaliteli (prompt, ideal yanıt) çiftiyle ince ayar; asistan formatını ekler. **(3) Ödül modeli** — insan sıralamalarından "iyi yanıt" skoru öğrenir; insan tercihini ekler. **(4) RLHF** — ödül modeline göre PPO ile optimize; tercihe-hizalanmış davranışı ekler. Her aşama daha az veri ama daha yüksek hizalama katar; ilki bilgiyi, sonraki üçü "yardımcı asistan" davranışını verir.

i Soru 2: Base model ile SFT/RLHF modeli arasındaki fark nedir? Hangi durumda hangisini kullanırsın?

Cevap: Base model internet metnini taklit eder — soruyu yanıtlamaz, *sürdürür* (bir soru verirsen daha çok soru üretebilir). Devasa genel bilgi taşır ama "asistan" değil. **SFT/RLHF modeli** (ChatGPT gibi) talimat izler, yardımcı yanıt verir. Kullanım: asistan davranışı, talimat-izleme, sohbet için **SFT/RLHF** modeli; **çeşitlilik** gereken (çok sayıda farklı/yaratıcı çıktı), ham tamamlama, veya kendi finetuning'inin temeli için **base model** (RLHF mode collapse nedeniyle çeşitliliği düşürür). Çoğu uygulama asistan modeli ister; araştırma/özel-eğitim base model ister.

i Soru 3: RLHF modelleri neden base modellerden daha az çeşitli çıktı üretir (mode collapse)?

Cevap: RLHF, modeli **ödülü maksimize etmeye** iter. Ödül modeli belli yanıt türlerini (kibar, yapılandırılmış, "güvenli") yüksek skorlar; model bu kalıplara yapışır ve çıktı dağılımının **entropisi düşer** (mode collapse) — aynı soruya hep benzer yanıtlar. Base model ise yalnızca internet metnini taklit ettiği için çok daha çeşitli (internetteki tüm üslupları kapsar). Sonuç bir denge (alignment tax): RLHF hizalama/yardımcılık kazandırır ama çeşitlilik kaybettirir. Bu yüzden "bana 30 farklı isim öner" gibi çeşitlilik görevlerinde base model bazen daha iyidir.

ℹ Soru 4: (Builder) 'Adım adım düşünelim' (chain-of-thought) demek neden modelin performansını artırır? Ders 7 (token başına sabit hesap) ile bağla.

Cevap: Ders 7'de gördük: transformer her token için **sabit** miktarda hesap yapar (sabit derinlik). Model arada "durup uzun uzun düşünemez" — tek bir token'da karmaşık bir akıl yürütmeyi sıkıştıramaz. Çözüm: modele **düşünmek için token** ver. "Adım adım düşünelim" deyince, model ara adımları (token olarak) üretir; her ara token, sonraki adım için ek hesap + bağlam sağlar (kendi ürettiğine geri bakar, Ders 7 self-attention). Yani CoT, sabit-derinlik kısıtını, **diziyi uzatarak** (test-time compute) aşar. Ayrıca "iyi düşünen" örnekleri taklit etmeye yöneltir ("imitate, ask for it"). Karpathy'nin deyişiyle: "transformers need tokens to think".

15.15 Egzersizler

Bu ders kavramsal olduğundan, egzersizler de **düşünme/deney** egzersizleridir (canlı kodlama yerine). Bir API (veya açık model) varsa pratikte deneyebilirsin.

Egzersiz 1 (Base vs asistan). Bir base model (tamamlama) ile bir asistan modele (ChatGPT) aynı soruyu sor: "Türkiye'nin başkenti nedir?". Base model'in soruyu *sürdürdüğünü* (belki başka sorular ürettiğini), asistanın *yanıtladığını* gözlemle. Aradaki farkı 4-aşamalı hatla açıkla.

Egzersiz 2 (Chain-of-thought). Zor bir aritmetik/mantık problemi seç. (a) Doğrudan sor. (b) "Adım adım düşünelim" ekleyerek sor. Ara adımların (token'ların) doğruluğu nasıl etkilediğini gözlemle — "düşünmek için token" sezgisini doğrula.

Egzersiz 3 (Self-consistency). Aynı zor soruyu 5 kez çözdür (sıcaklık > 0), cevapları topla, çoğunluk oyunu al. Tek seferlik cevaba göre güvenilirliğin arttığı durumları gözlemle. Ne zaman yardımcı olur, ne zaman olmaz?

Egzersiz 4 (Tool use / RAG). Modelin tek başına yapamayacağı bir soru kur (örn. büyük bir çarpım, veya güncel bir olay). (a) Bir araç (hesap makinesi / kod yürütücü) kullanan bir prompt tasarla. (b) İlgili bir belgeyi prompt'a koyup (RAG) modelin "getirilen bilgiden" yanıtlamasını sağla. Hallüsinasyonun azaldığını gözlemle.

Egzersiz 5 (Sonraki dersin habercisi). Tüm seri boyunca "token" kelimesini kullandık ama Ders 7'de naif bir **karakter** tokenizer kurmuştuk. Gerçek GPT'ler farklı tokenize eder. (a) "strawberry" kelimesinde kaç tane 'r' var? Bir LLM'e sor — neden bazen yanlış sayar? (İpucu: model karakterleri değil, token'ları görür; "strawberry" birkaç token'a bölünür, tek tek harfleri göremez.) (b) Metni token'lara bölmenin daha iyi bir yolu ne olabilir (karakter de değil, kelime de değil)? Bu sorular, Ders 9'da (**GPT Tokenizer / BPE**) gerçek tokenizer'ı sıfırdan kurmayı motive eder.

15.16 Sonraki Ders İçin Hazırlık

Ders 9: GPT Tokenizer'ı İnşa Etmek (BPE) — Andrej Karpathy

Ders 8 kavramsaldı; Ders 9 koda döner. Tüm seride "token" dedik ama Ders 7'de en naif (karakter) tokenizer'ı kurmuştuk. Ders 9'da gerçek GPT'lerin kullandığı tokenizer'ı — **byte-pair encoding (BPE)** — sıfırdan kuracağız: Unicode/UTF-8 byte'larından başlayıp en sık çiftleri birleştirerek bir sözlük inşa etmek. Ayrıca tiktoken,

sentencepiece gibi production tokenizer'larını ve LLM'lerin tuhaf davranışlarının (neden "strawberry"deki r'leri sayamaz) kökenini göreceğiz.

Ana konular:

- Unicode kod noktaları, UTF-8 byte kodlaması.
- BPE algoritması: en sık byte/token çiftini birleştir (train/encode/decode).
- GPT-2/GPT-4 regex split, tiktoken, sentencepiece, özel token'lar, tokenizer tuhaflıkları.

⚠ Ders 9 Öncesi Yapılacak

- Egzersizleri çöz — özellikle 5 ("strawberry" tuhaflığı).
- "GPT mimariyi Ders 7'de, asistan hattını Ders 8'de gördük; şimdi girdi-tokenizasyonu" çerçevesini hatırla.
- Not: Ders 9 tekrar canlı-kodlama; tokenizer ağ-dışı bir adım (autograd yok).

15.17 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Karpathy'de
Dört aşamalı hat	pretraining → SFT → ödül modeli → RLHF (sırayla, her biri hizalama ekler)	0m53
Pretraining	Devasa metinde next-token; compute'un $\approx$ %99'u; çıktı = base model	1m59
Base model	Güçlü genel bilgi ama asistan değil; soruyu sürdürür, yanıtlamaz	8m06
SFT	Az ama kaliteli (prompt, yanıt) çiftiyle ince ayar; asistan formatı	11m32
Ödül modeli	İnsan sıralamalarından "iyi yanıt" skoru öğrenir (yargılamak kolay)	12m59
RLHF (PPO)	Ödül modeline göre optimize; ChatGPT böyle. Yan etki: mode collapse	15m22
Mode collapse	RLHF modelleri çeşitliliği kaybeder (entropi düşer); base daha çeşitli	18m46
Token simülatörü	LLM iç monolog çalıştırmaz; "düşünmek için token" gerekir	23m13

Kavram	Tanım	Karpathy'de
Chain-of-thought	"Adım adım düşün" → ara token = ara hesap; Sistem 2'yi dışarıdan kur	24m57
Tool use / RAG	Araç (hesap/kod) + getirilen belge ile modelin eksiğini tamamla	≈31m
Limitler	Hallüsinasyon, jailbreak/injection, "ne bilmediğini bilmeme"	35m15

15.18 ML Builder Bağlantıları

8 köprü — State of GPT

1. **Pretraining (next-token)** → Ders 2-7'nin tam yaptığı (cross-entropy), dev ölçekte. İleriye: scaling laws (Ders 10).
2. **Base model** → Ders 7 GPT'nin dev hâli. İleriye: LLaMA/Mistral açık base modelleri, kendi SFT/LoRA temelin.
3. **SFT** → Ders 7 eğitiminin asistan-veri hâli. İleriye: LoRA/QLoRA ile erişilebilir ince ayar.
4. **Ödül modeli** → Stat 110 Bradley-Terry tercih. İleriye: DPO (ödül modelini atlar), RLAIF.
5. **RLHF (PPO)** → pekiştirmeli öğrenme + yargı-üretim asimetrisi. İleriye: Constitutional AI, modern hizalama.
6. **Token simülatörü / tokens-to-think** → Ders 7 sabit-derinlik hesap. İleriye: reasoning modelleri, test-time compute.
7. **CoT / self-consistency / ToT** → Ders 7 autoregressive + arama. İleriye: o1/o3 reasoning, ReAct ajanları.
8. **Tool use / RAG / constraint** → LLM uygulama mimarisi. İleriye: MCP, LangChain, agent framework'leri, OWASP Top 10 for LLMs.

15.19 Karpathy'nin Önerdiği Kaynaklar

Karpathy'nin bu ders/konuşma için referans verdiği kaynaklar:

- **ChatGPT:** openai.com/chatgpt — konuşmanın merkezindeki asistan.
- **Lambda (hesaplama):** lambda.ai — GPU bulut sağlayıcı (eğitim altyapısı bağlamında).
- (Kavramsal temel: OpenAI InstructGPT makalesi — dört aşamalı hat bu çalışmadan gelir.)

! Bu dersten tek bir şey alıp gideceksen

Ham bir pretrain edilmiş GPT (Ders 7), internet metnini taklit eden bir "token simülatörü"dür — başarmayı değil taklit etmeyi ister. Onu yardımcı bir asistana çevirmek insan örnekleri (SFT) + insan tercihleri (ödül modeli + RLHF) ile ek eğitim gerektirir; etkili kullanmak ise modele "düşünmek için token" verip (chain-of-thought) istediğin davranışı açıkça istemekten geçer.

16 GPT Tokenizer'ı Sıfırdan — Byte-Pair Encoding (BPE)

Tokenizer, metni ağa girmeden önce öğrenilmiş alt-kelime parçalarına (token) bölen ağ-dışı bir ön-işleme adımıdır; BPE bu sözlüğü UTF-8 byte'larından en sık çiftleri birleştirerek kurar — ve LLM'lerin yazım, sayma, dil tuhaflıklarının çoğu, model karakterleri değil token'ları gördüğü için bu adımdan doğar

Bölüm bilgisi

- **Karpathy'nin videosu:** YouTube — Let's build the GPT Tokenizer (≈134 dk)
- **Seri:** Neural Networks: Zero to Hero — Ders 9
- **Hoca:** Andrej Karpathy
- **Kaynak repo:** github.com/karpathy/minbpe
- **Okuma süresi:** ≈34 dk

16.1 Bu Derste Ne Var?

Ders 7'de bir GPT kurarken **naif** bir karakter tokenizer kullanmıştık (65 karakter → 65 id). Gerçek GPT'ler böyle yapmaz: **byte-pair encoding (BPE)** kullanırlar. Bu derste o tokenizer'ı sıfırdan kuruyoruz — ve LLM'lerin pek çok tuhaf davranışının (neden "strawberry"deki r'leri sayamaz, neden basit aritmetik/yazım zorlar) kökeninin tokenization olduğunu görüyoruz.

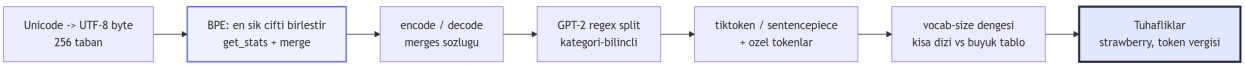
> *"Tokenization is my least favorite part of working with large language models, but unfortunately it is necessary to understand in some detail."* — Karpathy, 0:06

> *"Tokens are the fundamental unit — the atom of large language models. Everything is in units of tokens."* — Karpathy, 3:34

Büyük fikir: metni karakter değil, kelime de değil, **öğrenilmiş alt-kelime parçalarına** (token) bölmek. BPE, UTF-8 byte'larından başlayıp en sık geçen çiftleri tekrar tekrar birleştirerek bu sözlüğü kurar.

Dersin üç büyük fikri:

1. **Unicode → UTF-8 byte** — metni byte dizisine çevirmek (tokenizer'ın ham girdisi).
2. **BPE algoritması** — en sık byte/token çiftini birleştir, yeni token yarat, tekrarla (train/encode/decode).
3. **Production tokenizer'ları** — GPT-2/GPT-4 regex split, tiktoken, sentencepiece, özel token'lar ve tokenization tuhaflıkları.



Şekil 16.1: Ders 9'un kavram haritası: metni Unicode kod noktalarından UTF-8 byte'larına çevirirız (tokenizer'ın ham girdisi); BPE en sık byte çiftlerini birleştirerek (get_stats + merge) sözlüğü büyütür; eğitilen merges ile encode/decode yaparız; GPT-2 metni önce regex ile parçalara böler; production'da tiktoken ve sentencepiece, özel token'lar ve vocab-size dengesi devreye girer; sonuçta LLM tuhaflıklarının çoğu bu adımdan doğar. Slate akış + indigo dönüm noktaları (BPE algoritması ve tuhaflıklar).

Builder Notu — Geriye Ders 2 ve 7, İleriye Ders 10

Geriye (Ders 2, 7):

- **Naif tokenizer = Ders 7.** Ders 7'de karakter tokenizer (s2i/i2s, Ders 2'den) kurmuştuk; bu ders onu gerçek BPE ile değiştirir.
- **Ağ-dışı adım.** Tokenizer, sinir ağının **dışında** ayrı bir ön-işleme adımıdır — autograd/backprop YOK. Metin → token id'leri → (sonra) ağa girer.

İleriye: Tokenizer seçimi, modelin verimliliğini (sıkıştırma oranı), dil-adaletini (İngilizce-dışı diller daha çok token) ve tuhaf hatalarını (yazım, aritmetik) doğrudan belirler. Ders 10'da GPT-2'nin tokenizer'ını kullanacağız.
Tek cümleyle: Tokenizer, metni ağa girmeden önce öğrenilmiş alt-kelime parçalarına (token) bölen, ağ-dışı bir ön-işleme adımıdır; BPE bu sözlüğü UTF-8 byte'larından en sık çiftleri birleştirerek kurar — ve LLM'lerin pek çok tuhaflığı bu adımdan doğar.

16.2 Tokenization Neden Önemli?

Karpathy açıkça sevmediğini söyler ama şart olduğunu vurgular. Neden? Çünkü LLM'lerin pek çok "aptalık"ı tokenization'dan gelir:

> *"[LLMs are] not able to do spelling tasks very easily — that's usually due to tokenization."* — Karpathy, 4:54

Neden bir LLM "strawberry"deki r'leri sayamaz? Çünkü model karakterleri **görmez** — metni token'lar hâlinde görür. "strawberry" birkaç token'a bölünür; model tek tek harfleri ayırt edemez. Aynı şekilde basit aritmetik, İngilizce-dışı diller, kod girintileri — hepsinde tokenization sorun çıkarır. Bu yüzden tokenizer'ı anlamak, LLM davranışını anlamaktır.

Builder Notu — Model Token Görür, Karakter Değil

İleriye: "Modelin gördüğü şey token, karakter değil" — bir LLM uygulamacısının bilmesi gereken en pratik gerçeklerden. Yazım/sayma/format hatalarını teşhis ederken ilk bakılacak yer tokenization.

16.3 tiktokenizer ile Sezgi

Karpathy önce sezgi için bir web aracı gösterir: **tiktokenizer**. Bir metin yazıp hangi token'lara bölündüğünü görsel olarak görürsün. Gözlemler:

- Sık kelimeler tek token; nadir kelimeler parçalanır.
- Baştaki boşluk token'ın parçasıdır (`" token"` ile `"token"` farklı id).
- Sayılar tutarsız bölünür (`"127"` tek token, `"677"` iki token olabilir) — aritmetik zorluğunun kökü.
- İngilizce-dışı metin daha çok token'a bölünür (aynı anlam, daha çok token = daha pahalı, daha kısa etkin bağlam).

Bu görsel araç, "model neden böyle davranıyor" sorularının çoğunu açıklar.

> Builder Notu — Token Sayısı = Maliyet
>
> **İleriye:** Bir prompt'un kaç token olduğu, hem maliyeti (API token başı ücret) hem etkin bağlamı belirler. İngilizce-dışı dillerin daha çok token'a bölünmesi ("token vergisi"), çok dilli adalet sorunudur; Türkçe de bundan etkilenir.

16.4 String, Unicode ve UTF-8 Byte

Tokenizer'ın ham girdisi metindir; ama bilgisayar metni **byte** olarak görür. Python'da bir string, Unicode **kod noktalarından** (code point) oluşur; `ord` ile her karakterin kod noktasını alırız. Modeli evrensel kılmak için metni **UTF-8** ile byte dizisine çevirir (her karakter 1-4 byte).

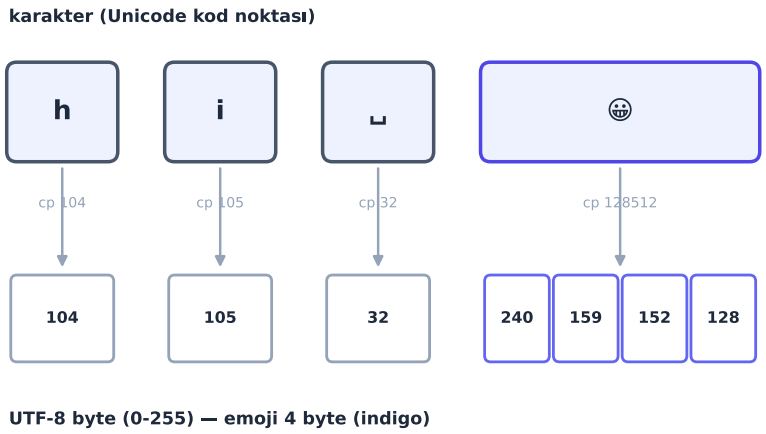
```
ord('h')                          # 104  (Unicode kod noktasi)
ord('')                           # 128512
list('hi '.encode('utf-8'))      # [104, 105, 32, 240, 159, 152, 128]  byte'lar (0-255)
```

UTF-8 neden? Çünkü 256 olası byte değeri (0-255) ile **her dilden her karakteri** temsil edebilir — Türkçe, Çince, emoji dahil. Tokenizer byte'lardan başlar: başlangıç sözlüğü = 256 byte. Sorun: byte-düzeyi diziler **çok uzun** (her karakter birkaç byte). Çözüm: en sık byte çiftlerini birleştirip sözlüğü büyütmek — BPE.

> Builder Notu — Byte-Level: Hiçbir Şey 'Bilinmeyen' Değil
>
> **İleriye:** "Byte-level" başlangıç, modern tokenizer'ların (GPT-2/4) standardı: hiçbir metin "bilinmeyen" (unknown) olmaz, çünkü en kötü ihtimalle byte'lara iner. UTF-8'in 256-byte tabanı, çok dilli evrenselliği sağlar — ama İngilizce-dışı diller daha çok byte/token harcar.

16.5 BPE Algoritması

Byte-pair encoding (BPE) fikri basit: byte dizisinde **en sık geçen ardışık çifti** bul, onu **yeni bir token** ile değiştir, tekrarla. Her birleştirme sözlüğü 1 büyütür ($256, 257, 258, \ldots$) ve diziyi kısaltır.



Şekil 16.2: Metin → Unicode kod noktası → UTF-8 byte (GERÇEK, `'hi 🤩'`). Tokenizer'ın ham girdisi byte'lardır. ASCII karakterler (h, i, boşluk) tek byte; ama emoji 🤩 (kod noktası 128512) **dört byte**'a açılır ($[240, 159, 152, 128]$). UTF-8'in 256 olası byte değeriyle her dilden her karakter temsil edilebilir — bu yüzden BPE byte'lardan başlar (hiçbir metin "bilinmeyen" olmaz). Sorun: byte dizileri uzun; BPE bu yüzden en sık çiftleri birleştirip kısaltır.

"Byte Pair Encoding (BPE) algorithm walkthrough." — Karpathy, 23:49

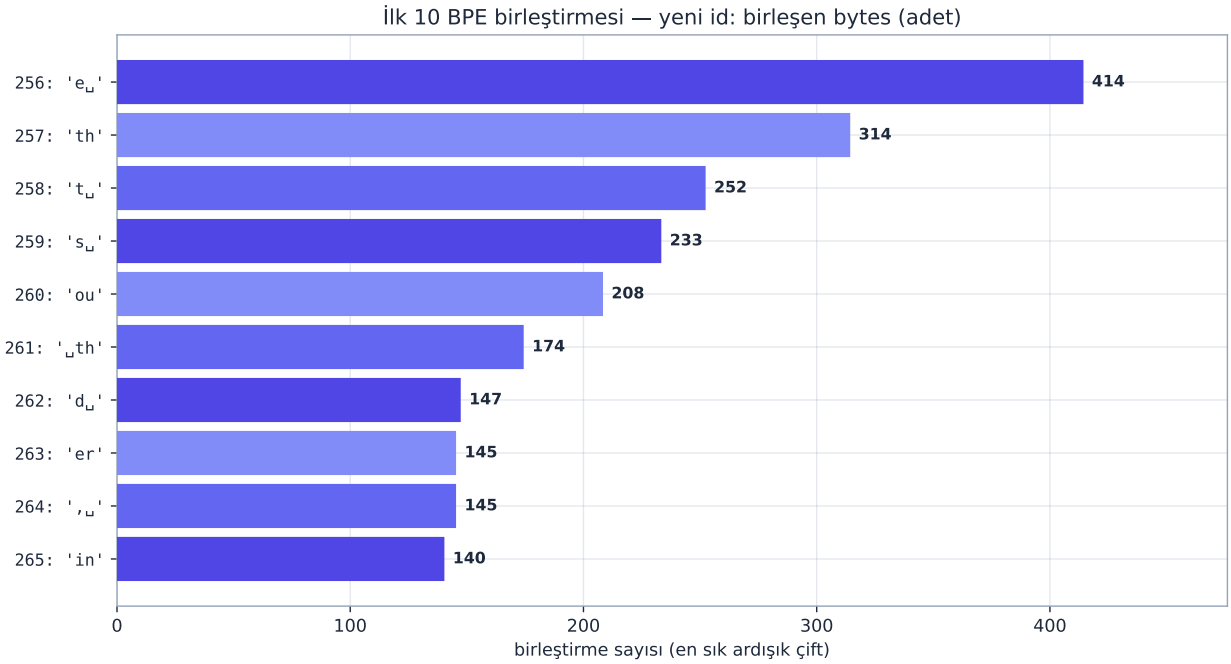
Oyuncak örnek: `"aaabdaaabac"` dizisinde en sık çift `"aa"`. Onu `Z` ile değiştir → `"ZabdZabac"`. Sonra en sık `"ab"` → `Y`: `"ZYdZYac"`. Diziler kısalır, sözlük büyür. Gerçek metinde de aynısı olur — aşağıda tiny Shakespeare üzerinde eğittiğimiz tokenizer'ın ilk birleştirmeleri (en sık İngilizce desenler tek token oluyor):

> Builder Notu — BPE = Sıkıştırma
>
> **İleriye:** BPE aslında bir **sıkıştırma** algoritmasıdır (veri sıkıştırmadan ödünç). Sözlük büyüklüğü bir denge: büyük sözlük = kısa diziler (verimli) ama büyük embedding tablosu + seyrek token'lar. GPT-2: $\approx$ 50K, GPT-4: $\approx$ 100K token.

16.6 BPE İmplementasyonu: get_stats, merge, Eğitim

BPE üç parçadan oluşur. **get_stats**: ardışık çiftleri say. **merge**: bir çifti yeni id ile değiştir. **Eğitim döngüsü**: en sık çifti bul, birleştir, tekrarla.

```
def get_stats(ids):
    counts = {}
    for pair in zip(ids, ids[1:]):   # ardisik ciftler
        counts[pair] = counts.get(pair, 0) + 1
    return counts
```



Şekil 16.3: BPE eğitiminin ilk birleştirmeleri (tiny Shakespeare ilk 15.000 karakter üzerinde, GERÇEK). Algoritma her adımda **en sık geçen ardışık byte çiftini** bulup yeni bir token'a birleştirir. İlk birleştirme `'e '` (414 kez — "e"+boşluk), sonra `'th'` (314), `'t '` (252), `'s '` (233), `'ou'` (208)… Klasik İngilizce desenler tek token oluyor; sık desenler sıkışır, nadir olanlar parçalı kalır. Her birleştirme sözlüğü 1 büyütür ($256 \to 257 \to \ldots$) ve diziyi kısaltır.

```python
def merge(ids, pair, idx):
    newids = []
    i = 0
    while i < len(ids):
        if i < len(ids) - 1 and ids[i] == pair[0] and ids[i+1] == pair[1]:
            newids.append(idx)   # cifti yeni token ile degistir
            i += 2
        else:
            newids.append(ids[i])
            i += 1
    return newids
```

"Implement first the function that finds the most common pair." — Karpathy, 28:32

```python
vocab_size = 276            # hedef sozluk (256 byte + 20 birlestirme)
num_merges = vocab_size - 256
ids = list(tokens)          # baslangic: byte id'leri
merges = {}                 # (cift) -> yeni id
for i in range(num_merges):
    stats = get_stats(ids)
    pair = max(stats, key=stats.get)   # en sik cift
    idx = 256 + i
    ids = merge(ids, pair, idx)
    merges[pair] = idx                  # birlestirme kaydi
```

`merges` sözlüğü, eğitilen tokenizer'ın "modeli"dir: hangi çiftin hangi yeni token'a birleştiğini tutar. encode/decode bunu kullanır.

> Builder Notu — get_stats = Ders 2 Bigram Sayımı
>
> **Geriye (Ders 2):** `get_stats`, Ders 2'deki bigram sayımının (çiftleri say) ta kendisi — ama amacı farklı (en sık çifti birleştirmek). `merges` sözlüğü, Ders 2'nin s2i'sinin BPE karşılığı.
> **İleriye:** Bu $\approx$ 30 satır, `minbpe`'nin çekirdeği. Production'da (tiktoken) aynı algoritma C/Rust'ta hızlandırılır; ama mantık birebir budur.

16.7 encode ve decode

Tokenizer, modelin **dışında** ve ondan **ayrı** eğitilir. Eğitilen `merges` ile iki yönlü çeviri yaparız: **decode** (id'ler → metin) ve **encode** (metin → id'ler).

decode: her id'yi byte'larına çevir, birleştir, UTF-8 çöz. Geçersiz byte dizilerinde `errors='replace'` kullanılır:

```
vocab = {idx: bytes([idx]) for idx in range(256)}
for (p0, p1), idx in merges.items():
    vocab[idx] = vocab[p0] + vocab[p1]   # birlestirilen token'in byte'lari

def decode(ids):
    tokens = b''.join(vocab[idx] for idx in ids)
    return tokens.decode('utf-8', errors='replace')   # gecersiz byte -> replacement
```

encode: metni byte'lara çevir, sonra `merges`'i (en erken öğrenilen birleştirmeden başlayarak) açgözlü uygula:

```
def encode(text):
    tokens = list(text.encode('utf-8'))
    while len(tokens) >= 2:
        stats = get_stats(tokens)
        pair = min(stats, key=lambda p: merges.get(p, float('inf')))   # en erken merge
        if pair not in merges:
            break
        tokens = merge(tokens, pair, merges[pair])
    return tokens
```

`errors='replace'` kritik: model rastgele token üretebilir, bunlar geçerli UTF-8 olmayabilir; `replace` çökme yerine replacement karakteri verir. Eğittiğimiz tokenizer kayıpsızdır (`decode(encode(text)) == text`) ve metni belirgin biçimde sıkıştırır:

> Builder Notu — Tokenizer Ayrı Eğitilir
>
> **İleriye:** "Tokenizer ayrı eğitilir" gerçeği önemli: tokenizer'ın eğitim verisi modelinkinden farklı olabilir (örn. çoğu İngilizce tokenizer, az kod görmüş → kodu verimsiz tokenize eder). `errors='replace'` (geçersiz start byte sorunu), gerçek dünyada sık karşılaşılan bir tuzak.

16.8 GPT-2 Regex Split Deseni

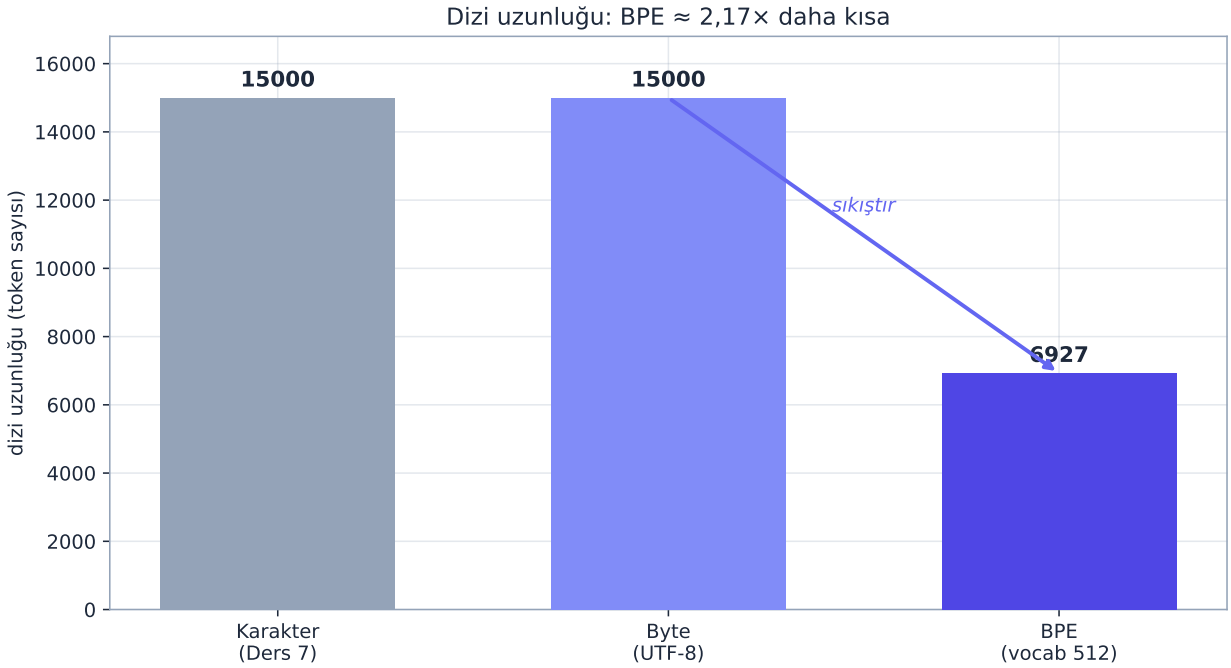
Naif BPE bir sorun yaratır: birleştirmeler kategori sınırlarını aşabilir (örn. "dog" + "." → "dog." tek token, ya da harf+boşluk birleşir). GPT-2, bunu önlemek için metni önce bir **regex** ile parçalara böler, BPE'yi her parça içinde ayrı çalıştırır (parçalar arası birleşme olmaz).

> *"Regex patterns to force splits across categories."* — Karpathy, 57:31

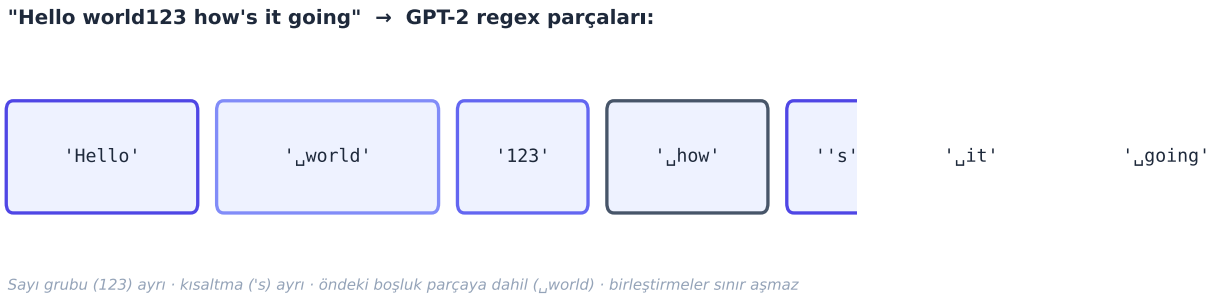
GPT-2'nin meşhur deseni (harf grupları, sayı grupları, noktalama, boşluklar ayrı tutulur):

```
import regex as re   # 're' degil 'regex' (Unicode ozellik siniflari icin)
gpt2pat = re.compile(r"""'s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s
re.findall(gpt2pat, "Hello world123 how's it going")
```

`\p{L}` (harfler), `\p{N}` (sayılar) Unicode özellik sınıflarıdır — bu yüzden standart `re` değil, `regex` modülü gerekir. Böylece "dog." → `["dog", "."]` (nokta harfe yapışmaz):



Şekil 16.4: Aynı 15.000 karakterlik metin, üç tokenizasyon altında dizi uzunluğu (GERÇEK). **Karakter** tokenizer (Ders 7): her karakter bir token → 15.000 token. **Byte** (UTF-8): bu metinde ASCII olduğu için yine 15.000. **BPE** (vocab 512): sık desenler tek token → yalnızca 6.927 token ($\approx 2{,}17\times$ sıkıştırma). Daha kısa dizi = aynı bağlam penceresine daha çok metin, daha verimli eğitim. Karakterden BPE'ye geçiş, gerçek GPT'lerin neden BPE kullandığının özüdür.



Şekil 16.5: GPT-2'nin regex split deseni (GERÇEK, `regex` modülü). Naif BPE birleştirmeleri kategori sınırlarını aşabilir (harf+noktalama, harf+boşluk birleşir). GPT-2 bunu önlemek için metni önce regex ile **parçalara** böler, BPE'yi her parça içinde ayrı çalıştırır. `"Hello world123 how's it going"` → görülen parçalar: harf grupları, sayı grubu (`123` ayrı), kısaltma (`'s`), öndeki boşluk parçaya dahil (`world`). Bu desen, sayıların tutarsız bölünmesinin (aritmetik zorluğu) ve boşluk davranışının kökenidir.

Builder Notu — Regex = GPT-2 Sözleşmesi

İleriye: Bu regex, GPT-2 tokenizer'ının sözleşmesidir; metni "neyin neyle birleşebileceği" açısından böler. Sayıların tutarsız bölünmesi (aritmetik zorluğu) ve boşluk davranışı bu desenden gelir. GPT-4 farklı (daha iyi) bir regex kullanır.

16.9 tiktoken: GPT-2 vs GPT-4

OpenAI'nin **tiktoken** kütüphanesi, GPT-2/GPT-4 tokenizer'larını (hızlı, Rust ile) sağlar. (Bu derste tiktoken'ı kavramsal anlatıyoruz; demolar kendi BPE'mizle yapıldı — algoritma birebir aynı.)

```
import tiktoken
enc = tiktoken.get_encoding('gpt2')        # GPT-2 tokenizer
enc = tiktoken.get_encoding('cl100k_base') # GPT-4 tokenizer
enc.encode("hello world")
```

Farklar: **(1)** GPT-4 sözlüğü daha büyük ($\approx$ 100K vs GPT-2 $\approx$ 50K) $\rightarrow$ daha kısa diziler. **(2)** GPT-4 regex'i geliştirildi: sayıları en fazla 3 haneye böler, büyük/küçük harfi daha iyi ele alır. **(3)** GPT-4 kod girintilerini (ardışık boşluklar) daha verimli tokenize eder. tiktoken yalnızca **encode/decode** yapar (eğitim yok); `merges`'i hazır gelir.

Builder Notu — tiktoken = Maliyet Hesabı

İleriye: tiktoken, production'da token sayma/maliyet hesabı için standart araç (OpenAI API). Tokenizer şeması (cl100k vs o200k), modelin verimliliğini ve dil-adaletini belirler; yeni modeller genelde daha büyük, daha dengeli sözlükler kullanır.

16.10 OpenAI gpt2 encoder.py

Karpathy OpenAI'nin yayımladığı gerçek GPT-2 tokenizer kodunu (`encoder.py`) gezer. İçinde bizim kurduğumuzla aynı parçalar var: bir `encoder` (token $\rightarrow$ id, bizim vocab'ımız), bir `bpe_merges` listesi (bizim `merges`), ve bir regex. Ek olarak bir `byte_encoder` katmanı vardır — byte'ları "görünür" karakterlere eşleyen kozmetik bir eşleme (örneğin boşluğu `Ġ` olarak gösterir).

Mesaj: kurduğumuz minik BPE, OpenAI'nin production kodunun çekirdeğiyle **aynı** — yalnızca birkaç pratik detay (byte_encoder, dosya formatı) eklenmiş.

Builder Notu — Sıfırdan Kur, Sonra Karşılaştır

Geriye (Ders 1, 7): "Sıfırdan kur, sonra gerçek kodla karşılaştır" deseni (Ders 1 micrograd vs PyTorch, Ders 4 `nn.Linear`) burada da: kendi BPE'miz vs OpenAI `encoder.py`. Artık production tokenizer'ı bir kara kutu değil.

16.11 Özel Token'lar

Normal token'lar BPE ile veriden öğrenilir. Ama bazı token'lar **elle eklenir** ve özel anlam taşır — bunlar veride geçmez, dışarıdan enjekte edilir:

- `<|endoftext|>`: belgeler arası sınır (pretraining'de, Ders 8). Model "burada yeni bir belge başlıyor" bilgisini alır.
- **FIM** (fill-in-the-middle): kod tamamlamada ortayı doldurmak için özel token'lar.
- Sohbet sınırları: `<|im_start|>` / `<|im_end|>` — kullanıcı/asistan dönüşlerini ayırır (ChatGPT formatı).

Özel token eklemek **model cerrahisi** gerektirir: token embedding tablosuna (Ders 3-7 `wte`) ve LM head'e (çıkış katmanı) yeni satırlar eklenir, bunlar eğitilir. Yani tokenizer değişikliği modele de dokunur.

> Builder Notu — Özel Token = Ders 8 Altyapısı
>
> **Geriye (Ders 7-8):** `<|endoftext|>` = Ders 8'deki pretraining belge-ayracı; sohbet token'ları = Ders 8'deki asistan formatının (SFT) altyapısı. `wte` genişletme = Ders 7'nin token embedding tablosuna ekleme.
> **İleriye:** Özel token'lar, chat template'lerin (Claude/GPT mesaj formatı), tool-use işaretçilerinin ve FIM kod modellerinin temelidir. Yanlış kullanımları (prompt injection ile özel token enjekte etme) bir güvenlik açığıdır.

16.12 minbpe ve sentencepiece

Karpathy iki gerçek araç gösterir. **minbpe** — kendi temiz BPE implementasyonu (bu dersin kodu). **sentencepiece** — Google'ın kütüphanesi, Llama 2'nin sözlüğünü eğitmekte kullanılmış.

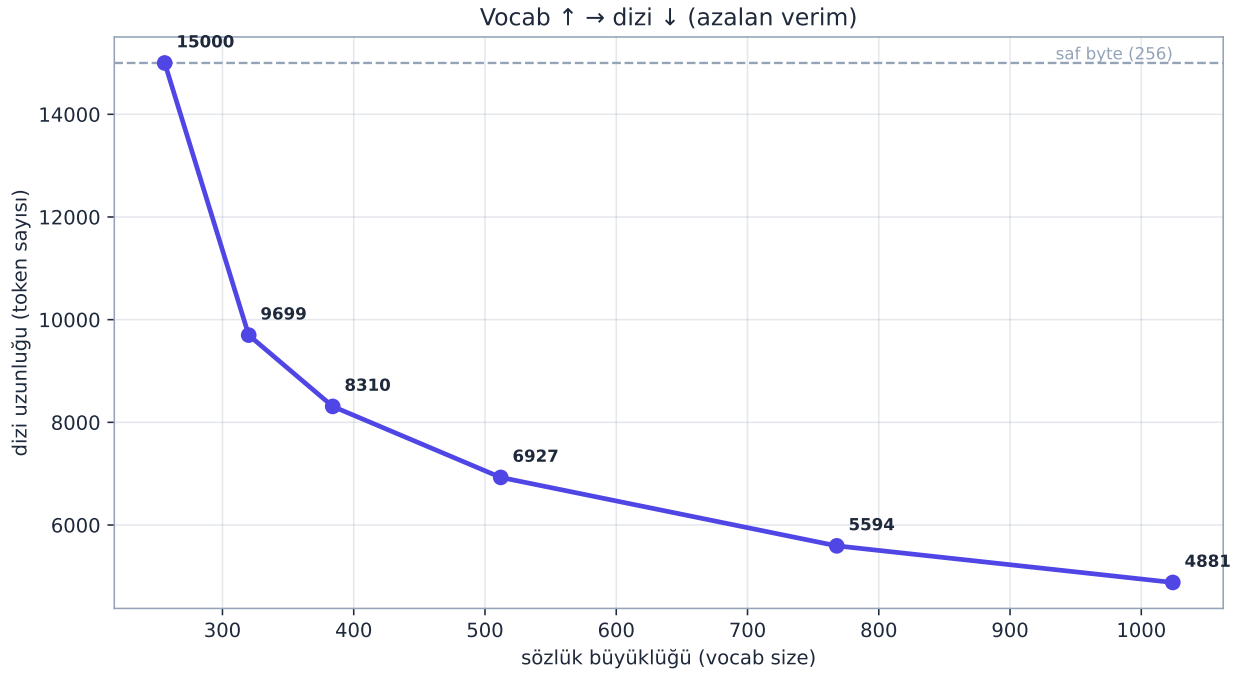
> *"sentencepiece library intro, used to train Llama 2 vocabulary."* — Karpathy, 1:28:41

Önemli fark: tiktoken **byte-level BPE** (önce UTF-8 byte, sonra merge); sentencepiece ise doğrudan **Unicode kod noktaları** üzerinde çalışır, nadir karakterler için `byte_fallback`'e iner. sentencepiece'in çok sayıda ayarı vardır (`add_dummy_prefix`, `character_coverage`, `<unk>`); bunlar tarihsel ve karmaşıktır — Karpathy dikkatli olmayı önerir.

> Builder Notu — İki Yaklaşım Yan Yana Yaşar
>
> **İleriye:** İki yaklaşım (tiktoken byte-level vs sentencepiece code-point + byte_fallback) sektörde yan yana yaşar. Kendi modelini eğitirken tokenizer kütüphanesi seçimi (ve onlarca ayarı) erken ve kalıcı bir karardır; yanlış ayar (örn. düşük character_coverage) bir dili sakatlayabilir.

16.13 Vocab-Size Dengeleri, Genişletme ve Multimodal

Sözlük büyüklüğü (vocab size) bir dengedir. Büyük sözlük: diziler kısalır (her token daha çok bilgi, daha verimli bağlam) — ama (a) embedding tablosu ve çıkış softmax'ı büyür, (b) her token daha az eğitim örneği görür (seyrekleşir). Küçük sözlük: tersine. Aşağıda kendi metnimizde bu dengeyi GERÇEK ölçtük:



Şekil 16.6: Sözlük büyüklüğü (vocab size) dengesi (GERÇEK, aynı 15.000 karakterlik metin). Sözlük büyüdükçe dizi kısalır: 256 token (saf byte) → 15.000; 512 → 6.927; 1024 → 4.881. Ama kazanç **azalan verim** gösterir (her yeni birleştirme giderek daha az kısaltır) ve büyük sözlük embedding tablosunu + softmax'ı büyütür, token'ları seyrekleştirir. Bu yüzden pratik bir orta nokta seçilir (GPT-2 $\approx$ 50K, GPT-4 $\approx$ 100K).

Sözlük genişletme: var olan bir modele yeni token eklemek (özel token'lar) mümkün — embedding + LM head'e satır ekle, yalnızca onları eğit (model cerrahisi). **Multimodal**: tokenizer fikri metne özel değil. Görüntü/ses de "token"a çevrilebilir — örneğin Sora, videoyu **görsel yamalara** böler, her yamayı bir token gibi işler. Aynı transformer, farklı modalitelerin token'larını işleyebilir.

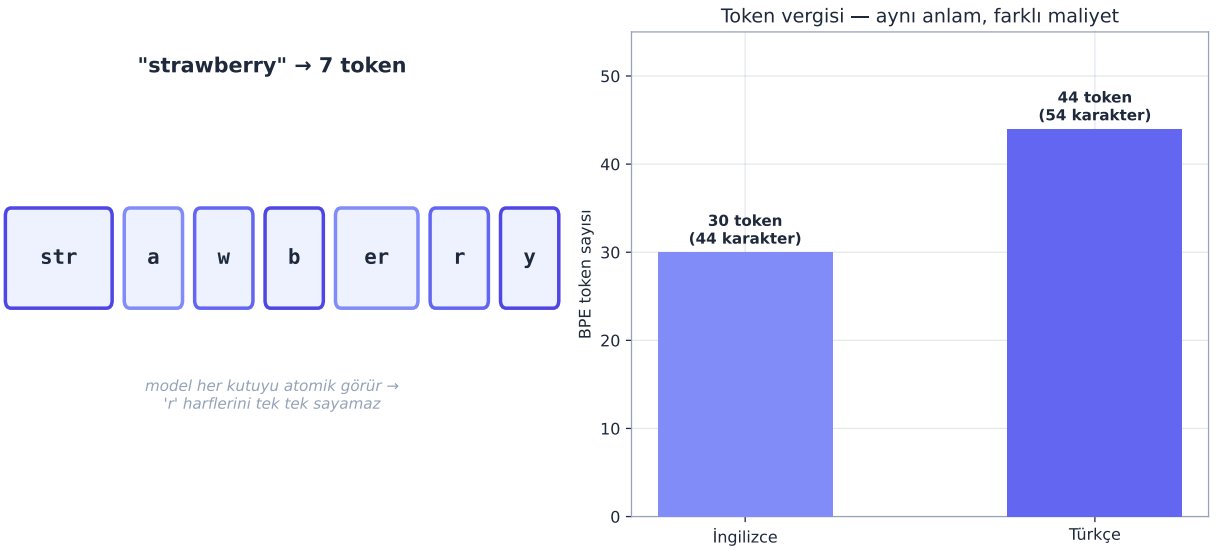
> Builder Notu — Vocab Size = Scaling Kararı
>
> **İleriye:** Vocab size, scaling/maliyet kararının parçası (Ders 10). Multimodal tokenization (görüntü→patch, ses→kod), GPT-4o/Gemini gibi modellerin temeli: tek transformer, çok modalite — hepsi "token dizisi" olarak.

16.14 Tokenization Tuhaflıkları

Karpathy, LLM'lerin meşhur "aptallıkları"nın tokenization kökenini sıralar:

- **Yazım/sayma** (strawberry'deki r'ler): model karakterleri değil token'ları görür.
- **Sondaki boşluk** (trailing whitespace): " " ayrı token; promptu boşlukla bitirmek modeli şaşırtır.
- **SolidGoldMagikarp**: eğitim verisinde geçen ama model eğitiminde hiç görülmeyen token'lar (örn. bir Reddit kullanıcı adı) → model onlarla karşılaşınca "bozulur" (eğitilmemiş embedding).
- **İngilizce-dışı / kod**: daha çok token = daha pahalı, daha kötü performans.

Aşağıda iki tuhaflığı kendi BPE'mizle GERÇEK gösteriyoruz — "strawberry"nin token bölünmesi ve Türkçe'nin "token vergisi":



Şekil 16.7: Tokenization'dan doğan iki tuhaflık (GERÇEK, kendi BPE'mizle). **Sol:** "strawberry" tek tek harflere değil, 7 token'a bölünür — model bu token'ları atomik görür, içlerindeki tek tek 'r' harflerini **göremez**; bu yüzden "kaç r var?" gibi karakter-düzeyli sorular zordur. **Sağ:** "token vergisi" — aynı anlamı taşıyan Türkçe cümle, İngilizce'den daha çok token'a bölünür (44 vs 30); İngilizce-dışı diller daha pahalı, daha kısa etkin bağlam. "Model karakteri değil token'ı görür."

Karpathy'nin önerisi: mümkünse hazır, iyi-test edilmiş bir tokenizer (tiktoken cl100k) kullan; kendi eğitecek-sen sentencepiece'i dikkatle ayarla. Ve tokenizasyonu bir gün ortadan kaldırmayı diler:

> *"Eternal glory goes to anyone who can get rid of it."* — Karpathy, 2:10:21

Builder Notu — Debug Çantasının İlk Maddesi

İleriye: Bu tuhaflıklar, bir LLM uygulamacısının debug çantasının ilk maddesi: "model neden bunu yapamıyor" → önce tokenization'a bak. SolidGoldMagikarp türü açıklar, güvenlik (adversarial token) açısından da önemli.

16.15 Bu Dersin Özeti

1. **Tokenizer**, metni ağa girmeden önce token'lara bölen, **ağ-dışı** bir ön-işleme adımıdır (autograd yok). LLM tuhaflıklarının çoğu buradan doğar.

2. **Unicode → UTF-8 byte**: metni 256-byte tabanına çevir (her dil/karakter temsil edilebilir); ama byte dizileri uzun.
3. **BPE**: en sık ardışık çifti bul, yeni token'a birleştir, tekrarla — sözlüğü büyüt, diziyi kısalt (sıkıştırma; bizde 15.000 byte → 6.927 token, $\approx 2{,}17\times$).
4. **İmplementasyon**: `get_stats` (çiftleri say) + `merge` (çifti değiştir) + eğitim döngüsü; `merges` sözlüğü tokenizer'ın modelidir.
5. **encode/decode**: metin → byte → açgözlü merge; id → byte → UTF-8 (`errors='replace'`); kayıpsız roundtrip.
6. **GPT-2 regex split**: metni kategorilere böl (harf/sayı/noktalama/boşluk ayrı) ki birleştirmeler sınır aşmasın (`\p{L}`, `\p{N}` → `regex` modülü).
7. **tiktoken** (GPT-2/GPT-4, cl100k_base) vs **sentencepiece** (Llama 2, byte_fallback); özel token'lar (`<|endoftext|>`, FIM, chat) model cerrahisiyle eklenir.
8. **Vocab size** bir denge (kısa dizi vs büyük tablo/seyrek token); multimodal tokenization (Sora görsel yamalar) aynı fikrin uzantısı.
9. **Tuhaflıklar**: yazım/sayma, trailing whitespace, SolidGoldMagikarp, İngilizce-dışı maliyet — hepsi tokenization kökenli.

! Tek Bir Cümle

Tokenizer, metni ağ-dışı bir adımda öğrenilmiş alt-kelime parçalarına (token) bölen bir ön-işlemedir; BPE bu sözlüğü UTF-8 byte'larından en sık çiftleri birleştirerek kurar — ve LLM'lerin yazım/sayma/dil tuhaflıklarının çoğu, "model karakterleri değil token'ları görür" gerçeğinden doğar.

16.16 Kontrol Soruları

i Soru 1: 'aaabdaaabac' dizisinde BPE'nin ilk birleştirmesi ne olur? Sonraki adım ne olabilir?

Cevap: En sık ardışık çift `"aa"`'dır; onu yeni bir token `Z` ile değiştir → `"ZabdZabac"`. **İlk birleştirme: (a, a) → Z.** Sonra yeni dizide en sık çift `"ab"`; onu `Y` ile değiştir → `"ZYdZYac"`. Her adım sözlüğü 1 büyütür (yeni token), diziyi kısaltır. Metinde de aynısı: en sık byte çiftleri (örn. bizim ölçümümüzde `"e "` 414 kez, `"th"` 314 kez) tek token olur — sık desenler sıkışır.

i Soru 2: Ders 7'deki karakter tokenizer ile BPE arasındaki temel denge nedir?

Cevap: Karakter tokenizer (Ders 7): küçük sözlük (65), ama **çok uzun diziler** — her karakter bir token, bağlam hızla dolar. **BPE**: sık desenleri (alt-kelimeler) tek token yapar → **çok daha kısa diziler** (bizde 15.000 karakter → 6.927 token), ama sözlük büyük (50K-100K) → büyük embedding tablosu + softmax. BPE ayrıca "bilinmeyen" sorununu çözer (en kötü byte'a iner). Denge: BPE, dizi-uzunluğu ile sözlük-boyutu arasında karakterden çok daha iyi bir orta nokta bulur — bu yüzden tüm gerçek GPT'ler BPE kullanır.

Soru 3: Bir LLM neden 'strawberry' kelimesindeki 'r' harflerini güvenilir sayamaz?

Cevap: Model metni **karakter** olarak görmez — **token** olarak görür. "strawberry" tek tek harflere değil, birkaç token'a bölünür (bizim BPE'mizde 7 token: `['str','a','w','b','er','r','y']`). Model bu token'ları birer atomik birim olarak işler; içlerindeki tek tek 'r' harflerini "göremez" (token'ın iç karakter yapısı modele şeffaf değil). Karakter-düzeyli bir işlem (sayma, ters çevirme, yazım) bu yüzden zordur. Aynı sebep: aritmetik, kafiye, anagram. Çözüm yolları: karakterleri açıkça ayır (boşlukla), veya araç kullandır.

Soru 4: (Builder) Sözlük büyüklüğünü (vocab size) artırmanın artıları ve eksileri nelerdir?

Cevap: Artı: Büyük sözlük → diziler kısalır (her token daha çok bilgi taşır) → aynı bağlam penceresine daha çok "metin" sığar, hesap daha verimli (bizde vocab $256 \to 1024$'te dizi $15.000 \to 4.881$). **Eksi:** (1) Embedding tablosu (Ders 3-7 `wte`) ve çıkış softmax'ı büyür → daha çok parametre + bellek; (2) her token daha az eğitim örneği görür → **seyrekleşir**, nadir token'lar az-eğitilmiş kalır (SolidGoldMagikarp riski); (3) çok büyük softmax yavaşlar. Bu yüzden bir orta nokta seçilir (GPT-2 $\approx$ 50K, GPT-4 $\approx$ 100K). Karar, model boyutu/veri/dil dağılımına bağlı bir denge — scaling kararının parçası (Ders 10).

16.17 Egzersizler

Egzersiz 1 (BPE eğitimi). `get_stats` ve `merge`'i yaz. Bir paragraf metni UTF-8 byte'larına çevir, vocab_size=276 (20 birleştirme) ile BPE eğit. `merges` sözlüğünü incele — hangi çiftler birleşti (sık desenler: `" th"`, `"in"`)?

Egzersiz 2 (encode/decode roundtrip). `encode` ve `decode`'u yaz. Rastgele bir metin için `decode(encode(text)) == text` olduğunu doğrula (kayıpsız roundtrip). Sonra `decode`'a rastgele id'ler ver — `errors='replace'` sayesinde çökmediğini gözlemle.

Egzersiz 3 (Geçersiz byte). Geçerli UTF-8 olmayan bir byte dizisi (örn. tek başına `0x80`) `decode` et. `errors='replace'` ile replacement karakteri ( ) döndüğünü, `errors='strict'` ile hata fırlattığını karşılaştır. Bu, modelin geçersiz token üretmesine karşı neden gerekli?

Egzersiz 4 (Token vergisi). Aynı anlamı taşıyan bir İngilizce ve Türkçe cümleyi tokenize et. Türkçe'nin daha çok token'a bölündüğünü (token vergisi) say ve gözlemle. "strawberry"yi tokenize edip kaç token olduğunu, harflerin nasıl bölündüğünü gör.

Egzersiz 5 (Sonraki dersin habercisi). Artık iki büyük parçaya sahibiz: **mimari** (Ders 7, transformer) ve **tokenizer** (Ders 9, BPE). Gerçek bir GPT-2 (124M) üretmek için daha ne gerekir? (a) Ölçek (kaç parametre, kaç token), (b) gerçek veri (internet metni), (c) eğitim hızı (GPU, mixed precision), (d) eğitim teknikleri (LR schedule, gradient clipping, dağıtık eğitim). Bu sorular, Ders 10'da (**GPT-2'yi Yeniden Üretmek**) tüm seriyi production-ölçek tek bir modelde birleştirmeyi motive eder.

16.18 Sonraki Ders İçin Hazırlık

Ders 10: GPT-2'yi (124M) Yeniden Üretmek (nanoGPT) — Andrej Karpathy

Serinin finali. Ders 7'nin transformer'ı + Ders 9'un tokenizer'ı + Ders 8'in pipeline kavramları, gerçek bir **GPT-2 124M** modelinde birleşir. Karpathy OpenAI'nin GPT-2'sini sıfırdan yeniden üretir: gerçek mimari (HuggingFace ağırlıklarıyla uyumlu), production eğitim hızı (TF32/bf16/torch.compile, FlashAttention), ve ciddi eğitim altyapısı (LR schedule, gradient clipping, weight decay, gradient accumulation, dağıtık eğitim — DDP).

Ana konular:

- GPT-2 `nn.Module` + HuggingFace ağırlık yükleme + weight tying.
- Eğitimi hızlandırma: TF32, bfloat16, torch.compile, FlashAttention.
- Hiperparametreler, gradient clipping, LR schedule, weight decay, gradient accumulation, DDP; FineWeb-EDU verisi + HellaSwag değerlendirmesi.

> ⚠ Ders 10 Öncesi Yapılacak
>
> - Egzersizleri çöz — özellikle 4 (token vergisi) ve 5 (GPT-2 için ne gerekir).
> - Ders 7 (transformer mimarisi) ve Ders 9 (BPE tokenizer) kodlarını hazır tut; Ders 10 ikisini birleştirir.
> - Ders 4 (init/mixed precision notları), Ders 8 (scaling/pipeline) — Ders 10'un production teknikleri için zemin.

16.19 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Karpathy'de
Tokenization önemi	LLM tuhaflıkları (yazım, sayma, dil) çoğu buradan; model token görür, karakter değil	0m06
tiktokenizer	Web aracı: metnin token'lara nasıl bölündüğünü görselleştirir	5m49
Unicode / UTF-8 byte	Metin → kod noktaları → UTF-8 byte (256 taban); tokenizer'ın ham girdisi	14m56
BPE algoritması	En sık ardışık çifti yeni token'a birleştir, tekrarla; sözlük büyür, dizi kısalır	23m49
get_stats + merge	Çiftleri say + en sık çifti yeni id ile değiştir; `merges` sözlüğü = tokenizer modeli	26m56
encode / decode	metin → byte → açgözlü merge; id → byte → UTF-8 (`errors='replace'`)	39m20

Kavram	Tanım	Karpathy'de
GPT-2 regex split	Metni kategorilere böl (harf/sayı/noktalama/boşluk); birleştirmeler sınır aşmaz	57m35
tiktoken (cl100k)	OpenAI GPT-2/GPT-4 tokenizer'ı; encode/decode (eğitim yok), Rust-hızlı	1h11m
Özel token'lar	endoftext, FIM, sohbet sınırları; elle eklenir (model cerrahisi)	1h18m
minbpe / sentencepiece	minbpe (byte-level BPE); sentencepiece (Llama 2, code-point + byte_fallback)	1h25m
Vocab size dengesi	Büyük: kısa dizi ama büyük tablo/seyrek; küçük: tersine (GPT-2 $\approx$ 50K, GPT-4 $\approx$ 100K)	2h23m
Tokenization tuhaflıkları	strawberry sayma, trailing whitespace, SolidGoldMagikarp, İngilizce-dışı maliyet	2h31m

16.20 ML Builder Bağlantıları

8 köprü — Tokenizer

1. **Tokenizer = ağ-dışı adım** → Ders 7 naif char tokenizer'ın gerçek hâli. İleriye: GPT-2 tokenizer (Ders 10).
2. **Unicode/UTF-8 byte** → temel veri temsili. İleriye: byte-level BPE, çok dilli evrensellik + token vergisi.
3. **BPE (get_stats)** → Ders 2 bigram sayımının çift-birleştirme amaçlı kullanımı. İleriye: tiktoken/sentencepiece production.
4. **encode/decode + merges** → Ders 2 s2i/i2s'in BPE karşılığı. İleriye: `errors='replace'` (geçersiz byte dayanıklılığı).
5. **GPT-2 regex** → kategori-bilinçli bölme. İleriye: aritmetik/kod/dil davranışının kökeni.
6. **Özel token'lar** → Ders 8 pretraining (endoftext) + asistan formatı (sohbet token'ları). İleriye: chat template, FIM, tool işaretçileri.
7. **Vocab size** → embedding tablosu (Ders 3-7 `wte`) + softmax boyutu dengesi. İleriye: scaling kararı (Ders 10).
8. **Tokenization tuhaflıkları** → "model token görür" gerçeği. İleriye: LLM debug çantasının ilk maddesi; adversarial token güvenliği.

16.21 Karpathy'nin Önerdiği Kaynaklar

Karpathy'nin bu ders için verdiği kaynaklar:

- **minbpe repo:** github.com/karpathy/minbpe — dersin kodu + adım adım egzersiz.
- **tiktokenizer (web):** tiktokenizer.vercel.app — token görselleştirme aracı.
- **tiktoken:** github.com/openai/tiktoken — OpenAI BPE kütüphanesi.
- **sentencepiece:** github.com/google/sentencepiece — Google tokenizer (Llama 2).

! Bu dersten tek bir şey alıp gideceksen

Tokenizer, metni ağa girmeden önce öğrenilmiş alt-kelime parçalarına (token) bölen, ağ-dışı bir ön-işleme adımıdır — BPE bu sözlüğü UTF-8 byte'larından en sık çiftleri birleştirerek kurar. Ve LLM'lerin yazım/sayma/dil tuhaflıklarının çoğu tek bir gerçekten doğar: **model karakterleri değil, token'ları görür.**

17 GPT-2'yi (124M) Yeniden Üretmek — Serinin Finali

Ders 7'nin transformer mimarisini ve Ders 9'un tokenizer'ını alıp production ölçeğine taşımak; çalışan bir model ile GPT-2'yi geçen ciddi bir eğitim arasındaki farkın büyük kısmı mimari değil, hız (TF32'den FlashAttention'a) ve ölçek (DDP, gradient accumulation, kaliteli veri) mühendisliğidir — micrograd'dan başlayan yolculuğun finali

i Bölüm bilgisi

- **Karpathy'nin videosu:** YouTube — Let's reproduce GPT-2 (124M) (≈241 dk — serinin en uzunu)
- **Seri:** Neural Networks: Zero to Hero — Ders 10 (**FİNAL**)
- **Hoca:** Andrej Karpathy
- **Kaynak repo:** github.com/karpathy/build-nanogpt · nanoGPT
- **Okuma süresi:** ≈50 dk (en uzun ders)

! Bu derste gerçek 124M eğitimi yok — gerçek-hesaplanabilir sayılar var

GPT-2 124M'i gerçekten eğitmek saatlerce çok-GPU gerektirir (bu render ortamında infeasible). Bu derste **gerçek-hesaplanabilir** sayıları üretiyoruz: parametre sayısı ($124{,}4$M — mimariyi kurup saydık), weight tying tasarrufu ($38{,}6$M), beklenen ilk loss ($\ln 50257 = 10{,}82$), residual init ($0{,}02/\sqrt{2N}$), LR schedule eğrisi. **Hız süreleri** ($1000 \rightarrow 93$**ms) Karpathy'nin A100 ölçümüdür** — figürde açıkça "A100 ölçümü" diye işaretlenir.

17.1 Bu Derste Ne Var?

Seri burada doruğuna ulaşıyor: gerçek bir **GPT-2 (124M)** modelini sıfırdan, OpenAI'nin orijinaliyle uyumlu biçimde yeniden üretiyoruz. Ders 7'de transformer mimarisini, Ders 9'da BPE tokenizer'ı kurmuştuk; bu ders ikisini birleştirip **production ölçeğine** taşıyor — yalnızca "çalışan" değil, **hızlı ve ciddi** bir eğitim hattıyla.

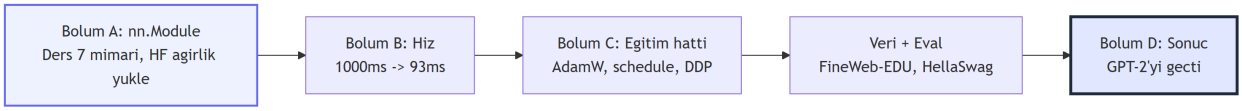
> *"Today we are going to reproduce the GPT-2 model, the 124 million [parameter version]."* — Karpathy, 0:00

Ders dört bölüme ayrılır:

1. **Bölüm A — nn.Module:** GPT-2'yi PyTorch'ta kur, OpenAI ağırlıklarını yükle, örnekle, eğitime başla (weight tying, init).

2. **Bölüm B — Hız:** Aynı modeli hızlandır (1000ms → 93ms): TF32, bfloat16, torch.compile, FlashAttention, "güzel sayılar".
3. **Bölüm C — Eğitim hattı:** AdamW, gradient clipping, LR schedule, weight decay, gradient accumulation, DDP (çok-GPU), FineWeb-EDU, HellaSwag.
4. **Bölüm D — Sonuçlar:** Sabahki eğitim sonuçları, llm.c, ve serinin kapanışı.

Büyük fikir: "çalışan bir model" ile "production-kalite eğitim" arasındaki fark — ve bu farkın büyük kısmı **mühendislik** (hız, kararlılık, ölçek).



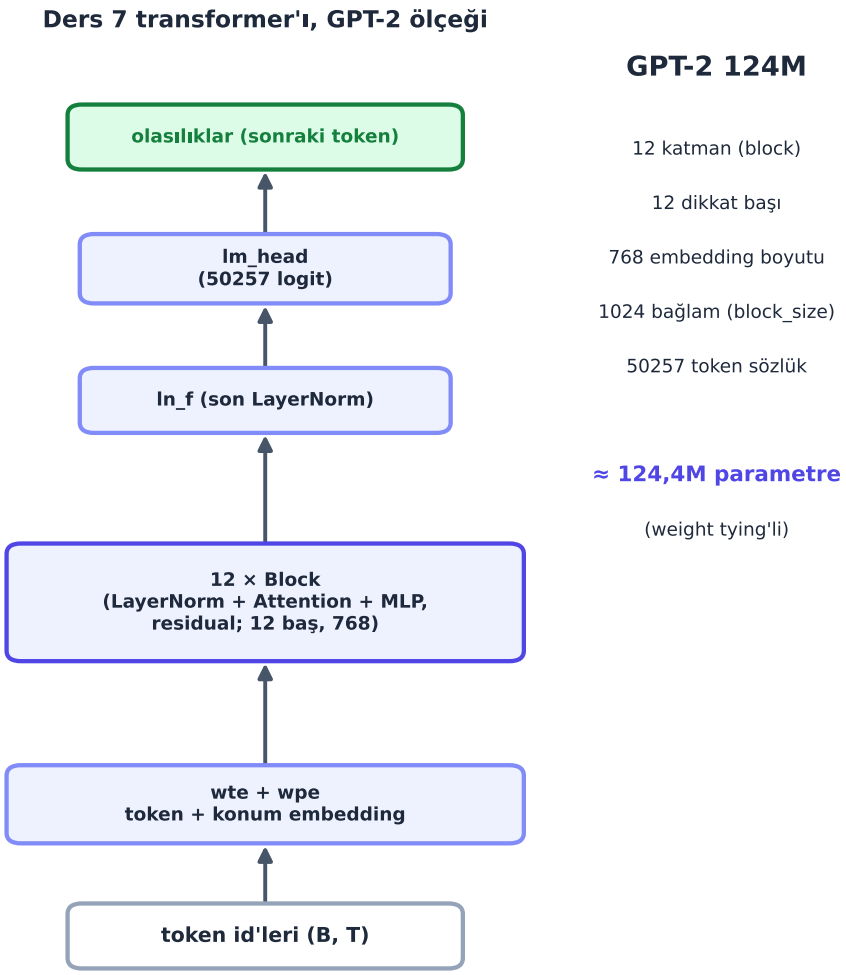
Şekil 17.1: Ders 10'un kavram haritası: GPT-2'yi PyTorch'ta kurarız (Ders 7 mimarisi, GPT-2 isimlendirmesi) ve OpenAI ağırlıklarını yükleriz; sonra aynı modeli hızlandırırız (TF32, bfloat16, torch.compile, FlashAttention); ciddi bir eğitim hattı kurarız (AdamW, gradient clipping, LR schedule, gradient accumulation, DDP); kaliteli veri (FineWeb-EDU) ve değerlendirme (HellaSwag) ile eğitir, sonuçta GPT-2'yi geçeriz. Slate akış + indigo dönüm noktaları (nn.Module ve sonuç).

> Builder Notu — Tüm Seri Burada Birleşir
>
> **Geriye (bütün seri):**
>
> - **Bölüm A = Ders 7.** Tüm attention/FFN/residual/LayerNorm Ders 7'de inşa edildi; bu ders onları GPT-2 ölçeğine taşır (12 katman, 768 boyut, 124M parametre).
> - **Tokenizer = Ders 9.** GPT-2 BPE tokenizer'ı (tiktoken `gpt2`), Ders 9'da kurduğumuz encode/decode hattı.
> - **Pretraining = Ders 8.** Ders 8'in kavramsal anlattığı 1. aşamanın (pretraining) **pratik gerçeği**: gerçek veri, gerçek compute.
> - **Sentez = Ders 1-6.** micrograd'ın autograd'ı, makemore'un dil modelleme çerçevesi — hepsi burada GPT-2'de buluşur.
>
> **Tek cümleyle:** GPT-2'yi yeniden üretmek, Ders 7'nin transformer'ını + Ders 9'un tokenizer'ını alıp production ölçeğine taşımaktır — ve "çalışan model" ile "ciddi eğitim" arasındaki farkın büyük kısmı, mimari değil **hız ve ölçek mühendisliğidir**.

17.2 Giriş ve GPT-2 Checkpoint'ini İnceleme

Hedef: 2019'da OpenAI'nin yayımladığı GPT-2'nin **124 milyon parametreli** (en küçük) versiyonunu yeniden üretmek. Önce orijinal OpenAI checkpoint'ini (HuggingFace üzerinden) inceleriz: model bir sözlük (`state_dict`) olarak gelir — ağırlık tensörleri ve adları. Bu adlar (`wte`, `wpe`, `h.0.attn...`, `ln_f`, `lm_head`) bizim kuracağımız yapıyı belirler.

GPT-2 124M mimarisi: **12 katman**, **12 dikkat başı**, **768 embedding boyutu**, **1024 bağlam uzunluğu**, **50257 token sözlük**. Bu sayılar Ders 7'deki char-GPT'nin büyütülmüş hâli — aynı mimari, gerçek ölçek:



Şekil 17.2: GPT-2 124M mimarisi — Ders 7'nin transformer'ının production ölçeği (GERÇEK parametre sayısı). Token embedding (`wte`) + konum embedding (`wpe`) toplanır, 12 transformer bloğundan (12 baş, 768 boyut) geçer, son LayerNorm (`ln_f`), sonra `lm_head` ile 50257 logit. Mimari Ders 7 ile birebir aynı — yalnızca isimler GPT-2 sözleşmesinde ve ölçek büyük. Toplam 124,4**M parametre** (weight tying'li). Bu sayılar OpenAI checkpoint'iyle uyumlu.

Builder Notu — Model = Adlandırılmış Ağırlık Sözlüğü

Geriye (Ders 7): OpenAI checkpoint'inin ağırlık adları (`wte`, `wpe`, attention, MLP, LayerNorm), Ders 7'de kurduğumuz bileşenlerin birebir adı. Bir model = ağırlık tensörlerinin adlandırılmış bir sözlüğü; mimariyi bu adlardan okuyabilirsin.
İleriye: "Önce referans checkpoint'i incele, sonra ona uyumlu kur" — production'da var olan bir modeli (LLaMA, Mistral) yüklemek/uyarlamak için standart ilk adım.

17.3 Bölüm A: GPT-2 nn.Module İmplementasyonu

GPT-2'yi PyTorch'ta kuruyoruz (Ders 7'nin mimarisini GPT-2 isimlendirmesiyle), OpenAI ağırlıklarını yüklüyoruz, örnekliyoruz, ve eğitime başlıyoruz.

17.3.1 nn.Module İskeleti

Modeli, OpenAI checkpoint'inin isimlendirmesine uyumlu kurarız: bir `GPTConfig` (hiperparametreler) ve bir `GPT(nn.Module)`. Yapı Ders 7'nin aynısı, yalnızca isimler GPT-2 sözleşmesinde.

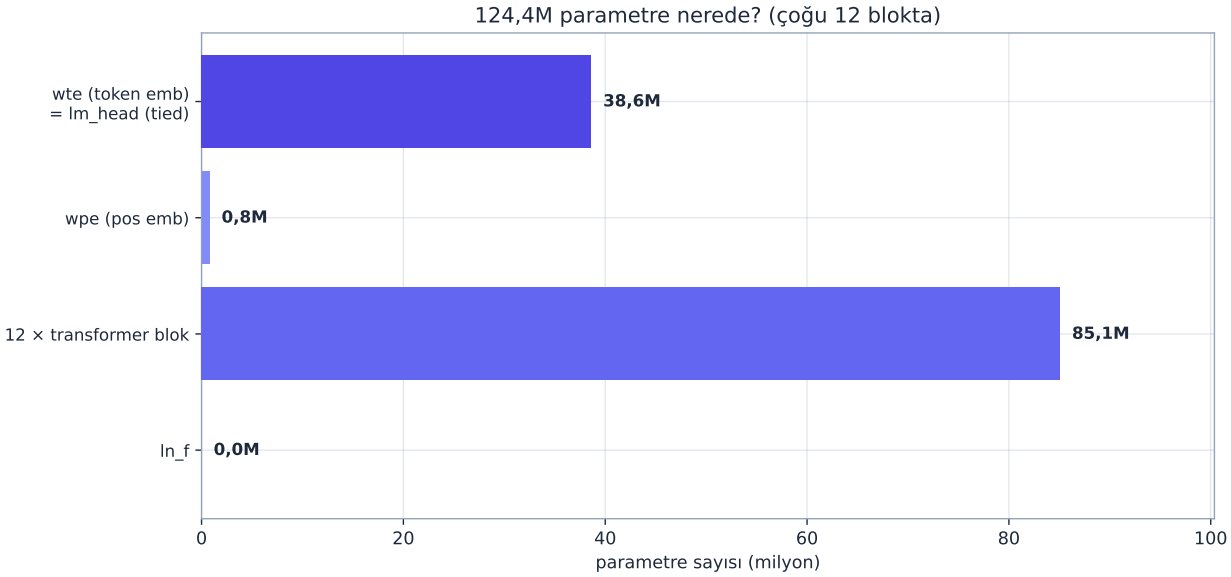
```
@dataclass
class GPTConfig:
    block_size: int = 1024     # baglam uzunlugu
    vocab_size: int = 50257    # BPE sozluk (Ders 9)
    n_layer: int = 12          # transformer blok sayisi
    n_head: int = 12           # dikkat basi
    n_embd: int = 768          # embedding boyutu

class GPT(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.transformer = nn.ModuleDict(dict(
            wte  = nn.Embedding(config.vocab_size, config.n_embd),  # token embedding
            wpe  = nn.Embedding(config.block_size, config.n_embd),  # positional embedding
            h    = nn.ModuleList([Block(config) for _ in range(config.n_layer)]),
            ln_f = nn.LayerNorm(config.n_embd),                     # son LayerNorm
        ))
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
```

`Block` (Ders 7): pre-norm LayerNorm → CausalSelfAttention → residual, sonra LayerNorm → MLP → residual. GPT-2'nin MLP'si **GELU** (tanh yaklaşık biçimi) kullanır — Ders 7'deki ReLU yerine.

```
class MLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.c_fc   = nn.Linear(config.n_embd, 4 * config.n_embd)   # 4x genisleme
```

```
        self.gelu    = nn.GELU(approximate='tanh')                # GPT-2 GELU
        self.c_proj = nn.Linear(4 * config.n_embd, config.n_embd)   # residual'a geri
```

Bu mimariyi kurup parametreleri saydığımızda $124{,}4\mathbf{M}$ çıkar — parametrelerin çoğu 12 blokta:



Şekil 17.3: GPT-2 124M parametrelerinin nerede yaşadığı (GERÇEK, analitik). Parametrelerin çoğu 12 **transformer bloğunda** (≈ 85M) — her blokta attention + MLP ($4\times$). Token embedding `wte` ($\approx 38{,}6$M) ikinci büyük; weight tying ile `lm_head` ile paylaşıldığı için **iki kez sayılmaz**. Konum embedding `wpe` ($\approx 0{,}8$M) ve son LayerNorm küçüktür. Toplam $\approx 124{,}4$M.

17.3.2 HuggingFace Ağırlıklarını Yükleme

Modeli kurduk; şimdi OpenAI'nin eğittiği ağırlıkları yükleriz (`from_pretrained`). Tek incelik: OpenAI bazı ağırlıkları (attention/MLP projeksiyonları) **transpoze** saklamış (eski TensorFlow `Conv1D` mirası); yüklerken `.t()` ile çevirir.

```
@classmethod
def from_pretrained(cls, model_type):
    transposed = ['attn.c_attn.weight', 'attn.c_proj.weight',
                  'mlp.c_fc.weight', 'mlp.c_proj.weight']
    # ... her anahtar icin: transposed ise sd[k].t() kopyala, degilse dogrudan
```

Yükleme başarılıysa, modelimiz OpenAI GPT-2'nin **tam kopyası** — mimarimizin doğruluğunun kanıtı (Ders 1'in "micrograd = PyTorch" karşılaştırmasının GPT-2 ölçeğindeki hâli).

17.3.3 İleri Geçiş, Cross-Entropy ve Tek-Batch Overfit

İleri geçiş Ders 7'nin aynısı: `wte` + `wpe`, bloklar, `ln_f`, sonra `lm_head`. Hedef verilince cross-entropy:

```python
def forward(self, idx, targets=None):
    pos = torch.arange(0, idx.size(1), device=idx.device)
    x = self.transformer.wte(idx) + self.transformer.wpe(pos)   # token + konum
    for block in self.transformer.h:
        x = block(x)
    logits = self.lm_head(self.transformer.ln_f(x))             # (B, T, vocab)
    loss = None
    if targets is not None:
        loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1))
    return logits, loss
```

Rastgele başlatılmış modelde ilk loss $\approx -\ln(1/50257) = \mathbf{10{,}82}$ olmalı (uniform, Ders 4'ün "beklenen başlangıç loss"u). Sonra Ders 3'ten tanıdık sanity-check: tek küçük batch'e **kasten overfit** et, loss ≈ 0'a insin.

> *"[Batch] 4 and 32 right now, just because we're debugging — we just want to have a single batch that's very small, and all of this should now look familiar."* — Karpathy, 52:10

17.3.4 Weight Tying ve Başlatma

İki önemli detay, GPT-2'yi sadık üretmek için. **Weight tying:** token embedding (`wte`) ile dil-modeli başı (`lm_head`) **aynı matrisi paylaşır**.

> *"What's happening here is a common weight tying scheme that actually comes from [the original paper]."* — Karpathy, 1:07:58

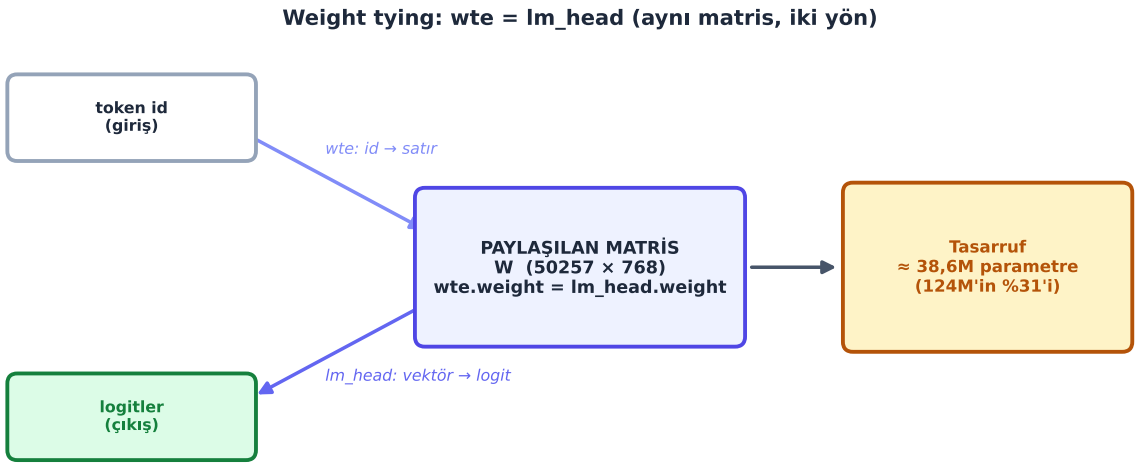
```python
self.transformer.wte.weight = self.lm_head.weight   # ayni tensor (Ders 2: one-hot@W = satir secim
```

Başlatma: GPT-2 ağırlıkları std$=0{,}02$ normal ile başlatılır. Kritik ek: **residual projeksiyonları** katman sayısıyla ölçeklenir — residual akışına (Ders 7) her katman katkı eklediği için varyans katlanarak büyümesin:

$$\text{std} = 0{,}02 \times \frac{1}{\sqrt{2N}}$$

Burada N = katman sayısı; 2 ise her blokta 2 residual ekleme (attention + MLP). $N = 12$ için bu $\approx 0{,}00408$ (Ders 4 Kaiming'in residual'a özel hâli).

> **Builder Notu — Weight Tying = Ders 2; Init Scaling = Ders 4**
>
> **Geriye (Ders 2, 4, 7):** Weight tying, Ders 2'nin "one-hot $\times W$ = satır seçimi" gözleminden gelir (giriş ve çıkış aynı matris olabilir). Residual init scaling = Ders 4 varyans kontrolü, Ders 7 residual akışına uygulanmış ($1/\sqrt{2N}$ ile her katmanın katkısı dengelenir).
> **İleriye:** `from_pretrained`, tüm modern iş akışının temeli: hazır base model yükle, üstüne kendi ince ayarını (Ders 8 SFT/LoRA) ekle. Bu küçük detaylar (tying, init), "sadık reprodüksiyon" ile "yaklaşık" arasındaki farktır.



Şekil 17.4: Weight tying: token embedding (`wte`, giriş — id'den vektöre) ile dil-modeli başı (`lm_head`, çıkış — vektörden logitlere) **aynı ağırlık matrisini paylaşır**. İkisi de aynı "token ↔ vektör" eşlemesini yapar (biri ileri, biri ters — Ders 2: one-hot $\times W$ = satır seçimi). Paylaşmak $50257 \times 768 \approx 38{,}6$M parametre tasarrufu sağlar (124M'lik modelin $\%31$'i!) ve genellemeyi iyileştirir.

17.4 Bölüm B: Hız Optimizasyonları

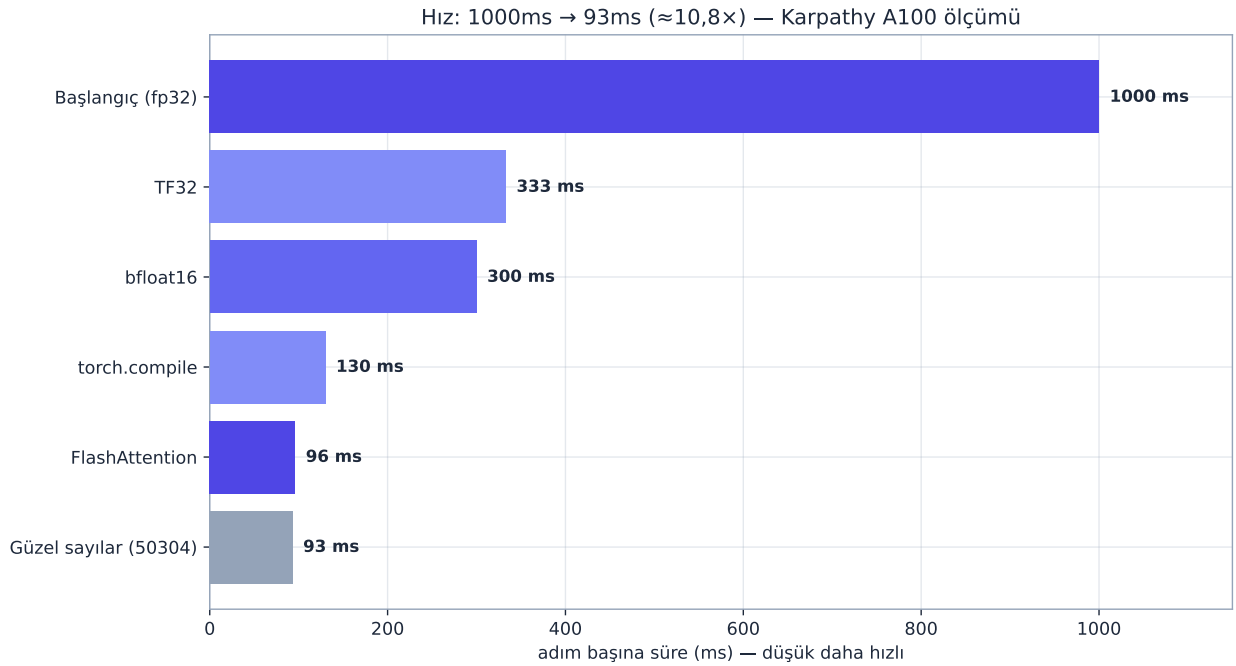
GPT-2'yi kurduk, eğitebiliyoruz — ama **yavaş** (adım başına ≈ 1000ms). Şimdi aynı modeli, **sonucu değiştirmeden** hızlandırıyoruz. Temel sorun:

> *"The workloads for training are memory bound — which means the tensor cores that do all these extremely fast multiplications [sit idle, waiting for data]."* — Karpathy, 1:27:29

Yani GPU çoğu zaman "bellek-bağlı" (memory bound): hesap değil, **veri taşıma** darboğaz. Hızlandırma adımları (her biri veriyi azaltır veya turları birleştirir; **süreler Karpathy'nin A100 ölçümü, illüstratif**):

- **TF32** (`set_float32_matmul_precision('high')`): Tensor Cores, $\approx 3\times$ matmul hızı $\rightarrow$ 333ms.
- **bfloat16** (`torch.autocast`): geniş üs aralığı (scaler'sız; fp16 scaler ister) $\rightarrow$ 300ms.
- **torch.compile**: kernel füzyonu + Python overhead kaldırma (HBM gidip-gelmeyi azaltır) $\rightarrow$ 130ms.
- **FlashAttention** (`F.scaled_dot_product_attention`): $T \times T$ matrisini HBM'de **hiç oluşturmaz** (online softmax, $O(T)$ bellek) $\rightarrow$ 96ms.
- **Güzel sayılar**: vocab $50257 \rightarrow 50304$ (128'in katı; GPU hizalaması) $\rightarrow$ 93ms.

```
torch.set_float32_matmul_precision('high')                    # TF32
with torch.autocast(device_type='cuda', dtype=torch.bfloat16): # bf16
    logits, loss = model(x, y)
model = torch.compile(model)                                   # kernel fuzyonu
y = F.scaled_dot_product_attention(q, k, v, is_causal=True)    # FlashAttention
```



Şekil 17.5: Aynı modeli hızlandırma: adım başına süre 1000ms → 93ms ($\approx$ 10,8×). Sonuç değişmez, yalnızca donanım daha verimli kullanılır: **TF32** (Tensor Cores), **bfloat16** (autocast), **torch.compile** (kernel füzyonu), **FlashAttention** ($T \times T$ matrisini bellekte oluşturmaz), **güzel sayılar** (vocab 50304, 128'in katı). Çoğu kazanç "memory bound" darboğazını (veri taşımayı) azaltmaktan gelir — modern ML mühendisliğinin kalbi. **NOT: bu süreler Karpathy'nin A100 ölçümüdür** (bu ortamda üretilemez, illüstratif).

Builder Notu — Hız = Ders 5 Füzyon Ruhu + Donanım

Geriye (Ders 5, 7): torch.compile/FlashAttention, Ders 5'teki "atomik graf yerine füzyonlu hesap" fikrinin donanım hâli — ara değerleri materyalize etme, tek geçişte hesapla. FlashAttention, Ders 7'nin softmax$(QK^\top/\sqrt{d})V$ formülünü **aynen** yapar ama dev ara matrisi saklamaz.
İleriye: "Memory bound vs compute bound", GPU performansının temel kavramı. Mixed precision (bf16) + torch.compile + FlashAttention, tüm modern büyük-model eğitiminin standardı; FP8, vLLM, TensorRT bir sonraki katman.

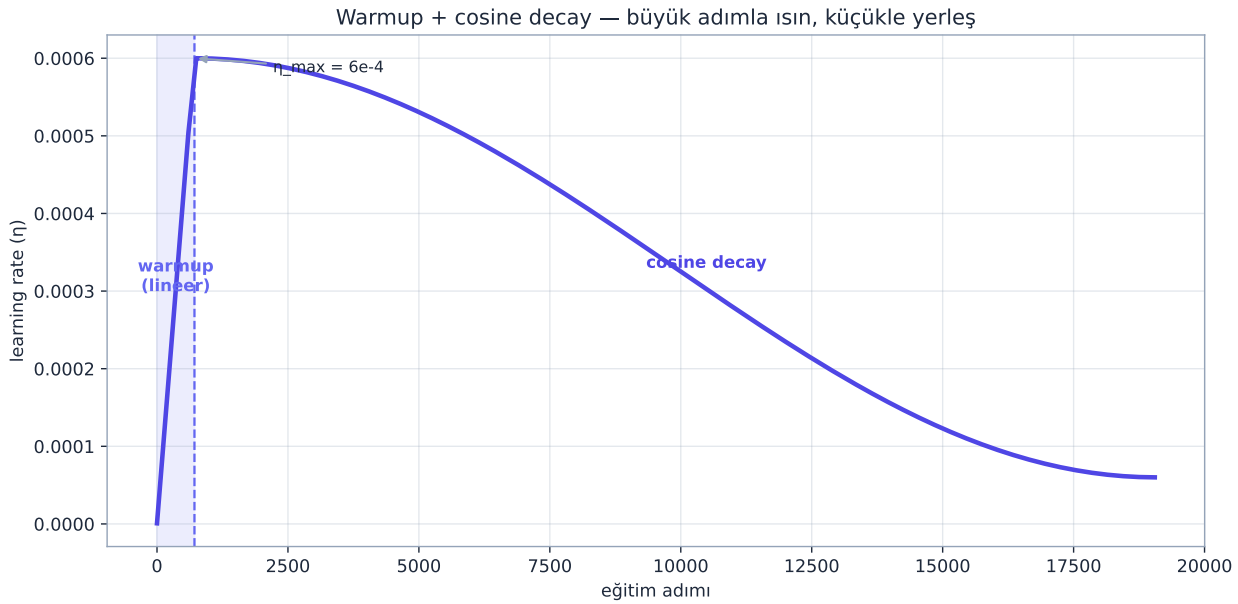
17.5 Bölüm C: Hiperparametreler, Eğitim Hattı ve DDP

GPT-2'yi kurduk (A) ve hızlandırdık (B). Şimdi **ciddi bir eğitim hattı** — GPT-3 makalesinin hiperparametreleri (GPT-2 belirsizdir).

AdamW + gradient clipping: betas $(0{,}9, 0{,}95)$, eps 10^{-8}; tüm gradyanların küresel normunu $1{,}0$'a kırp (kötü bir batch'in modeli sarsmasını önle).

```
optimizer = torch.optim.AdamW(model.parameters(), lr=6e-4, betas=(0.9, 0.95), eps=1e-8)
norm = torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)   # gradient clipping
```

Learning rate schedule: başta **warmup** (lr'yi 0'dan yükselt), sonra **cosine decay** (minimuma indir):



Şekil 17.6: Learning rate schedule: **warmup** (lineer, lr'yi 0'dan yavaşça yükselt — erken kararsızlığı önle) + **cosine decay** (yavaşça minimuma indir — sonda ince ayar). GERÇEK analitik eğri (Karpathy/GPT-3 config: 715 adım warmup, 19073 adım toplam, $\eta_{\max} = 6 \times 10^{-4}$, $\eta_{\min} = 6 \times 10^{-5}$). Sezgi: büyük adımlarla ısın, sonra küçük adımlarla yerleş — Ders 3'teki "lr decay" fikrinin production hâli.

Weight decay (0,1): yalnızca 2 boyutlu ağırlıklara (matrisler, embedding'ler), bias/LayerNorm'a değil (Ders 2 L2 regularization). **Gradient accumulation:** GPT-3 batch başına $\approx$ 0,5M token kullanır (tek GPU'ya sığmaz); büyük batch'i micro-batch'lere böl, gradyanları **biriktir** (Ders 1 +=!), loss'u `grad_accum_steps`'e böl (ortalamayı koru):

```
loss_accum = 0.0
for micro in range(grad_accum_steps):
    x, y = loader.next_batch()
    with torch.autocast(device_type='cuda', dtype=torch.bfloat16):
        logits, loss = model(x, y)
    loss = loss / grad_accum_steps      # ortalamayi koru!
    loss.backward()                      # gradyanlar BIRIKIR (Ders 1)
optimizer.step()                         # tum micro-batch'ler sonrasi tek adim
```

DDP (DistributedDataParallel): modeli her GPU'ya kopyala, her GPU farklı veri dilimini işlesin, gradyanları **ortala** (all-reduce). **Veri:** FineWeb-EDU (kalite-filtreli eğitsel web, sample-10BT). **Değerlendirme:** validation loss (Ders 3 train/dev/test) + **HellaSwag** benchmark.

> Builder Notu — Gradient Accumulation = Ders 1 += ; DDP = Stat 110 Ortalama
>
> **Geriye (Ders 1-4):** Eğitim döngüsü micrograd'ın aynısı; gradyan birikmesi = Ders 1 += (zero_grad'ı yalnız tam adımdan önce); `/grad_accum_steps` = ortalamayı koruma; AdamW + clip = Ders 4; LR decay = Ders 3 + Calculus (cosine); DDP all-reduce = Stat 110 örneklem ortalaması (her GPU bir alt-örnek). Weight decay = Ders 2 L2.
> **İleriye:** Warmup + cosine, gradient clipping, seçici weight decay, gradient accumulation, DDP — neredeyse tüm modern LLM eğitiminin standardı. Daha büyük modeller için FSDP (model paralelliği), tensor/pipeline parallelism gelir.

17.6 Bölüm D: Sonuçlar ve Kapanış

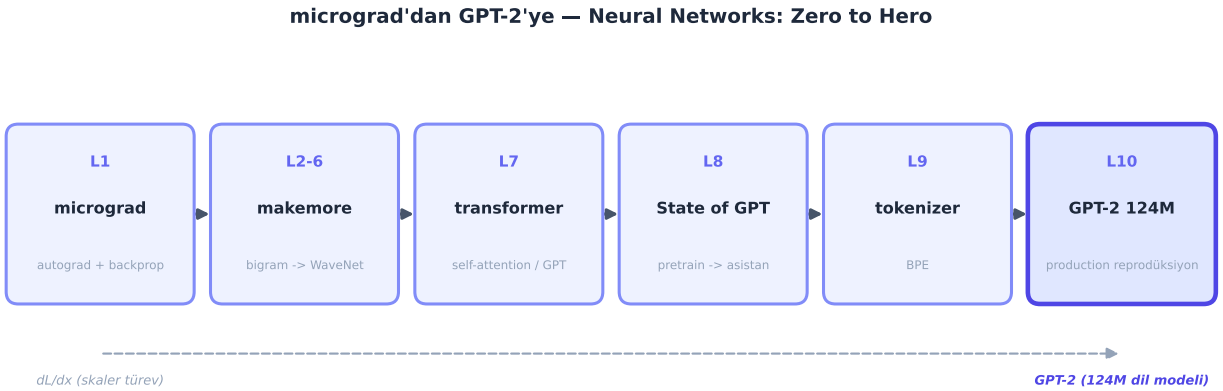
Karpathy eğitimi gece boyunca (bir bulut GPU kümesinde, örn. 8$\times$ A100) koşturur ve sabah sonuçlara bakar. Sonuç çarpıcı: model yalnızca OpenAI'nin **GPT-2 124M'ini geçmekle** kalmaz, bazı ölçülerde **GPT-3 124M'e** bile yaklaşır — daha az token görmesine rağmen. Sebep: daha kaliteli veri (FineWeb-EDU) + iyi eğitim hattı (Bölüm B+C).

Yani 2019'da OpenAI'nin (kapalı) ürettiği bir modeli, sıfırdan, açık araçlarla, birkaç yüz dolarlık compute ile yeniden ürettik — ve geçtik. Karpathy ayrıca **llm.c**'yi tanıtır: aynı GPT-2 eğitimini PyTorch yerine **ham C/CUDA** ile yapan kod (Ders 1'in "her şey efficiency" + Ders 5'in "elle backward" felsefesinin en uç hâli).

> Builder Notu — Açık Reprodüksiyon Alanı Demokratikleştirir
>
> **İleriye:** "Daha az token ama daha iyi veri/hat ile eski modeli geçmek", scaling laws + veri kalitesi araştırmasının pratik kanıtı. `build-nanogpt` (commit-by-commit) + `nanoGPT` (production-temiz), kendi GPT'ni eğitmenin başlangıç noktası — artık tüm hat senin elinde: tokenizer $\rightarrow$ mimari $\rightarrow$ eğitim $\rightarrow$ değerlendirme.

17.7 Seri Kapanışı

Bu, **Neural Networks: Zero to Hero** serisinin son dersiydi. Yolculuk: bir skaler türevden (Ders 1 micrograd) gerçek bir dil modeline (Ders 10 GPT-2 124M) — her adım bir öncekinin üstüne kuruldu.



Şekil 17.7: **Neural Networks: Zero to Hero — tüm serinin yolculuğu.** Bir skaler türevden (Ders 1 micrograd, autograd + backprop) gerçek bir dil modeline (Ders 10 GPT-2 124M) — her ders bir öncekinin üstüne kuruldu: makemore (Ders 2-6: bigram → MLP → BatchNorm → manuel backprop → WaveNet) dil modellemenin temellerini, transformer (Ders 7) self-attention'ı, State of GPT (Ders 8) pretrain→asistan hattını, tokenizer (Ders 9) BPE'yi verdi. Hepsi Ders 10'da GPT-2'de birleşir. Karpathy'nin tek mesajı: sinir ağları sihir değil — anlaşılabilir, kurulabilir, gerisi verimliliktir.

Calculus zincir kuralı (backprop), 18.06 matris çarpımı (her katman), Stat 110 olasılık (cross-entropy, varyans) — üç matematik kursu + 10 ders, tek bir GPT-2'de birleşti. Karpathy'nin tek mesajı baştan sona aynıydı: **sinir ağları sihir değil; anlaşılabilir, kurulabilir, ve "gerisi sadece verimlilik"tir.**

> *"Eternal glory goes to anyone who can get rid of [tokenization]."* — Karpathy, Ders 9, 2:10:21 (serinin esprili ruhu)

Builder Notu — Builder Yolu Açık

İleriye: Buradan: kendi modelini eğit, ince ayarla (Ders 8 SFT/LoRA), değerlendir (HellaSwag), modern teknikleri (FSDP, RLHF/DPO, reasoning) ekle, veya araştırmaya geç. micrograd'dan GPT-2'ye tüm hat artık senin.

17.8 Bu Dersin Özeti

1. **GPT-2 124M reprodüksiyonu**: Ders 7 transformer + Ders 9 tokenizer + Ders 8 pipeline, production ölçeğinde (12 katman, 768, 50257, $\approx 124{,}4$M parametre).

2. **Bölüm A (nn.Module)**: GPT-2'yi OpenAI isimlendirmesiyle kur, HF ağırlıklarını yükle (transpoze tuzağı), (B,T)+cross-entropy ile eğit, tek-batch overfit; **weight tying** (wte=lm_head, $\approx 38{,}6$M tasarruf) + **residual init scaling** ($0{,}02/\sqrt{2N} \approx 0{,}00408$).
3. **Bölüm B (hız,** $1000 \rightarrow 93$**ms)**: memory-bound; TF32 → bf16 → torch.compile → FlashAttention → güzel sayılar (50304). $\approx 10{,}8\times$ (A100 ölçümü).
4. **Bölüm C (eğitim hattı)**: GPT-3 hiperparametreleri, AdamW + gradient clipping, warmup+cosine LR schedule, seçici weight decay, **gradient accumulation** (Ders 1 +=), **DDP** (all-reduce), FineWeb-EDU, validation + HellaSwag.
5. **Bölüm D (sonuçlar)**: model GPT-2 124M'i geçti, GPT-3 124M'e yaklaştı (kaliteli veri + iyi hat); llm.c (ham C/CUDA); build-nanogpt.
6. Büyük ders: "çalışan model" ile "ciddi eğitim" arasındaki fark çoğunlukla **mimari değil, hız ve ölçek mühendisliğidir**.
7. Tüm seri burada birleşir: micrograd → makemore → transformer (Ders 7) → State of GPT (Ders 8) → tokenizer (Ders 9) → **GPT-2 (Ders 10)**.

! Tek Bir Cümle

GPT-2'yi yeniden üretmek, Ders 7'nin transformer mimarisini ve Ders 9'un tokenizer'ını alıp production ölçeğine taşımaktır; ve "çalışan bir model" ile "GPT-2'yi geçen ciddi bir eğitim" arasındaki farkın büyük kısmı mimari değil, **hız (**$1000 \rightarrow 93$**ms) ve ölçek (DDP, gradient accumulation, kaliteli veri) mühendisliğidir** — micrograd'dan başlayan yolculuğun finali.

17.9 Kontrol Soruları

i Soru 1: Weight tying nedir (wte = lm_head)? Neden yapılır, ne kazandırır? Ders 2 ile bağla.

Cevap: Weight tying, token embedding tablosu (`wte`, giriş: id → vektör) ile dil-modeli başının (`lm_head`, çıkış: vektör → logitler) **aynı ağırlık matrisini paylaşmasıdır**. Neden: ikisi de aynı "token $\leftrightarrow$ vektör" eşlemesini yapar — biri ileri (Ders 2: id seçer W'nin satırını), biri ters. Kazanç: **(1)** $\approx 38{,}6$M parametre tasarrufu (vocab × n_embd $= 50257 \times 768$ — 124M'lik modelin $\%31$'i!); **(2)** daha iyi genelleme. Ders 2'deki "one-hot $\times W$ = W'nin satırını seçer" gözleminin doğrudan sonucu: aynı W, hem satır-seçimi (giriş) hem logit-üretimi (çıkış) için kullanılabilir.

i Soru 2: Hız yolculuğunda (1000ms→93ms) en büyük kazançlar neden mümkün oldu? 'Memory bound' ne demek?

Cevap: Memory bound = darboğaz hesap değil, **veri taşıma**. GPU'nun Tensor Core'ları çok hızlı, ama veriyi yavaş HBM bellekten yeterince hızlı çekemeyince **boşta beklerler**. Bu yüzden en büyük kazançlar, veri taşımayı azaltan optimizasyonlardan geldi: **TF32/bf16** (daha az byte → $1000 \rightarrow 300$ms), **torch.compile** (kernel füzyonu, HBM gidip-gelmeyi azalt → 130ms), **FlashAttention** ($T \times T$ matrisi HBM'de hiç oluşturma → 96ms). Hepsi aynı matematiği yapar (sonuç değişmez), yalnızca donanımı daha verimli kullanır. Modern hızlandırma çoğunlukla "daha az hesap" değil, "belleği daha az meşgul et".

Soru 3: Gradient accumulation'da loss neden grad_accum_steps'e bölünür? Bölünmezse ne olur?

Cevap: Cross-entropy, bir batch üzerinde **ortalama** alır (toplam değil). Büyük batch'i micro-batch'lere bölüp gradyanları topladığımızda, her micro-batch kendi içinde ortalama hesaplar; bunları toplayınca efektif olarak **toplam** elde ederiz, ortalama değil — yani gradyan grad_accum_steps kat büyük olur. `loss = loss / grad_accum_steps` ile her micro-batch'in katkısını ölçekleyerek, biriken gradyanın gerçek büyük-batch ortalamasına eşit olmasını sağlarız. Bölünmezse: efektif öğrenme oranı grad_accum_steps kat büyür (gradyan şişer), eğitim kararsızlaşır/ıraksar. (Gradyanların birikmesi = Ders 1'in += kuralı; bölme = ortalamayı koruma.)

Soru 4: (Builder) Bu ders, Ders 7'den (Sıfırdan GPT) gerçekte ne ekliyor? Mimari mi değişti?

Cevap: Mimari neredeyse aynı — Ders 7'nin transformer'ı (attention, FFN, residual, LayerNorm), yalnızca GPT-2 isimlendirmesi + büyük boyut (12 katman, 768, 124M) + GELU. Eklenen asıl şey **mühendislik**: **(1) Sadakat** (HF ağırlık yükleme, weight tying, doğru init); **(2) Hız** (TF32, bf16, torch.compile, FlashAttention — $\approx 10\times$); **(3) Ölçek/kararlılık** (LR schedule, gradient clipping, weight decay, gradient accumulation, DDP); **(4) Veri/değerlendirme** (FineWeb-EDU, HellaSwag). Yani Ders 7 "model nasıl çalışır"ı, Ders 10 "production'da nasıl ciddi eğitilir"i öğretir. "Çalışan model"den "GPT-2'yi geçen model"e fark, mimaride değil, bu mühendislik katmanlarındadır — serinin en büyük dersi.

17.10 Egzersizler

Egzersiz 1 (GPT-2'yi yükle ve çalıştır). `from_pretrained('gpt2')` ile OpenAI GPT-2'yi yükle (HuggingFace). Bir önek ("Hello, I'm a language model,") ver, top-k örneklemeyle metin üret. Tutarlı çıktı = mimarin doğru. tiktoken `gpt2` ile encode/decode et (Ders 9).

Egzersiz 2 (Sıfırdan eğit, overfit). Rastgele başlatılmış bir `GPT(GPTConfig())` kur. İlk loss'un $\approx \ln(50257) \approx 10{,}8$ olduğunu doğrula. Tek küçük batch'e overfit et (50 adım), loss'un ≈ 0'a indiğini gözlemle.

Egzersiz 3 (Hız ölçümü). Adım başına süreyi (ms) ölç (`torch.cuda.synchronize()` + zaman). Sırayla ekle: TF32, `autocast(bfloat16)`, `torch.compile`, `F.scaled_dot_product_attention` (FlashAttention). Her birinin hızlanmasını ölç.

Egzersiz 4 (Eğitim hattı). LR schedule (warmup + cosine), gradient clipping (norm 1,0), seçici weight decay, gradient accumulation ekle. FineWeb-EDU sample-10BT'nin bir parçasında eğit, validation loss'u izle. (Çok GPU varsa DDP ile dene.)

Egzersiz 5 (Seri finali — ne inşa edeceksin?). micrograd'dan (Ders 1) GPT-2'ye (Ders 10) tüm yolculuğu tamamladın. Şimdi kendi projen: (a) Kendi veri setinde (şiir, kod, kendi yazıların) bir nanoGPT eğit; (b) Bir base model'i kendi görevin için ince ayarla (Ders 8: SFT/LoRA); (c) llm.c'yi okuyup en alt seviyeyi gör. Artık tüm hat senin: tokenizer → mimari → eğitim → değerlendirme. **Ne inşa edeceksin?**

17.11 Sonraki Adımlar

Bu, serinin son dersiydi — "sonraki ders" yok, ama **sonraki proje** var. Karpathy'nin önerdiği yol:

- **build-nanogpt** (commit-by-commit) ve **nanoGPT** (production-temiz) ile kendi GPT'ni eğit.
- Bir base model'i kendi göreviniz için ince ayarla (Ders 8: SFT/LoRA).
- **llm.c**'yi okuyup en alt seviyeyi (ham C/CUDA) gör.
- Modern teknikleri ekle: FSDP, RLHF/DPO (Ders 8), reasoning (Ders 8 CoT'nin gelişmişi).

Seriden Sonra

- Üç matematik temelini (Calculus zincir kuralı, 18.06 matris çarpımı, Stat 110 olasılık) ve 10 dersi bir araya getirdin.
- "Sinir ağları sihir değil" — artık her katmanı sıfırdan kurabilir, production'a taşıyabilirsin.
- Builder yolu açık: kendi modelini eğit, değerlendir, paylaş.

17.12 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Karpathy'de
GPT-2 124M nn.Module	Ders 7 transformer, GPT-2 isimlendirmesi (wte/wpe/h/ln_f/lm_head); 12 katman, 768, 50257	13m49
HF ağırlık yükleme	`from_pretrained`; state_dict kopyala, bazı tensörleri transpoze et (TF Conv1D)	28m13
Weight tying	wte = lm_head (aynı matris); $\approx 38{,}6$M tasarruf + genelleme	1h06m
Residual init scaling	std $= 0{,}02/\sqrt{2N} \approx 0{,}00408$; residual akış varyansını kontrol	1h13m
TF32 (Tensor Cores)	`set_float32_matmul_precision('high')`; $\approx 3\times$ matmul (1000 $\rightarrow$ 333ms)	1h28m
bfloat16 / autocast	Geniş üs aralığı (scaler'sız); fp16 scaler ister (300ms)	1h39m
torch.compile	Kernel füzyonu + Python overhead kaldırma (130ms)	1h48m
FlashAttention	$T \times T$ matrisi HBM'de oluşturma; online softmax, $O(T)$ bellek (96ms)	2h00m
Güzel sayılar	vocab 50257 $\rightarrow$ 50304 (128'in katı); GPU hizalaması (93ms)	2h06m

Kavram	Tanım	Karpathy'de
AdamW + gradient clip	betas $(0{,}9, 0{,}95)$; küresel grad normu $1{,}0$'a kırp	2h15m
LR schedule	warmup (lineer) + cosine decay (min $\%10$)	2h21m
Gradient accumulation	Micro-batch'leri biriktir (Ders 1 +=), loss/accum böl; büyük batch taklidi	2h34m
DDP (çok-GPU)	Modeli kopyala, gradyanları ortala (all-reduce); torchrun	2h46m
FineWeb-EDU + HellaSwag	Kaliteli eğitsel veri + sağduyu benchmark'ı; val loss + benchmark izle	3h10m

17.13 ML Builder Bağlantıları

9 köprü — GPT-2 Reprodüksiyon

1. **GPT-2 nn.Module** → Ders 7 transformer'ın production ölçeği. İleriye: GPT-3/4, LLaMA (aynı iskelet, daha büyük).
2. **HF ağırlık yükleme** → Ders 9 model cerrahisi. İleriye: base model yükle + ince ayar (Ders 8).
3. **Weight tying** → Ders 2 "one-hot $\times W$ = satır seçimi". İleriye: parametre verimliliği.
4. **Residual init scaling** → Ders 4 Kaiming + Ders 7 residual akışı. İleriye: derin model kararlılığı.
5. **Mixed precision + torch.compile + FlashAttention** → Ders 5 füzyon ruhu + donanım. İleriye: FP8, vLLM, TensorRT.
6. **AdamW + grad clip + LR schedule** → Ders 1 optimizer + Ders 3-4 lr/clip. İleriye: büyük-model eğitiminin standardı.
7. **Gradient accumulation** → Ders 1 += + Ders 3 minibatch. İleriye: büyük efektif batch.
8. **DDP** → Ders 1 SGD + Stat 110 örneklem ortalaması. İleriye: FSDP, tensor/pipeline parallelism.
9. **FineWeb-EDU + HellaSwag** → Ders 3 train/dev/test + Ders 8 veri kalitesi. İleriye: veri pipeline'ı + lm-eval-harness.

17.14 Karpathy'nin Önerdiği Kaynaklar

Karpathy'nin bu ders için verdiği kaynaklar:

- **build-nanogpt:** github.com/karpathy/build-nanogpt — dersin commit-by-commit kodu.
- **nanoGPT:** github.com/karpathy/nanoGPT — production-temiz versiyon.
- **llm.c:** github.com/karpathy/llm.c — aynı eğitim, ham C/CUDA.
- **Attention is All You Need:** arxiv 1706.03762; **GPT-3 makalesi:** arxiv 2005.14165.

! Bu dersten — ve tüm seriden — tek bir şey alıp gideceksen

GPT-2'yi yeniden üretmek, Ders 7'nin transformer'ını ve Ders 9'un tokenizer'ını alıp production ölçeğine taşımaktır — ve "çalışan model" ile "GPT-2'yi geçen ciddi eğitim" arasındaki farkın büyük kısmı mimari değil, **hız (**1000**ms**$\rightarrow$ 93**ms) ve ölçek (DDP, gradient accumulation, kaliteli veri) mühendisliğidir.** micrograd'dan başlayan yolculuk burada, gerçek bir GPT-2'de tamamlanır: **sinir ağları sihir değil — anlaşılabilir, kurulabilir, ve gerisi sadece verimliliktir.**