AI and Deep Learning

Day 1: the whole day

Day 1 of 4

Fatih Kansoy
Worcester College, University of Oxford

What today covers

I Inputs and targets. What the model receives, what it must produce, and what it never sees.

II Units and layers. One weighted sum, repeated until it becomes a matrix.

III Nonlinearity and depth. Why an activation function is not optional.

IV Forward pass and loss. One example in, one number out.

V Backpropagation. How that one number reaches every parameter.

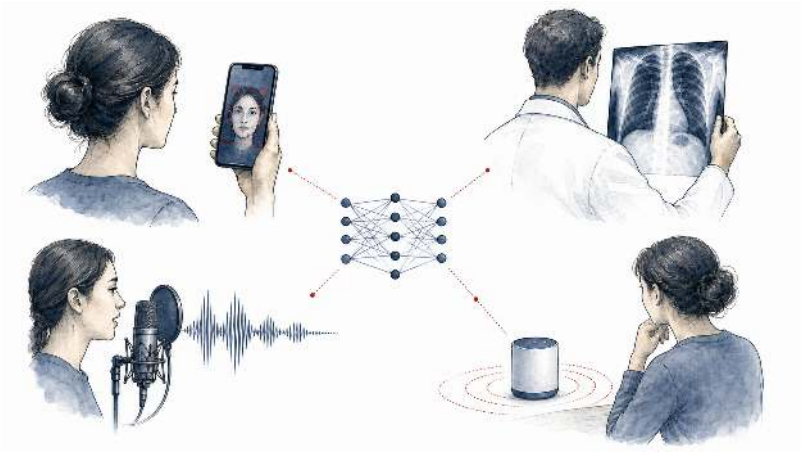
VI Generalisation. Why a lower training loss is not yet good news.

VII The whole system. Everything in one place, and the question Day 2 opens with.

> One photograph and one small network run through all seven. The numbers never restart.

Parts I to III build the object. Parts IV and V train it. Part VI asks whether to believe the result.

Where deep learning is already working



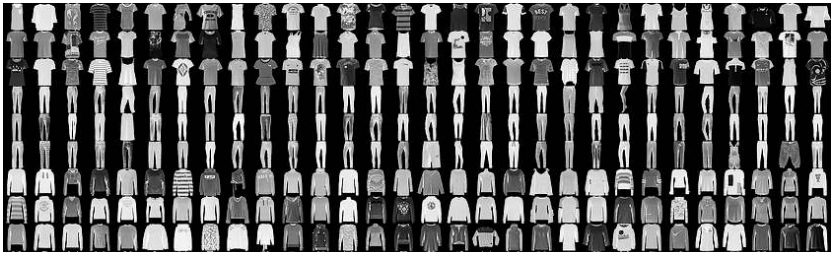
input data $\longrightarrow$ **learned representation** $\longrightarrow$ prediction

Part 0

The problem

A retailer, seventy thousand photographs, ten boxes.

The catalogue team has a problem

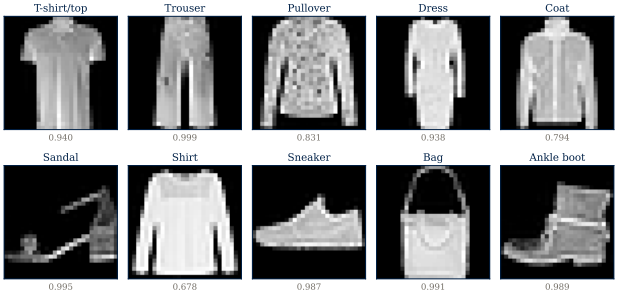


An online clothing retailer photographs every product. Before a product can go on the site, somebody looks at the photograph and files it: *T-shirt, trouser, ankle boot* … — ten boxes.

- Thousands of new products every week.
- Filing by hand is slow, boring, and inconsistent across staff.
- A misfiled product disappears from search — unsold stock, avoidable returns.

> **The task has three parts: a photograph as input, one of ten categories as output, and many correctly labelled examples.**

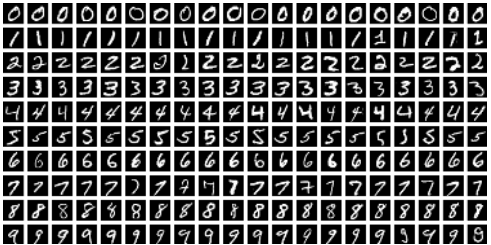
Seventy thousand photographs of clothes



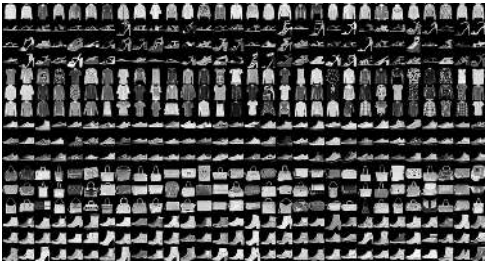
- They have an archive of 70,000 past photographs, each already filed by a human.
- Categories: T-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, ankle boot.
- Nobody wrote a rule for any of the ten names.
- **The task:** *can the archive do the filing? name the category of a photograph never seen.*

Fashion-MNIST: 70,000 small, grey, centred photographs of ten kinds of clothes, released by Zalando Research.

From MNIST to Fashion-MNIST



MNIST dataset example



Fashion-MNIST dataset example

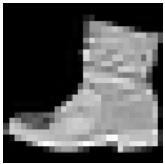
"Hello World" benchmark for beginners and experts testing computer vision and machine learning models.

First idea: write the rule

"A boot has a chunky sole and covers the ankle."

. . . what if the sole is slim or the laces are hidden or the photo is dark.

Humans recognise the object without stating a complete rule; software still needs an explicit procedure.



- **High tops?** — high-top trainers too.
- **Thin soles?** — flat sandals too.
- A fixed visual rule will fail on some of the 70,000 photographs.

> **Because appearances vary, the classification rule must be learned from many labelled examples.**

Variation within a single category

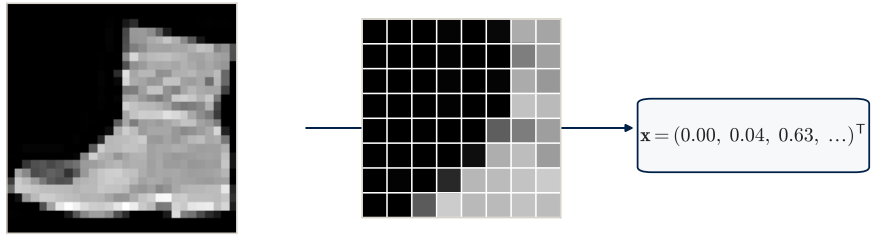
No rule written by hand survives all of this, and the next photograph is not in the picture.

Part I

Inputs, targets, and tensors

What the model receives, what it must produce, and what it never sees.

The input as a vector



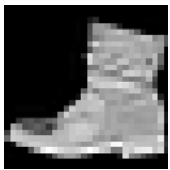
$$\boldsymbol{x} = (x_1, x_2, \ldots, x_{784})^\top \in \mathbb{R}^{784}, \qquad x_j = u_j/255 \in [0, 1]$$

one number per pixel, $28 \times 28 = 784$

the raw value u_j runs 0 to 255

The machine never sees a photograph — it sees 784 numbers, and flattening forgets which pixels were neighbours.

Encoding the category



the photograph

ankle boot

the observed category

$y = 9$

its number, 0 to 9

the only one of these four that cannot do arithmetic

$\boldsymbol{y} = (0, \ldots, 0, \mathbf{1})^\top$

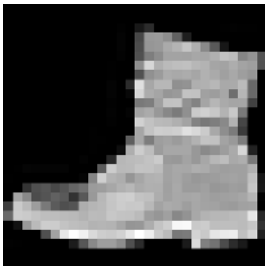
ten entries, the last is 1

All say “ankle boot”.

- Category 9 is not nine times category 1, not one more than category 8.

The target side is now settled. With the input from earlier, one example is the pair $(\boldsymbol{x}, \boldsymbol{y})$; next we stack many of them.

A labelled example



$\boldsymbol{x}$ the input

+

ankle boot

y the observed category

The pair $(\boldsymbol{x}, y)$ is one labelled example. Fashion-MNIST holds 70,000 of them in ten categories, and "ankle boot" is category 9.

Examples arrive stacked

- $\mathbf{X} \in \mathbb{R}^{B \times 784}$ — B photographs, 784 numbers each.
- Row i of both matrices is one example.
- $\mathbf{Y} \in \{0, 1\}^{B \times 10}$ — the same B labels.
- Row i of both matrices is one example.
- One arithmetic pass per batch.

Batch size B is a choice the experimenter makes — it is not a property of the data.

A batch of examples

$$\mathbf{X} = \begin{bmatrix} 0.00 & 0.04 & 0.63 & \cdots & 0.12 \\ 0.00 & 0.00 & 0.21 & \cdots & 0.00 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0.31 & 0.55 & 0.00 & \cdots & 0.07 \end{bmatrix}$$

$\mathbf{X} \in \mathbb{R}^{B \times 784}$
B examples × 784 pixels

$$\mathbf{Y} = \begin{array}{c} \text{category} \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} 0 & 1 & 2 & \cdots & 8 & 9 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & \mathbf{1} \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 1 & \cdots & 0 & 0 \end{bmatrix} \end{array}$$

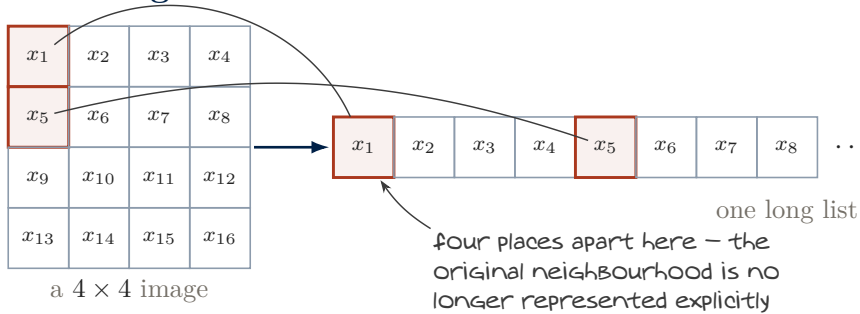
$\mathbf{Y} \in \{0, 1\}^{B \times 10}$
B examples × 10 categories

numbering starts at 0, so category 9 is the tenth column

- Row i of $\mathbf{X}$ and row i of $\mathbf{Y}$ are the same example. Here row 1 is our ankle boot.
- B is the **batch size**: how many examples are pushed through together. It is a choice, not a property of the data.
- The whole batch is processed in a single operation, far faster than B separate ones.

Inputs $\mathbf{X}$ and targets $\mathbf{Y}$ are ready. The next slide opens the machine that connects them, and names the numbers inside it that training is allowed to change.

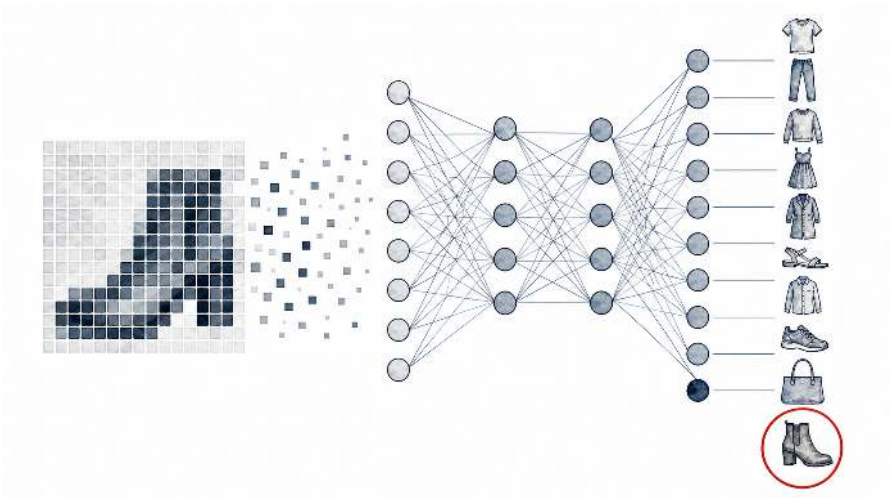
What flattening discards



- **Why flattening is used.** A dense layer multiplies a list by a matrix, so the grid has to become one line first.
- **What flattening loses.** x_1 sits directly above x_5 in the picture, yet four places apart in the list, and 28 apart for a real 28×28 image. Their original adjacency is no longer explicit.

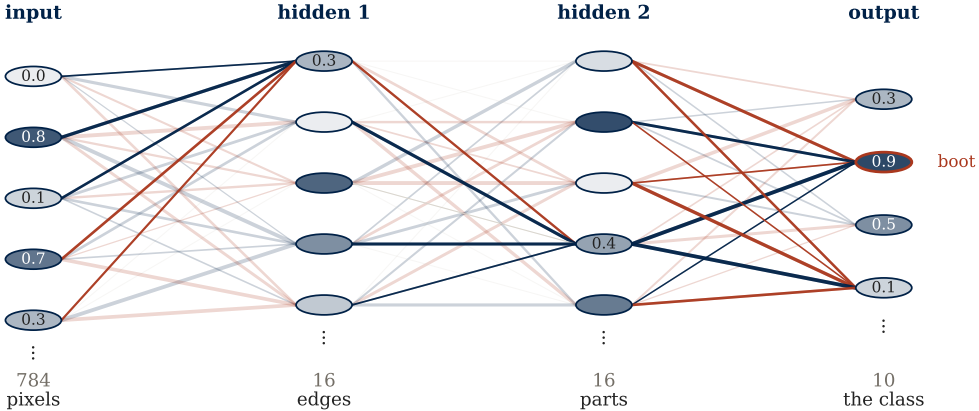
Day 2 puts the geometry back.

From pixels to a predicted category



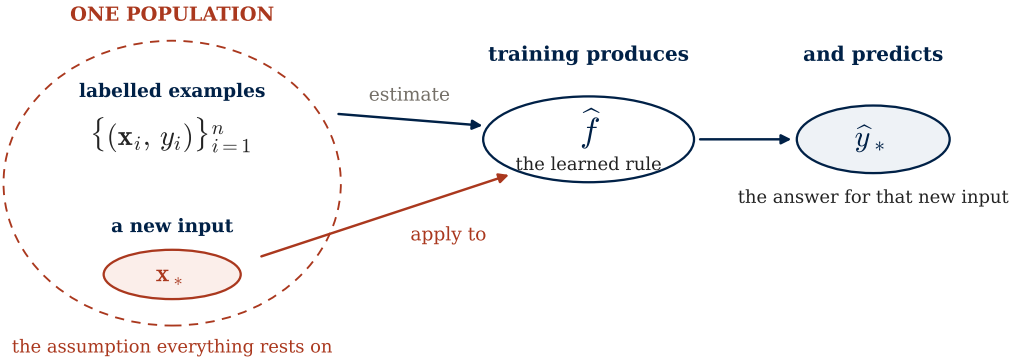
pixels $\longrightarrow$ learned representations $\longrightarrow$ class scores $\longrightarrow$ predicted class

A network is layers of neurons



each circle holds one number, its **activation**, how lit up that unit is; each line carries a **weight**; every unit adds its weighted inputs, shifts the total by its own **bias**, then applies an **activation function** (Part III)

The supervised learning problem



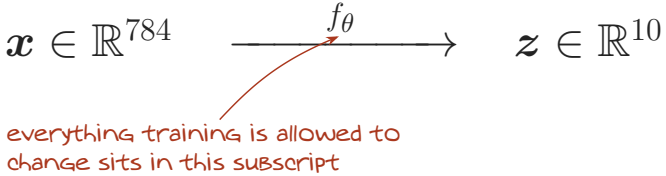
the new input $\boldsymbol{x}_*$

Doing well here is called **generalisation**. Not one of the examples used for fitting; any rule can memorise those, so they prove nothing on their own.

the same population

The dashed boundary is an assumption, and it is the first one to fail in practice. Part VI returns to both.

Architecture and parameters



The architecture: specified

- how many layers, and how many units in each
- which activation function (Part III)
- what connects to what

Specified before training begins.

The parameters θ: estimated

For a $784 \to 16 \to 16 \to 10$ network,

$\theta = (W_1, \boldsymbol{b}_1,\ W_2, \boldsymbol{b}_2,\ W_3, \boldsymbol{b}_3)$

shapes $16{\times}784,\ 16,\ 16{\times}16,\ 16,\ 10{\times}16,\ 10$

Each W is a matrix of weights, each $\boldsymbol{b}$ a vector of biases. Part II builds them from a single unit.

Estimated from the training data.

The ten outputs are scores; the reported category is the largest of them. Part IV turns the scores into probabilities.

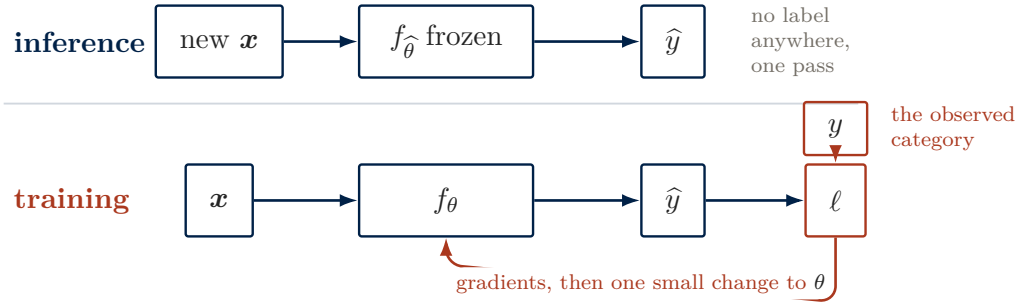
How many numbers is that?

$$\underbrace{16\times784+16}_{\text{layer 1}}+\underbrace{16\times16+16}_{\text{layer 2}}+\underbrace{10\times16+10}_{\text{layer 3}}=\boxed{13{,}002}$$

That is $12{,}560+272+170$: almost all of it in the first layer, where each of the 784 inputs meets each of the 16 units.

Thirteen thousand numbers, in a deliberately small network. Nobody sets these by hand, and finding them is what the rest of the day is about.

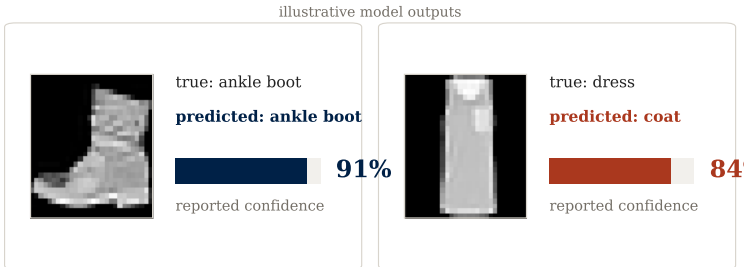
Training and inference



The forward pass is the same computation in both lanes; training only adds the label, the loss, and the loop back to the parameters.

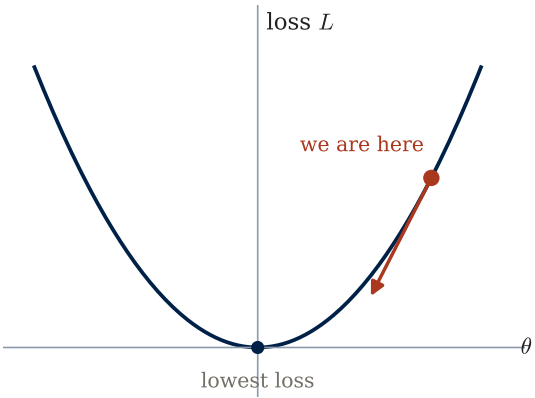
Interpreting the output score

A trained network outputs a predicted class and a **confidence** score. Neither guarantees correctness.



- Softmax produces the confidence, but a high value does not guarantee a correct answer.
- Interpreting 84% as “correct about 84 times in 100” requires calibration, studied in Part VI.
- Confidence does not measure whether the input resembles the training data.

The gradient: which way and how much



The loss measures the current prediction error. For each of the 13,002 parameters, the **gradient** gives two pieces of information.

- **sign** – the direction in which the parameter should change to reduce the loss.
- **magnitude** – how sensitive the loss is to a small change in that parameter.

Gradient descent updates all parameters a small step in the direction that reduces the loss, then repeats the calculation.

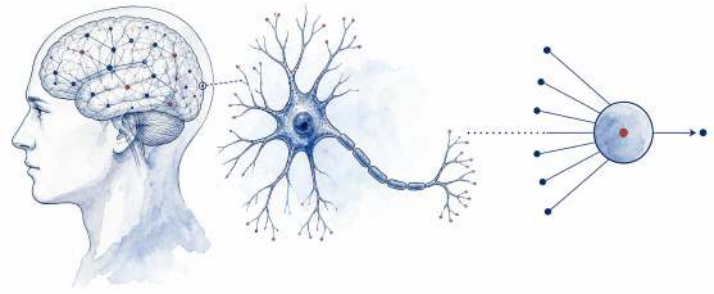
Computing all 13,002 slopes at once is what **backpropagation** does, in Part V.

Part II

Artificial neurons and dense layers

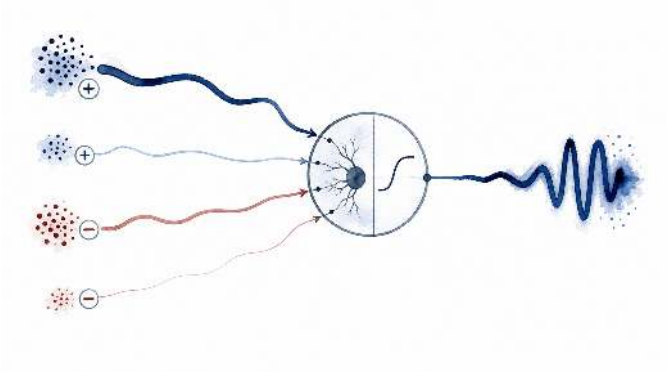
The whole network is one small calculation, repeated with different numbers.

Biological inspiration and its limits



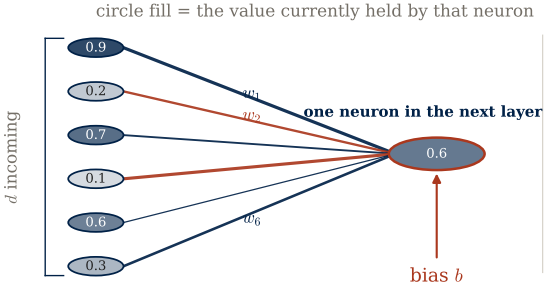
A brain cell collects signals from many other cells and fires only when they are strong enough. The units we build keep this one idea and borrow the name. But a real neuron is far more complicated, so this is where the comparison ends.

The unit as weighted evidence



Some inputs count for more than others, some count against, and the unit responds only when the balance is strong enough.

The artificial unit



1. weighted sum

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b$$

2. activation

$$a = g(z)$$

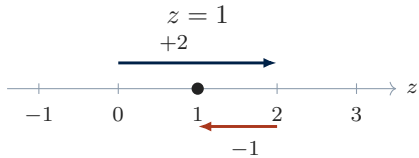
The number held by the target neuron is its activation.

$\boldsymbol{x}$ incoming activations · $\boldsymbol{w}$ learned weights · b learned bias · z pre-activation · a output activation

Worked example: one unit

$$\boldsymbol{x} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \qquad \boldsymbol{w} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \qquad b = 0$$

$$\begin{aligned} z &= \boldsymbol{w}^\top \boldsymbol{x} + b \\ &= \underbrace{(1)(2)}_{+2} + \underbrace{(-1)(1)}_{-1} + \underbrace{0}_{b} \\ &= \boxed{1} \end{aligned}$$



Input 1 pushes *for* by +2 and input 2 pushes *against* by −1; the balance lands at +1.

A weight's sign is the direction an input pushes and its size is how hard, and neither says anything about cause. Whether the unit passes this +1 forward is the activation's job, on the next slide.

ReLU: the rectified linear unit

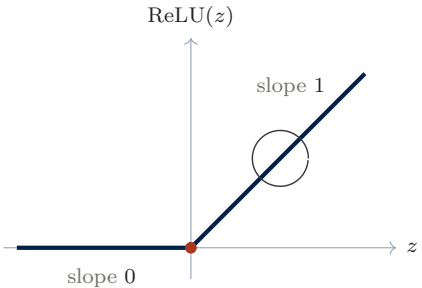
ReLU stands for the *rectified linear unit*: *rectified* because it keeps a positive input and blocks a negative one, the way a rectifier does to a current; *linear* because the positive part passes through unchanged; *unit* because this is just the neuron's own activation.

$$\mathrm{ReLU}(z) = \max(0, z)$$

A negative z gives output 0.

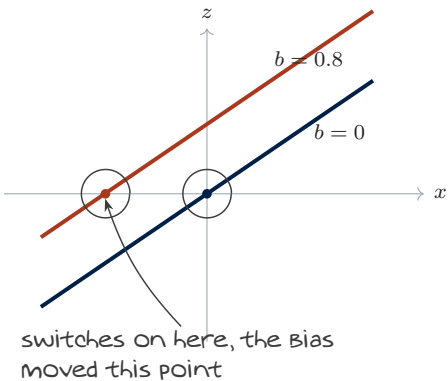
A positive z passes through unchanged.

Our unit had $z = 1$, so it passes on 1.



Part III asks why a nonlinear response is needed at all, and what the alternatives cost.

The role of the bias



$$z = wx + b$$

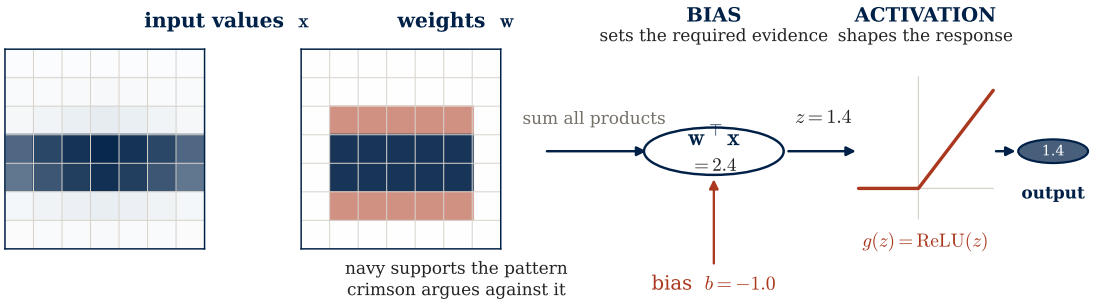
The weight w tilts the line; the bias b slides it up or down.

The unit fires when $z > 0$, so it switches on where the line crosses zero. Set $z = 0$ and solve: $x = -b/w$.

Raise the bias and that crossing slides left, so the unit fires on weaker evidence.

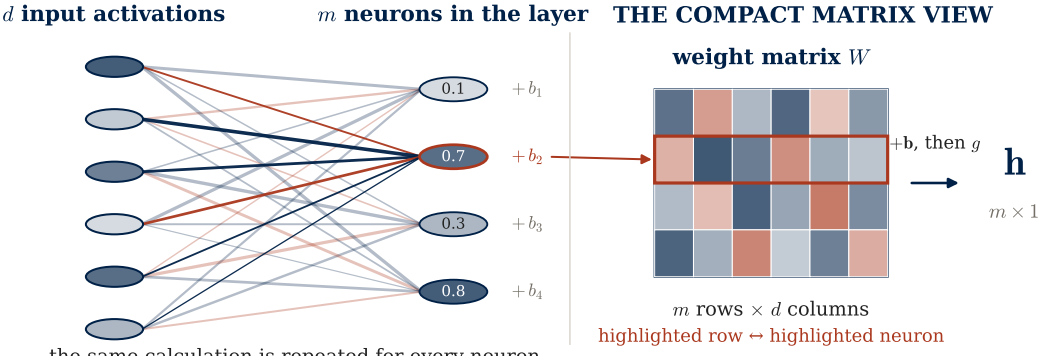
the bias is not the threshold, it moves the threshold

Weights, bias, and activation



The weights describe the pattern to detect; the bias controls how strong the match must be; the activation determines what value is passed forward.

From a unit to a dense layer



$$\underbrace{\boldsymbol{z}}_{m\times 1} = \underbrace{W}_{m\times d}\,\underbrace{\boldsymbol{x}}_{d\times 1} + \underbrace{\boldsymbol{b}}_{m\times 1}, \qquad \boldsymbol{h} = g(\boldsymbol{z})$$

row j of W is unit j's weights, and its scalar output a from before is now entry $h_j = g(z_j)$ of the stacked vector $\boldsymbol{h}$

Worked example: a layer of three units

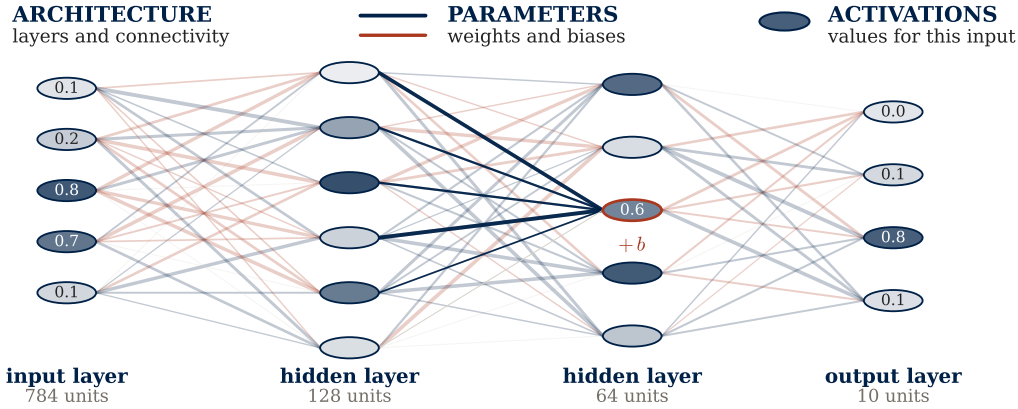
$$\boldsymbol{x} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \qquad W_1 = \begin{bmatrix} 1 & -1 \\ 0.5 & 0.5 \\ -1 & 2 \end{bmatrix}, \qquad \boldsymbol{b}_1 = \begin{bmatrix} 0 \\ 0.5 \\ -0.5 \end{bmatrix}$$

row 1 of W, entry 1 of $\boldsymbol{b}$ – this row alone computes $z_{1,1}$

$$\begin{aligned}
z_{1,1} &= 1(2) - 1(1) + 0 &&= 1 &&\longrightarrow h_1 = \mathrm{ReLU}(1) = 1 \\
z_{1,2} &= 0.5(2) + 0.5(1) + 0.5 &&= 2 &&\longrightarrow h_2 = \mathrm{ReLU}(2) = 2 \\
z_{1,3} &= -1(2) + 2(1) - 0.5 &&= -0.5 &&\longrightarrow h_3 = \mathrm{ReLU}(-0.5) = 0
\end{aligned}$$

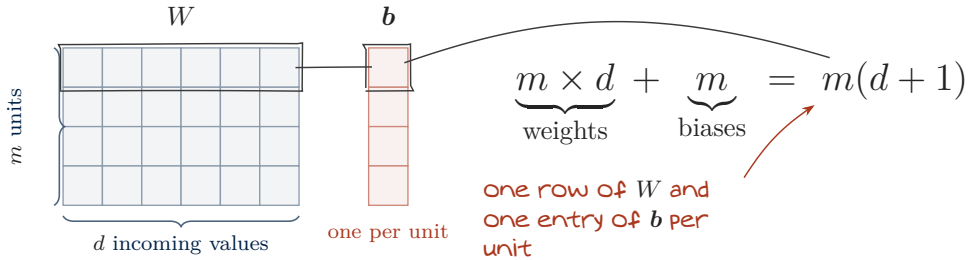
The red row reproduces the earlier single-unit calculation. Unit 3 is inactive for this input; a different input may activate it.

Architecture, parameters, and activations



Architecture fixes the layers and connections. Training estimates the weights and biases. Activations are recomputed for every input.

Counting a layer's parameters



Put a real layer through it: $m = 16$ units reading $d = 784$ pixels.

$$\underbrace{16 \times 784}_{\text{weights}} + \underbrace{16}_{\text{biases}} = 12{,}544 + 16 = \boxed{12{,}560}$$

One small layer, twelve and a half thousand numbers, none set by hand. That is why the rest of the day is about learning them.

Part III

Nonlinear activation functions and depth

A second layer adds something new only when a nonlinear activation separates it from the first.

Why add a second layer?

$$\boldsymbol{x} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \xrightarrow{W_1,\ \boldsymbol{b}_1} \boldsymbol{z}_1 = \begin{bmatrix} 1 \\ 2 \\ -0.5 \end{bmatrix} \xrightarrow{\text{ReLU}} \boldsymbol{h} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} \xrightarrow{W_2,\ \boldsymbol{b}_2} \boldsymbol{z}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

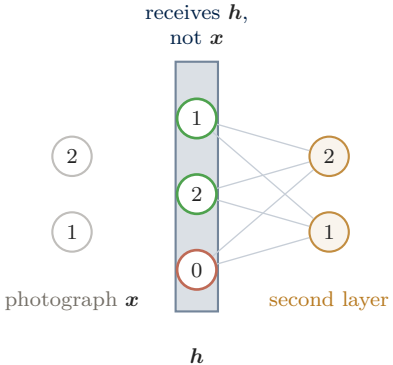
The first-layer computation from Part II remains unchanged; the second weight layer is the new component:

$$W_2 = \begin{bmatrix} 1 & 0.5 & -1 \\ 0 & 0.5 & 1 \end{bmatrix}, \qquad \boldsymbol{b}_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \qquad \boldsymbol{z}_2 = W_2\boldsymbol{h} + \boldsymbol{b}_2.$$

Has the network gained anything it could not do before?

First inspect the three numbers that enter layer 2. We keep these values fixed: Part IV turns $\boldsymbol{z}_2$ into probabilities, and Part V computes the gradients.

Layer 2 sees the hidden representation

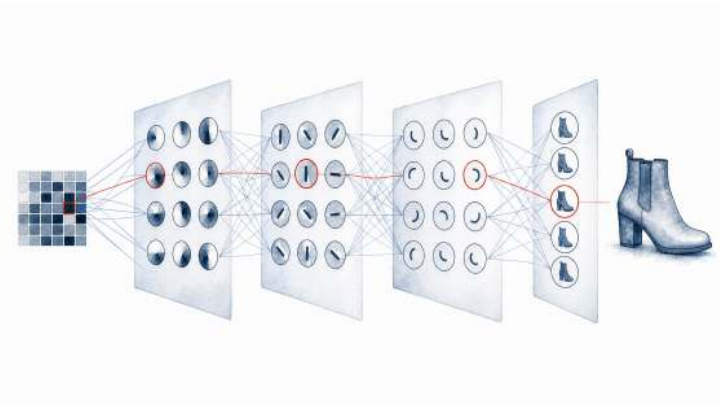


> $\boldsymbol{h}$ is a **representation**: a short numerical description of the photograph learned by the network.

- Only $\boldsymbol{h}$ passes to the second layer.
- Layer 2 receives three numbers: $(1, 2, 0)$.
- The original photograph is no longer directly available.

Later layers see only this description. It must keep the information needed for the task.

Each layer rewrites the input

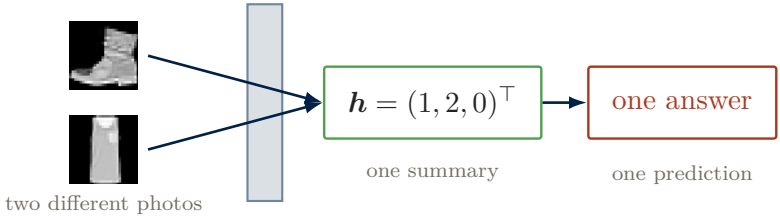


pixels $\longrightarrow$ **edges** $\longrightarrow$ **parts** $\longrightarrow$ a boot

Feature coordinates such as edges are learned rather than specified by hand. Each layer uses the representation produced below it.

Each layer must keep useful information. Once a layer removes it, later layers cannot recover it.

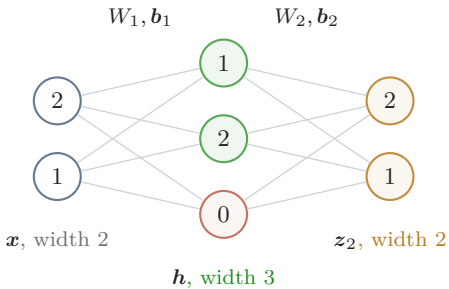
Later layers cannot recover lost information



- A layer knows only the summary $\boldsymbol{h}$.
- Same $\boldsymbol{h} \Rightarrow$ same answer — always.
- Merging two boot images may remove unimportant differences such as lighting.
- Merging a boot and a dress removes the class difference; later layers cannot undo it.
- **Training learns** what $\boldsymbol{h}$ keeps — Part V.

Training decides which information the representation keeps.

Depth and width



Depth counts the arrows: how many weight layers the signal passes through.

Width counts the units in one column.

This network has **depth** 2, from the two arrows W_1 and W_2, with **widths** $2, 3, 2$. The input and output widths are set by the problem; the hidden width 3 is a design choice.

Parameters are a third count: the weights and biases *inside* the arrows, $m(d+1)$ per layer.

$$\underbrace{3(2+1)}_{W_1,\ b_1} + \underbrace{2(3+1)}_{W_2,\ b_2} = 9 + 8 = \boxed{17}.$$

The activation changes what passes between layers

$$\boldsymbol{x} \xrightarrow{W_1, \boldsymbol{b}_1} \boldsymbol{z}_1 \xrightarrow{g} \boldsymbol{h} \xrightarrow{W_2, \boldsymbol{b}_2} \boldsymbol{z}_2$$

In the running example, g is ReLU, applied to each entry:

$$\boldsymbol{z}_1 = (1, 2, -0.5)^\top \xrightarrow{\text{ReLU}} \boldsymbol{h} = (1, 2, 0)^\top.$$

ReLU keeps the positive values 1 and 2, and replaces -0.5 with 0.

The weight layers mix their inputs. The activation decides which values pass to the next layer.

Identity activation leaves the signal unchanged

Replace ReLU with the **identity function**:

$$g(\boldsymbol{z}) = \boldsymbol{z} \qquad \text{for every } \boldsymbol{z}.$$

The identity activation passes its input through unchanged.

$$\boldsymbol{x} = (2,1)^\top \xrightarrow{W_1, \boldsymbol{b}_1} \boldsymbol{z}_1 = (1,2,-0.5)^\top \xrightarrow{g(\boldsymbol{z})=\boldsymbol{z}} \boldsymbol{h} = (1,2,-0.5)^\top$$

$$\boldsymbol{h} = (1,2,-0.5)^\top \xrightarrow{W_2, \boldsymbol{b}_2} \boldsymbol{z}_2 = (2.5,0.5)^\top.$$

The activation leaves $\boldsymbol{h} = \boldsymbol{z}_1$. The second weight layer still maps $\boldsymbol{h}$ to $\boldsymbol{z}_2$.

If the activation changes nothing, do we still need the second weight layer?

Equivalent networks agree for every input

Follow the calculation through both layers

$$\underbrace{\boldsymbol{z}_1 = W_1\boldsymbol{x} + \boldsymbol{b}_1}_{\text{first weight layer}}, \qquad \underbrace{\boldsymbol{h} = \boldsymbol{z}_1}_{\text{identity activation}}, \qquad \underbrace{\boldsymbol{z}_2 = W_2\boldsymbol{h} + \boldsymbol{b}_2}_{\text{second weight layer}}.$$

$$\Longrightarrow \qquad \boldsymbol{z}_2 = W_2 \underbrace{(W_1\boldsymbol{x} + \boldsymbol{b}_1)}_{\boldsymbol{h}=\boldsymbol{z}_1} + \boldsymbol{b}_2.$$

Matching the output for $\boldsymbol{x}_0 = (2, 1)^\top$ checks only one case. A one-layer replacement must match every possible input.

Two networks are **equivalent** when they give the same output for every input:

$$W_2(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 = \widetilde{W}\boldsymbol{x} + \widetilde{\boldsymbol{b}} \qquad \text{for every } \boldsymbol{x}.$$

Next, combine the old weights and biases into one new matrix and one new bias.

With identity activation, two layers become one

Multiply out and group the terms:

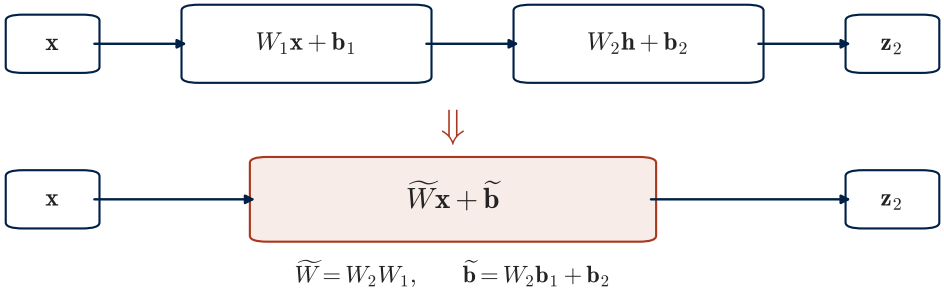
$$
\begin{aligned}
\boldsymbol{z}_2 &= W_2(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \\
&= W_2W_1\boldsymbol{x} + W_2\boldsymbol{b}_1 + \boldsymbol{b}_2 && \text{multiply out} \\
&= \underbrace{(W_2W_1)}_{\widetilde{W}}\boldsymbol{x} + \underbrace{(W_2\boldsymbol{b}_1 + \boldsymbol{b}_2)}_{\widetilde{\boldsymbol{b}}}.
\end{aligned}
$$

$$
\boxed{\widetilde{W} = W_2W_1} \qquad \boxed{\widetilde{\boldsymbol{b}} = W_2\boldsymbol{b}_1 + \boldsymbol{b}_2}
$$

▸ Full numerical derivation

With identity activation, the two layers do the same job as one matrix plus bias. ReLU stops this merge.

The same computation with one shorter network



two matrix-plus-bias steps $\equiv$ one matrix-plus-bias step for every $\boldsymbol{x}$

Nothing changes the signal between the two weight layers, so they can be combined.

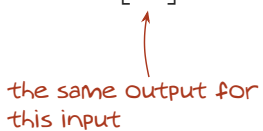
Checking the result with numbers

The combined layer has:

$$W_2 W_1 = \begin{bmatrix} 2.25 & -2.75 \\ -0.75 & 2.25 \end{bmatrix}, \qquad W_2 \boldsymbol{b}_1 + \boldsymbol{b}_2 = \begin{bmatrix} 0.75 \\ -0.25 \end{bmatrix}$$

Two layers, identity activation | **One layer, same input**

$$W_2(W_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 = \begin{bmatrix} 2.5 \\ 0.5 \end{bmatrix} \quad = \quad (W_2 W_1)\boldsymbol{x} + (W_2 \boldsymbol{b}_1 + \boldsymbol{b}_2) = \begin{bmatrix} 2.5 \\ 0.5 \end{bmatrix}$$



▸ Numerical parameter derivation

The numbers confirm one input. The algebra proves that the two forms match every input. ReLU prevents this merge.

A nonlinear activation makes depth useful

The identity case showed that weight layers alone can merge. A nonlinear activation keeps them separate.

If $g(\boldsymbol{z}) = \boldsymbol{z}$

$$W_2(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 = \widetilde{W}\boldsymbol{x} + \widetilde{\boldsymbol{b}}$$

The layers merge into one rule.

One layer is enough.

$\longrightarrow$

If $g = \mathrm{ReLU}$

$$W_2\,\mathrm{ReLU}(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$

Cannot generally merge.

Two layers can represent more complex functions.

Depth helps because ReLU changes the signal between the weight layers.

Activation and depth work together

A stack of weight layers can learn more when nonlinear activations separate the layers.

$$\underbrace{W_1\boldsymbol{x} + \boldsymbol{b}_1}_{\text{mix}} \xrightarrow{g} \underbrace{W_2\boldsymbol{h} + \boldsymbol{b}_2}_{\text{mix again}}$$

The activation's role

- Changes the signal with a nonlinear rule.
- Examples: ReLU, sigmoid, tanh.
- Stops neighboring weight layers from merging.

Depth's role

- Repeats this nonlinear change.
- Builds new features from earlier ones.
- Can represent more complex functions.

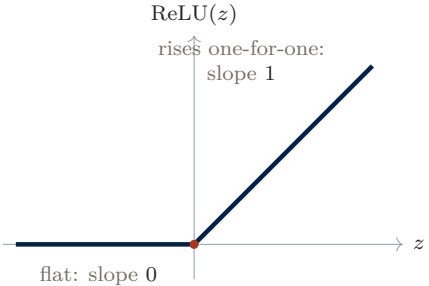
ReLU stops neighboring layers from merging; depth then combines several nonlinear steps.

ReLU uses two straight-line rules

ReLU uses one rule when $z \leq 0$ and another when $z > 0$.

$$\mathrm{ReLU}(z) = \max(0, z) = \begin{cases} 0 & z \leq 0, \\ z & z > 0. \end{cases}$$

- Negative input: ReLU blocks it.
- Positive input: ReLU passes it unchanged.



ReLU is **nonlinear**: no single straight-line rule describes both pieces across the whole real line.

The bend at $z = 0$ is why the two weight layers cannot be replaced by one matrix plus bias.

ReLU's derivative is 0 or 1

The **derivative** is the local slope: how much the output changes when z changes a little.

When $z < 0$

$$\mathrm{ReLU}(z) = 0 \quad \Longrightarrow \quad \mathrm{ReLU}'(z) = 0$$

The output is flat: changing z a little changes nothing.

When $z > 0$

$$\mathrm{ReLU}(z) = z \quad \Longrightarrow \quad \mathrm{ReLU}'(z) = 1$$

The output rises one-for-one: adding 0.1 to z adds 0.1 to the output.

$$\mathrm{ReLU}'(z) = \begin{cases} 0 & z < 0, \\ 1 & z > 0. \end{cases}$$

At $z = 0$, the two slopes meet and disagree, so there is no unique derivative. Implementations commonly use $\mathrm{ReLU}'(0) = 0$.

In backpropagation, slope 1 lets the gradient pass; slope 0 stops it.

Weights and bias produce the score z

The input supplies the values (x_1, x_2). The unit supplies its own weights and bias.

Input **Example parameters**

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad + \qquad \boldsymbol{w} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \qquad b = -1$$

$$\begin{aligned} z &= \boldsymbol{w}^\top \boldsymbol{x} + b \\ &= \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - 1 \\ &= \underbrace{1}_{w_1} x_1 + \underbrace{2}_{w_2} x_2 + \underbrace{(-1)}_{b}. \end{aligned}$$

The **weights** say how much each input counts; the **bias** shifts the total. Training learns these values.

> **The unit computes z first. ReLU then keeps a positive score or replaces it with zero.**

The same ReLU unit on three inputs

The same weights $(1, 2)$ and bias -1 are applied to each input point.

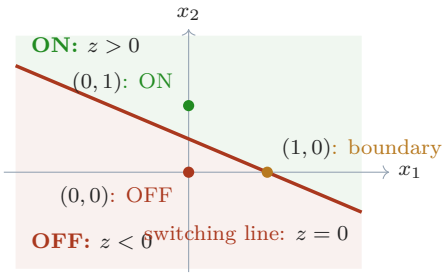
$$\underbrace{(x_1, x_2)}_{\text{one input point}} \longrightarrow \underbrace{z = x_1 + 2x_2 - 1}_{\text{weighted score}} \xrightarrow{\text{ReLU}} \underbrace{a = \text{ReLU}(z)}_{\text{unit's output}}$$

One unit, three possible input points

input (x_1, x_2)	score z	output a	state
$(0, 0)$	-1	0	**OFF**
$(1, 0)$	0	0	**boundary**
$(0, 1)$	1	1	**ON**

The input space contains all pairs (x_1, x_2). Every pair receives a score, and the pairs with $z = 0$ form the switching line.

One ReLU unit splits the input space



The switching line

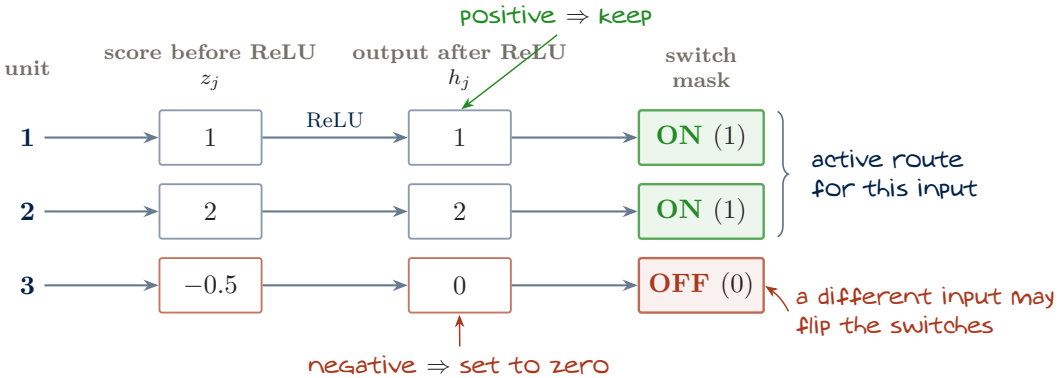
(using $w_1 = 1,\ w_2 = 2,\ b = -1$)

$$z = x_1 + 2x_2 - 1 = 0 \quad \Longrightarrow \quad x_1 + 2x_2 = 1.$$

- For $z < 0$, ReLU outputs 0; for $z > 0$, it outputs z.
- Weights tilt the line; the bias shifts it.

One ReLU unit creates one learned ON/OFF split. A layer combines many such splits.

Different inputs turn on different units



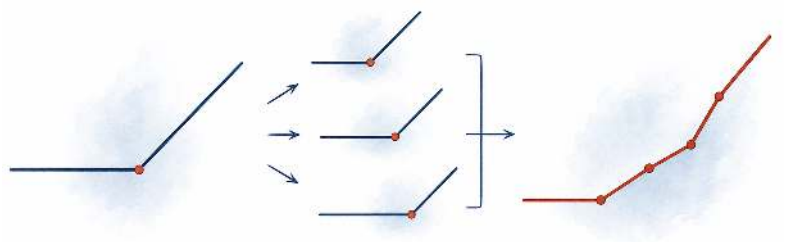
Mask $(1, 1, 0)$ means **ON, ON, OFF**. Each input has its own mask.

Prediction: OFF contributes nothing.

Training: OFF receives no gradient; always OFF means a **dead unit**.

Each input turns on its own set of units, creating a different route through the same network.

From one ReLU bend to many



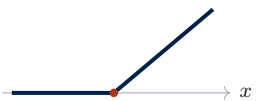
- One ReLU can change the function at one location.
- Many ReLUs can create changes at many locations.
- More bends let the network follow more complicated patterns.

Step 1: place bends at different locations

First give each hidden unit its own bend location.

- A general unit computes $h = \mathrm{ReLU}(wx + b)$.
- To isolate the role of the bias, the pictures below fix $w = 1$.
- Then $h = \mathrm{ReLU}(x + b)$, and the bend occurs where $x + b = 0$, or $x = -b$.

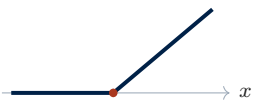
$b = +1$



bend at $x = -1$

$\mathrm{ReLU}(x + 1)$

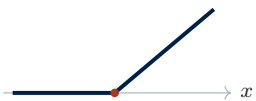
$b = 0$



bend at $x = 0$

$\mathrm{ReLU}(x)$

$b = -1$



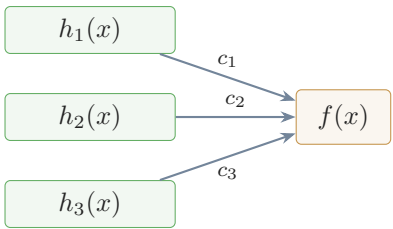
bend at $x = 1$

$\mathrm{ReLU}(x - 1)$

The biases are chosen by hand for this demonstration. During training, the network learns the weights and biases that place the bends.

Step 2: weight and add the bends

The next layer combines the separate hidden-unit outputs into one function.



1. Each unit has its own $h_j(x) = \text{ReLU}(w_j x + b_j)$.
2. The next layer weights these outputs by c_1, c_2, c_3.
3. It adds them and b_{out}.

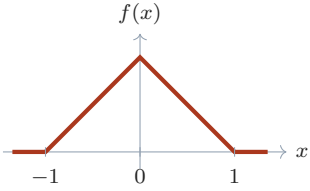
$$f(x) = b_{\text{out}} + c_1 h_1(x) + c_2 h_2(x) + c_3 h_3(x).$$

The result is **piecewise linear**: straight pieces joined at bends.

Training learns where the bends are through (w_j, b_j), and how to combine them through c_j.

Example: three bends form a triangle

Apply Step 1 and Step 2 with three hidden ReLU units.



- Hidden weights: $w_1 = w_2 = w_3 = 1$.
- Biases: $(b_1, b_2, b_3) = (+1, 0, -1)$, placing bends at $(-1, 0, 1)$.
- Next-layer weights: $(c_1, c_2, c_3) = (+1, -2, +1)$, making the total rise, fall, then flatten.

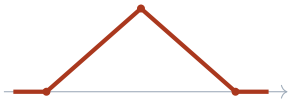
$$h_1 = \mathrm{ReLU}(x+1), \qquad h_2 = \mathrm{ReLU}(x), \qquad h_3 = \mathrm{ReLU}(x-1).$$

$$f(x) = h_1 - 2h_2 + h_3 = \mathrm{ReLU}(x+1) - 2\,\mathrm{ReLU}(x) + \mathrm{ReLU}(x-1).$$

We chose these weights by hand to show the mechanism. In a real problem, training learns the weights from data to create a useful output shape.

More ReLUs give the model more possible shapes

The triangle used three hidden units. A **wider** layer contains more ReLU units.



three units, three bends

make the layer wider

more bends, more detail

- Each added unit supplies another adjustable bend.
- Training learns (w_j, b_j) to place the bends and c_j to combine them.
- More units give training more possible output shapes to choose from.

> **More width increases capacity: what the network could learn. It does not guarantee that training will find a better function.**

How flexible can one wide hidden layer be if suitable weights are found?

A wide hidden layer can represent flexible functions

A wider layer provides more adjustable bends. With suitable weights, those bends can follow increasingly complicated targets.

> With enough ReLU units, one hidden layer can approximate any continuous function on a bounded input range as closely as desired.

- Each additional ReLU provides another possible bend.
- Many bends can follow the target more closely.
- This representation result is called **universal approximation**.

> **Universal approximation says what the architecture can represent when suitable weights exist.**

Capacity is not the same as learning

The theorem describes the network's **capacity**. Training is a separate problem.

Capacity: what is possible

- Suitable weights exist.
- More width provides more possible shapes.
- The target can be represented closely.

Learning: what training finds

- Training must find useful weights from data.
- More units do not guarantee an easier fit.
- They do not guarantee better results on new data.

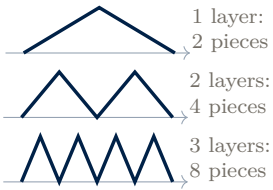
> **More ReLUs give the model more options. They do not automatically make the model learn better.**

If one wide layer can already be so flexible, why add more layers?

Depth can create complexity more efficiently

- **Width**: add more ReLU units to the same layer.
- **Depth**: stack another ReLU layer.
- **This example**: every layer contains two ReLUs.

hidden layers	total ReLUs	straight pieces
1	2	2
2	4	4
3	6	8
k	$2k$	2^k



- Each new layer adds two ReLUs, but doubles the straight pieces.

▸ The exact construction

Width adds complexity piece by piece; depth can multiply complexity by composing layers.

From flexible functions to classification

Results established so far

- Linear layers alone collapse to one linear rule.
- ReLU adds bends and nonlinear transformations.
- More layers can compose those transformations.

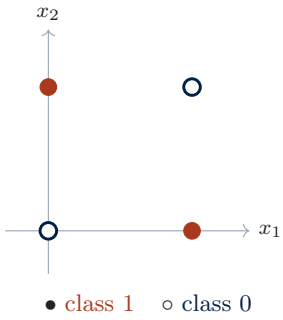
What this means for classification

- A linear rule gives one straight decision boundary.
- Some class patterns cannot be separated by one line.
- ReLU layers can reshape the inputs before the final boundary.

Place four points at the corners of a square and give opposite corners the same class. This pattern is called XOR.

Can one straight line separate the two classes?

A linear classifier uses one score and one boundary



- Use the same pre-activation notation as before:

$$z = w_1 x_1 + w_2 x_2 + b.$$

- $z > 0$ predicts class 1; $z < 0$ predicts class 0.
- $z = 0$ is the decision boundary—one straight line in the (x_1, x_2) plane.

The score z only decides which side of one line an input lies on. It does not reshape the inputs.

Can one such line separate the four XOR points?

The XOR requirements contradict one another

Evaluate the same score $z = w_1x_1 + w_2x_2 + b$ at all four inputs:

input	class	score z	required sign
$(0,0)$	0	b	$b < 0$
$(0,1)$	1	$w_2 + b$	$w_2 + b > 0$
$(1,0)$	1	$w_1 + b$	$w_1 + b > 0$
$(1,1)$	0	$w_1 + w_2 + b$	$w_1 + w_2 + b < 0$

Add the class-1 requirements

$$(w_2 + b) + (w_1 + b) = w_1 + w_2 + 2b > 0.$$

Add the class-0 requirements

$$b + (w_1 + w_2 + b) = w_1 + w_2 + 2b < 0.$$

The same expression cannot be positive and negative. Therefore one line cannot separate the XOR classes.

Adding more matrix-plus-bias layers does not help: they still collapse to one linear rule.

▸ See a two-layer example

Two ReLUs create a new representation

The hidden layer first computes two linear scores and then applies ReLU:

$$z_1 = x_1 + x_2, \qquad h_1 = \mathrm{ReLU}(z_1),$$
$$z_2 = x_1 + x_2 - 1, \quad h_2 = \mathrm{ReLU}(z_2).$$

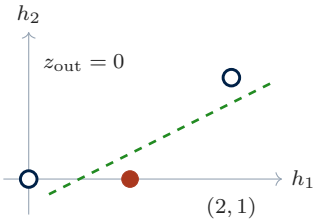
- (h_1, h_2) becomes the new representation of the input.
- ReLU clips negative scores to zero, so this transformation is not linear.

input	class	h_1	h_2	new point
$(0,0)$	0	0	0	$(0,0)$
$(0,1)$	1	1	0	$(1,0)$
$(1,0)$	1	1	0	$(1,0)$
$(1,1)$	0	2	1	$(2,1)$

The two class-1 inputs now land at the same hidden point $(1,0)$, while the class-0 inputs land elsewhere.

The ReLU layer has changed the geometry. Can one line separate the new points?

One line now separates the hidden points



The output layer uses another linear score:

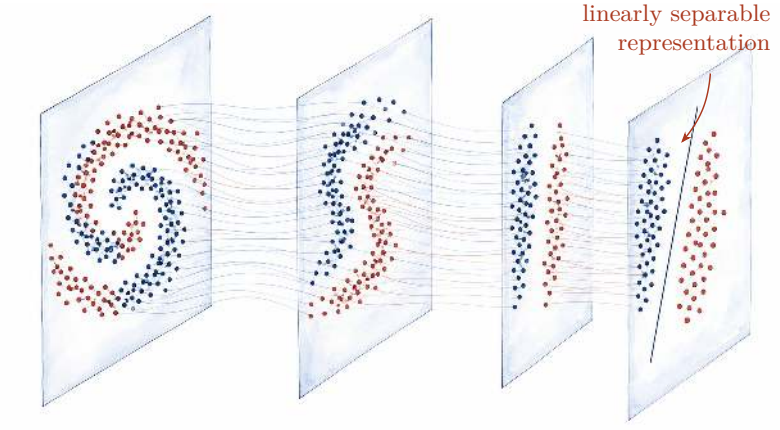
$$z_{\text{out}} = h_1 - 2h_2 - \tfrac{1}{2}.$$

hidden point	z_{out}	prediction
$(0, 0)$	$-\frac{1}{2}$	class 0
$(1, 0)$	$+\frac{1}{2}$	class 1
$(2, 1)$	$-\frac{1}{2}$	class 0

- $z_{\text{out}} > 0$ gives class 1.
- $z_{\text{out}} < 0$ gives class 0.

ReLU makes XOR separable by changing the representation. The final classifier still uses one straight boundary.

Layers can untangle two spirals



The spirals overlap so that no line separates them in the original coordinates. Each layer reshapes the representation until one line works.

ReLU worked—but is it the only choice?

In the XOR example, every hidden unit followed the same pattern:

$$\underbrace{z = \boldsymbol{w}^\top \boldsymbol{x} + b}_{\text{linear score}} \qquad \longrightarrow \qquad \underbrace{h = \mathrm{ReLU}(z)}_{\text{nonlinear feature}}.$$

ReLU did two important things:

1. It stopped the weight layers from collapsing into one linear rule.
2. It changed the representation until a straight boundary could work.

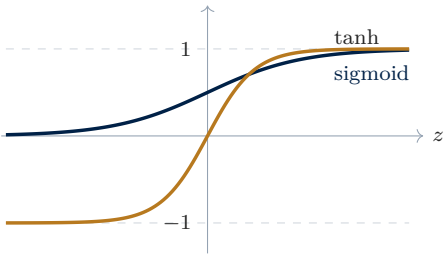
ReLU is one possible choice. Other nonlinear activations use

$$h = g(z), \qquad g \in \{\mathrm{ReLU},\ \sigma,\ \tanh, \ldots\}.$$

Which activation gives useful nonlinearity *and* allows the network to learn effectively?

Sigmoid and tanh also add nonlinearity

For the same score $z = \boldsymbol{w}^\top \boldsymbol{x} + b$, replace $h = \mathrm{ReLU}(z)$ by $h = \sigma(z)$ or $h = \tanh(z)$.



- **Sigmoid:** $\sigma(z) = \dfrac{1}{1 + e^{-z}}$, with outputs between 0 and 1.
- **Tanh:** a similar smooth curve, with outputs between -1 and 1.
- Both create smooth bends rather than ReLU's sharp bend.
- Because they are nonlinear, they also prevent linear layers from collapsing.

Sigmoid, tanh and ReLU can all reshape a representation. Their main practical difference is how they pass the learning signal.

Flat regions pass almost no learning signal

Training uses the slope $g'(z)$ to decide how much an earlier weight should change.

Near the centre

$$\sigma'(0) = \tfrac{1}{4}.$$

The curve changes with z, so a learning signal passes through.

In the flat tails

$$\sigma'(z) \approx 0.$$

The output barely changes, so earlier weights receive almost no signal.

Across several layers, these slopes are multiplied:

$$\text{signal to an early layer} \;\propto\; g_1'(z_1)g_2'(z_2)\cdots g_L'(z_L).$$

If several slopes are near zero, their product is near zero. This is **saturation**.

> **Sigmoid and tanh add useful nonlinearity, but saturation can make deep hidden layers learn slowly. ReLU has slope (1) on its active side.**

Choose the activation for the layer's job

Hidden layers: build features

- **ReLU** is a strong default: simple, nonlinear and slope 1 when active.
- **Leaky ReLU or GELU** are alternatives when ordinary ReLUs remain inactive or another smooth shape works better.
- **Sigmoid and tanh** can be used, but may saturate in deep hidden layers.

Output layer: match the prediction

- **Sigmoid**: one probability for a two-class outcome.
- **Softmax**: probabilities across several classes.
- **Tanh**: an output deliberately restricted to $(-1, 1)$.

The hidden activation shapes what the network can learn; the output activation gives the prediction the required form.

Part IV introduces softmax and shows how output scores become probabilities.

Part III: main lessons

1. **Depth needs nonlinearity.** Without an activation, any stack of matrix-plus-bias layers collapses into one such rule.

2. **Activations reshape the representation.** ReLU, sigmoid and tanh allow the network to create bends so that simpler boundaries can work.

3. **Width and depth create complexity differently.** Width adds bends piece by piece; depth can multiply complexity by composing layers.

4. **Representation is not the same as learning.** The activation's slope affects whether useful signals reach earlier layers.

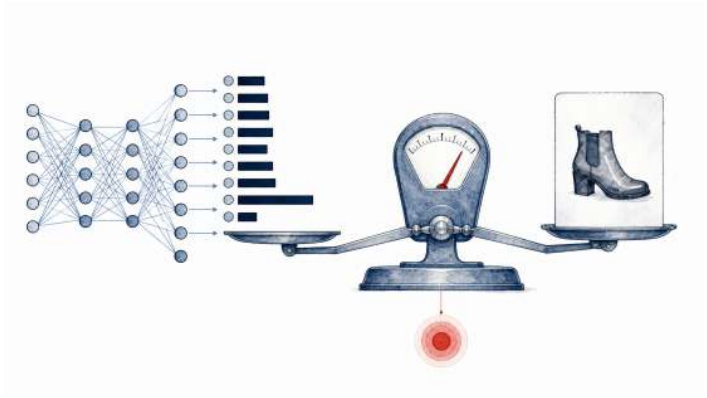
Part IV takes the network's final scores and turns them into probabilities and one loss value that training can reduce.

Part IV

Forward propagation and loss functions

One example produces one loss value. Training adjusts the parameters to reduce it.

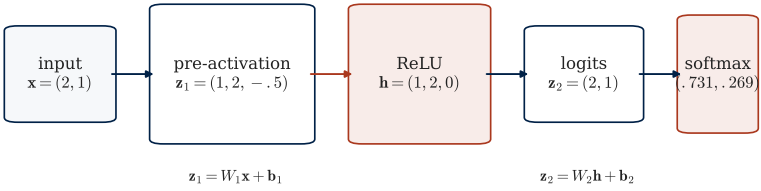
The forward pass ends with a loss



model scores + observed label $\longrightarrow$ **one loss value**

The loss measures the disagreement between the prediction and the observed label. Training adjusts the parameters to reduce it.

The complete forward pass has four stages



1. The first matrix produces hidden scores $\boldsymbol{z}_1$.
2. ReLU produces the hidden representation $\boldsymbol{h}$.
3. The second matrix produces class scores $\boldsymbol{z}_2$.
4. Softmax and cross-entropy turn those scores into one loss value.

Part III explained how the hidden representation is built. Part IV completes the calculation from class scores to a training signal.

Read W_1 one row at a time

$$\boldsymbol{z}_1 = W_1\,\boldsymbol{x} + \boldsymbol{b}_1 = \begin{bmatrix} 1 & -1 \\ 0.5 & 0.5 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.5 \\ -0.5 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ -0.5 \end{bmatrix}$$

W: weights $\quad$ $\boldsymbol{x}$: input $\quad$ $\boldsymbol{b}$: bias $\quad$ the result $\boldsymbol{z}_1$

$$z_{1,1} = 1(2) - 1(1) + 0 = 1$$

$$z_{1,2} = 0.5(2) + 0.5(1) + 0.5 = 2$$

$$z_{1,3} = -1(2) + 2(1) - 0.5 = -0.5$$

Each *row* of W_1 is one unit's weights, and each entry of $\boldsymbol{b}_1$ is that unit's bias. The input $\boldsymbol{x}$ is shared by all three. Three rows, three units, three numbers out, all at once.

The shapes check: $(3 \times 2)(2 \times 1) + (3 \times 1) = (3 \times 1)$.

Pre-activation and representation are different objects

$$\underbrace{\boldsymbol{z}_1 = \begin{bmatrix} 1 \\ 2 \\ -0.5 \end{bmatrix}}_{\text{pre-activation}} \xrightarrow{\text{ReLU, entry by entry}} \underbrace{\boldsymbol{h} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}}_{\text{representation}}$$

$\boldsymbol{z}_1$ is what the weights produced. $\boldsymbol{h}$ is what the layer passes on. They differ in exactly the entries ReLU switched off, and Part V will need both of them, separately.

This distinction is essential for backpropagation: $\boldsymbol{z}_1$ determines which ReLU units can pass a gradient.

The second matrix produces one score per class

$$\boldsymbol{z}_2 = W_2\,\boldsymbol{h} + \boldsymbol{b}_2 = \begin{bmatrix} 1 & 0.5 & -1 \\ 0 & 0.5 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

W: weights $\quad$ $\boldsymbol{h}$: hidden features $\quad$ $\boldsymbol{b}$: bias $\quad$ the result $\boldsymbol{z}_2$

$$z_{2,1} = 1(1) + 0.5(2) - 1(0) + 0 = 2$$
$$z_{2,2} = 0(1) + 0.5(2) + 1(0) + 0 = 1$$

Row j of W_2 says how strongly each hidden feature $\boldsymbol{h}$ counts towards class j. The two outputs are the **logits**: scores on no particular scale, not yet probabilities.

the top score 2 is class 1, the bottom score 1 is class 2

Logits rank classes, but they are not probabilities

The final matrix produced two **logits**:

$$z_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

What the logits tell us

- Class 1 has the larger score.
- The model therefore prefers class 1.
- The score gap is $2 - 1 = 1$.

Why they are not probabilities

- Logits can be negative or larger than 1.
- Here they sum to 3, not 1.
- Their scale has no direct probability meaning.

Before comparing the prediction with the label, convert the logits into positive numbers that sum to one while preserving their order.

Softmax converts scores in two steps

Start with any vector of logits $\boldsymbol{z} = (z_1, \ldots, z_K)$.

1. **Exponentiate each score:**

$$a_j = e^{z_j}.$$

   Every a_j is positive, and larger logits remain larger.

2. **Divide by their total:**

$$p_j = \frac{a_j}{\sum_k a_k} = \frac{e^{z_j}}{\sum_k e^{z_k}}.$$

   The resulting probabilities satisfy $p_j > 0$ and $\sum_j p_j = 1$.

Softmax does not choose a class by itself. It expresses the model's relative preference for every class as a probability distribution.

Softmax depends only on differences between logits

These three logit vectors have the same gap between class 1 and class 2:

$$(2, 1), \qquad (102, 101), \qquad (0, -1).$$

Therefore all three produce the same probabilities:

$$\boldsymbol{p} = (0.731,\ 0.269).$$

The algebra shows why. Adding the same constant c to every logit gives

$$\frac{e^{z_j+c}}{\sum_k e^{z_k+c}} = \frac{e^c e^{z_j}}{e^c \sum_k e^{z_k}} = \boxed{\frac{e^{z_j}}{\sum_k e^{z_k}}}.$$

Softmax measures how the classes compare with one another. A common shift changes none of those comparisons.

Apply softmax to the two logits

1. Exponentiate

$$\begin{bmatrix}2\\1\end{bmatrix} \longrightarrow \begin{bmatrix}e^2\\e^1\end{bmatrix} = \begin{bmatrix}7.389\\2.718\end{bmatrix}.$$

The values are positive and keep the same order.

2. Normalize

$$7.389 + 2.718 = 10.107,$$

$$\boldsymbol{p} = \begin{bmatrix}7.389/10.107\\2.718/10.107\end{bmatrix} = \begin{bmatrix}0.731\\0.269\end{bmatrix}.$$

- Both entries are positive and they sum to 1.
- The larger logit remains the larger probability.

A one-point logit advantage becomes a 73% versus 27% probability split.

Prediction and label play different roles

Model prediction

$$\boldsymbol{p} = \begin{bmatrix} 0.731 \\ 0.269 \end{bmatrix}.$$

The largest probability is for class 1, so the model predicts

class 1.

Observed label

$$\boldsymbol{y} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

The 1 marks the observed category, so the correct answer is

class 2.

The prediction is computed from $\boldsymbol{x}$, the weights and the biases.

The label is then used to evaluate that prediction.

> **Softmax tells us what the model believes. The label tells us which probability should have been large.**

The one-hot label selects the correct-class probability

Multiply each predicted probability by the matching entry of the label:

class	predicted p_j	label y_j	$y_j p_j$
1	0.731	0	0
2	0.269	1	0.269

The zero removes the probability for class 1; the one keeps the probability for the observed class:

$$p_{\text{true}} = \sum_j y_j p_j = \boldsymbol{y}^\top \boldsymbol{p} = \boxed{0.269}.$$

The label enters the forward calculation here. During inference there is no label, so the calculation stops at the predicted probabilities.

Cross-entropy turns probability into a penalty

The loss should be small when p_{true} is large and large when p_{true} is small. Cross-entropy uses

$$\ell = -\log p_{\text{true}}.$$

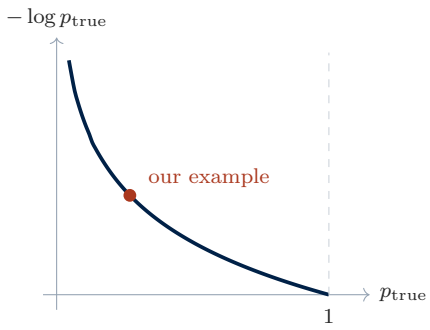
For this example,

$$\ell = -\log(0.269) = \boxed{1.313}.$$

$$\begin{aligned} p_{\text{true}} = 1 \quad &\Rightarrow \quad \ell = 0 \\ p_{\text{true}} = 0.269 \quad &\Rightarrow \quad \ell = 1.313 \\ p_{\text{true}} = 0.01 \quad &\Rightarrow \quad \ell = 4.605 \end{aligned}$$

High probability on the correct class gives a small loss; low probability gives a large loss.

The logarithm gives the loss useful properties



1. The loss is 0 at $p_{\text{true}} = 1$ and grows without bound as $p_{\text{true}} \to 0$. Confident errors receive the largest penalty.

2. Products of probabilities become sums of losses, which can be averaged across a batch.

3. Together with softmax, cross-entropy gives a simple gradient for the output logits.

For the worked example, $p_{\text{true}} = 0.269$ gives $\ell = 1.313$.

Accuracy treats every wrong prediction alike

	logits z_2	p	predicted	correct?	loss
near miss	(1.0, 0.9)	(0.525, 0.475)	class 1	**no**	0.744
our example	(2.0, 1.0)	(0.731, 0.269)	class 1	**no**	1.313
confident error	(3.0, 0.0)	(0.953, 0.047)	class 1	**no**	3.049

- Accuracy assigns all three examples the same result: incorrect.
- Cross-entropy distinguishes a near miss from a confident error.
- Unlike accuracy, cross-entropy changes smoothly as the logits change.

Accuracy summarizes final decisions. Cross-entropy supplies a differentiable signal for improving the parameters.

Average the example losses to obtain the objective

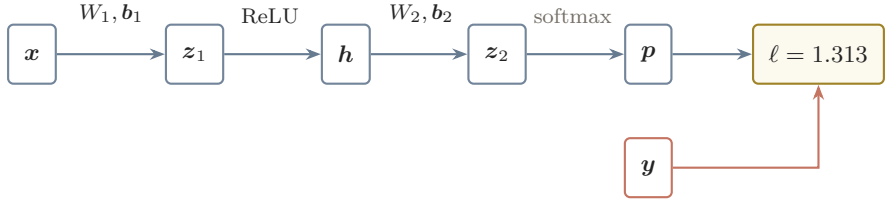
$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell_i = -\frac{1}{n} \sum_{i=1}^{n} \log p_{i,\, y_i}$$

Each example produces one loss ℓ_i. Their average is the objective used to evaluate the current parameters on a batch of n examples.

> θ denotes the network's 17 weights and biases. With the data held fixed, training changes θ in order to reduce $L(\theta)$.

Using the mean rather than the sum keeps the scale of the objective comparable across different batch sizes.

The loss completes the forward computation



Every arrow represents a differentiable calculation, and the chain ends at one loss value.

Part V follows this graph backwards and computes how the loss changes with each of the 17 parameters.

Part IV: main lessons

1. **Logits are relative class scores.** Their order indicates the preferred class, but they are not probabilities.

2. **Softmax normalizes the logits.** It produces positive probabilities that sum to one and depend only on score differences.

3. **The label selects the correct-class probability.** In the example, class 2 selects $p_{\text{true}} = 0.269$.

4. **Cross-entropy produces one penalty.** $\ell = -\log p_{\text{true}} = 1.313$ for this example.

5. **Training minimizes the average loss.** This connects the forward computation to parameter updates.

The full chain is now complete: $\boldsymbol{x} \to \boldsymbol{z}_1 \to \boldsymbol{h} \to \boldsymbol{z}_2 \to \boldsymbol{p} \to \ell$.

Appendix

Appendix

Optional derivations that support the main results.

Appendix: combine the two weight matrices

With identity activation $g(z) = z$, substitute the first layer into the second:

$$\boldsymbol{z}_2 = W_2(W_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 = \underbrace{(W_2W_1)}_{\widetilde{W}}\boldsymbol{x} + \underbrace{(W_2\boldsymbol{b}_1 + \boldsymbol{b}_2)}_{\widetilde{\boldsymbol{b}}}.$$

For the matrices in the running example,

$$W_1 = \begin{bmatrix} 1 & -1 \\ 0.5 & 0.5 \\ -1 & 2 \end{bmatrix}, \qquad W_2 = \begin{bmatrix} 1 & 0.5 & -1 \\ 0 & 0.5 & 1 \end{bmatrix}.$$

Each entry of W_2W_1 is a row–column dot product. For example,

$$\widetilde{W}_{11} = (1)(1) + (0.5)(0.5) + (-1)(-1) = 2.25.$$

Applying the same calculation to all four entries gives

$$\widetilde{W} = W_2W_1 = \begin{bmatrix} 2.25 & -2.75 \\ -0.75 & 2.25 \end{bmatrix}.$$

▸ Combine the biases

Appendix: combine the two bias vectors

The running example uses

$$\boldsymbol{b}_1 = \begin{bmatrix} 0 \\ 0.5 \\ -0.5 \end{bmatrix}, \qquad \boldsymbol{b}_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Multiplying and adding gives

$$\widetilde{\boldsymbol{b}} = W_2 \boldsymbol{b}_1 + \boldsymbol{b}_2 = \boxed{\begin{bmatrix} 0.75 \\ -0.25 \end{bmatrix}}.$$

Combining the matrix and bias results,

$$\boldsymbol{z}_2 = \underbrace{\begin{bmatrix} 2.25 & -2.75 \\ -0.75 & 2.25 \end{bmatrix}}_{\widetilde{W}} \boldsymbol{x} + \underbrace{\begin{bmatrix} 0.75 \\ -0.25 \end{bmatrix}}_{\widetilde{\boldsymbol{b}}}.$$

For $\boldsymbol{x} = (2, 1)^\top$, both the original two layers and this single matrix-plus-bias rule give $(2.5, 0.5)^\top$.

Back to the main slide

Appendix: two affine layers still make one line

Suppose there is no activation between these two matrix-plus-bias layers.

First layer

$$h_1 = x_1 + x_2, \qquad h_2 = 2x_1 - x_2 + 1.$$

Second layer

$$z = 3h_1 - 2h_2 + 1.$$

Substitute the first-layer formulas into the second:

$$\begin{aligned} z &= 3(x_1 + x_2) - 2(2x_1 - x_2 + 1) + 1 \\ &= -x_1 + 5x_2 - 1. \end{aligned}$$

- The two layers reduce to one matrix-plus-bias rule.
- Its boundary, $-x_1 + 5x_2 - 1 = 0$, is still one straight line.
- Further layers without an activation collapse in the same way.

> **Extra affine layers can change the line, but they cannot create a nonlinear boundary.**

◂ Back to the XOR slide

Appendix: two ReLUs make one tent

Work on the bounded input range $0 \le x \le 1$. Start with two hidden ReLU units:

$$h_1(x) = \mathrm{ReLU}(x), \qquad h_2(x) = \mathrm{ReLU}(x - \tfrac{1}{2}).$$

Combine them at the output:

$$T(x) = 2h_1(x) - 4h_2(x) = 2\,\mathrm{ReLU}(x) - 4\,\mathrm{ReLU}(x - \tfrac{1}{2}).$$

- For $0 \le x \le \frac{1}{2}$, only h_1 is active, so $T(x) = 2x$: the line rises from 0 to 1.
- For $\frac{1}{2} \le x \le 1$, both units are active, so $T(x) = 2 - 2x$: the line falls from 1 to 0.

One layer with two ReLUs creates a tent with two straight pieces.

▸ Why the pieces double

Appendix: depth doubles the pieces

Feed the output $T(x)$ into another copy of the same two-ReLU construction:

$$\underbrace{T(x)}_{2\text{ pieces}} \longrightarrow \underbrace{T(T(x))}_{4\text{ pieces}} \longrightarrow \underbrace{T(T(T(x)))}_{8\text{ pieces}}.$$

- Each existing piece runs through the rising and falling sides of the next tent.
- Therefore every new layer doubles the number of straight pieces.
- Each new layer costs only two additional ReLUs.

hidden layers	total ReLUs	straight pieces
1	2	2
2	4	4
3	6	8
k	$2k$	$\boxed{2^k}$

With k layers, this network uses $2k$ ReLUs and creates 2^k straight pieces. A one-hidden-layer network would need at least $2^k - 1$ ReLUs to create that many pieces.

Back to the slide