Why Deep Learning?

Lecture 1 · ES 667: Deep Learning

Prof. Nipun Batra
IIT Gandhinagar · Aug 2026

READING · PRINCE UDL · CH 1 · CH 3

Lecture 1

This lecture mirrors UDL **Ch 1** (introduction) and **Ch 3** (shallow networks). Read both — slides go brisk on the parts the book covers well, deep on what it doesn't.

PART 1

The big picture

What is deep learning, and why now?

A question to open the semester

You already built classifiers in ES 654 — logistic regression, SVMs, decision trees.

Q. Point any of them at a raw 224×224 colour photo.

POP QUIZ

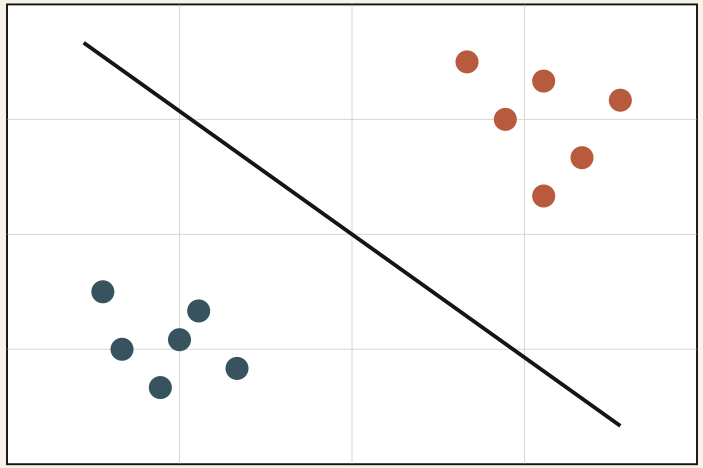
Input dimension: $224 \times 224 \times 3 = 150{,}528$ features.

Think before the next slide: *why* would that fail?

Linear · works for separable data, fails for curved data

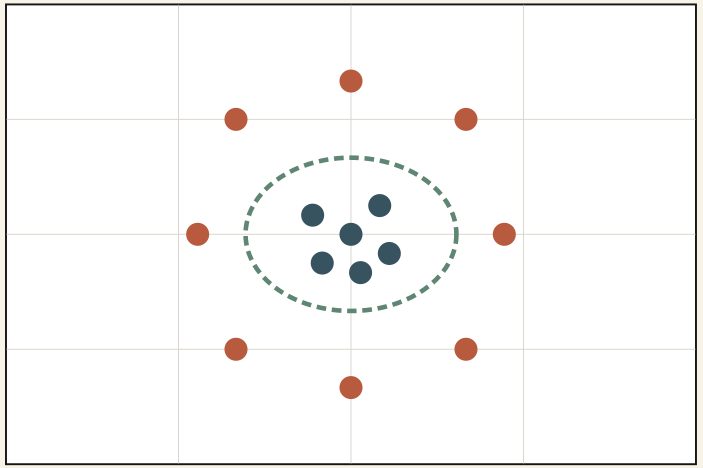
A linear classifier can split linearly-separable data · not arbitrary patterns

Linearly separable



A line separates the two classes · linear classifier ✓

Non-linearly separable



No straight line works · we need a curved boundary · non-linear net

Real images live in 150,000-dimensional space and look like the right panel · we need depth + nonlinearities.

Why raw pixels break a linear classifier

Why a linear classifier over raw pixels fails

A 1-PIXEL SHIFT LOOKS LIKE A COMPLETELY DIFFERENT INPUT

IMAGE A

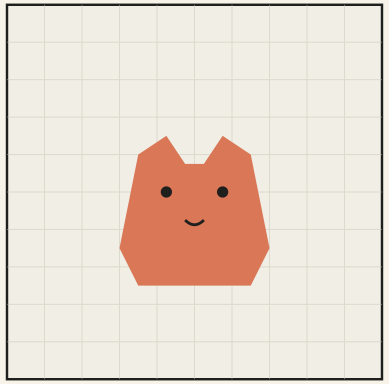
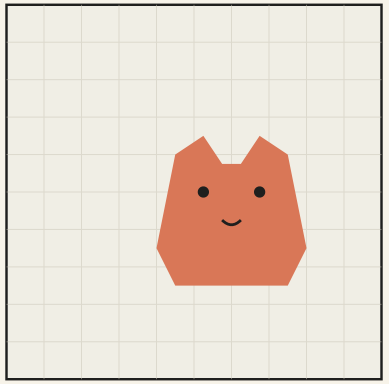
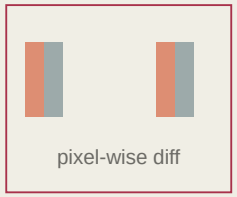


IMAGE B · SAME CAT, SHIFTED 1 PIXEL



LINEAR CLASSIFIER SEES

$\|x_A - x_B\| \approx$ LARGE



a linear SVM treats them as unrelated inputs

*A classifier over raw pixels has **no spatial prior** — it must relearn translation from scratch for every class.*

Pop quiz · which model would *you* pick?

A friend asks · "I have $50{,}000$ raw 224×224 photos labelled as cat/dog. Which model do you reach for?"

POP QUIZ

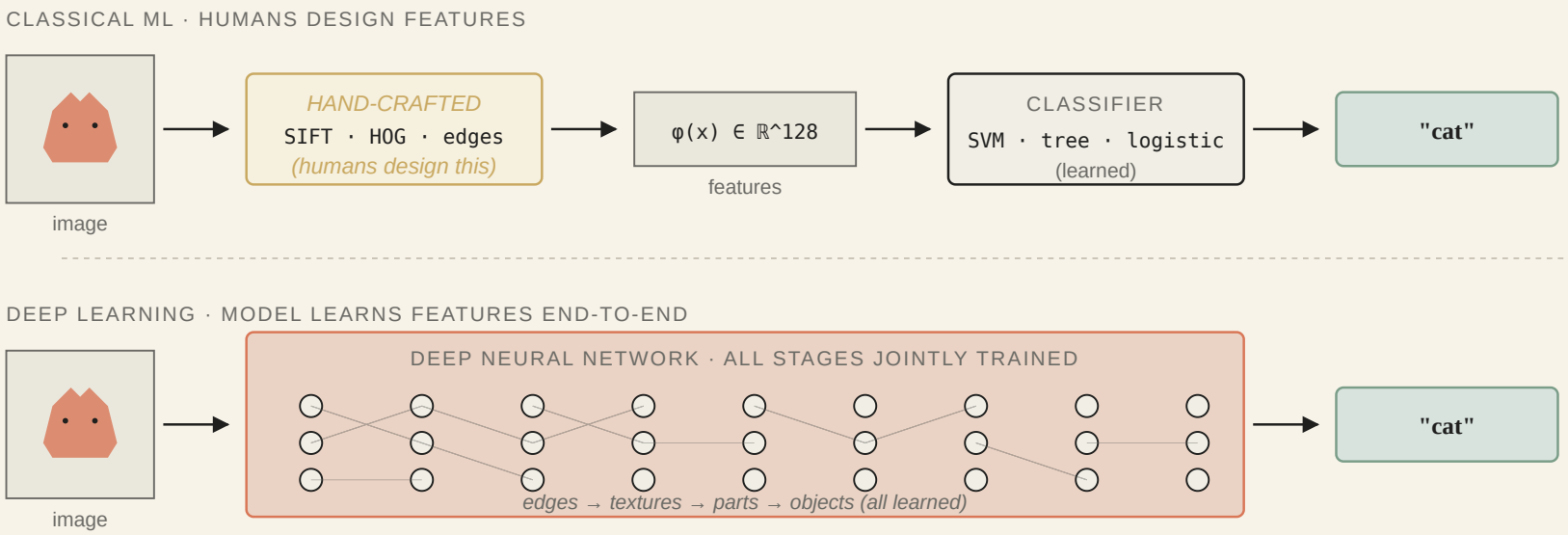
(a) Logistic regression on raw pixel intensities.
(b) Hand-engineer SIFT/HOG features, then SVM.
(c) Train a CNN end-to-end.

Stop and decide before the next slide. *Why* did you pick what you picked?

There is no trick · we want you to feel where each option helps and hurts. The next slides explain why **the right answer changed in 2012** — and why this whole course exists.

Classical ML vs deep learning

Classical ML vs deep learning — same input, different pipelines



*The **entire stack** — feature extractor and classifier — is trained together with one loss and one optimizer.*

Deep learning, in one sentence

INTUITION

Deep learning = representation learning with differentiable, composable modules, trained end-to-end by gradient descent.

Every word matters. We will unpack all of them this semester.

What representation learning buys you

Classical ML:

$$\text{raw data} \rightarrow \text{human-designed features} \rightarrow \text{classifier}$$

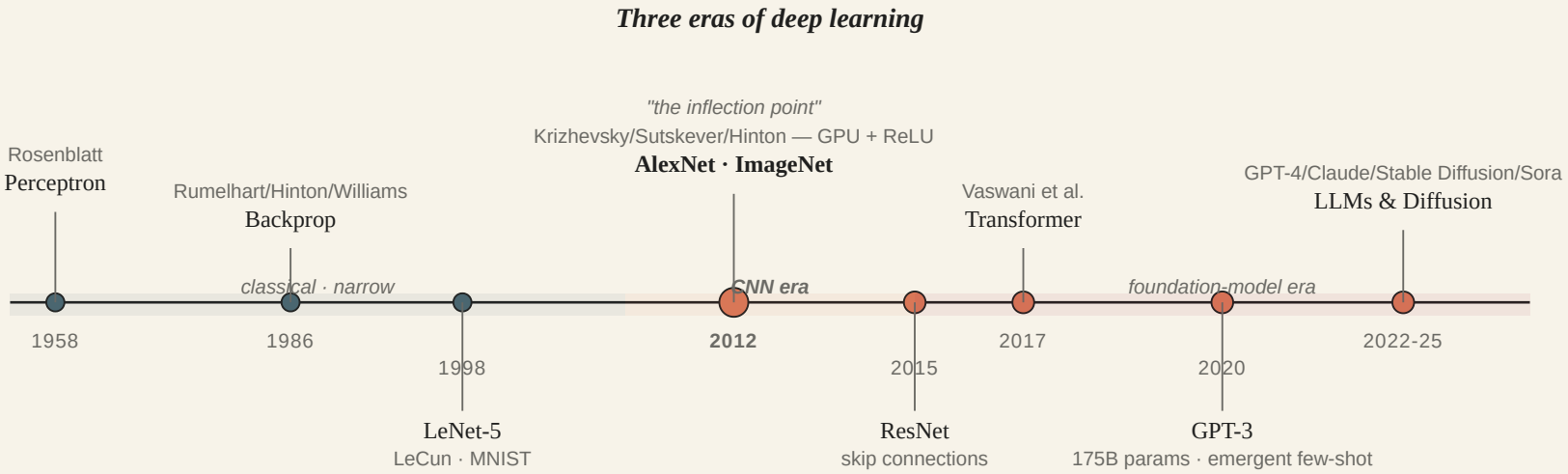
Deep learning:

$$\text{raw data} \rightarrow \text{learned layers} \rightarrow \text{representation} \rightarrow \text{prediction}$$

KEY IDEA

The key change is not "more parameters". It is that each layer learns a transformation: pixels become edges, edges become parts, parts become object evidence. Good representations keep task-relevant variation and suppress nuisance variation.

Three eras of deep learning



ImageNet · the turning point (2012)

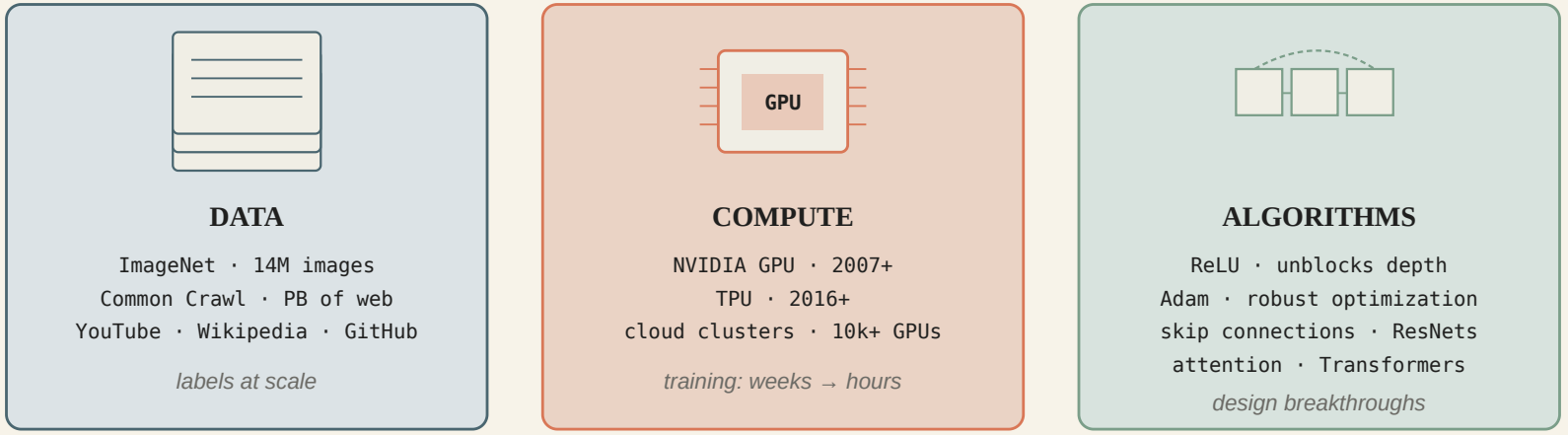
YEAR	WINNER	TOP-5 ERROR	METHOD
2010	NEC-UIUC	28.2%	SIFT + Fisher
2011	XRCE	25.8%	Hand-crafted
2012	**AlexNet**	**16.4%**	**8-layer CNN**
2015	ResNet	**3.6%**	152-layer CNN

Human top-5 error: ~5.1%. **AlexNet cut error by ~10 points in one year.**

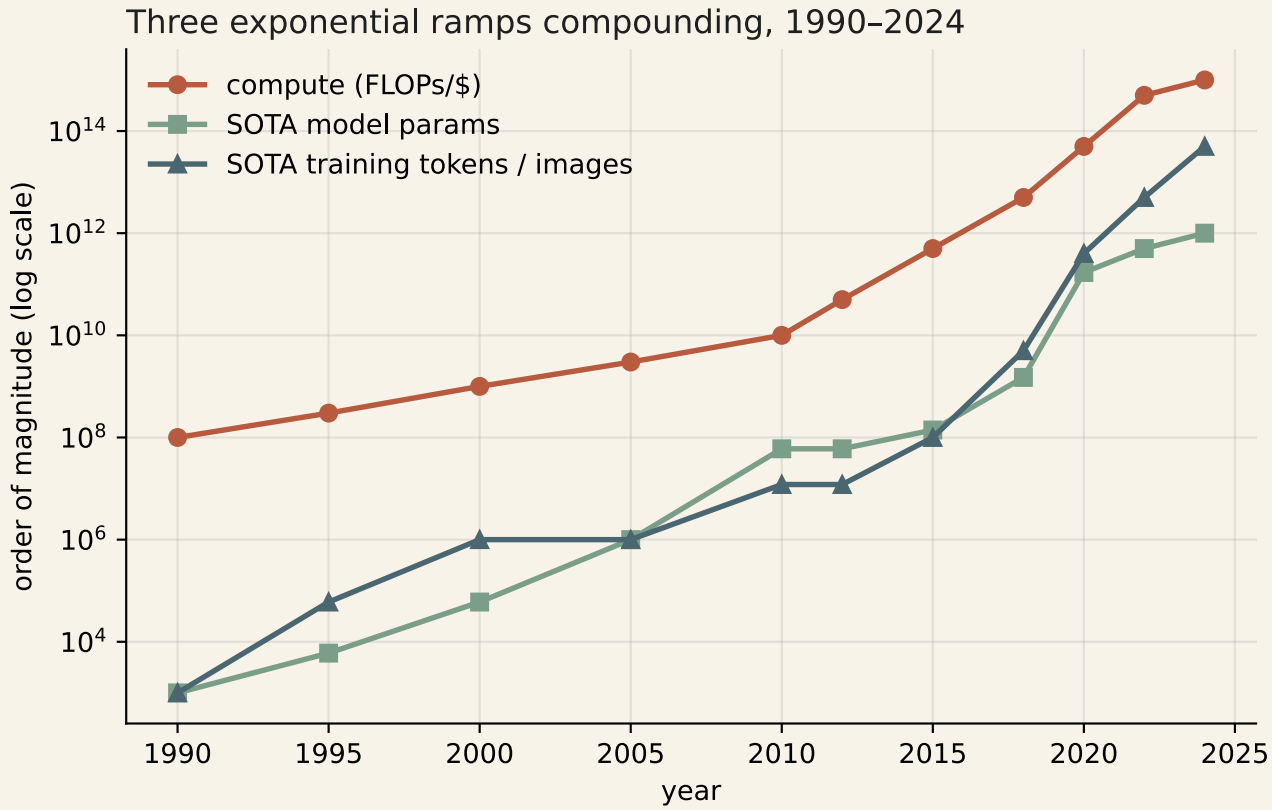
Why now? Three ingredients compounded

Three ingredients compounded 2009–2017 · remove any one and the timeline stalls

DATA × COMPUTE × ALGORITHMS



Why now · concrete numbers



DERIVATION

	1990	2024	GROWTH
Compute ($/FLOP best-case)	$\sim$10⁸ FLOPs/$	$\sim$10¹⁵ FLOPs/$	$10^7\times$
Datasets (frontier)	60k MNIST digits	5T LLM tokens	$10^8\times$

Learning outcomes · for this lecture

By the end of this lecture you will be able to:

1. Articulate **why DL works now but did not earlier**: compute, data, algorithms.
2. Describe **representation learning** as learning transformations, not just classifiers.
3. Explain why stacked **linear** layers collapse and why nonlinear layers do not.
4. Compute a small MLP's forward pass, parameter count, and tensor shapes.
5. Derive why softmax + cross-entropy gives gradient $\hat{\mathbf{y}} - \mathbf{y}$.
6. State the three axes that make deep networks useful: **expressivity, trainability, generalization**.

Course roadmap

IN PRACTICE

24-lecture arc

Foundations → optimization → regularization → **CNNs** → **detection** → sequences → **attention** → **Transformers** → **LLMs** → **vision-language** → VAEs → GANs → diffusion → efficient inference.

- **Framework:** PyTorch, exclusively.
- **Primary textbook:** Prince, *Understanding Deep Learning* (2023) — free PDF.
- **Style:** math + code + intuition + examples.
- **Assessment:** 4 quizzes · 4 assignments · attendance · bonus.

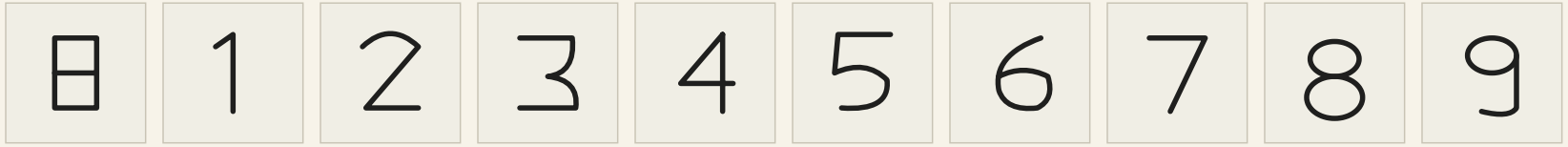
PART 2

MLP recap

The building block you already know — tightened up

Our running example

MNIST — our running example for today's lecture



0 1 2 3 4 5 6 7 8 9

PROBLEM STATEMENT

input: 28 × 28 grayscale pixels → flatten to $x \in \mathbb{R}^{784}$

label: $y \in \{0, 1, 2, \ldots, 9\}$ (10 classes)

goal: learn $f_\theta : \mathbb{R}^{784} \to \mathbb{R}^{10}$ dataset: 60,000 train + 10,000 test

Tying back to Lecture 0 · what changed

In **L0** we showed every loss is **NLL of an assumed conditional distribution** ·

TASK	CONDITIONAL	LOSS
Regression	$Y \mid \mathbf{x} \sim \mathcal{N}(\boldsymbol{\theta}^\top \mathbf{x}, \sigma^2)$	MSE
Binary classification	$Y \mid \mathbf{x} \sim \text{Bernoulli}(\sigma(\boldsymbol{\theta}^\top \mathbf{x}))$	BCE
K-class	$Y \mid \mathbf{x} \sim \text{Categorical}(\text{softmax}(W\mathbf{x}))$	CE

KEY IDEA

Logistic regression and softmax classification are 1-layer neural nets. You've already met the entire framework. Today we add **hidden layers** so the conditional distribution can be a more complex function of $\mathbf{x}$.

The probabilistic story is unchanged · pick a distribution, write NLL, optimize. Only $p_\theta(y \mid \mathbf{x})$ becomes more expressive.

From linear models to neurons

You already know **linear regression / classification** from ES 654:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b$$

A neuron is just this · plus a non-linear "squashing" function:

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

KEY IDEA

That's the only new ingredient. Everything in this course builds on top · stack many of these neurons, learn the weights, you get a deep network.

Worked example · one neuron forward pass

DERIVATION

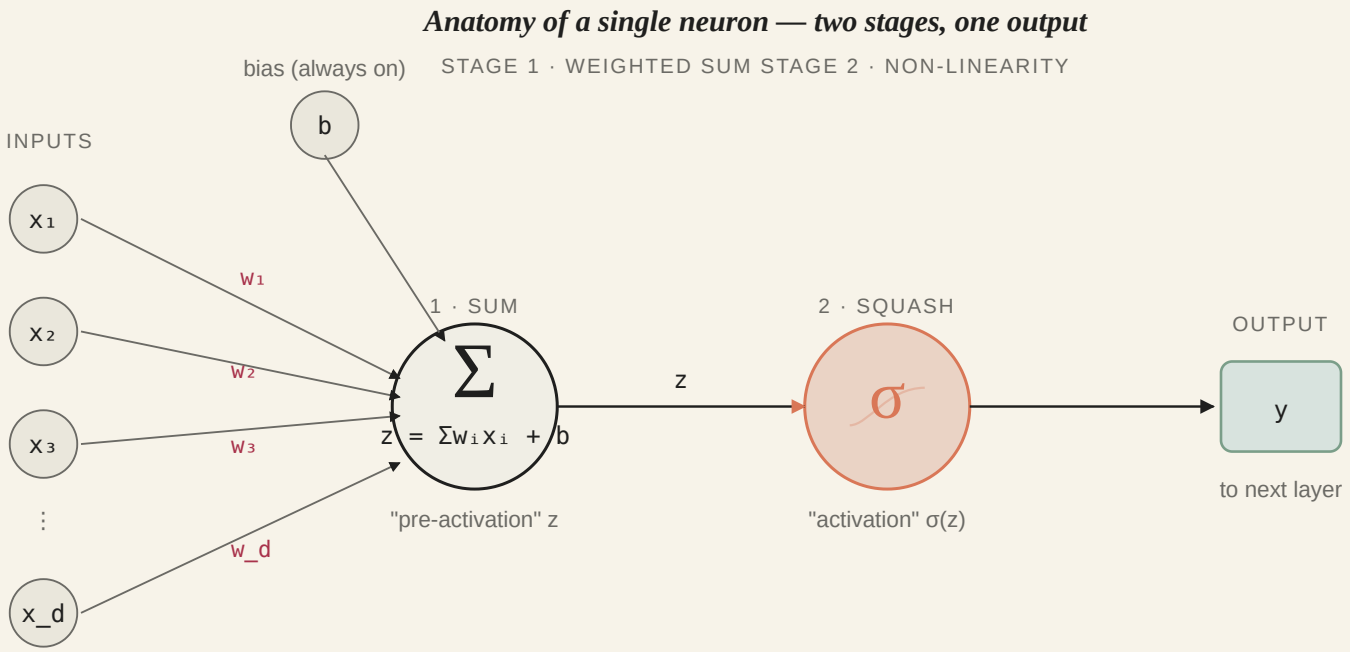
Input · $x = [0.5, -1.0]$ · Weights · $w = [0.8, 0.2]$ · Bias · $b = 0.1$

Pre-activation · $z = (0.8)(0.5) + (0.2)(-1.0) + 0.1 = 0.4 - 0.2 + 0.1 = 0.3$

Activation (sigmoid) · $\sigma(0.3) = 1/(1 + e^{-0.3}) \approx 0.574$

This 1-neuron, 2-input setup is exactly what we just had in linear regression · plus the sigmoid making it 0–1 instead of any real number. Stack 500 of these and you have an MLP layer.

The single neuron — anatomy



$$y = \sigma\left(\sum w_i x_i + b\right)$$

In vector form

$$y = \sigma\Big(\underbrace{\mathbf{w}^\top \mathbf{x} + b}_{\text{pre-activation } z} \Big)$$

- $\mathbf{x} \in \mathbb{R}^d$ — inputs
- $\mathbf{w} \in \mathbb{R}^d$ — learned weights (how much each input matters)
- $b \in \mathbb{R}$ — learned bias (threshold shift)
- $\sigma(\cdot)$ — non-linearity (squash)

  Q. Why the non-linearity? What breaks without it?

Why we need a non-linearity · the magnifying-glass analogy

KEY IDEA

Stack two magnifying glasses · you get a bigger image. Still just a *bigger linear* version of the original.

Stack two linear layers · same story. The composition of linear maps is just another linear map. No new patterns become learnable.

A non-linearity is like adding a **prism** · it bends the input in a way no linear stack can replicate. Each layer can learn a new *kind* of feature.

This is why every deep network has activation functions between linear layers. Without them, depth is wasted.

Let's prove it · stacking linear layers collapses

A tiny 2-layer network with no nonlinearity:

DERIVATION

Layer 1 · $h = W_1 x + b_1$
Layer 2 · $y = W_2 h + b_2$

Substitute h into the second equation:
$y = W_2(W_1 x + b_1) + b_2$

Distribute W_2:
$y = (W_2 W_1)x + (W_2 b_1 + b_2)$

Define · $W_{\text{eff}} = W_2 W_1$ (just another matrix) and $b_{\text{eff}} = W_2 b_1 + b_2$ (just another vector):

$$y = W_{\text{eff}} x + b_{\text{eff}}$$

The 2-layer network is exactly one linear layer. The depth was useless.

Worked numeric example · the collapse

DERIVATION

$$x = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \cdot W_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot b_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot W_2 = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \cdot b_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Forward through the 2 layers:

- $h = W_1 x + b_1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$
- $y = W_2 h + b_2 = \begin{bmatrix} 9 \\ 9 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 9 \\ 10 \end{bmatrix}$

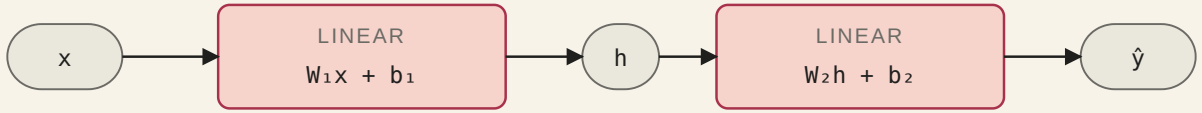
Equivalent single layer:

- $W_{\text{eff}} = W_2 W_1 = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$
- $b_{\text{eff}} = W_2 b_1 + b_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$
- Check · $W_{\text{eff}} x + b_{\text{eff}} = \begin{bmatrix} 7 \\ 8 \end{bmatrix} + \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 9 \\ 10 \end{bmatrix}$ · **same!**

Without σ · depth gives nothing

Without σ: two layers = one layer (depth gives nothing)

STACKING TWO LINEAR LAYERS

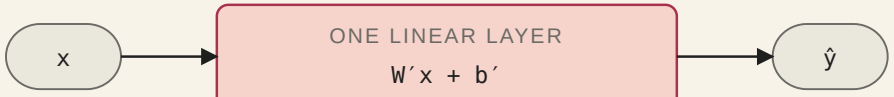


$$\hat{y} = W_2(W_1x + b_1) + b_2 = (W_2W_1)x + (W_2b_1 + b_2)$$

group the products into one matrix, the biases into one vector — it is literally a single linear layer

⇓

EQUIVALENT TO



Conclusion · a depth-L linear network has the capacity of a single linear layer. ***Non-linearity is what makes depth meaningful.***

XOR · the canonical "linear can't, MLP can" example

XOR · 2D binary inputs, output $= x_1 \oplus x_2$ ·

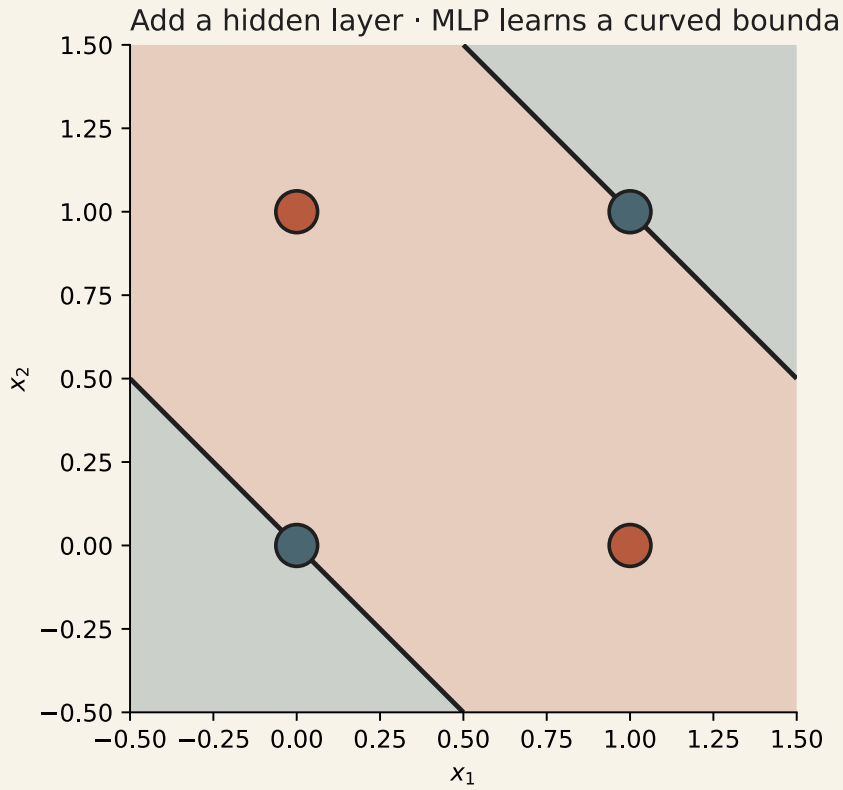
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

KEY IDEA

XOR is the historical "minor scandal" that **broke perceptrons** in the 1969 Minsky-Papert critique — and motivated the multi-layer / hidden-unit revolution of the 1980s.

Try mentally · can you find a single line $w_1 x_1 + w_2 x_2 + b = 0$ that puts $\{(0,1),(1,0)\}$ on one side and $\{(0,0),(1,1)\}$ on the other? Spoiler · *no straight line works*.

XOR · linear fails, two-layer MLP succeeds



DERIVATION

Left · Three attempts at a linear separator. Whichever line you draw, two same-class points end up on opposite sides.

Right · A 2-layer MLP with **2 hidden ReLU units** carves out a curved decision region · all four points correctly classified.

XOR · build a 2-layer MLP by hand

A 2-input, 2-hidden, 1-output MLP with ReLU ·

DERIVATION

$$\mathbf{h} = \mathrm{ReLU}(W_1\mathbf{x} + \mathbf{b}_1), \qquad \hat{y} = \mathbf{w}_2^\top \mathbf{h} + b_2$$

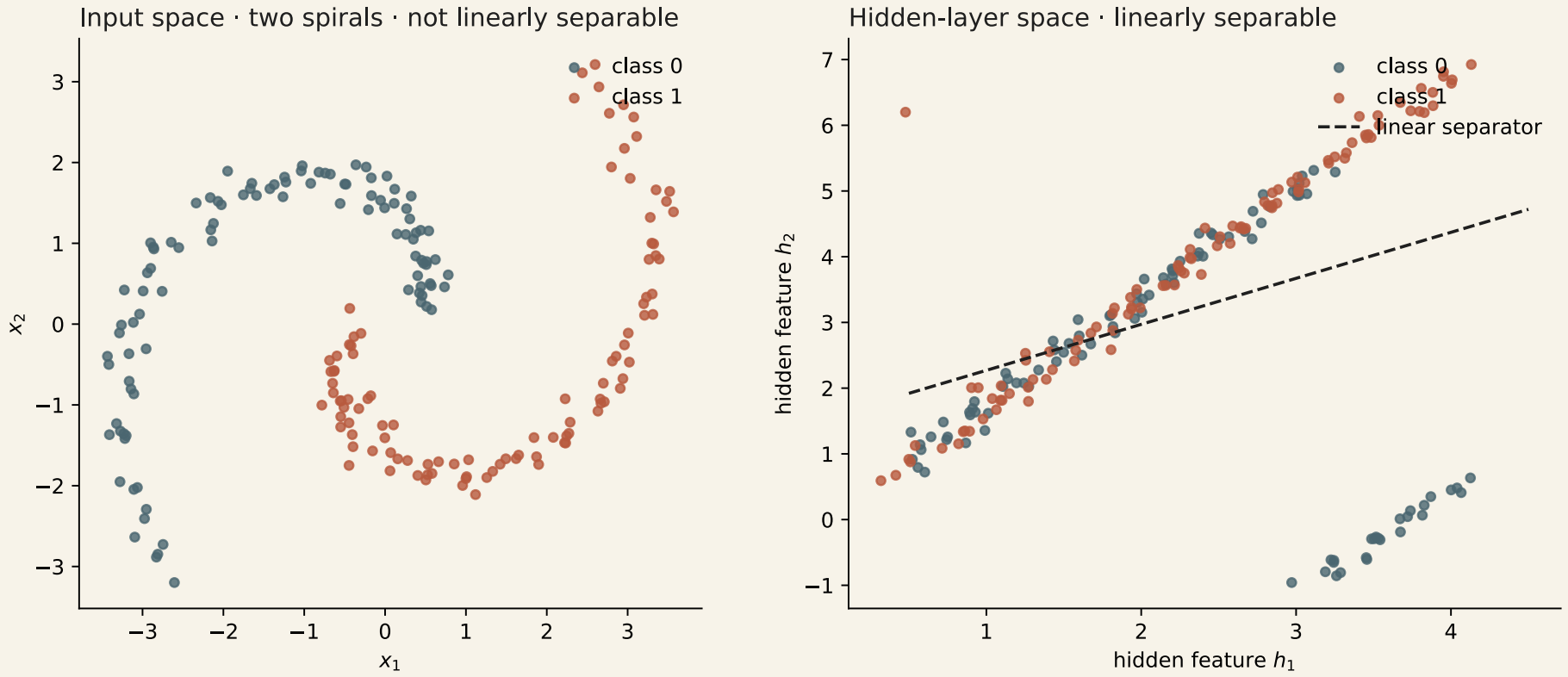
Choose by hand ·

$$W_1 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \quad \mathbf{w}_2 = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, \quad b_2 = 0$$

$\mathbf{x}$	$W_1\mathbf{x} + \mathbf{b}_1$	$\mathbf{h} = \mathrm{ReLU}(\cdot)$	$\mathbf{w}_2^\top \mathbf{h} + b_2$
$(0, 0)$	$(0, -1)$	$(0, 0)$	0
$(0, 1)$	$(1, 0)$	$(1, 0)$	1
$(1, 0)$	$(1, 0)$	$(1, 0)$	1
$(1, 1)$	$(2, 1)$	$(2, 1)$	$2 - 2 = 0$

Outputs $0, 1, 1, 0$ — **exactly XOR**. The hidden ReLU "kink" gives the curved decision region we drew. Two neurons + one non-linearity is the smallest network that solves it.

Feature-space transformation · the deeper view

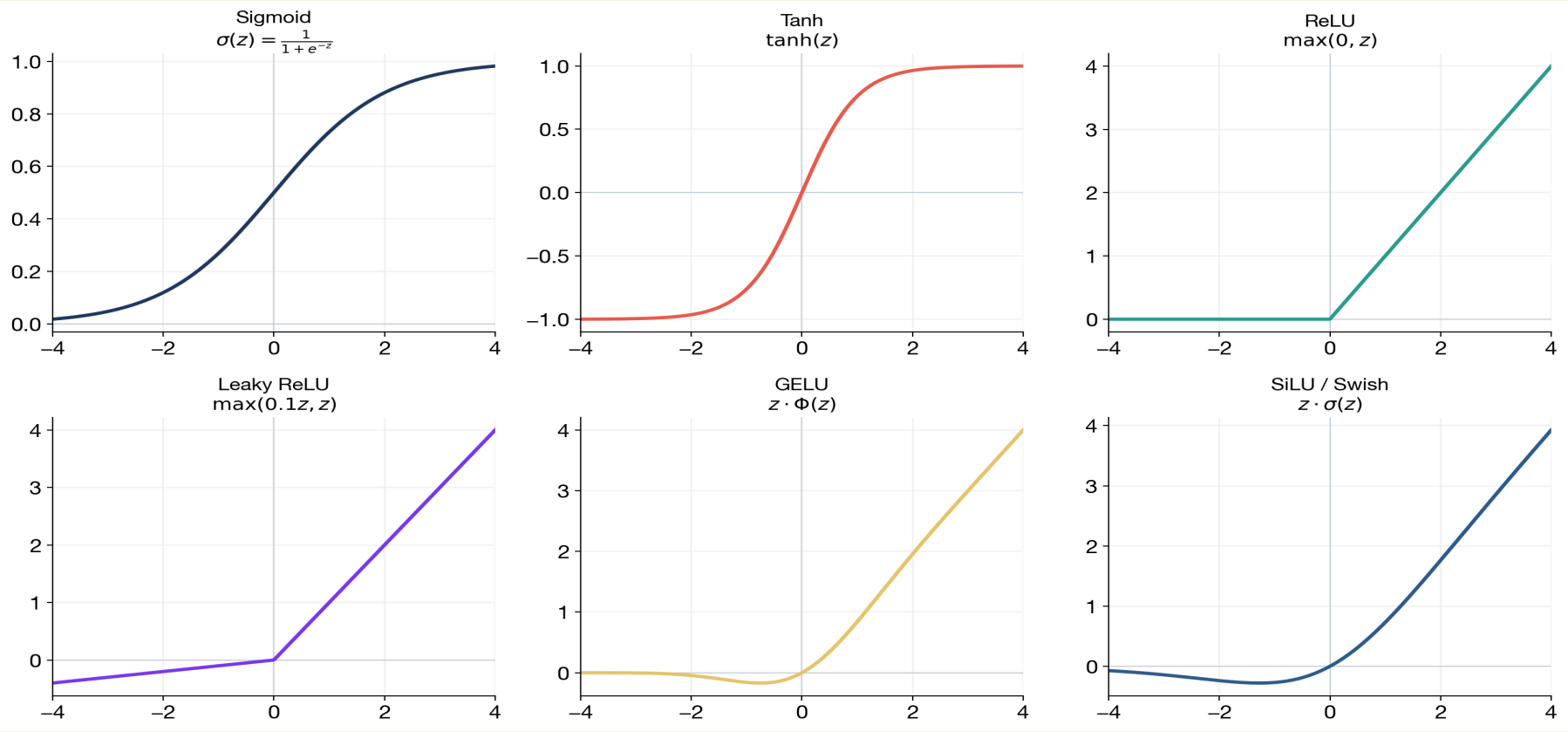


DERIVATION

A hidden layer **reshapes the data** so the next (linear) layer's job becomes easy.

Left · two interlocking spirals · no straight line separates them.
Right · the same data after a learned hidden layer · spirals are *unrolled* into bands that a linear classifier separates trivially.

Activation functions at a glance



NAME	FORMULA	WHERE YOU SEE IT
Sigmoid	$1/(1+e^{-z})$	gates
Tanh	$\tanh(z)$	RNNs

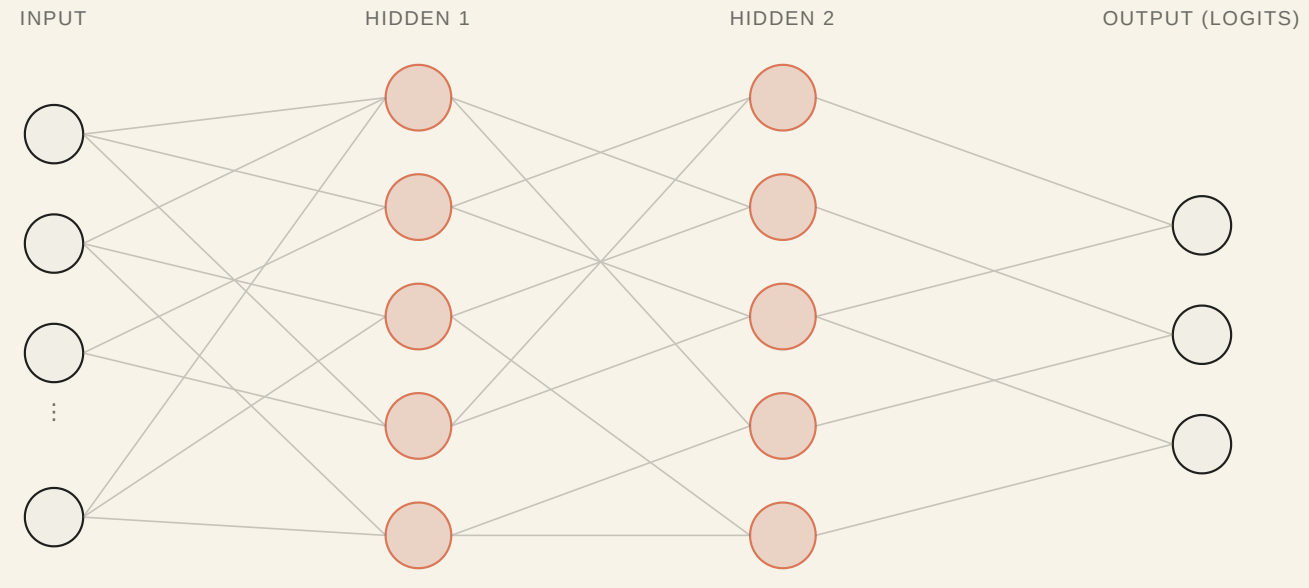
Activation functions · what can go wrong

ACTIVATION	FAILURE MODE	PRACTICAL CONSEQUENCE
Sigmoid	Saturates; derivative ≤ 0.25	vanishing gradients
Tanh	Saturates for large $	z
ReLU	zero gradient for $z < 0$	dead units if updates are harsh
GELU / SiLU	smoother but costlier	common in Transformers, less common in tiny CNNs

KEY IDEA

An activation is not decoration. It controls both the representation and the backward gradient flow. Modern defaults are ReLU-family for CNNs and GELU / SwiGLU-family for Transformers.

Stacking neurons → MLP



Parameters: 256·784 + 256 + 256·256 + 256 + 10·256 + 10 = **269,322**

Parameter count — do this in your head

For MNIST with hidden sizes 256, 256:

$$\underbrace{256 \times 784}_{W_1} + \underbrace{256}_{b_1} + \underbrace{256 \times 256}_{W_2} + \underbrace{256}_{b_2} + \underbrace{10 \times 256}_{W_3} + \underbrace{10}_{b_3} = \boxed{269{,}322}$$

A tiny MNIST model has ~270k parameters. GPT-3 has 175 billion — $6 \times 10^5 \times$ more.

Batched matrix form · the shapes that matter

For a mini-batch:

$$X \in \mathbb{R}^{B \times d_{\text{in}}}, \quad W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}, \quad b \in \mathbb{R}^{d_{\text{out}}}$$

$$Z = XW + b \quad \Rightarrow \quad Z \in \mathbb{R}^{B \times d_{\text{out}}}$$

MNIST batch of 64:

TENSOR	SHAPE
flattened input X	64×784
first weight W_1	784×256
hidden activations H_1	64×256
final logits	64×10

Most PyTorch bugs in early DL are shape bugs. Check the batch dimension first.

MLP in PyTorch

```python
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, d_in=784, d_h=256, d_out=10):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d_in, d_h), nn.ReLU(),
            nn.Linear(d_h,  d_h), nn.ReLU(),
            nn.Linear(d_h,  d_out),     # raw logits — no softmax here
        )

    def forward(self, x):
        return self.net(x)
```

Q. Why no activation after the last `Linear`?

The last layer is bare · the #1 beginner bug

For classification:

- Output should be K raw scores (logits).
- `nn.CrossEntropyLoss` internally applies `log_softmax`.
- Adding your own softmax → **double softmax** → frozen loss near $\log K$.

> WATCH OUT
>
> Symptom to memorize: training loss stuck at ~2.30 (= $\log 10$) and refusing to move.
> Fix: remove the extra softmax.

PART 3

Losses and backprop

The math that makes learning possible

From scores to probabilities · the goal

The network outputs raw "scores" (logits) for each class · arbitrary real numbers like $[2.0, 1.0, 0.1]$.

For classification, we need a **valid probability distribution** · all values ≥ 0, summing to 1.

KEY IDEA

Two problems to solve:

1. **Make values positive** · `exp(·)` does this for any real input.
2. **Make them sum to 1** · divide by the total.

Together · the **softmax** function.

Softmax · worked numeric example

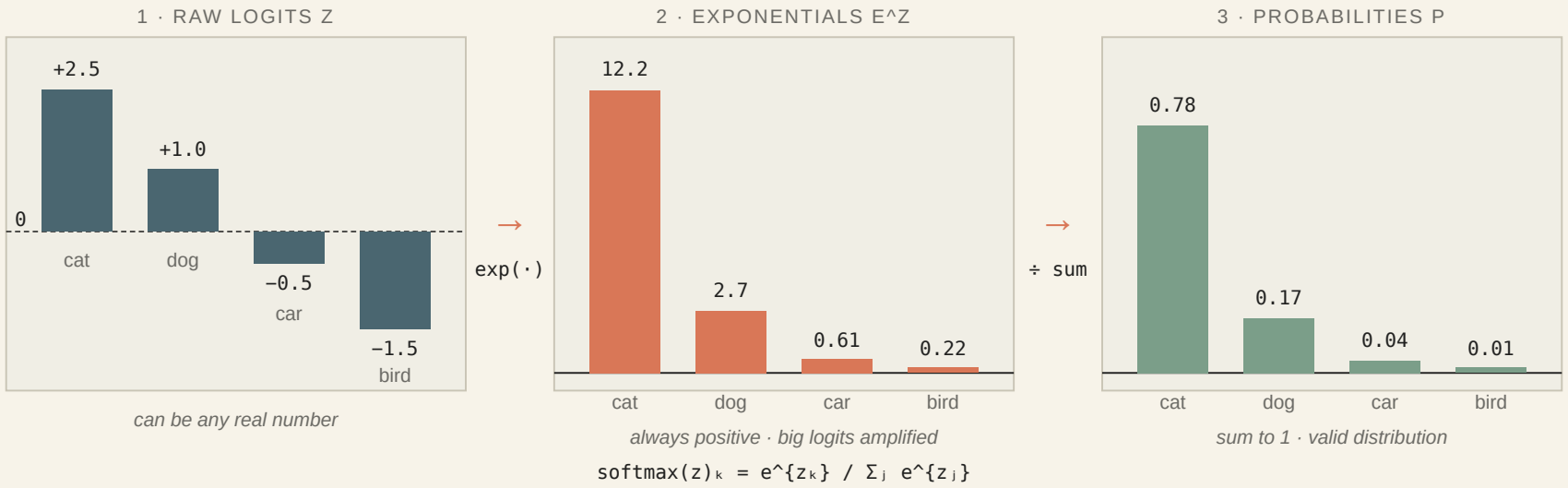
DERIVATION

Logits · $z = [2.0, 1.0, 0.1]$.

Step 1 · exponentiate · $e^z = [e^{2.0}, e^{1.0}, e^{0.1}] = [7.39, 2.72, 1.11]$
Step 2 · sum · $7.39 + 2.72 + 1.11 = 11.22$
Step 3 · normalize · $\hat{y} = [7.39/11.22, 2.72/11.22, 1.11/11.22] = [0.66, 0.24, 0.10]$

Note · the relative ranking is preserved (the largest logit becomes the largest probability) · but the values are now interpretable as probabilities. The softmax is the standard last layer for classification.

Softmax · three acts



Big logit → big probability. Negative logit → small positive probability. Never negative, always sums to 1.

IN PRACTICE

▶ Interactive: drag the temperature slider, watch the distribution morph — softmax-temperature.

Why exponentiate?

1. Logits can be **negative**; raw ratios misbehave. e^{z_k} is always positive.
2. Softmax **amplifies** differences — biggest logit dominates smoothly.
3. It falls out of **maximum likelihood** for categorical outputs (next).

Cross-entropy from MLE

One example $(\mathbf{x}, y)$, true class c. Maximize data likelihood → minimize negative log-likelihood:

$$\mathcal{L}(\theta) = -\log P(y = c \mid \mathbf{x}; \theta) = -\log \hat{y}_c$$

With one-hot $\mathbf{y}$:

$$\mathcal{L} = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

DERIVATION

This **is** cross-entropy. MLE hands it to us for free — we did not invent it.

Why cross-entropy is the right score

Accuracy only asks whether the top class is correct. Cross-entropy asks whether the whole probability distribution is honest.

Suppose the true class is class 0:

PREDICTION	CE LOSS
$[0.70, 0.20, 0.10]$	$-\log 0.70 = 0.36$
$[0.99, 0.005, 0.005]$	$-\log 0.99 = 0.01$

But if the label is class 1:

PREDICTION	CE LOSS
$[0.70, 0.20, 0.10]$	$-\log 0.20 = 1.61$
$[0.99, 0.005, 0.005]$	$-\log 0.005 = 5.30$

KEY IDEA

Cross-entropy rewards calibrated confidence and punishes confident wrong answers. That is why it is a proper scoring rule for classification.

Push-pull intuition · the gradient

Logits $z = [z_1, \ldots, z_K]$. Loss $\mathcal{L}$. We need $\partial \mathcal{L} / \partial z_k$ · "how should we change z_k to lower the loss?"

KEY IDEA

- If k is the **correct** class · we want its probability to be 1 · gradient should **push** z_k **up**.
- If k is a **wrong** class · we want its probability to be 0 · gradient should **push** z_k **down**.

The next slide derives the formula. The result · $\partial \mathcal{L} / \partial z_k = \hat{y}_k - y_k$ · does exactly the push-pull above.

Deriving · softmax + CE gradient · step by step

DERIVATION

Setup · $\mathcal{L} = -\sum_j y_j \log \hat{y}_j \cdot \hat{y}_j = e^{z_j} / \sum_i e^{z_i}$.

Want · $\partial \mathcal{L} / \partial z_k$. Chain rule · $\partial \mathcal{L} / \partial z_k = \sum_j (\partial \mathcal{L} / \partial \hat{y}_j)(\partial \hat{y}_j / \partial z_k)$.

Part 1 · $\partial \mathcal{L} / \partial \hat{y}_j = -y_j / \hat{y}_j$.

Part 2 · case $j = k$ (own logit, quotient rule):
$\partial \hat{y}_k / \partial z_k = \hat{y}_k (1 - \hat{y}_k)$

Part 2 · case $j \neq k$:
$\partial \hat{y}_j / \partial z_k = -\hat{y}_j \hat{y}_k$

Combine · $\partial \mathcal{L} / \partial z_k = -y_k (1 - \hat{y}_k) + \sum_{j \neq k} y_j \hat{y}_k = -y_k + \hat{y}_k \sum_j y_j$

Since $\sum_j y_j = 1$ for one-hot · $\boxed{\partial \mathcal{L} / \partial z_k = \hat{y}_k - y_k}$

Worked numeric · the gradient

DERIVATION

Logits $z = [2.0, 1.0, 0.1]$. Softmax · $\hat{y} = [0.66, 0.24, 0.10]$. True label · class 0, $y = [1, 0, 0]$.

$$\partial \mathcal{L} / \partial z = \hat{y} - y = [0.66 - 1, 0.24 - 0, 0.10 - 0] = [-0.34, 0.24, 0.10]$$

SGD step · $z \leftarrow z - \eta \cdot (\hat{y} - y)$

- z_0 has negative gradient · SGD pushes it **up** (good, correct class).
- z_1, z_2 have positive gradient · SGD pushes them **down** (good, wrong classes).

The gradient is bounded between -1 and 1 per logit · stable. No exploding gradients from the loss itself.

The elegant softmax + CE gradient

$$\boxed{\frac{\partial \mathcal{L}}{\partial z_k} = \hat{y}_k - y_k}$$

Prediction minus target. Same form as logistic regression — no accident.

What that gradient actually looks like

Backprop · the blame game

The forward pass computes the loss. **Backprop figures out who to blame.**

KEY IDEA

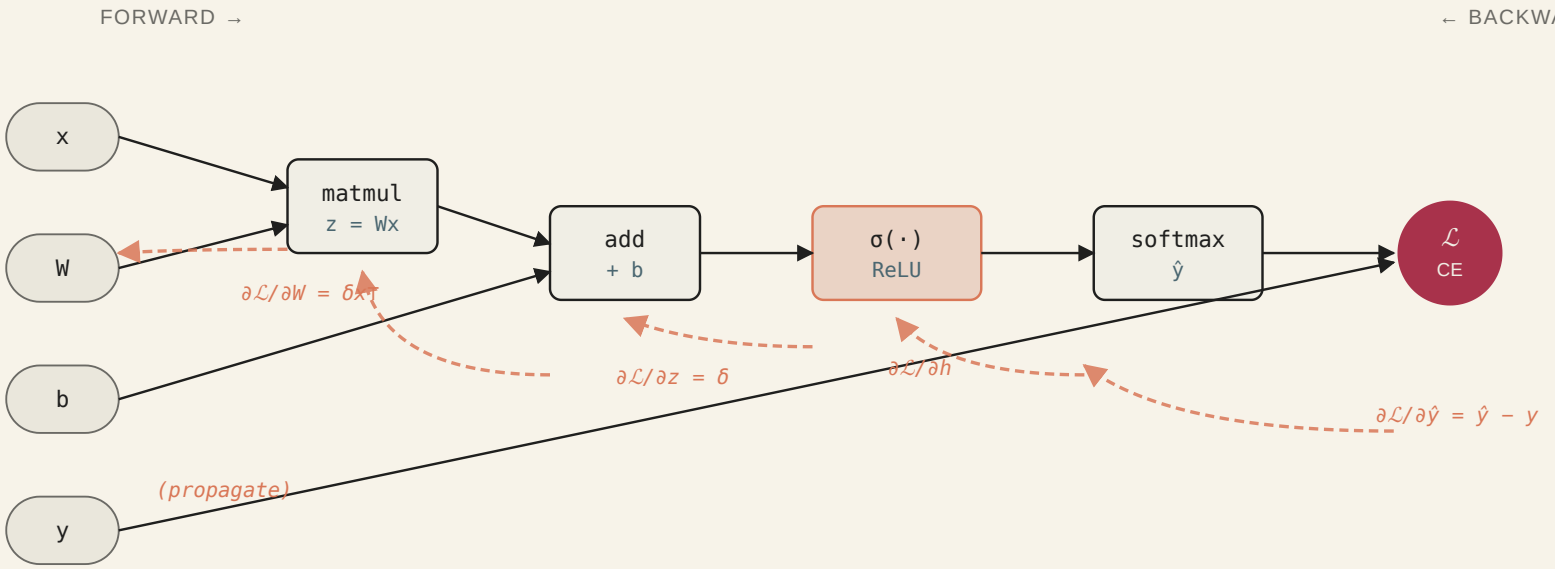
For each weight, ask · *how much did this weight contribute to the final error?*

Walk backwards from the loss through the network · at each layer, distribute "blame" proportionally to how much each weight affected the layer's output. Adjust weights to reduce that blame.

That's the entire idea. The math (chain rule) is mechanical · the *concept* is "trace responsibility backwards."

Backpropagation · the computational view

Every network is a DAG of differentiable ops. Forward computes values; backward computes gradients in reverse order.



EVERY NODE = DIFFERENTIABLE OP. CHAIN RULE RUNS THE DASHED PATH RIGHT→LEFT.

Backprop · the blame distributor

For a layer $z = Wx + b$, suppose the next layer sends back · "your output's error signal is $\delta = \partial \mathcal{L} / \partial z$."

KEY IDEA

We must figure out three things:

- How much was **W**'s fault? · $\partial \mathcal{L} / \partial W$
- How much was **b**'s fault? · $\partial \mathcal{L} / \partial b$
- How much should we blame the **previous layer's** output x? · $\partial \mathcal{L} / \partial x$ · this is the message we pass back.

The next slide does these one at a time, with full chain rule.

Deriving · the linear-layer backward pass

For one element · $z_i = \sum_j W_{ij} x_j + b_i$.

DERIVATION

1. Bias · $\partial z_i / \partial b_i = 1 \to \partial \mathcal{L} / \partial b_i = \delta_i \to \partial \mathcal{L} / \partial b = \delta$

2. Weight · $\partial z_i / \partial W_{ij} = x_j \to \partial \mathcal{L} / \partial W_{ij} = \delta_i x_j \to \partial \mathcal{L} / \partial W = \delta\, x^\top$ (outer product)

3. Input · x_j affects every output through W_{ij} · sum over outputs:
$\partial \mathcal{L} / \partial x_j = \sum_i \delta_i \cdot W_{ij} \to \partial \mathcal{L} / \partial x = W^\top \delta$

These three lines are the entire backward pass for a `Linear` layer in PyTorch.

Worked numeric · linear-layer backward

DERIVATION

$x = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \cdot W = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}$ · upstream gradient $\delta = \begin{bmatrix} 0.5 \\ -0.1 \end{bmatrix}$.

Weight gradient · $\delta x^\top = \begin{bmatrix} 0.5 \\ -0.1 \end{bmatrix} \begin{bmatrix} 2 & 3 \end{bmatrix} = \begin{bmatrix} 1.0 & 1.5 \\ -0.2 & -0.3 \end{bmatrix}$

Bias gradient · $\delta = \begin{bmatrix} 0.5 \\ -0.1 \end{bmatrix}$

Input gradient (passed to previous layer) · $W^\top \delta = \begin{bmatrix} 0.1 & 0.3 \\ 0.2 & 0.4 \end{bmatrix} \begin{bmatrix} 0.5 \\ -0.1 \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.06 \end{bmatrix}$

These three numbers are what `loss.backward()` computes for one Linear layer · in pure NumPy you could write it in 3 lines.

The local-gradient rule · three lines

For $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$ with upstream $\boldsymbol{\delta} = \partial \mathcal{L} / \partial \mathbf{z}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \boldsymbol{\delta}\, \mathbf{x}^\top, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \boldsymbol{\delta}, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \mathbf{W}^\top \boldsymbol{\delta}$$

KEY IDEA

These three lines are the **entire** backward pass of a linear layer. Everything else is repeating this rule through activations and stacking.

End-to-end worked numeric · 2-layer MLP, one example

Tiny 2-1-1 net with sigmoid hidden, sigmoid output. Input $x = 0.5$, target $y = 1$ (binary). Initial weights $w_1 = 0.4,\ b_1 = 0,\ w_2 = 0.6,\ b_2 = 0$. LR $\eta = 0.5$.

DERIVATION

Forward ·

- $z_1 = 0.4 \cdot 0.5 + 0 = 0.2$, $h = \sigma(0.2) \approx 0.5498$
- $z_2 = 0.6 \cdot 0.5498 = 0.3299$, $\hat{y} = \sigma(0.3299) \approx 0.5817$
- BCE loss $L = -\log 0.5817 \approx \mathbf{0.5414}$

Backward (using the BCE+sigmoid identity $\partial L/\partial z_2 = \hat{y} - y$) ·

- $\delta_2 = \hat{y} - y = 0.5817 - 1 = -0.4183$
- $\partial L/\partial w_2 = \delta_2 \cdot h = -0.4183 \cdot 0.5498 \approx -0.2300$
- $\partial L/\partial h = w_2 \cdot \delta_2 = 0.6 \cdot (-0.4183) = -0.2510$
- $\delta_1 = \partial L/\partial h \cdot \sigma'(z_1) = -0.2510 \cdot 0.5498 \cdot 0.4502 \approx -0.0621$
- $\partial L/\partial w_1 = \delta_1 \cdot x = -0.0621 \cdot 0.5 \approx -0.0311$

Update · $w_1 \to 0.4156,\ w_2 \to 0.7150.$

Forward at step 1 · $\hat{y} \approx 0.5959$, $L \approx 0.5176$ **— loss dropped ✓**

Same rule with batches

For a batch:

$$Z = XW + b$$

$$X \in \mathbb{R}^{B \times d_{\text{in}}}, \quad W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}, \quad \Delta = \frac{\partial \mathcal{L}}{\partial Z} \in \mathbb{R}^{B \times d_{\text{out}}}$$

The linear-layer backward pass becomes:

$$\frac{\partial \mathcal{L}}{\partial W} = X^\top \Delta, \quad \frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^{B} \Delta_i, \quad \frac{\partial \mathcal{L}}{\partial X} = \Delta W^\top$$

KEY IDEA

Backprop is mostly matrix multiplication. PyTorch is not doing magic here; it is applying these local rules through the computation graph.

PART 4

Why go deep?

A teaser for Lecture 2

One layer is enough, in principle

A single hidden layer can approximate any continuous function (UAT — next lecture).

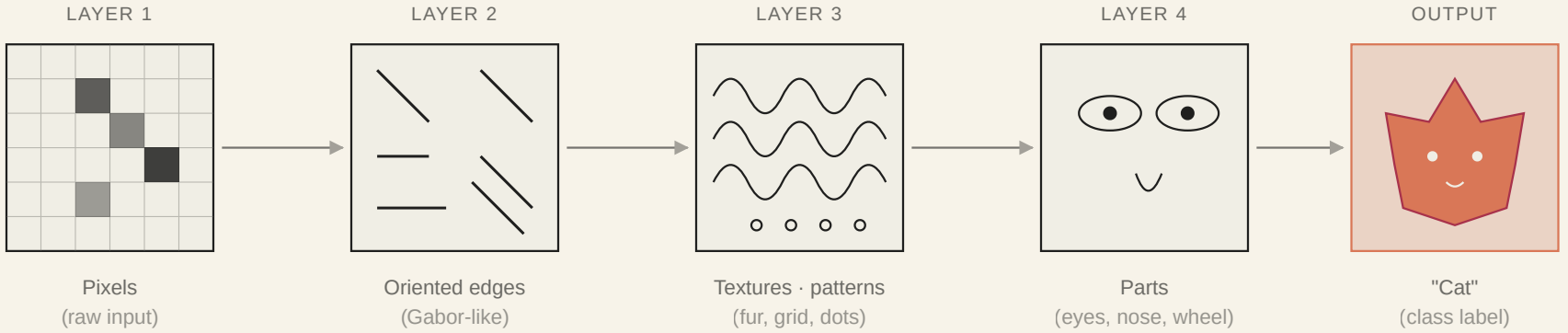
Q. So why do we ever use more than one?

Give an honest answer before turning the page.

Depth ⇒ hierarchical features

Deep networks learn a hierarchy of features

RAW INPUT → EDGES → TEXTURES → PARTS → OBJECTS



Each layer composes the previous. No human hand-crafts edges, textures, or parts — the network discovers them from data.

ZEILER & FERGUS 2014 · OLAH ET AL. 2017

Biology inspired the hierarchy

Hubel & Wiesel (Nobel 1981) — the cat visual cortex is hierarchical.

BRAIN REGION	ROUGHLY ANALOGOUS LAYER	DETECTS
V1	early layer	oriented edges
V2	next layer	textures, junctions
V4	mid layer	shapes, parts
IT	late layer	objects, faces

Biology suggested that hierarchical visual processing is useful. Modern neural networks are not literal brain models: the optimizer, data scale, loss function, and hardware are engineered.

Backprop as broken telephone

KEY IDEA

Think of backprop as a "telephone game" played backwards from the loss.

Each layer has to pass the **error signal** to the layer before it. If each layer multiplies the message by something less than 1 (e.g., 0.25 for sigmoid), then by the time the signal reaches the early layers it's a faint mumble.

Those early layers stop learning. This is the **vanishing gradient** problem · the topic of the next slide and a recurring theme through the course (RNN, deep MLPs, deep Transformers).

Why σ' shrinks · let's compute it

For sigmoid · $\sigma(z) = 1/(1+e^{-z})$.

DERIVATION

$$\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \sigma(z)\cdot(1-\sigma(z))$$

Maximum value · at $z=0$, $\sigma(0)=0.5$, so $\sigma'(0) = 0.5\cdot 0.5 = \mathbf{0.25}$.

For $|z| \geq 3$ · $\sigma'(z) \leq 0.045$ (saturated regions).

Every layer with sigmoid multiplies the backward-flowing gradient by something ≤ 0.25. After 5 layers, the gradient is shrunk by at least $0.25^5 \approx 0.001$. After 10 layers · $\approx 10^{-6}$. The earliest layers stop learning.

Worked numeric · the gradient vanishes

A 5-layer sigmoid net. Assume all weights = 1 and inputs are in saturating regions so $\sigma'(z) \approx 0.1$. Gradient at output = 1.

DERIVATION

LAYER	LOCAL FACTOR $W \cdot \sigma'$	GRADIENT SIGNAL
Output	—	1.0
L4	$1 \cdot 0.1 = 0.1$	0.1
L3	0.1	0.01
L2	0.1	0.001
L1	0.1	**0.0001**

The first layer gets 10^{-4} of the signal · its weights barely move. **Learning stalls in the early layers.** This is exactly why ReLU (derivative 0 or 1) replaced sigmoid in deep nets · and why ResNet's skip connections exist.

Depth has a cost · vanishing gradients

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \prod_{l=1}^{L} \frac{\partial \mathbf{h}_l}{\partial \mathbf{h}_{l-1}} \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}}$$

Each sigmoid factor ≤ 0.25:

DEPTH L	UPPER BOUND ON GRADIENT MAGNITUDE
5	10^{-3}
10	10^{-6}
20	10^{-12}

Early layers effectively stop learning. **This blocked depth for 20 years.**

The fix · ReLU

$$\mathrm{ReLU}(z) = \max(0, z), \quad \mathrm{ReLU}'(z) \in \{0, 1\}$$

For active neurons the gradient is exactly 1 — no shrinkage.

But ReLU alone is not enough for very deep networks:

PROBLEM	TOOL
activation variance shrinks or explodes	Xavier / He initialization
gradients weaken through many layers	residual connections
optimization is scale-sensitive	normalization layers

Lecture 2 derives why these tools make depth trainable.

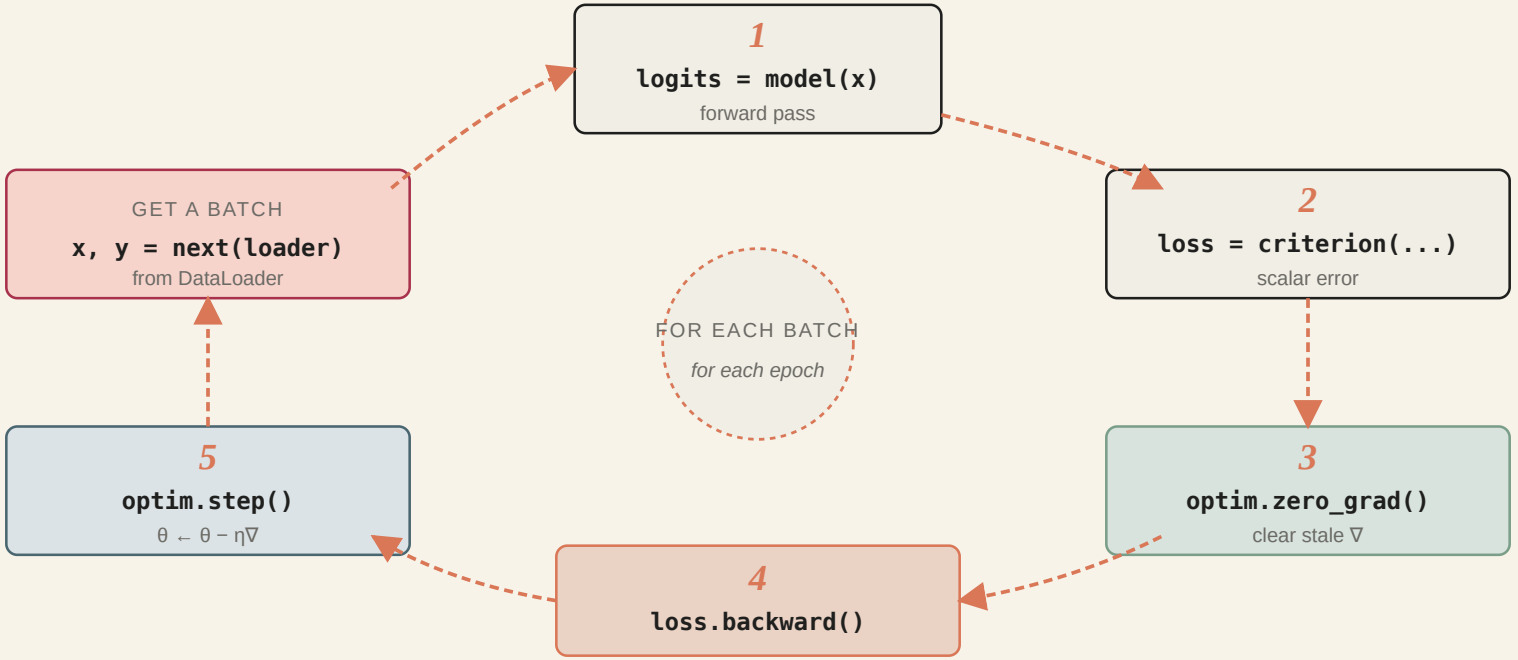
PART 5

The training loop

The five lines you'll type every day this semester

The training cycle

The training loop · five steps, repeated for every batch



The PyTorch training loop · code

```
model     = MLP(784, 256, 10).to('cuda')
criterion = nn.CrossEntropyLoss()
optim     = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

for epoch in range(10):
    for x, y in loader:
        x, y = x.view(-1, 784).to('cuda'), y.to('cuda')
        logits = model(x)                  # 1. forward
        loss   = criterion(logits, y)      # 2. loss
        optim.zero_grad()                  # 3. zero grads
        loss.backward()                    # 4. backward
        optim.step()                       # 5. update
```

Every real training script is a variation on this.

One common bug · `zero_grad`

Q. What if you forget `optimizer.zero_grad()`?

WATCH OUT

PyTorch **accumulates** gradients by default. Without zeroing, each backward adds to the previous `.grad`. After k batches you are stepping $k\times$ the real gradient — updates explode silently.

The single most common PyTorch bug.

Train mode vs eval mode

Some layers behave differently during training and evaluation.

MECHANISM	`MODEL.TRAIN()`	`MODEL.EVAL()`
Dropout	randomly masks activations	uses all activations
BatchNorm	updates running statistics	uses stored statistics
Autograd	tracks gradients if enabled	still tracks unless disabled

```
model.eval()
with torch.no_grad():
    logits = model(x_val)
```

WATCH OUT

For validation and test: use `model.eval()` and `torch.no_grad()`. Otherwise your measured performance may be noisy, slower, or wrong.

Train / val / test

How to split your data — and what each split is for

ALL AVAILABLE DATA · E.G. 70,000 MNIST IMAGES

TRAIN · UPDATE Θ

- computes **∇ loss** → updates weights
- seen every epoch
- loss always ↓ with capacity

VAL · TUNE HYPERPARAMS

- LR, width, depth, dropout
- early-stopping signal
- never touches optimizer

TEST · REPORT ONCE

- final honest estimate
- locked until submission

peek twice → leakage

Rule of thumb: validation is your *practice exam*; test is the *final* you take once.

Splits must match the real question

Random splitting is not always honest.

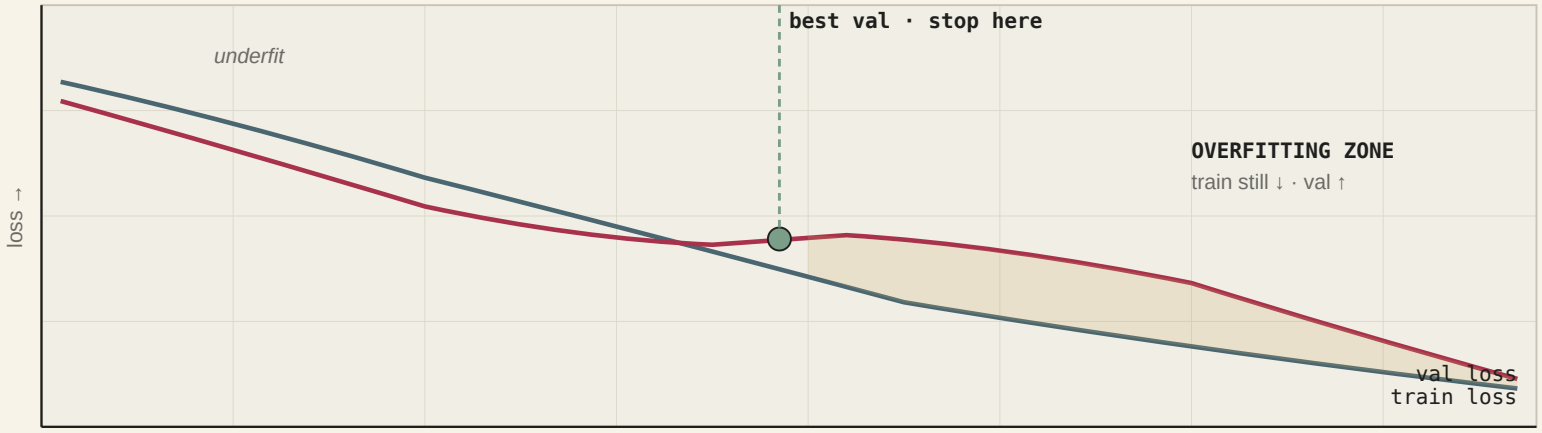
DATA TYPE	BAD SPLIT	BETTER SPLIT
medical images	random image split	split by patient
video frames	random frame split	split by video / scene
recommender logs	random row split	split by user or time
documents	random paragraph split	split by document / source
sensors	random window split	split by device / location

KEY IDEA

Validation should answer the question: "Will this model work on new cases we actually care about?"

Training loss ↓ ≠ model better

The gap between training and validation loss · what every training run looks like



train loss monotonically drops · val loss drops then rises · the gap is overfitting · stop at the val minimum

Overfitting · the rule

- Training loss monotonically drops.
- Validation loss drops, then rises.
- The gap **is** overfitting.

WATCH OUT

Never tune on the test set.

Test = final exam you take once.
Validation = practice exams, take many times.
Training = studying.

Putting it all together · the L01 master sentence

DERIVATION

A neural network is just a **stack of (linear → non-linearity)** blocks · the linear layer mixes features, the non-linearity bends the space. **Backprop** assigns blame layer-by-layer using the chain rule, and **SGD** updates parameters in the descent direction.

STEP	WHAT HAPPENS	WHERE IT CAME FROM
1 · Forward	$\mathbf{a}^{(\ell)} = \sigma(W^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)})$	composition of layers
2 · Loss	$\mathcal{L} = -\log p(y \mid x; \theta)$	NLL of the chosen distribution (L00)
3 · Backward	$\delta^{(\ell-1)} = (W^{(\ell)})^{\top}\delta^{(\ell)} \odot \sigma'$	chain rule on the comp. graph
4 · Update	$\theta \leftarrow \theta - \eta \nabla_{\theta}\mathcal{L}$	SGD

Sigmoid + BCE = L00's logistic-MLE, just with a hidden layer in front. Softmax + CE = L00's multiclass NLL. **Same probabilistic story, more layers.**

Practice problems

Try these on paper; verify with the notebooks.

DERIVATION

P1. A 1-hidden-layer MLP for MNIST has $784 \to 512 \to 10$. How many parameters in total (weights + biases)? How many FLOPs in one forward pass for a single image?

P2. Show that for sigmoid $\sigma(z) = 1/(1+e^{-z})$, $\sigma'(z) = \sigma(z)(1-\sigma(z))$ and the maximum value of σ' is $1/4$ at $z = 0$.

P3. Build a 2-hidden-unit ReLU MLP that computes the **AND** function ($y = x_1 \cdot x_2$). State the weights explicitly.

P4. A 3-class classifier outputs logits $\mathbf{z} = [3, 1, -1]$ for an example with true class 0. Compute (a) the softmax probabilities, (b) the cross-entropy loss, (c) the gradient on each logit using $\partial L/\partial z_k = \hat{y}_k - y_k$.

P5. Why does PyTorch's `CrossEntropyLoss` take **logits** rather than probabilities as input? Give two reasons (numerical and gradient-related).

P6. A 10-layer sigmoid network has weights initialized as $\mathcal{N}(0, 1)$. Estimate the magnitude of the gradient at the first layer when the upstream gradient at the loss is 1. (Hint · use $|\sigma'| \leq 0.25$ and the chain rule.)

Summary · Lecture 1 — summary

- **Deep learning = representation learning.** Layers learn transformations that preserve task signal and suppress nuisance variation.
- **Why now:** data + compute + algorithms compounded 2009–2017.
- **Neuron = sum + squash.** Stack them and non-linearity keeps depth meaningful.
- **Softmax + CE from MLE:** $\partial \mathcal{L} / \partial \mathbf{z} = \hat{\mathbf{y}} - \mathbf{y}$.
- **Backprop** = local gradient rules plus matrix multiplication, repeated layer by layer.
- **Depth needs engineering:** activation choice, initialization, residual connections, normalization.
- **Training loop:** forward → loss → zero_grad → backward → step.
- **Evaluation:** train / validation / test splits must match the deployment question.

Read before Lecture 2

Prince · Understanding Deep Learning — Ch 4 (deep networks), Ch 7 (gradients and initialization), Ch 11 (residual networks). Free PDF at udlbook.github.io.

Next lecture

Why depth, ResNets, Xavier / He initialization derived from first principles.

NOTEBOOK

1a · `01a-micrograd.ipynb` — scalar autograd engine from scratch (Karpathy-style).