USC University of Southern California

# ECE 105: Introduction to Electrical Engineering

Lecture 19-20

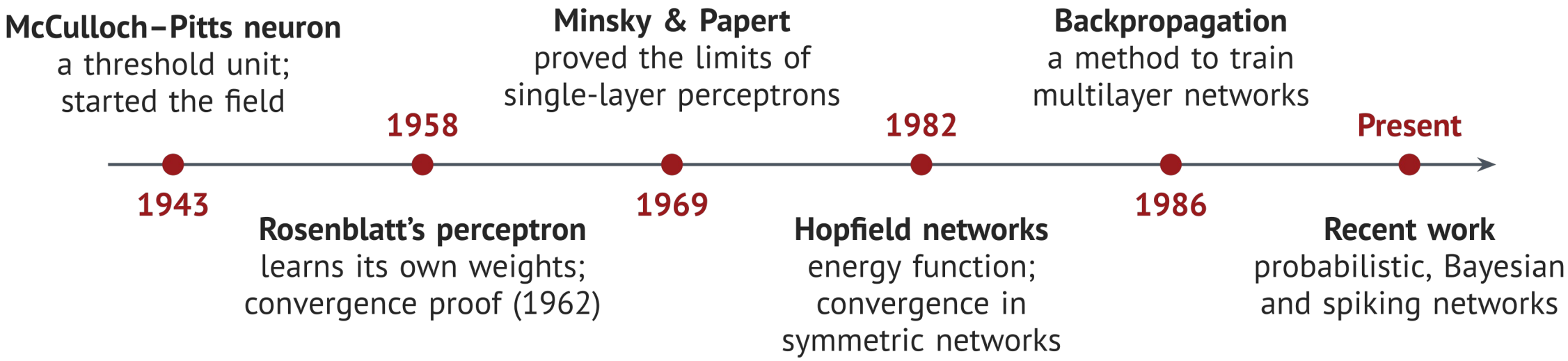
Neural Networks

Yasser Khan

Rehan Kapadia

# Timeframe

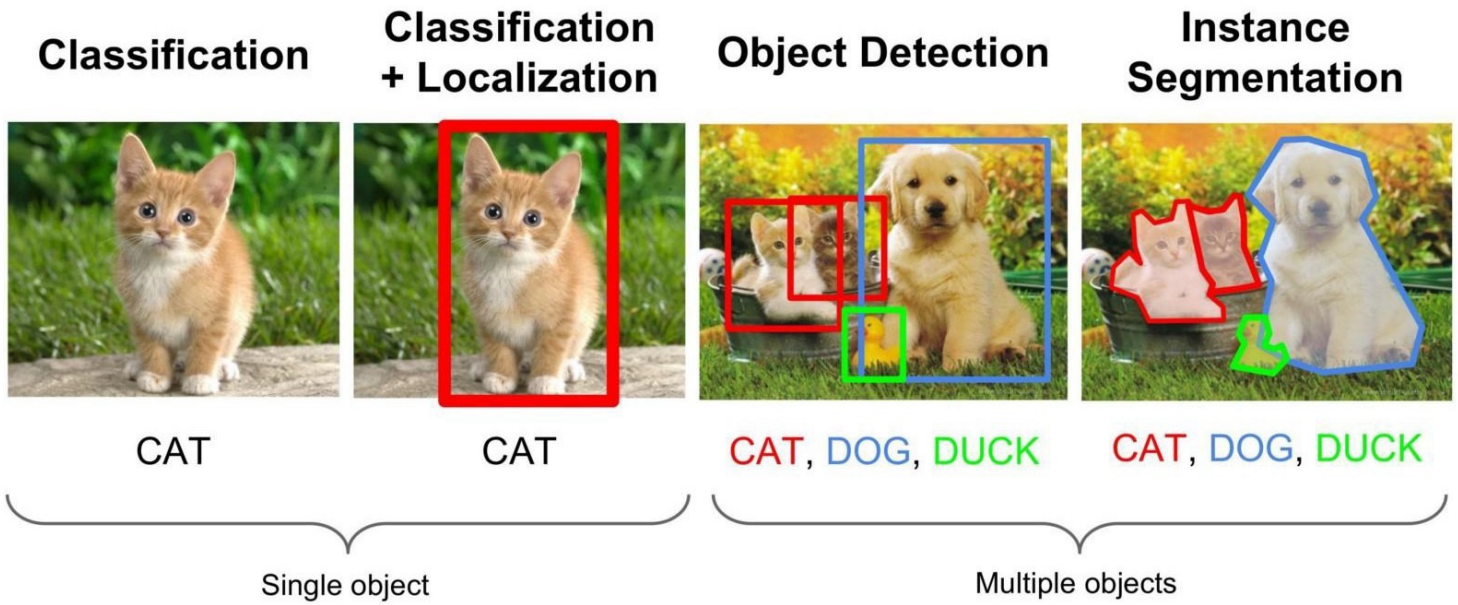
From a 1943 model neuron to backpropagation in 1986: the ideas behind today's networks.

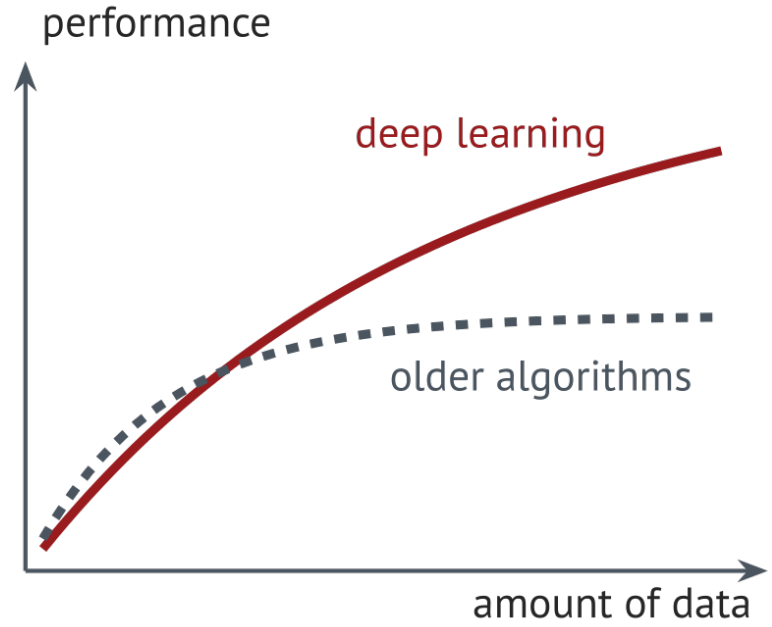



**The thread of these lectures:** one neuron (1943), a perceptron that learns (1958), its XOR limit (1969), and the fix: hidden layers trained by backpropagation (1986).

# Why Deep Neural Networks?

Deep networks excel at image and speech recognition, and they keep improving as data grows.




Qualitative trend: older methods level off as data grows; deep networks keep climbing.

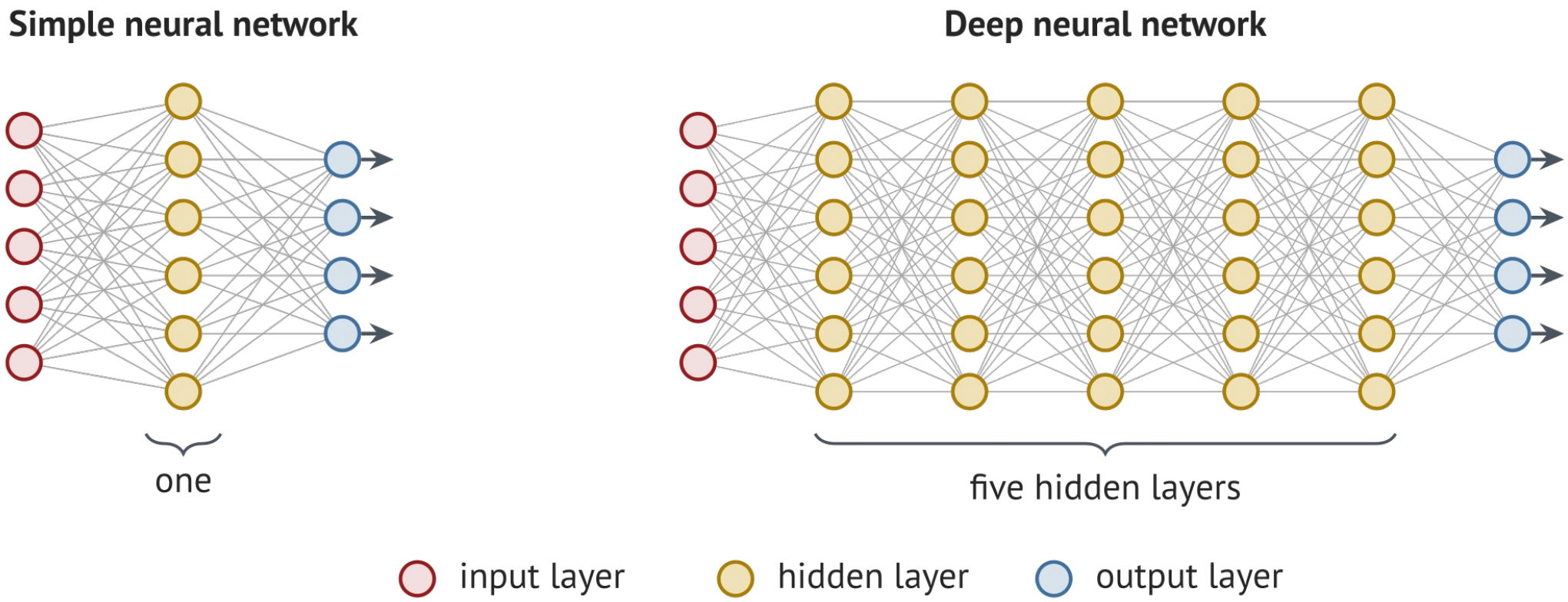
- Deep neural networks are excellent for some classes of problems: **image recognition**, **speech recognition** and more
- As the amount of data grows, the performance of deep learning keeps improving

Image from Medium: “Review of Deep Learning Algorithms for Object Detection”

# What's so deep about Deep Neural Networks?

USC University of Southern California

"Deep" means many layers of neurons between the input layer and the output layer.



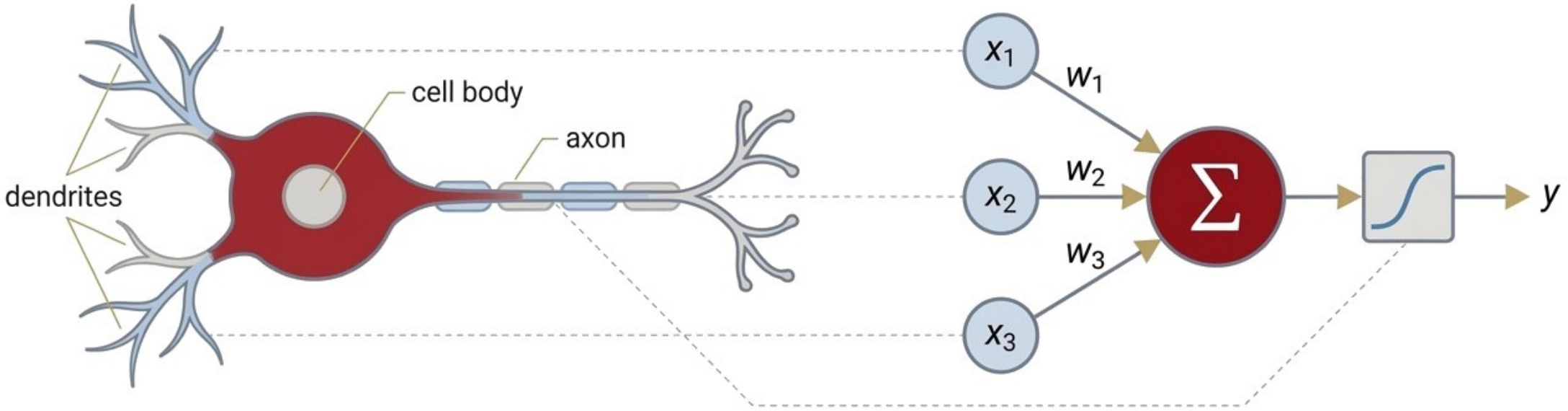

- The "deep" refers to the **number of layers** of units between the input layer and the output layer
- Since this is a neural network, the units are called **neurons**

becominghuman.ai: Deep learning made easy with Deep Cognition

# Fundamental Unit of a Neural Network

USC University of Southern California

A neuron multiplies each input by a weight, adds a bias, and passes the sum through a function.




A biological neuron (left) and the artificial neuron it inspired (right): weighted inputs, a sum, an activation, one output.

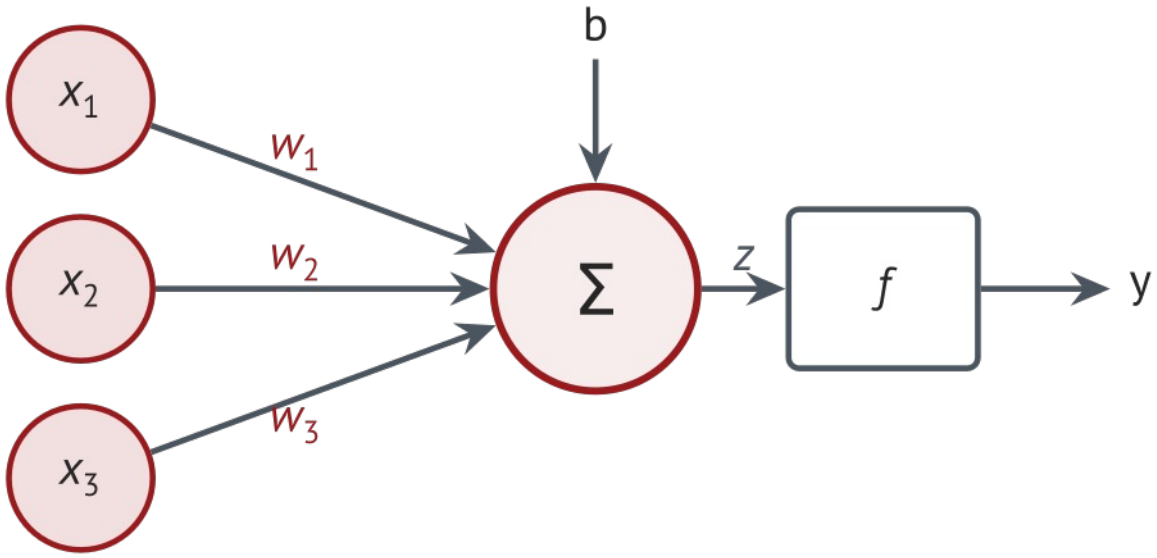
**Mathematical definition**

$$y = f(w_1x_1 + w_2x_2 + w_3x_3 + b)$$

- $x_i$: the **inputs**; y: the **output**
- $w_i$: the **weights**, also called synaptic weights
- b: the **bias**, a constant added to the sum
- f: the **neuron function** that maps inputs to output

# Fundamental Unit of a Neural Network

Many functions can serve as the neuron's f; the three most popular are sigmoid, ReLU and tanh.




**Pre-activation, then activation**

$$z = \mathbf{w} \cdot \mathbf{x} + b, \qquad y = f(z)$$

**Sigmoid**

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

**Rectified linear unit (ReLU)**

$$f(z) = \max(0, z)$$

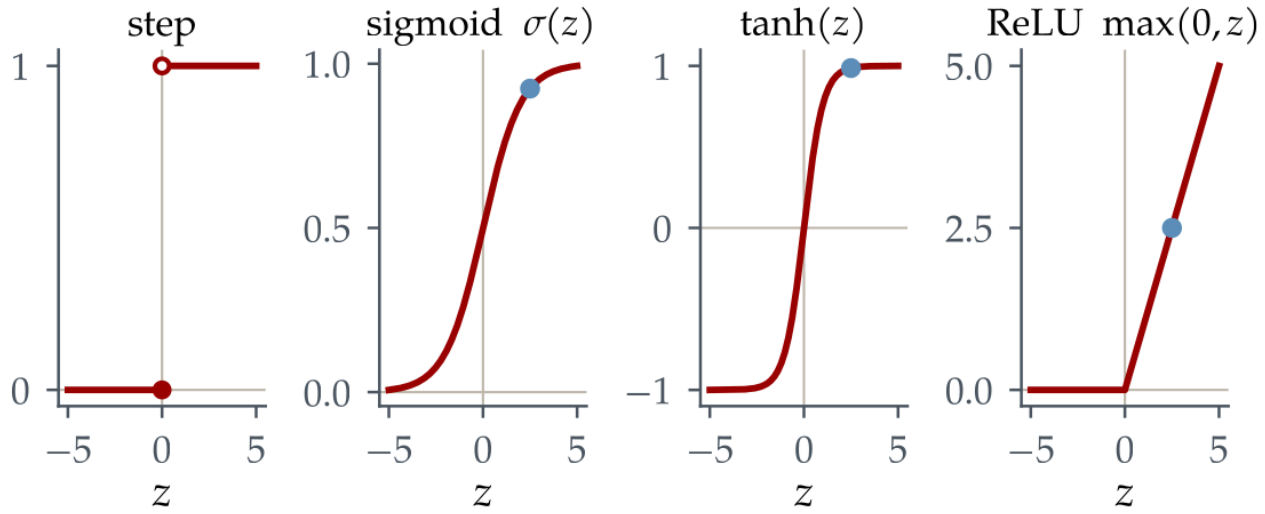
**Hyperbolic tangent**

$$\tanh(z) = 2\,\sigma(2z) - 1$$

# Activation functions of a Neural Network

USC University of Southern California

Each activation squashes or clips the weighted sum z; its shape sets how well a network learns.

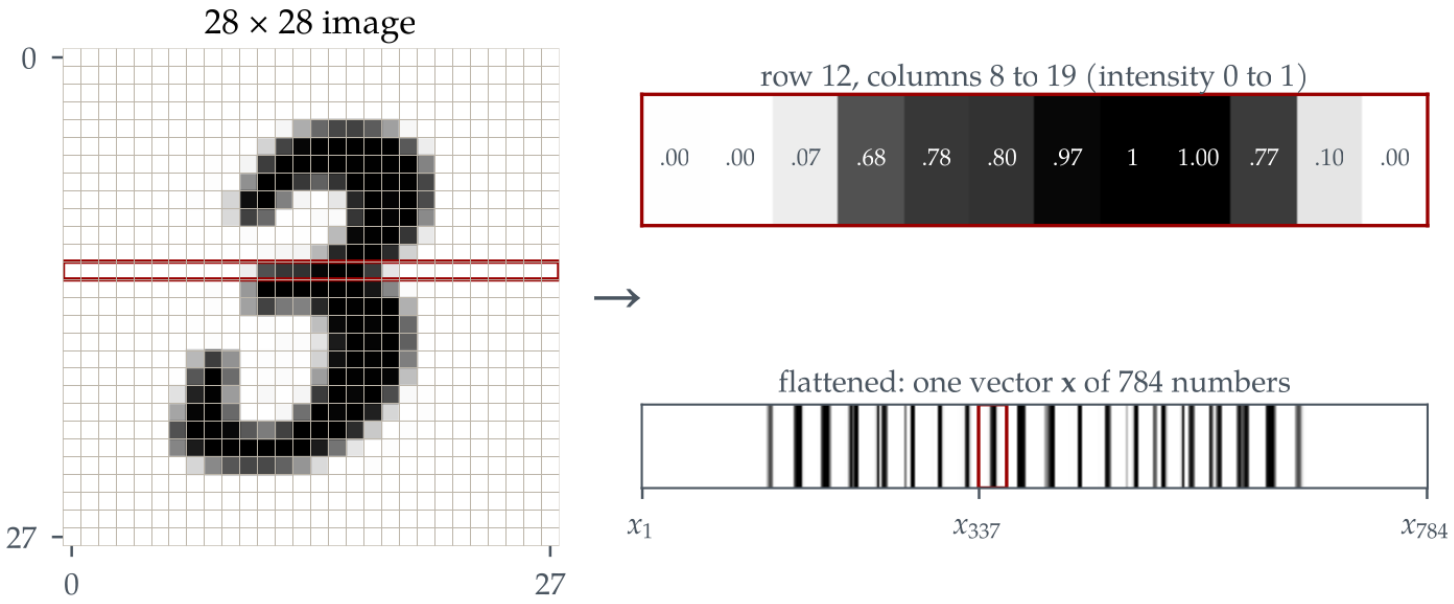



Blue dots: outputs at z = 2.5, where σ = 0.924, tanh = 0.987 and ReLU = 2.5.

| Function | Range | Shape | Pros | Cons |
|---|---|---|---|---|
| **Step** | 0 or 1 | jumps from 0 to 1 at z = 0 | clean yes/no decision | zero slope: cannot learn by small steps |
| **Sigmoid σ(z)** | (0, 1) | S-curve, σ(0) = 0.5 | smooth; reads as a probability | saturates for large \|z\|; not zero-centered |
| **tanh(z)** | (−1, 1) | S-curve centered on 0 | zero-centered | still saturates for large \|z\| |
| **ReLU max(0, z)** | [0, ∞) | 0 for z < 0, z for z > 0 | cheapest; no saturation for z > 0 | "dead" neurons stuck at 0 |

# Operation of a neural network

An image is a grid of pixels; each pixel becomes one input, connected to every hidden neuron.




Row 12 of a 28 × 28 digit with its values; reading all 28 rows gives the input vector x of 784 numbers.

- An image is made of a set number of pixels: 28 × 28 = **784**
- **Each pixel becomes an input**
- 8-bit greyscale: each pixel is 0 to 255, scaled to 0 to 1
- Each input connects to **all** the neurons of the hidden layer
- Adjusting the weights eventually lets the network recognize the digits

**Pixel value to input**

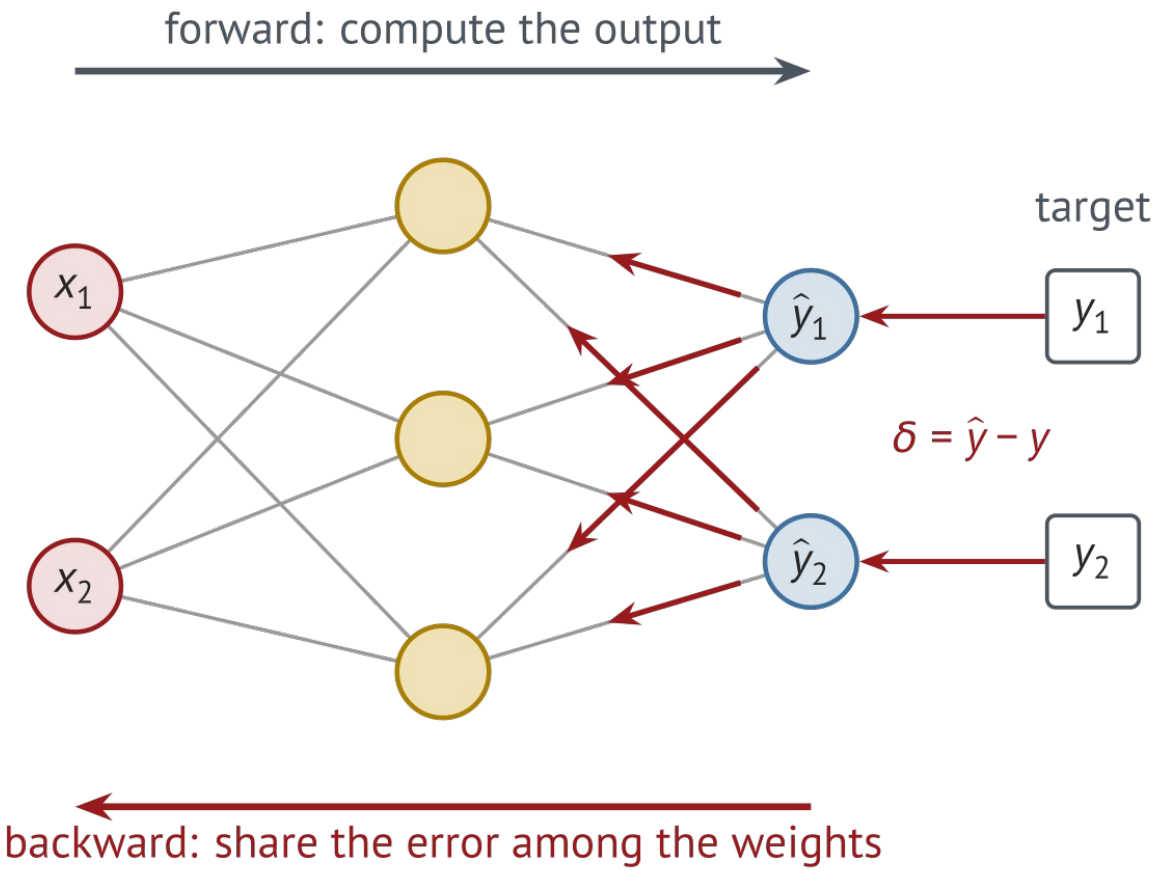
$$x_i = \frac{p_i}{255}, \qquad p_i = 0, 1, \ldots, 255$$

Source: becominghuman.ai

# Training of an Artificial Neural Network

USC University of Southern California

Backpropagation sends the output error backward to decide how much to change each weight.




A form of **supervised learning**:

1. Initialize all the weights, then feed the training data in
2. Each training example generates an output ŷ
3. We know the correct output y, so we can define an error
4. Go backward: assign part of that error to the weights between each neuron and the ones before it
5. Adjust the weights to move toward the correct answer, and repeat

Figure idea: rasbt, visual backpropagation (GitHub)

# Handwriting Recognition

USC University of Southern California

We read these digits without effort, but our brain uses an enormous network to do it.



- We can easily recognize these numbers
- But this uses a huge number of neurons (**about a billion or more**) and an even larger number of connections, or synapses (**hundreds of billions**)

**The network we build in this lecture**

784 inputs, 15 hidden and 10 output neurons:

**11,935** weights and biases in all

# Approach

USC University of Southern California

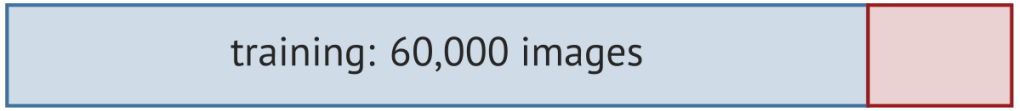
Learn from a large set of labeled examples, then test on images the network has never seen.



Handwritten digits from the MNIST dataset.

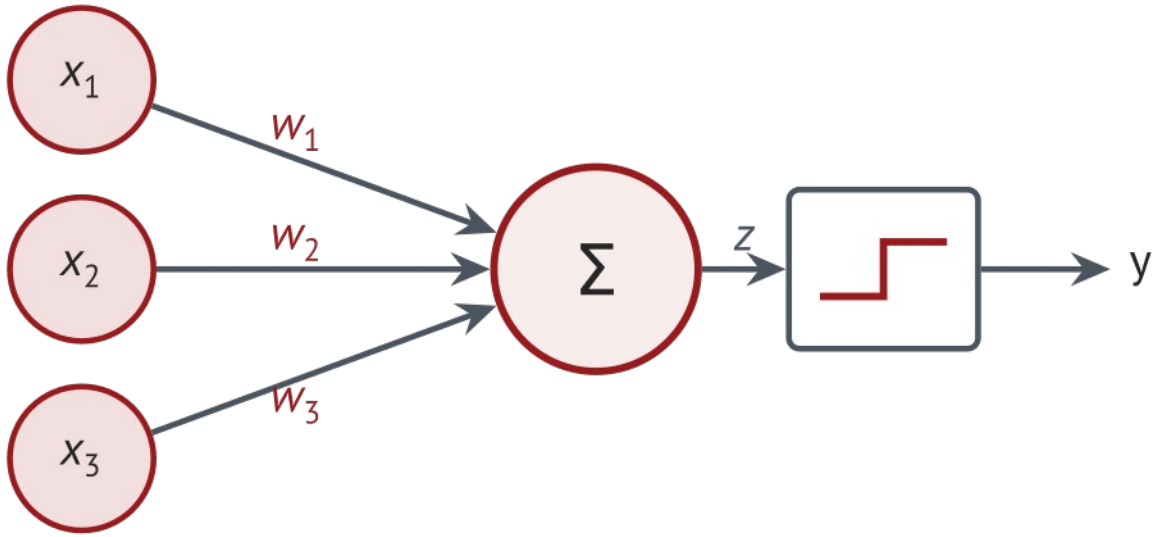
- Need a **large set of training data**: many digits, each with its correct label
- Also need a separate **test set** on which to check the trained model

**MNIST split**



# Perceptron

A perceptron uses a step: it outputs 1 only when the weighted sum is above a threshold.




- Same unit as before, with a **step** function
- Inputs $x_i$ and output y are 0 or 1
- The weights set how much each input counts
- The **threshold** sets how much evidence is enough

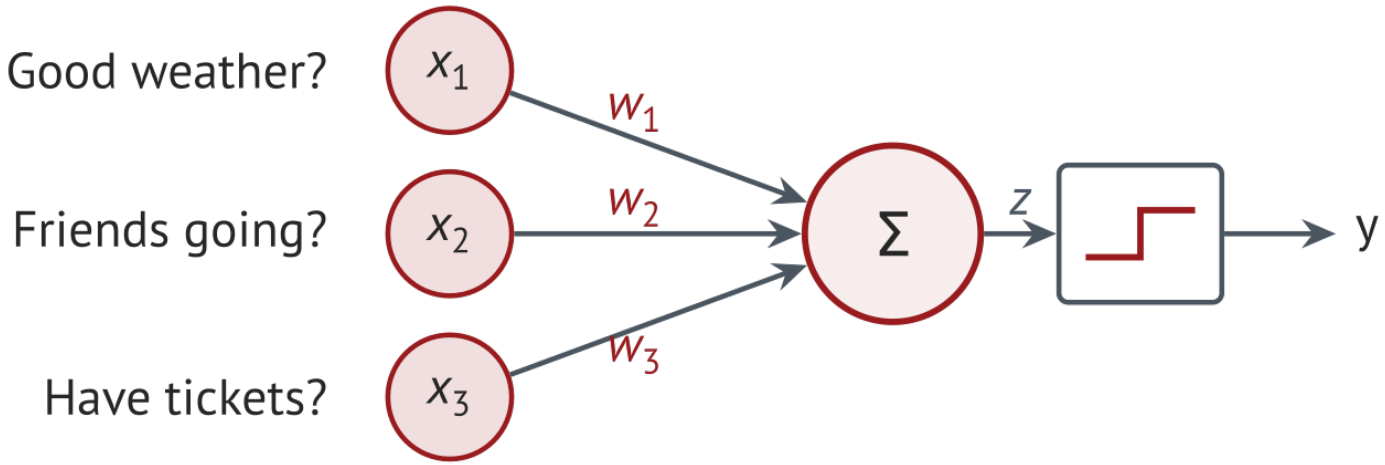
**Perceptron rule**

$$y = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq \text{threshold} \\ 1 & \text{if } \sum_i w_i x_i > \text{threshold} \end{cases}$$

# What can we answer with a perceptron?

USC University of Southern California

Do we want to go to Coachella? Three yes/no factors feed one perceptron that answers yes or no.

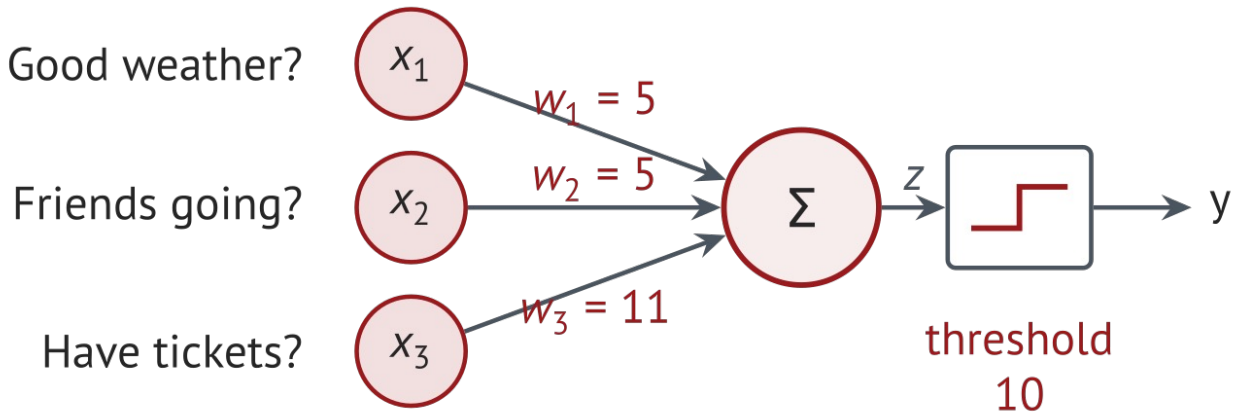



- Do we want to go to Coachella?
- Each variable is either **1 (yes)** or **0 (no)**
- The decision making happens in the **weights**
- We have to assign the weights and a **threshold** value

**Example input:** good weather, friends going, tickets in hand: $x_1 = x_2 = x_3 = 1$. The output y = 1 means go.

# How would we decide this?

Threshold 10 with weights 5, 5 and 11: the tickets alone decide.



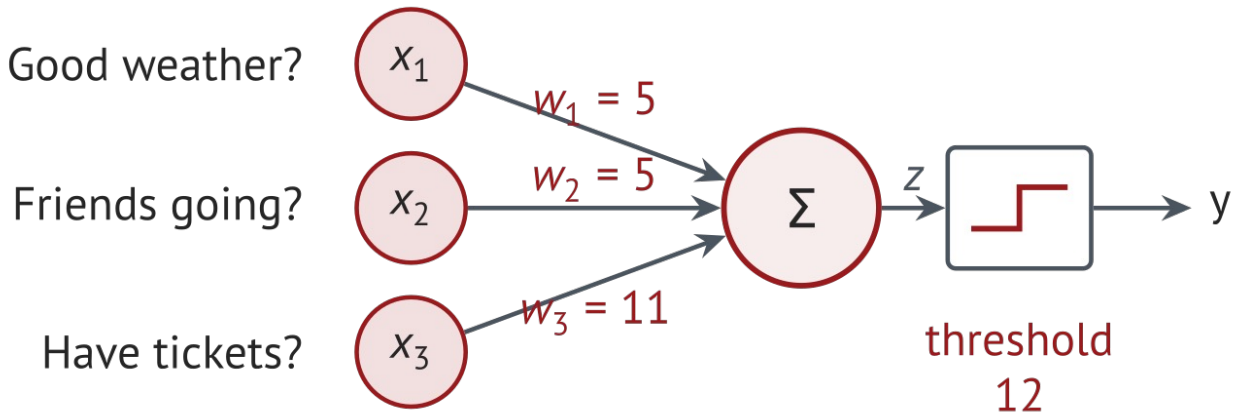

| $x_1$ | $x_2$ | $x_3$ | $\Sigma w_i x_i$ | go? |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 ≤ 10 | no |
| **0** | **0** | **1** | **11 > 10** | **yes** |
| 0 | 1 | 0 | 5 ≤ 10 | no |
| **0** | **1** | **1** | **16 > 10** | **yes** |
| 1 | 0 | 0 | 5 ≤ 10 | no |
| **1** | **0** | **1** | **16 > 10** | **yes** |
| 1 | 1 | 0 | 10 ≤ 10 | no |
| **1** | **1** | **1** | **21 > 10** | **yes** |

**$w_3$ = 11 determines the output.** Weather and friends together give 10, which is not more than 10. Either you have tickets and go, or you don't and you stay home.

# How would we decide this?

USC University of Southern California

Threshold 12 with weights 5, 5 and 11: tickets plus at least one other factor.



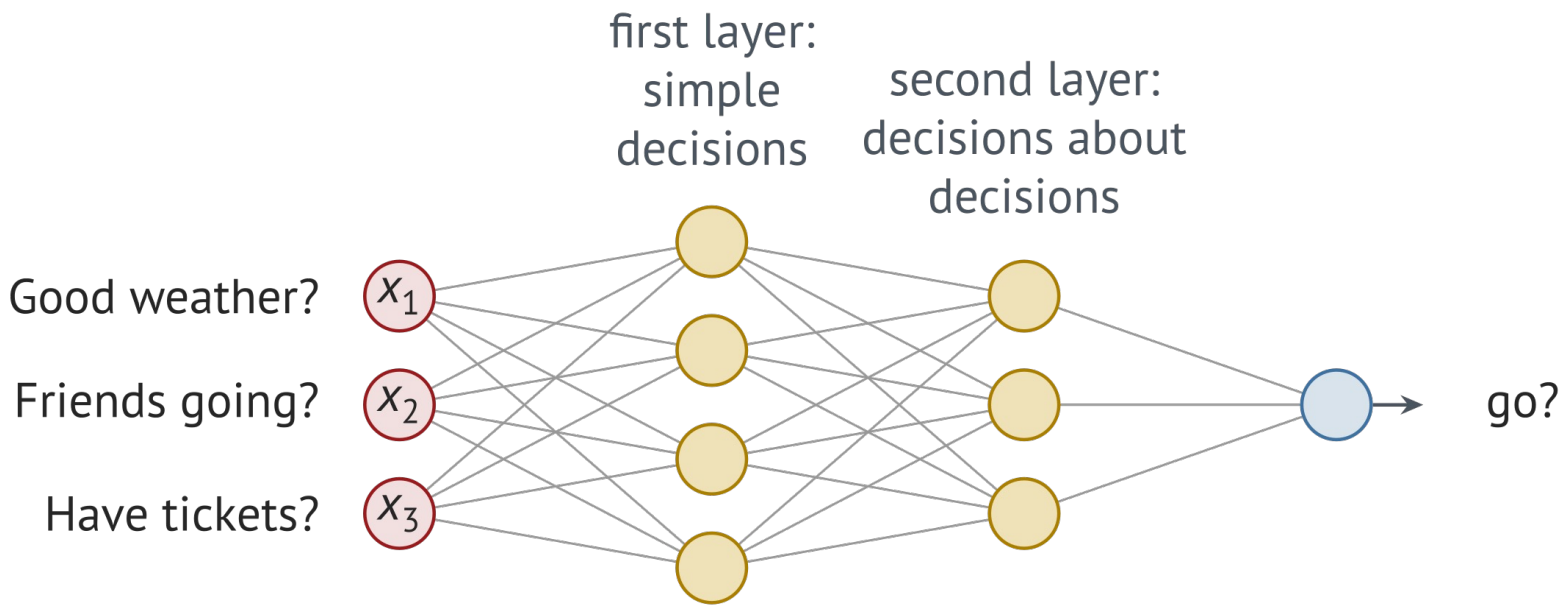

| $x_1$ | $x_2$ | $x_3$ | $\Sigma w_i x_i$ | go? |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 ≤ 12 | no |
| 0 | 0 | 1 | 11 ≤ 12 | no |
| 0 | 1 | 0 | 5 ≤ 12 | no |
| **0** | **1** | **1** | **16 > 12** | **yes** |
| 1 | 0 | 0 | 5 ≤ 12 | no |
| **1** | **0** | **1** | **16 > 12** | **yes** |
| 1 | 1 | 0 | 10 ≤ 12 | no |
| **1** | **1** | **1** | **21 > 12** | **yes** |

**Tickets AND at least one other factor.** 5 + 5 = 10 is a no; 11 alone is a no; 11 + 5 = 16 and 5 + 5 + 11 = 21 are both a yes.

# Addition of other nodes increases complexity

USC University of Southern California

More neurons and layers let a network weigh more inputs and make more nuanced decisions.

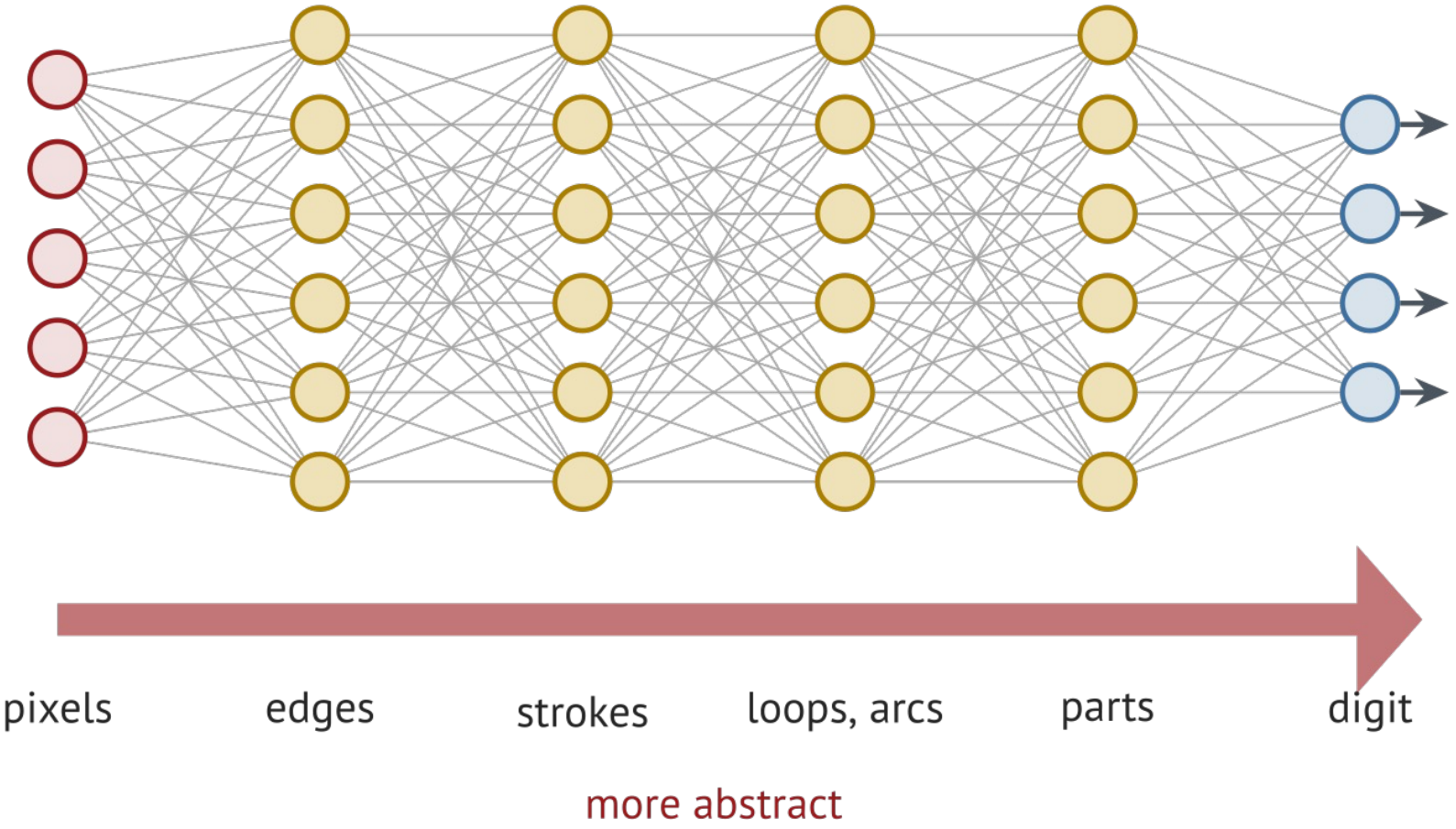



- We can now make decisions based on **more inputs**
- Each layer weighs the decisions of the layer before it
- A more nuanced way of getting to the output

# How does information change through layers?

USC University of Southern California

Through the layers of a deep network, decisions are about more and more abstract items.




- Early layers respond to **simple, local** patterns
- Later layers combine them into **larger, more abstract** ones
- The output layer decides about the whole input

# Perceptrons

Write the inputs and weights as vectors: the weighted sum is just their dot product.

**Inputs and weights as vectors**

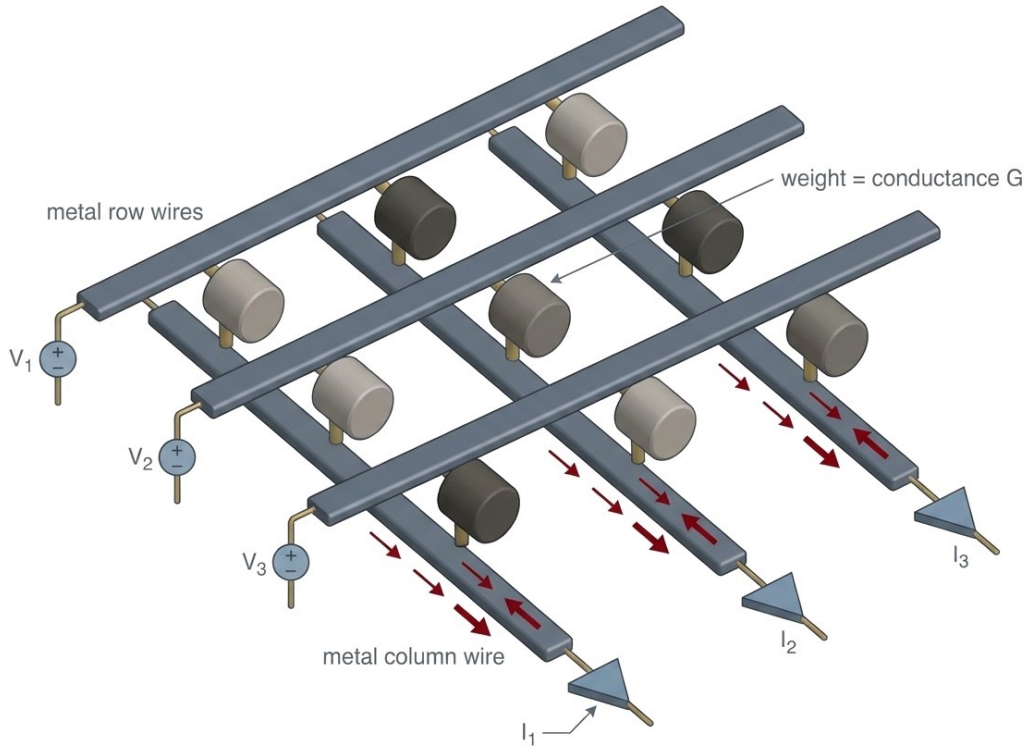
$$\mathbf{x} = (x_1, x_2, x_3), \qquad \mathbf{w} = (w_1, w_2, w_3)$$

**The sum is a dot product**

$$\mathbf{w} \cdot \mathbf{x} = w_1 x_1 + w_2 x_2 + w_3 x_3$$

**Perceptron rule with b = −threshold**

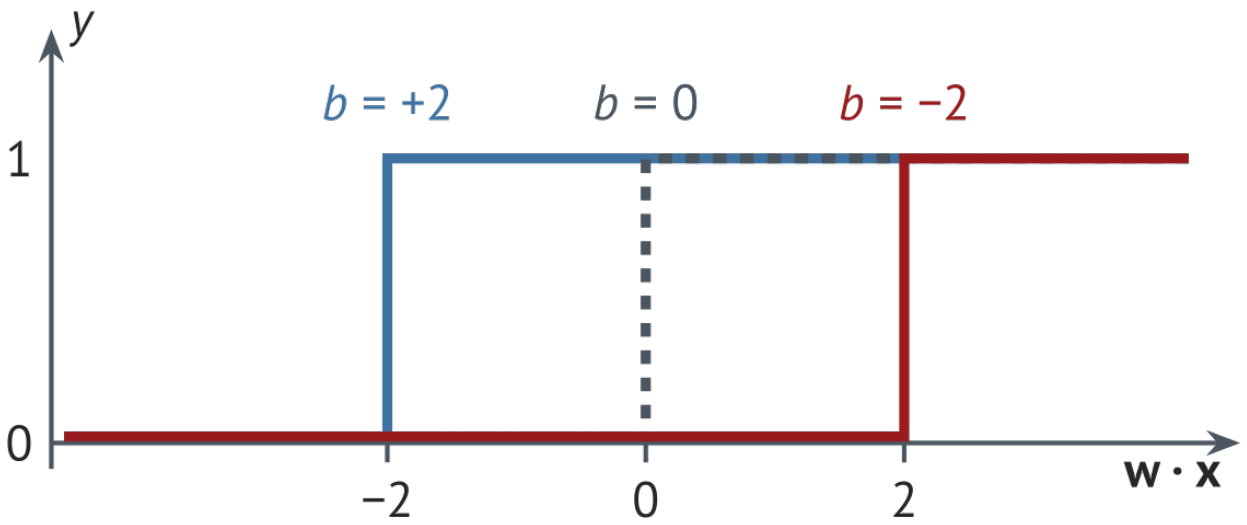
$$y = \begin{cases} 0 & \text{when } \mathbf{x} \cdot \mathbf{w} + b \leq 0 \\ 1 & \text{when } \mathbf{x} \cdot \mathbf{w} + b > 0 \end{cases}$$




In hardware: a resistor crossbar computes dot products in one step. Each conductance is a weight; Ohm's law multiplies and KCL adds the column currents.

# Perceptrons

The bias b sets how easily a perceptron fires: the step sits where w·x = −b.



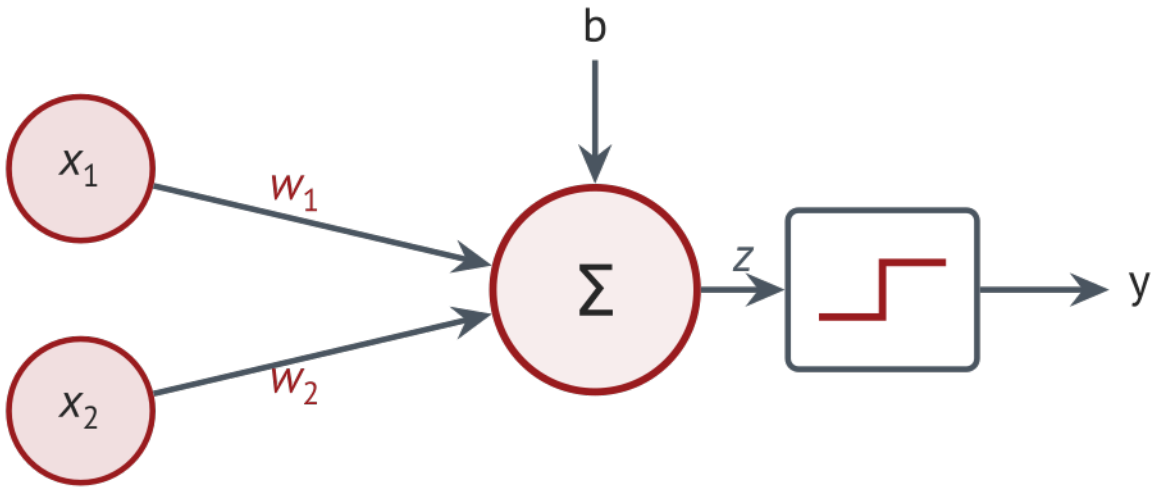

- y = 1 only when x·w + b > 0
- b can be thought of as the **bias**: b = −threshold
- **Large positive b**: a neuron that fires on most inputs
- **Negative b**: a neuron that rarely fires

**Coachella again:** a threshold of 10 is b = −10 and a threshold of 12 is b = −12; the more negative b, the more evidence the perceptron needs before it says yes.

# Perceptrons for logical functions?

USC University of Southern California

A two-input perceptron can act as a logic gate: we only need the right weights and bias.




**Target truth tables**

| $x_1$ | $x_2$ | AND | OR | NAND |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

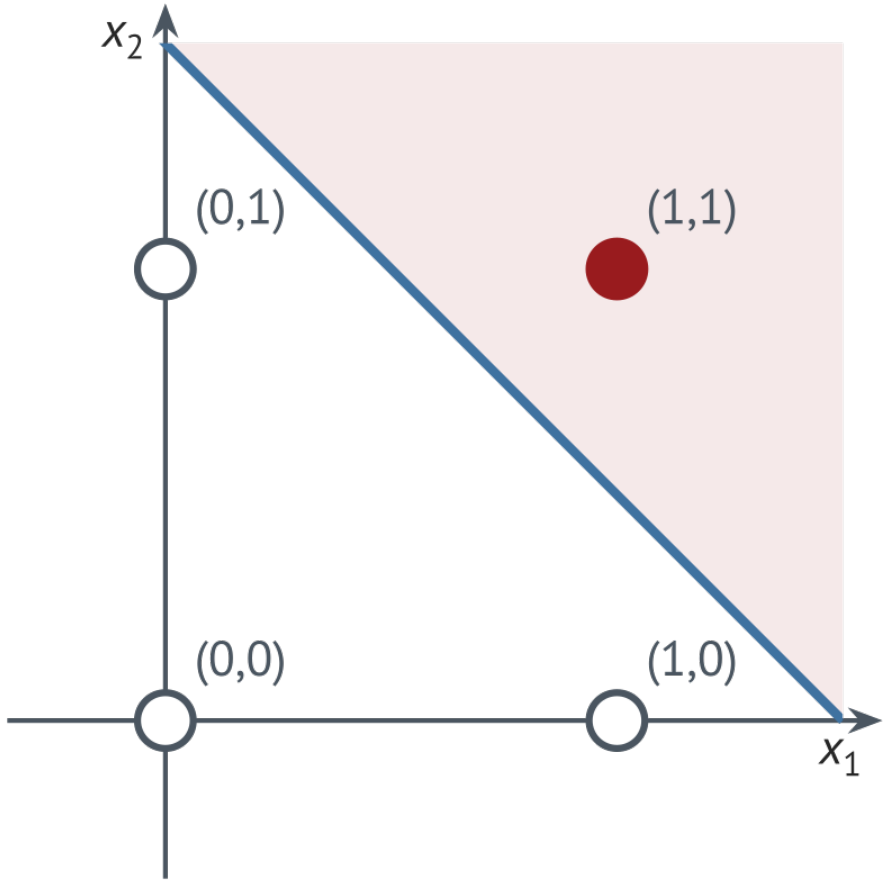
- How could we implement **AND**, **OR** and **NAND**?
- Find $w_1$, $w_2$ and b for each gate

TRY IT

Start with AND: which inputs need $z > 0$?

# AND gate

With w = (1, 1) and b = −1.5, only (1, 1) lands on the positive side of the line.




Filled: y = 1. Shaded: w·x + b > 0. Line: w·x + b = 0.

**AND gate: weights and bias**

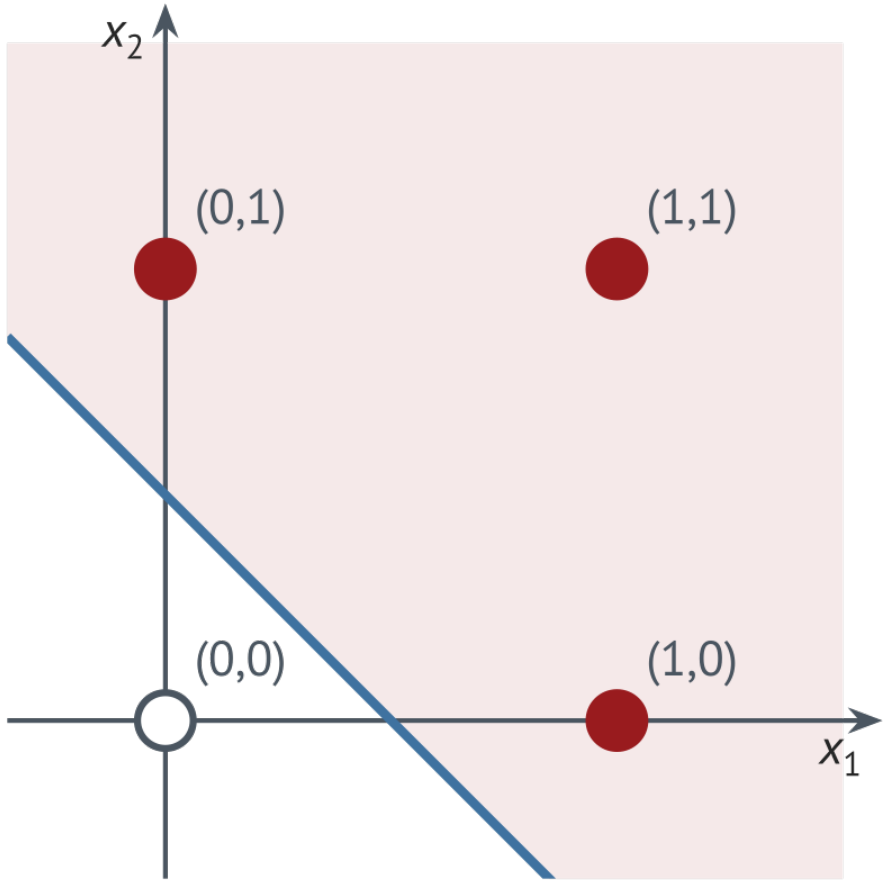
$$w_1 = 1, \quad w_2 = 1, \quad b = -1.5$$

| $x_1$ | $x_2$ | $z = w_1x_1 + w_2x_2 + b$ | y |
|---|---|---|---|
| 0 | 0 | 0 + 0 − 1.5 = −1.5 | 0 |
| 0 | 1 | 0 + 1 − 1.5 = −0.5 | 0 |
| 1 | 0 | 1 + 0 − 1.5 = −0.5 | 0 |
| **1** | **1** | **1 + 1 − 1.5 = 0.5** | **1** |

**Decision boundary:** $x_1 + x_2 = 1.5$. Any b with $-2 < b \leq -1$ works; −1.5 is the middle.

# OR gate

Lower the bias to −0.5 and the line moves down: any input with a 1 now gives y = 1.




Filled: y = 1. Shaded: w·x + b > 0. Line: w·x + b = 0.

**OR gate: weights and bias**

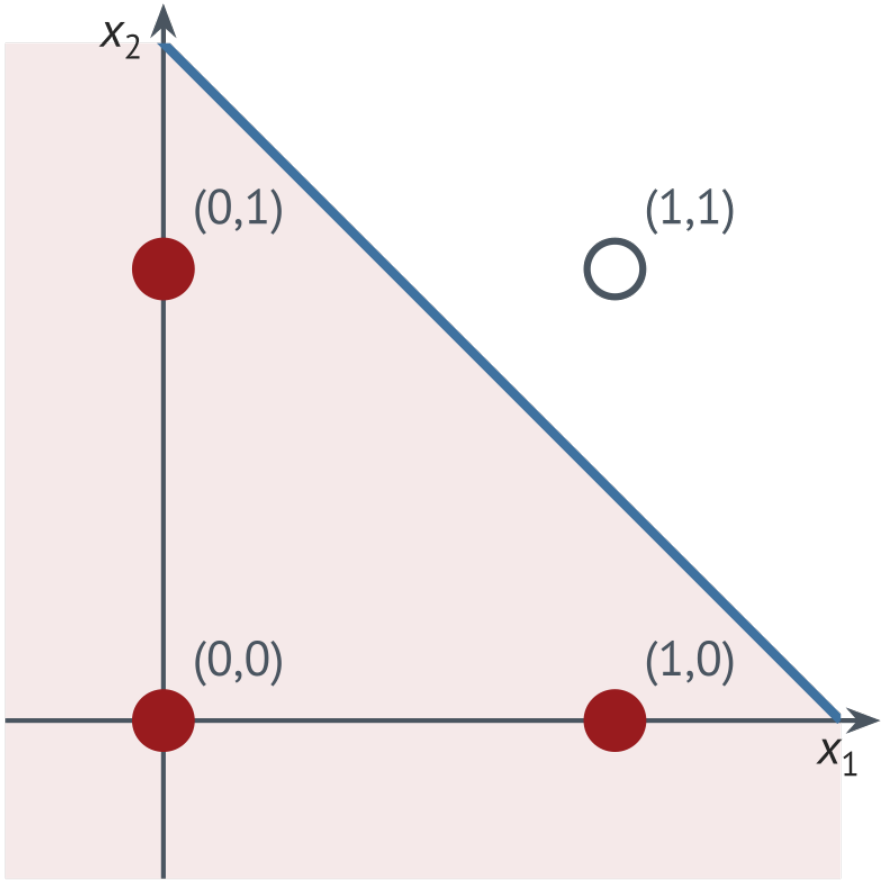
$$w_1 = 1, \quad w_2 = 1, \quad b = -0.5$$

| $x_1$ | $x_2$ | $z = w_1x_1 + w_2x_2 + b$ | y |
|---|---|---|---|
| 0 | 0 | 0 + 0 − 0.5 = −0.5 | 0 |
| **0** | **1** | **0 + 1 − 0.5 = 0.5** | **1** |
| **1** | **0** | **1 + 0 − 0.5 = 0.5** | **1** |
| **1** | **1** | **1 + 1 − 0.5 = 1.5** | **1** |

**Decision boundary:** $x_1 + x_2 = 0.5$. Same weights as AND; only the bias moved.

# NAND gate

Flip the sign of every weight and of the bias of AND, and every output flips: NAND.




Filled: y = 1. Shaded: w·x + b > 0. Line: w·x + b = 0.

**NAND gate: weights and bias**

$$w_1 = -1, \quad w_2 = -1, \quad b = 1.5$$

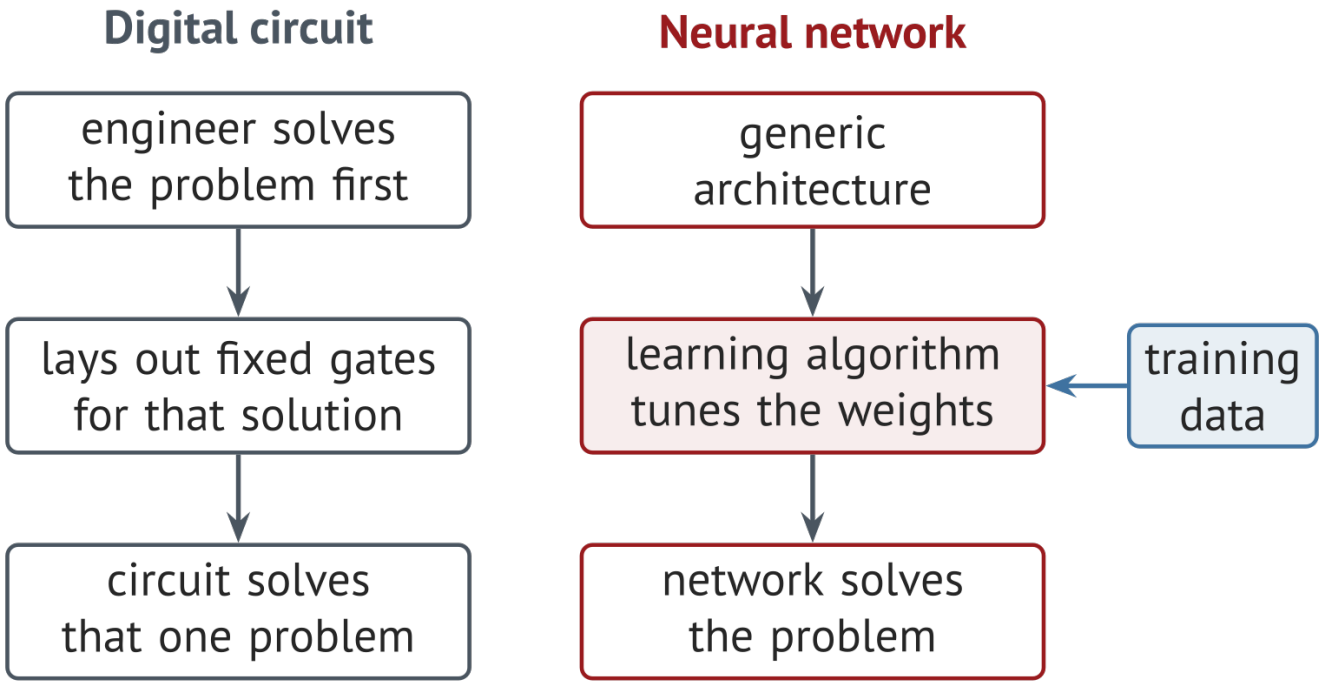
| $x_1$ | $x_2$ | $z = w_1x_1 + w_2x_2 + b$ | y |
|---|---|---|---|
| **0** | **0** | **0 + 0 + 1.5 = 1.5** | **1** |
| **0** | **1** | **0 − 1 + 1.5 = 0.5** | **1** |
| **1** | **0** | **−1 + 0 + 1.5 = 0.5** | **1** |
| 1 | 1 | −1 − 1 + 1.5 = −0.5 | 0 |

**NOT of AND:** 1 except at (1, 1). Same line $x_1 + x_2 = 1.5$, with the sides swapped.

# So are perceptrons just more complex NAND gates?

USC University of Southern California

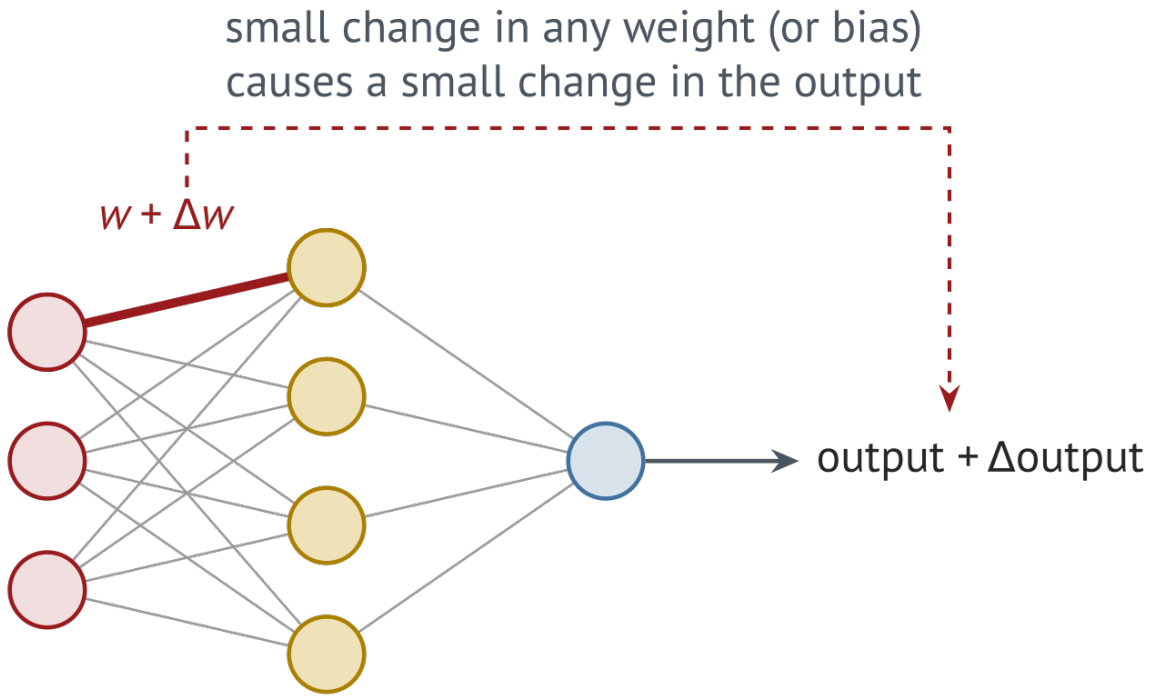
No: their strength is a learning algorithm that tunes the weights from the input data alone.




- **No.** The key strength lies in the algorithms that tune the weights
- The tuning uses only the input data, not a human
- A circuit: we presolve the problem, then lay out the gates to solve it
- Learning: we create a basic architecture and let it tune itself to get the right answer

# Sigmoid Neurons

To learn, a small change in one weight should cause a small change in the output.



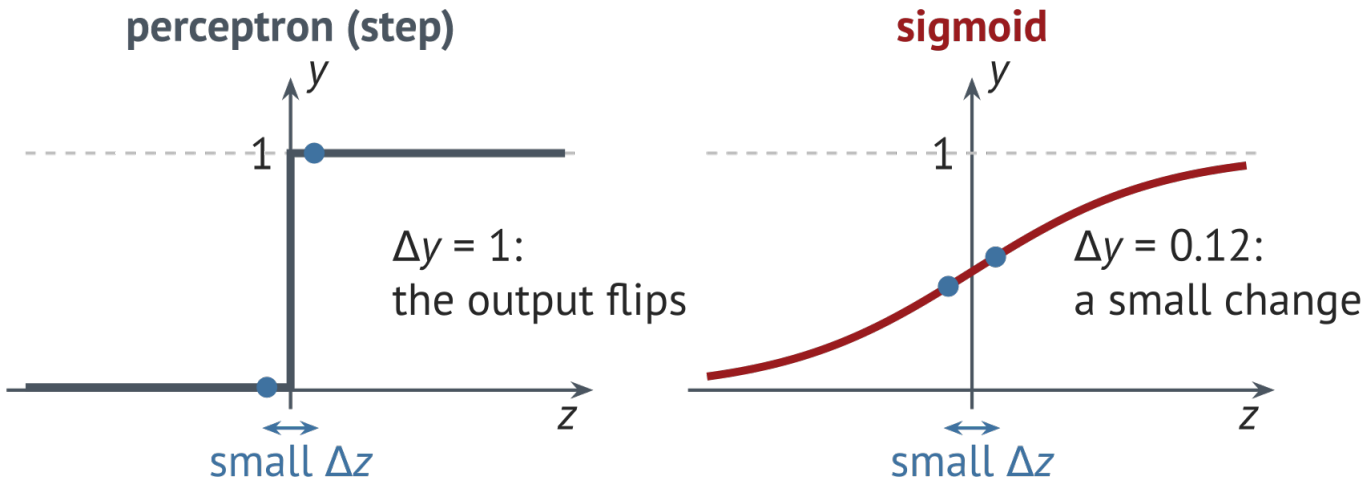

- If we want to learn, a small change in the weight of one synapse should give a **small change in the output**
- Example: we feed in a handwritten digit; it is a **3**
- Our network classifies it as a **4**
- We would like to change the weights slightly and move the output a little toward 3

**Small changes in, small change out**

$$\Delta\text{output} \approx \sum_j \frac{\partial\,\text{output}}{\partial w_j}\Delta w_j + \frac{\partial\,\text{output}}{\partial b}\Delta b$$

# Sigmoid Neurons

With perceptrons a tiny weight change can flip an output, so learning by small steps fails.

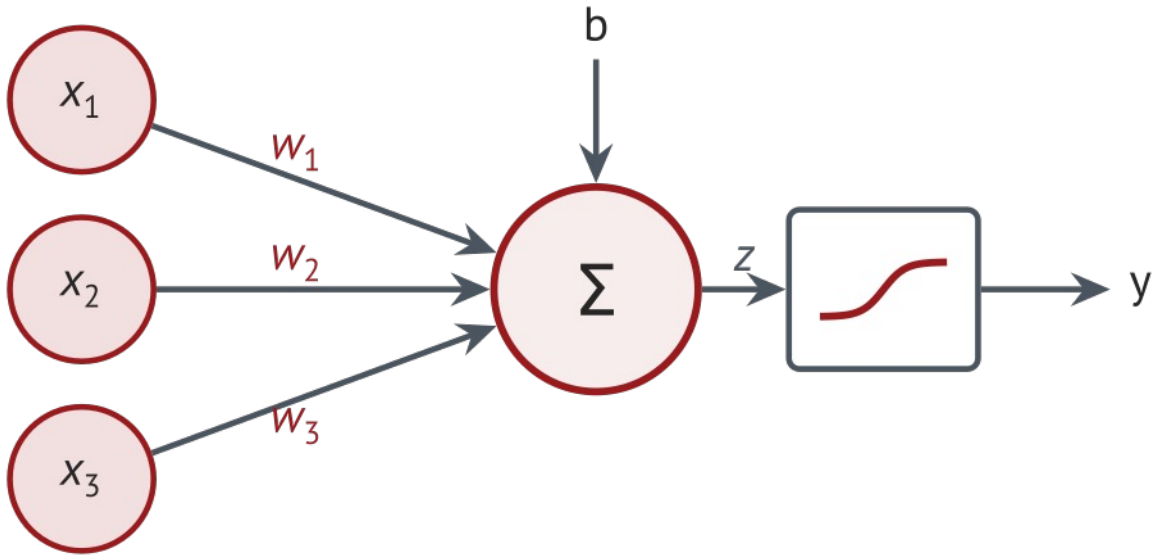



- With a perceptron, a small weight change can **invert** the neuron's output
- The flip propagates through the network and can change everything in a way that is hard to predict
- You fix the misclassified 3 but break other digits

**So** with perceptrons it is hard to find an algorithm that learns complex data. To get around this, the **sigmoid neuron** was introduced.

# Sigmoid Neurons

A sigmoid neuron looks the same, but its inputs and output can be anything between 0 and 1.



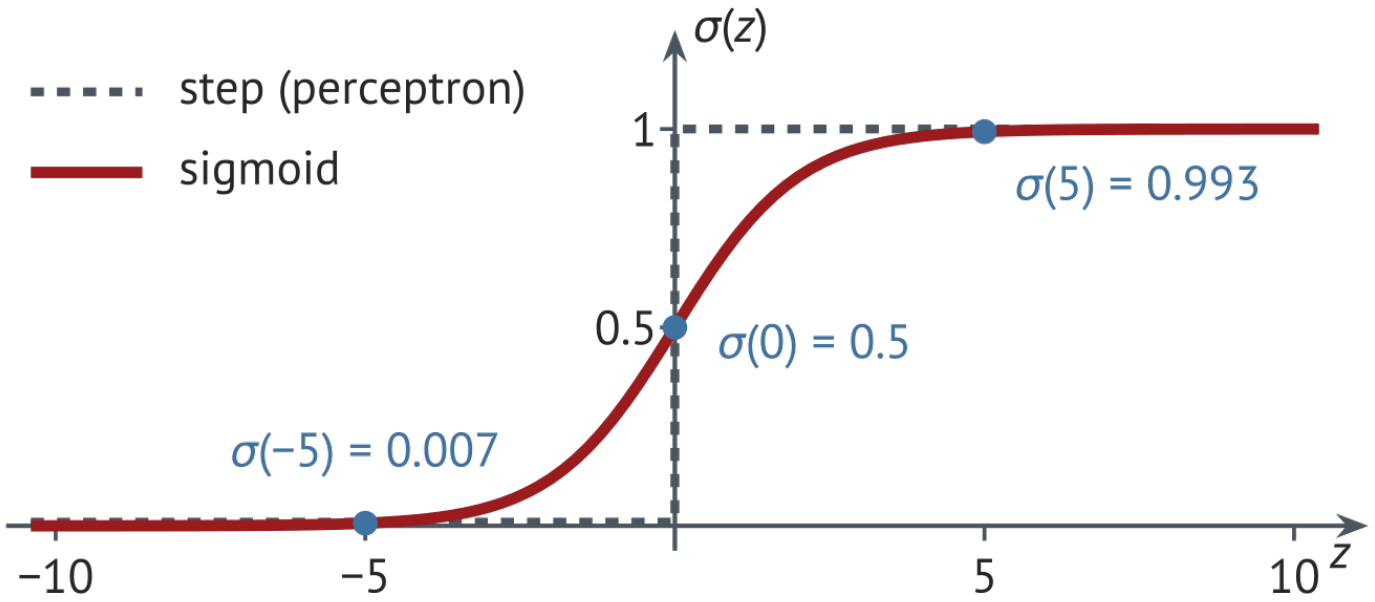

- Looks the same as before
- However, the inputs can now be **anything between 0 and 1**, not just 0 or 1
- The output is σ(z), also between 0 and 1

**The equation is now a little more complex**

$$y = \sigma(\mathbf{x} \cdot \mathbf{w} + b) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

# Similarities between Perceptrons and Sigmoids

Far from z = 0 a sigmoid acts like a perceptron; near zero it smooths out the sudden jump.




**Sigmoid**

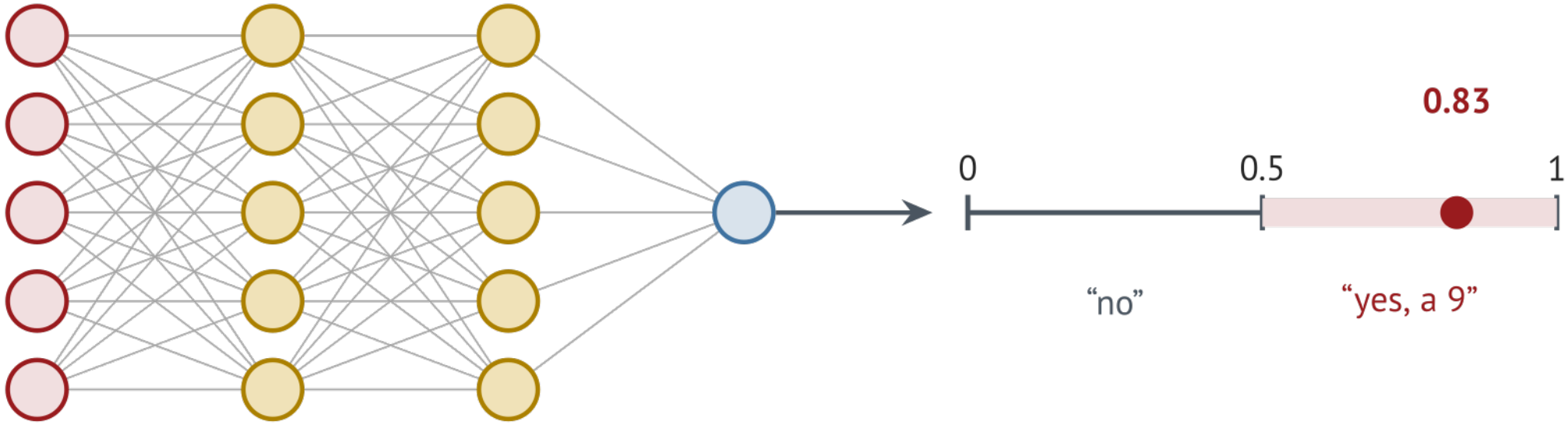
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- z large and positive: σ(z) ≈ 1
- z large and negative: σ(z) ≈ 0
- Key difference: the sigmoid **smooths out** the sudden change at z = 0

# Sigmoid Neurons

USC University of Southern California

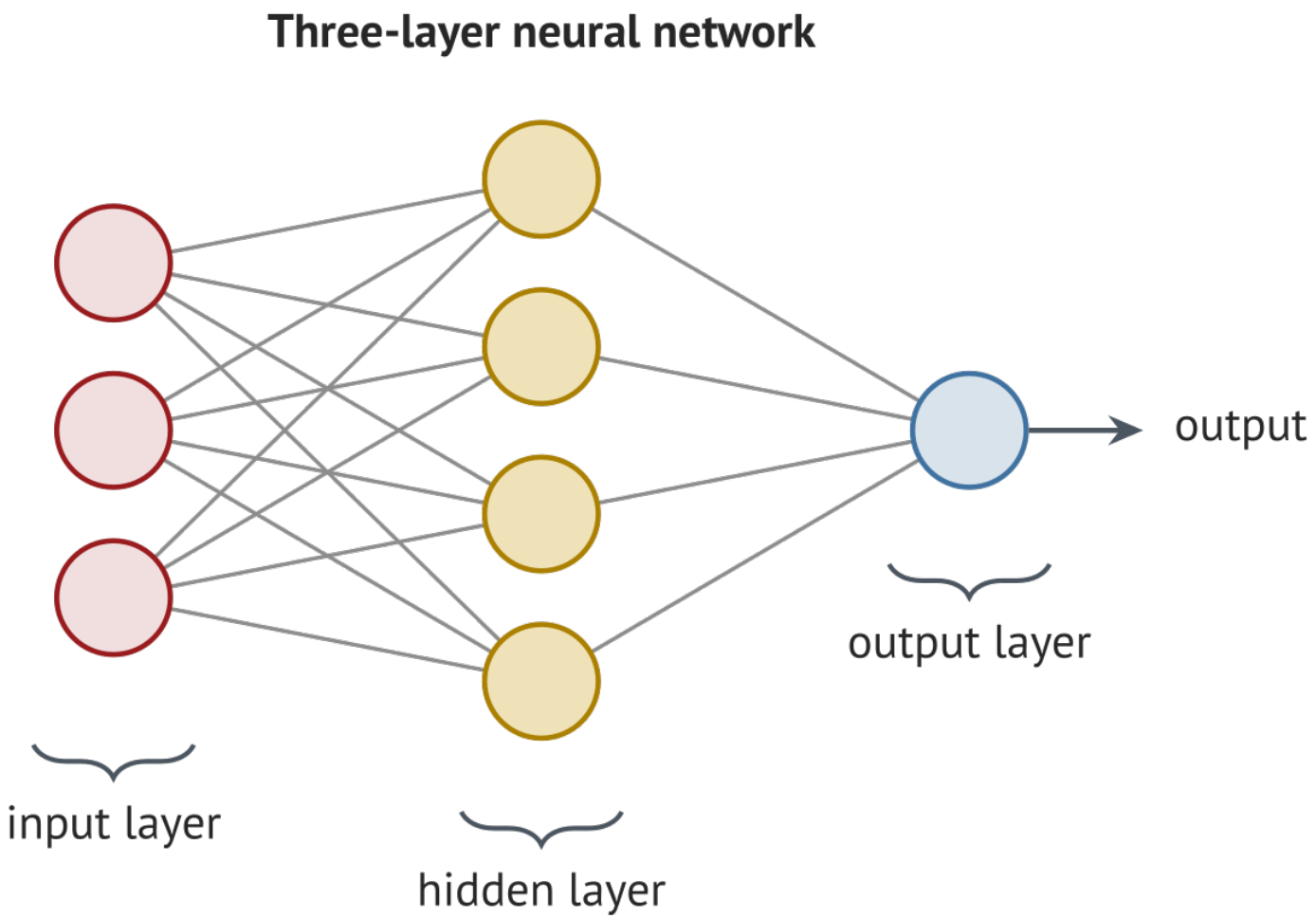
A sigmoid output is a number, not a yes or no, so we choose a rule for reading it.




- **Perceptron** neurons are easy to evaluate: 0 or 1; yes or no
- **Sigmoids** are continuous, so we have to set up a criterion for what the output neuron is saying
- For example: if the output is greater than 0.5, the input is a 9

# Neural Network Architecture

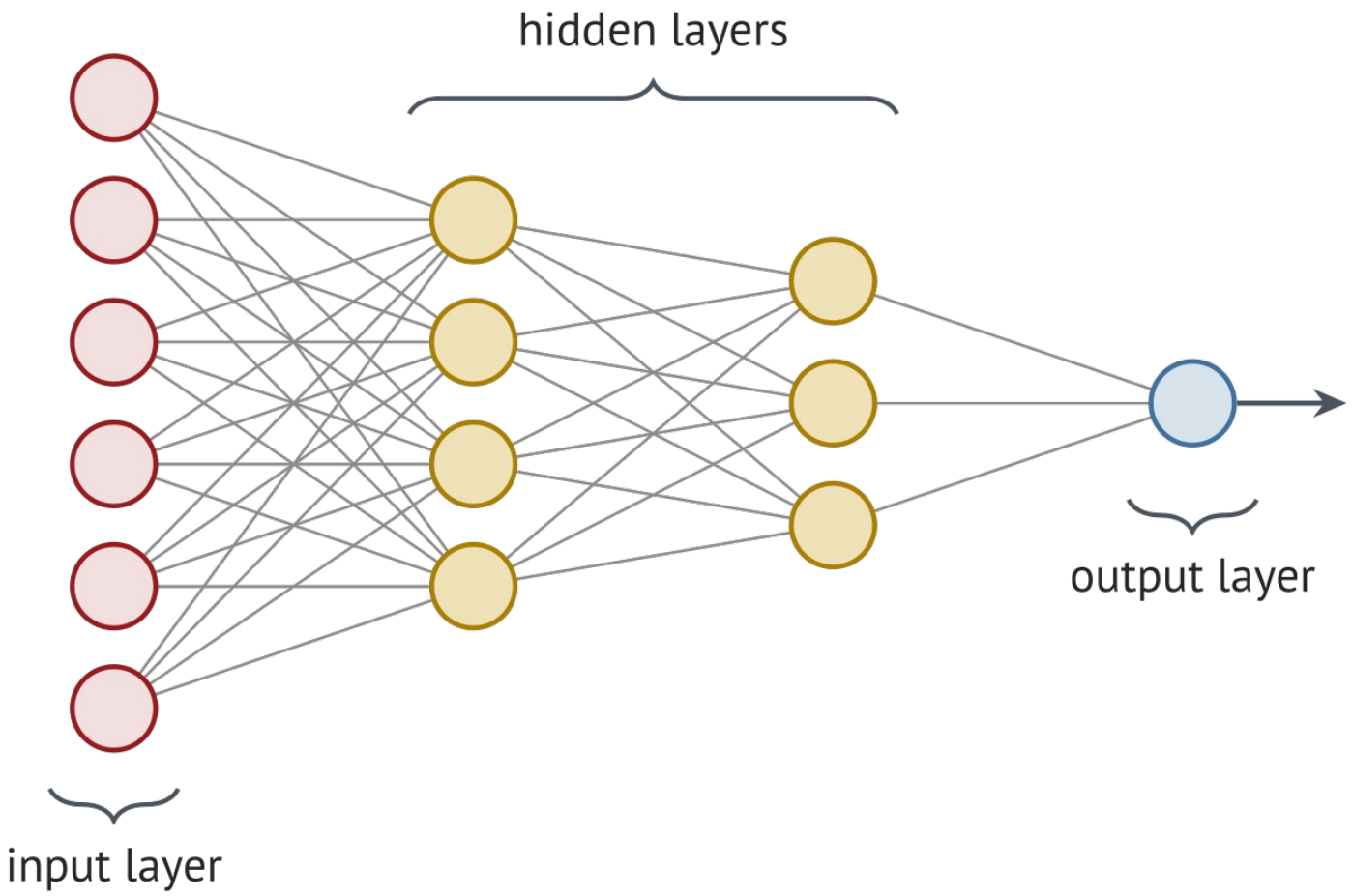
Every feed-forward network has an input layer, one or more hidden layers, and an output layer.




- **Input layer**: holds the data, one neuron per input; no computation
- **Hidden layer**: neither input nor output
- **Output layer**: gives the answer
- Every neuron feeds every neuron of the next layer

# Four Layer Neural Network

Add a second hidden layer and we have a four-layer network, a multi-layer perceptron (MLP).




- This is a **multi-layer perceptron (MLP)**
- The name stuck, even though its neurons are usually sigmoids or ReLUs

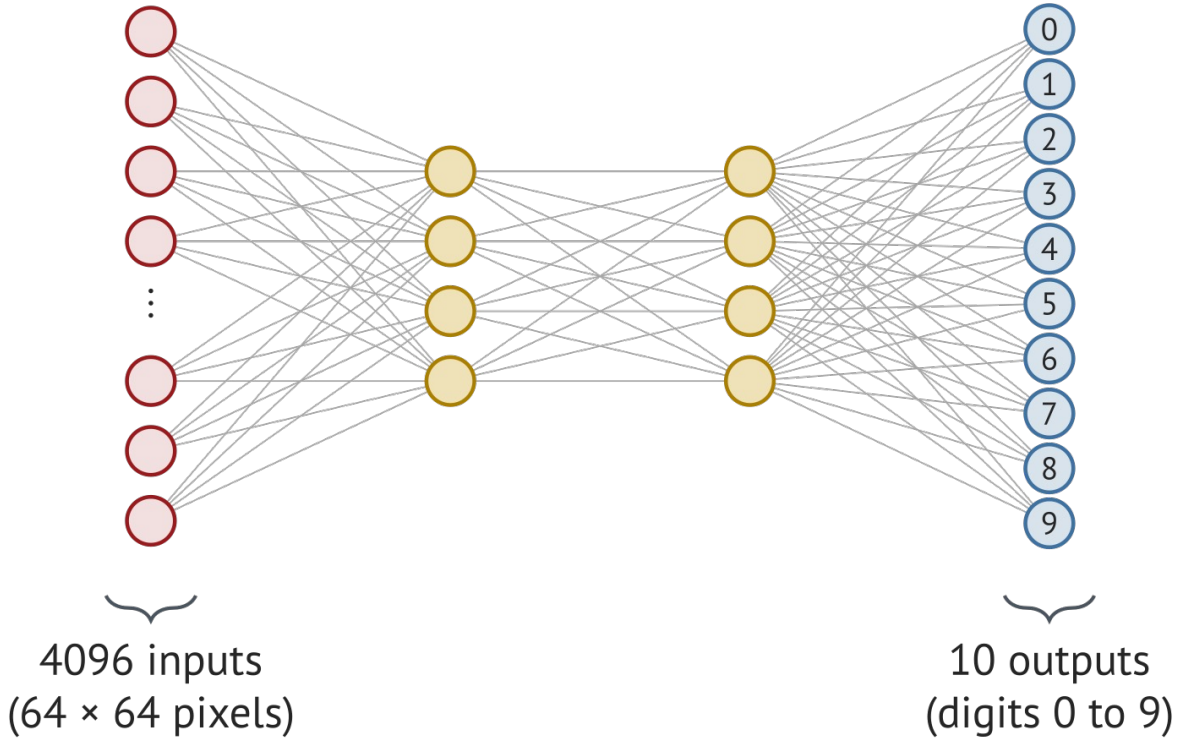
**Each layer**

$$\mathbf{h} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

# How do we design our Neural Network Topology?

USC University of Southern California

The data fix the input layer and the task fixes the output layer.



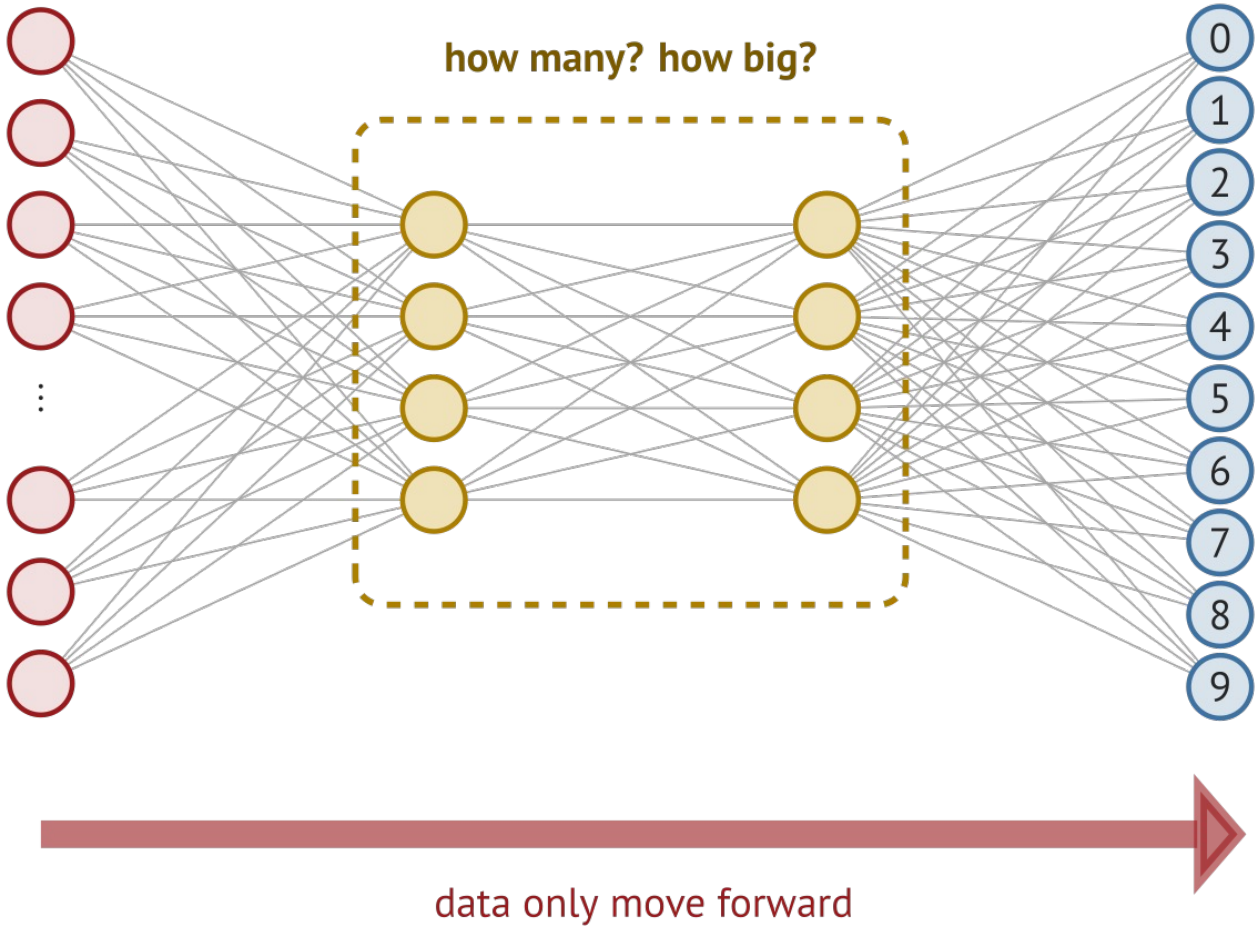

64 × 64 = 4096 inputs  10 outputs

- **Inputs** depend on the number and type of inputs
  - 64 × 64 greyscale image: **4096** input neurons, each 0 to 1, the intensity of one pixel
- **Outputs**: for handwritten digits use **10** outputs (0, 1, …, 9)
  - Only one neuron should output “yes”
  - “Yes” means the output is above some value, e.g. 0.5

# How do we design our Neural Network Topology?

USC University of Southern California

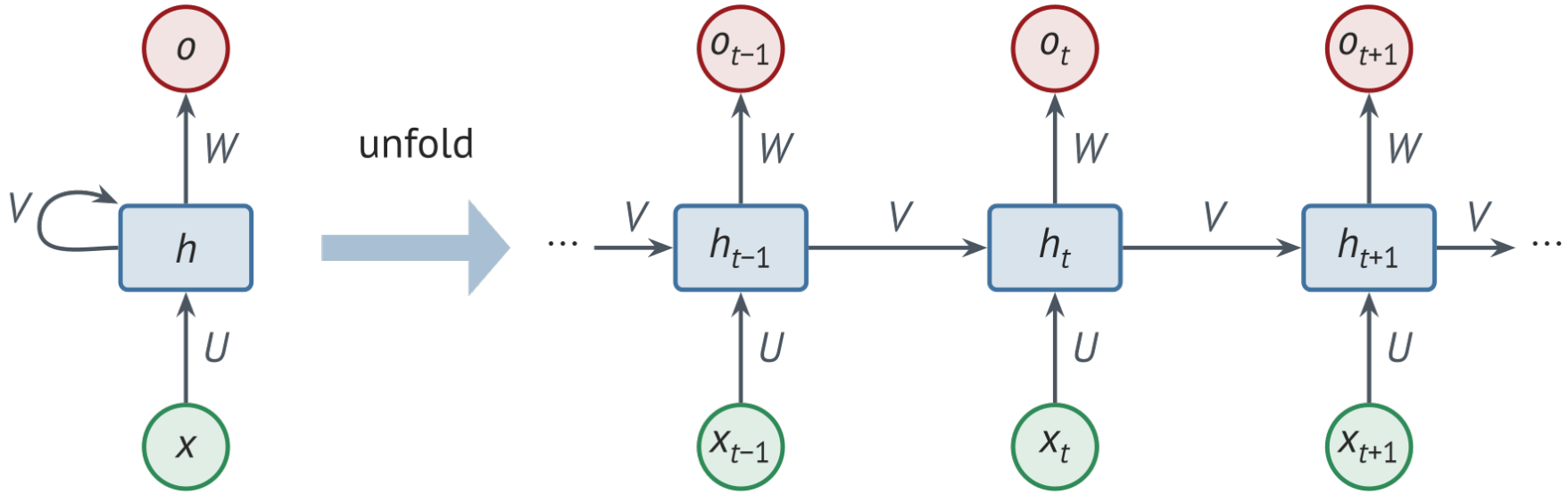
The hidden layers are the art: there is no formula, so we try sizes and keep what works.




- **Hidden layers?**
  - This part is art, not science
  - Driven by heuristics that depend on the problem
- **Note:** these are **feed-forward** neural networks
  - All inputs move forward, never backward, during operation

# Aside: Recurrent Neural Networks

In a recurrent network a neuron's output is fed back as an input, which gives the network memory.

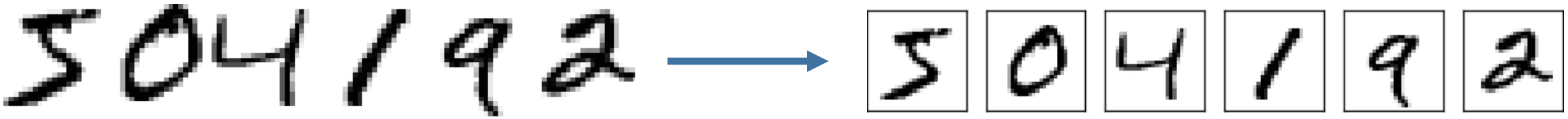



- The output of a neuron is also an **input**: the network has **memory**
- Behavior depends on time and can evolve from one step to the next
- The feedback needs a time step: an output cannot be fed back in if it were evaluated instantaneously

# Getting back to handwriting

USC University of Southern California

A written number is several digit images; we focus on classifying a single digit.



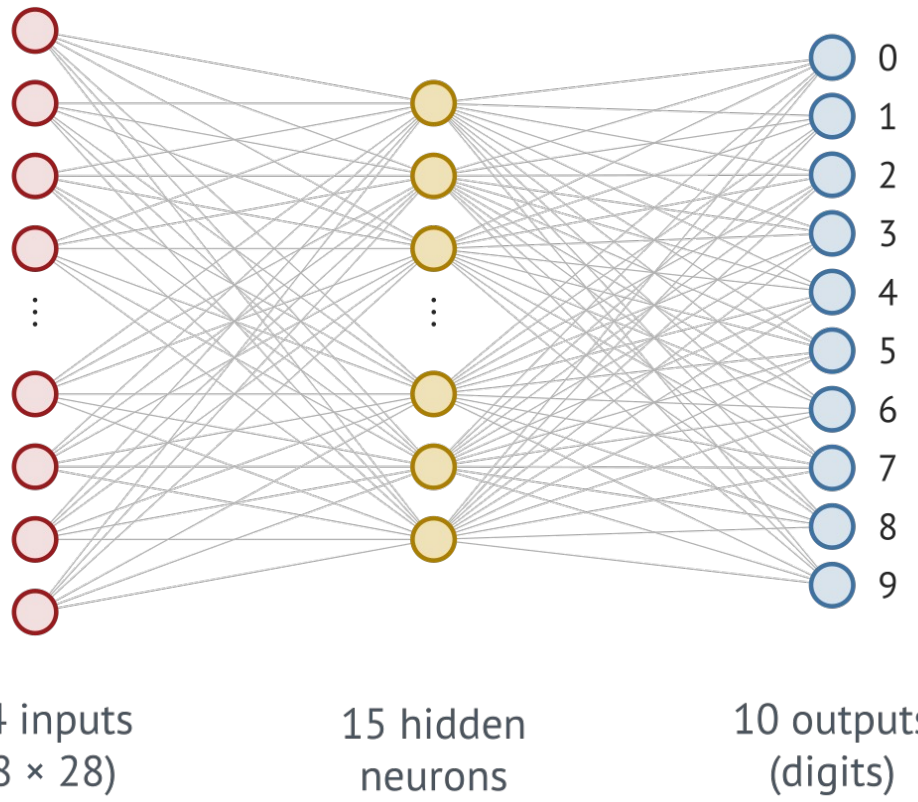
Segmented: one image per digit

- This number is **6 different images**
- Automatically recognizing and segmenting them is a bit outside the scope of what we cover here, and relatively doable
- So we will focus on a **single digit**
- With a digit classifier, segmenting can be done by cutting the image into many candidate pieces, scoring each segmentation with the classifier, and keeping the best

# An ANN for Handwriting

USC University of Southern California

784 pixel inputs, one hidden layer of 15 neurons, and 10 outputs, one for each digit.




- **Inputs**: 28 × 28 pixel scans of handwritten digits: **784** input neurons, greyscale with 0 = white and 1 = black
- **One hidden layer** of only **15** neurons; try other sizes to find what works
- **Output layer**: **10** neurons, one per digit
- For a given digit, one neuron should be 1 and the others 0; in reality we pick the neuron with the highest value

**Weights and biases to learn**

$$(784 \times 15 + 15) + (15 \times 10 + 10) = 11\,935$$

# An ANN for Handwriting

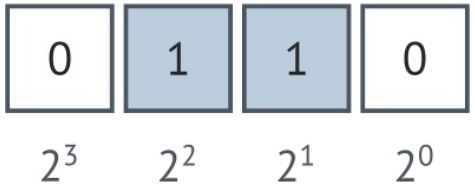
Four binary outputs could code ten digits, so why use ten output neurons?

**10 outputs: one per digit**



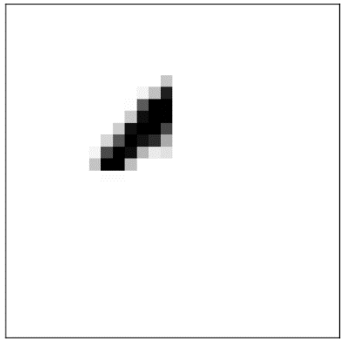

the digit 6 lights one neuron

**4 outputs: a binary code**




$0110_2$ = 6: the network must find the digit, then encode it

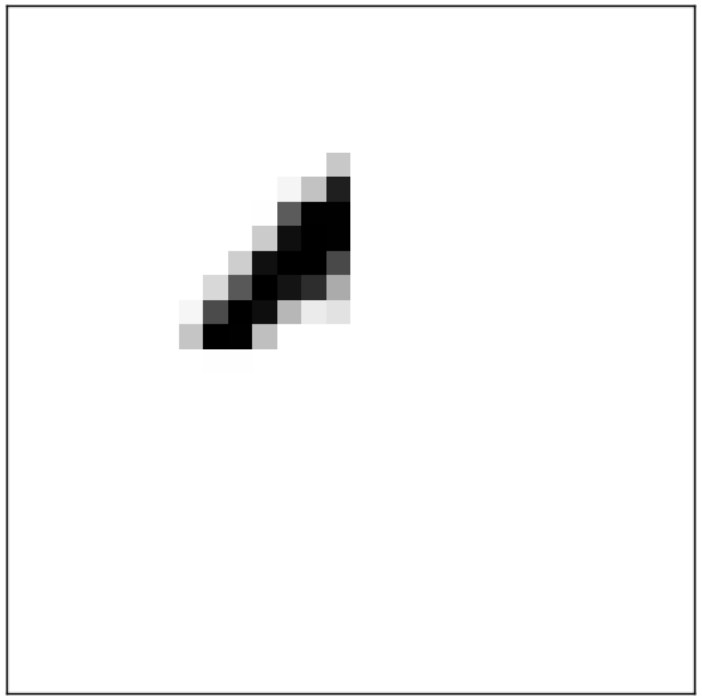
- **Why do 10 neurons work better than 4?**
- The '0' output neuron sums up the hidden-layer outputs and decides whether the digit is a 0
- So what are the hidden neurons doing?
- Heuristic: the **first hidden neuron** detects whether a certain pattern is present



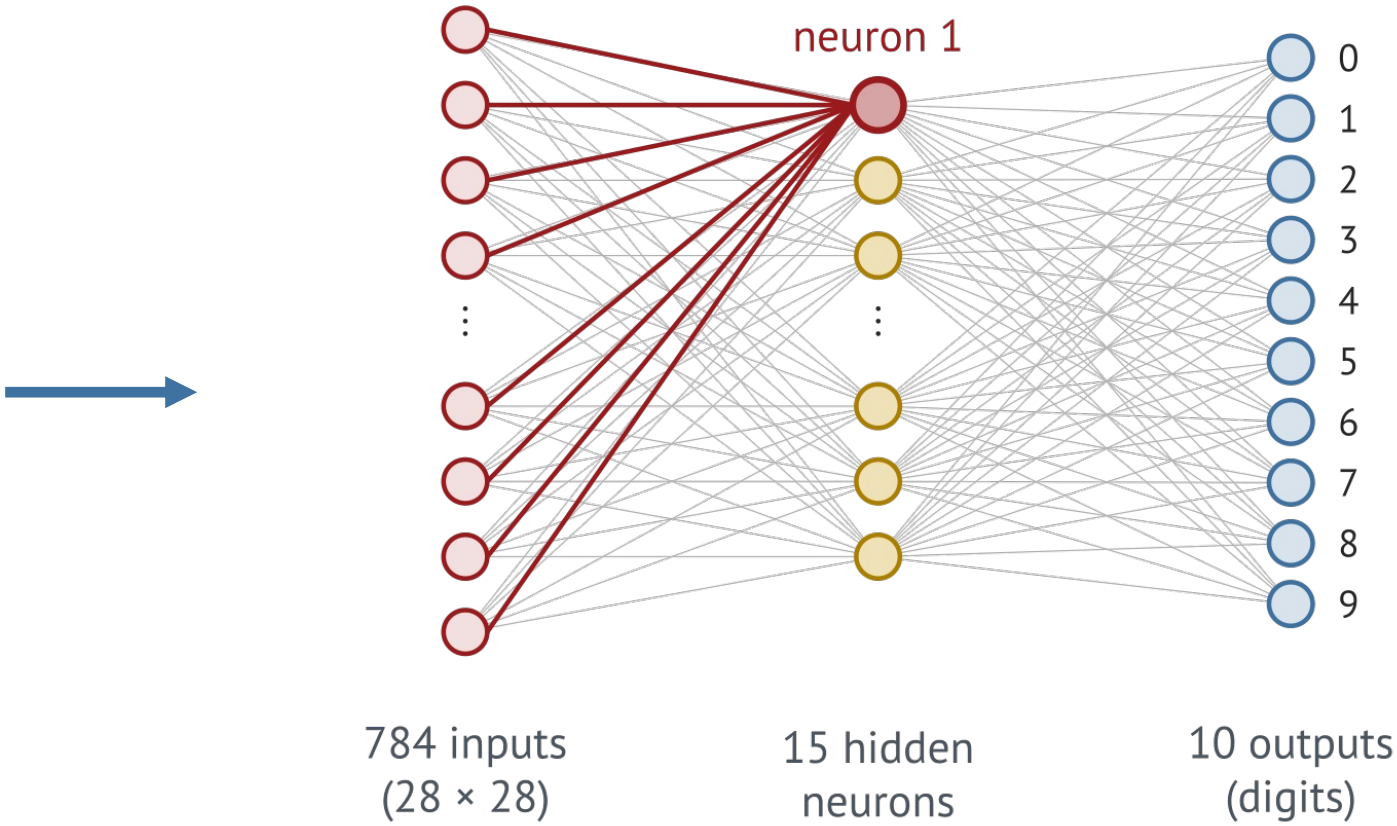
A pattern the first hidden neuron might look for: a short stroke.

# How does the hidden layer neuron find a specific image?

Every pixel connects to every hidden neuron, so each one can look for its own pattern.



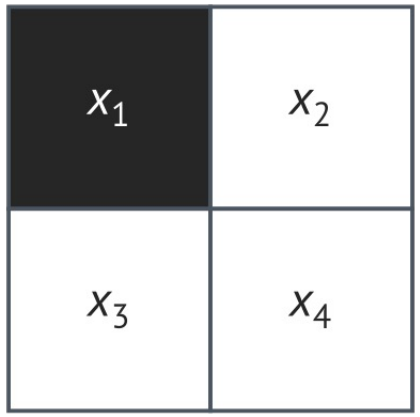
The pattern we want neuron 1 to detect.

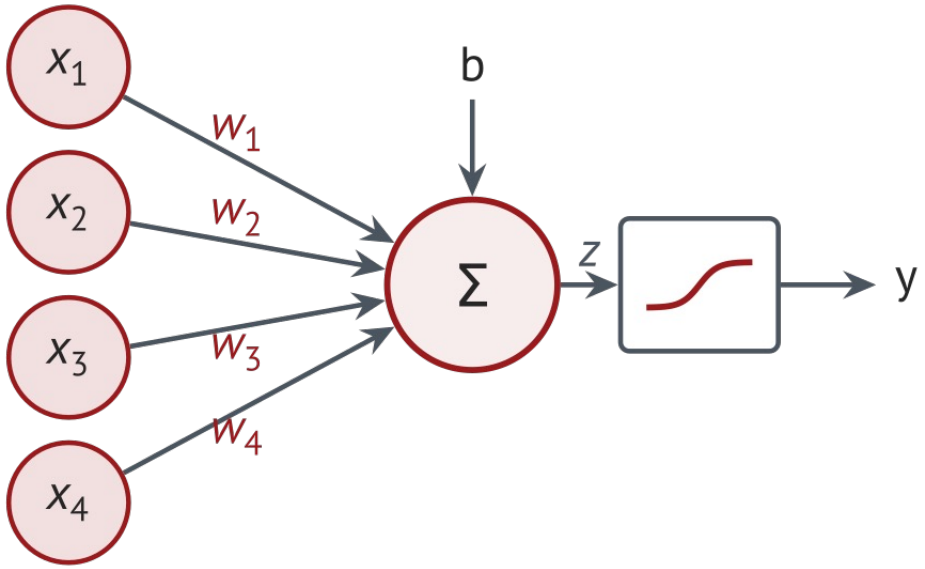



- Every input is connected to the hidden layer: **how would we weight and bias** neuron 1 so it finds that pattern?

# Let's do a simple example

Design one hidden neuron that tells us whether the top-left pixel of a 2 × 2 image is filled.




**EXAMPLE**

- Four inputs, $x_1$ to $x_4$: 0 = light, 1 = dark
- We want the output y to be **high** if $x_1$ is dark and $x_2$ to $x_4$ are light
- Choose $w_1$ to $w_4$ and b

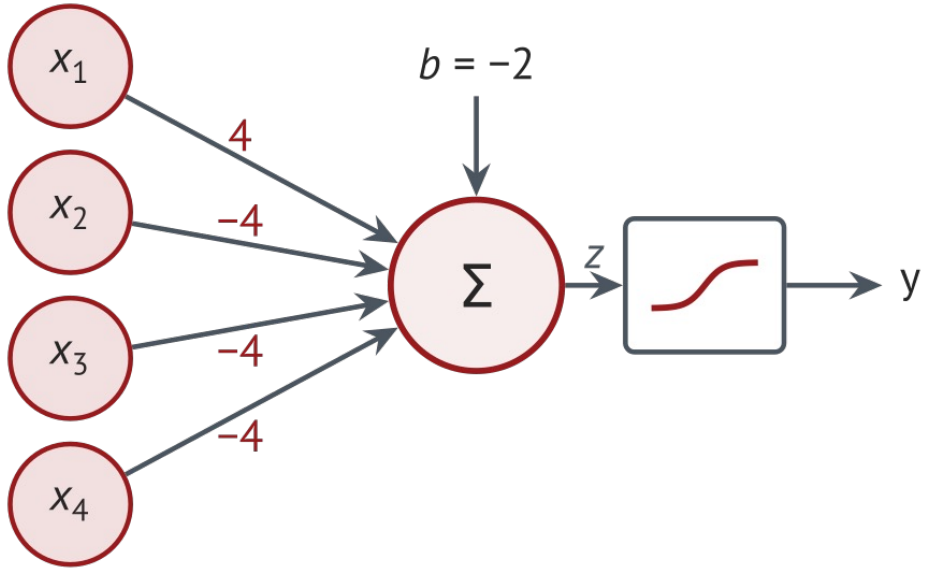
**Recall: the sigmoid neuron**

$$y = \sigma(\mathbf{x} \cdot \mathbf{w} + b), \qquad \sigma(z) = \frac{1}{1 + e^{-z}}$$

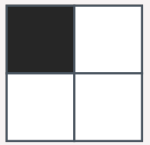
# Let's do a simple example

Weight $x_1$ heavily, make the other weights negative, and use a negative bias.

**SOLUTION**



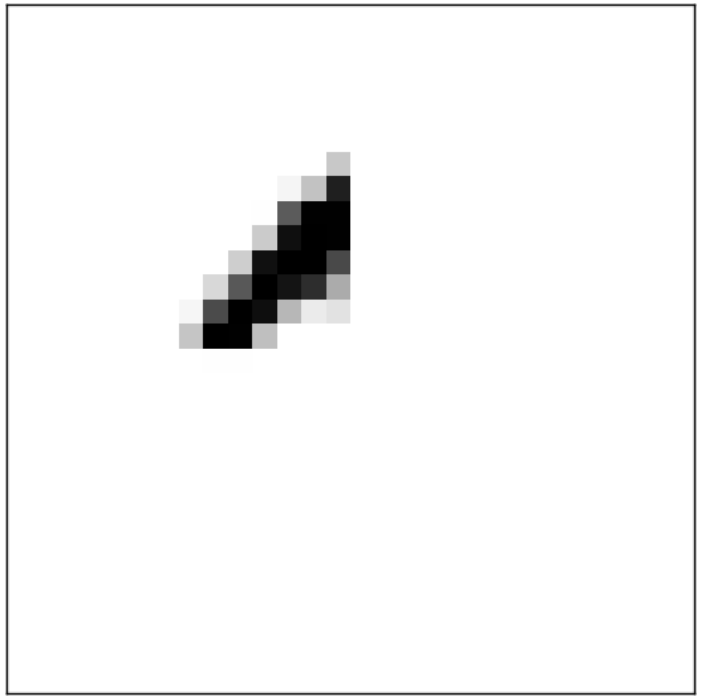

1. Weight $w_1$ heavily: $w_1 = 4$
2. Dramatically lower the weights of $x_2$ to $x_4$: −4 each
3. Bias b = −2 keeps a blank image low

| | Image | x | z | y = σ(z) |
|---|---|---|---|---|
|  | **Perfect match** | **(1, 0, 0, 0)** | **2** | **0.881** |
|  | Blank image | (0, 0, 0, 0) | −2 | 0.119 |
|  | All black | (1, 1, 1, 1) | −10 | ≈ 0 |
|  | **Noisy match** | **(0.8, 0.1, 0.1, 0)** | **0.4** | **0.599** |

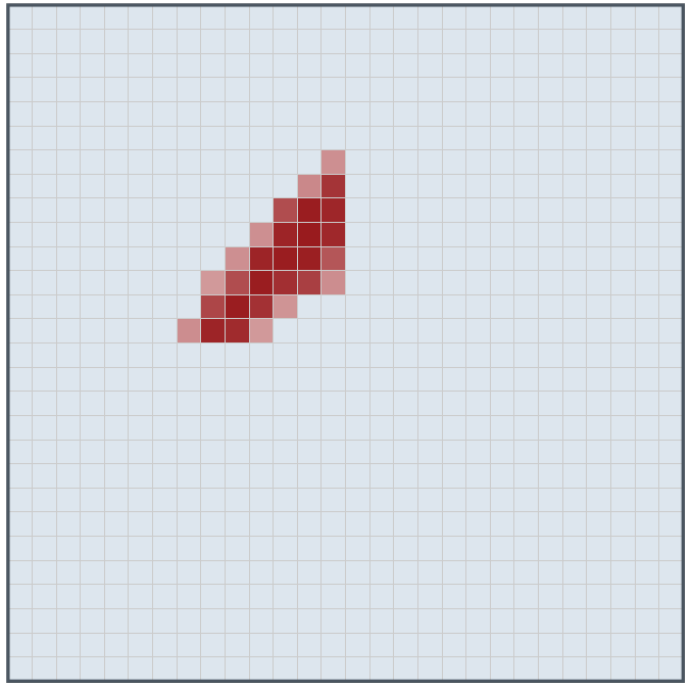
High (> 0.5) only for the match and the noisy match.

# Back to the task at hand

Give the pattern's pixels large positive weights and every other pixel a negative weight.



Pattern (input image)



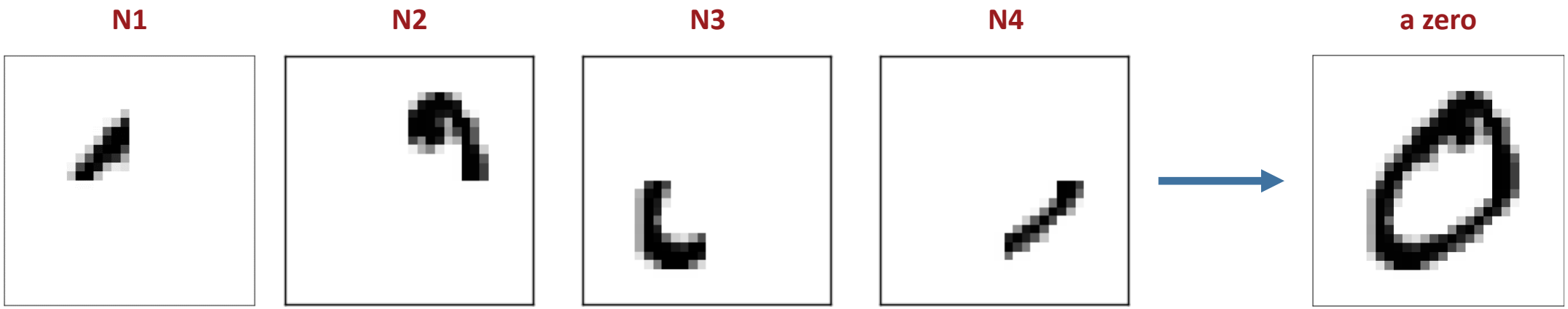
Weights of neuron 1: red positive, blue negative

- Weight the pixels that make up this pattern **heavily**
- Reduce the weights of the **other pixels**
- The weights form a **template**: w·x is large only when the image matches

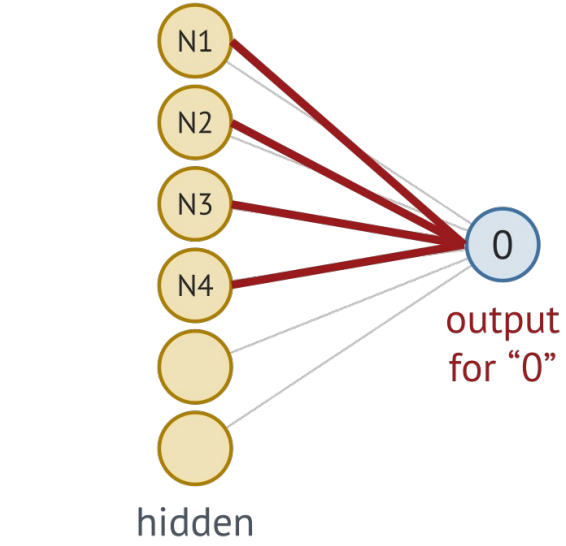
$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) \approx 1 \text{ only when the stroke is present}$$

# Other hidden neurons could be detecting other images

Four hidden neurons each find one arc; when all four fire, the “0” output neuron fires.

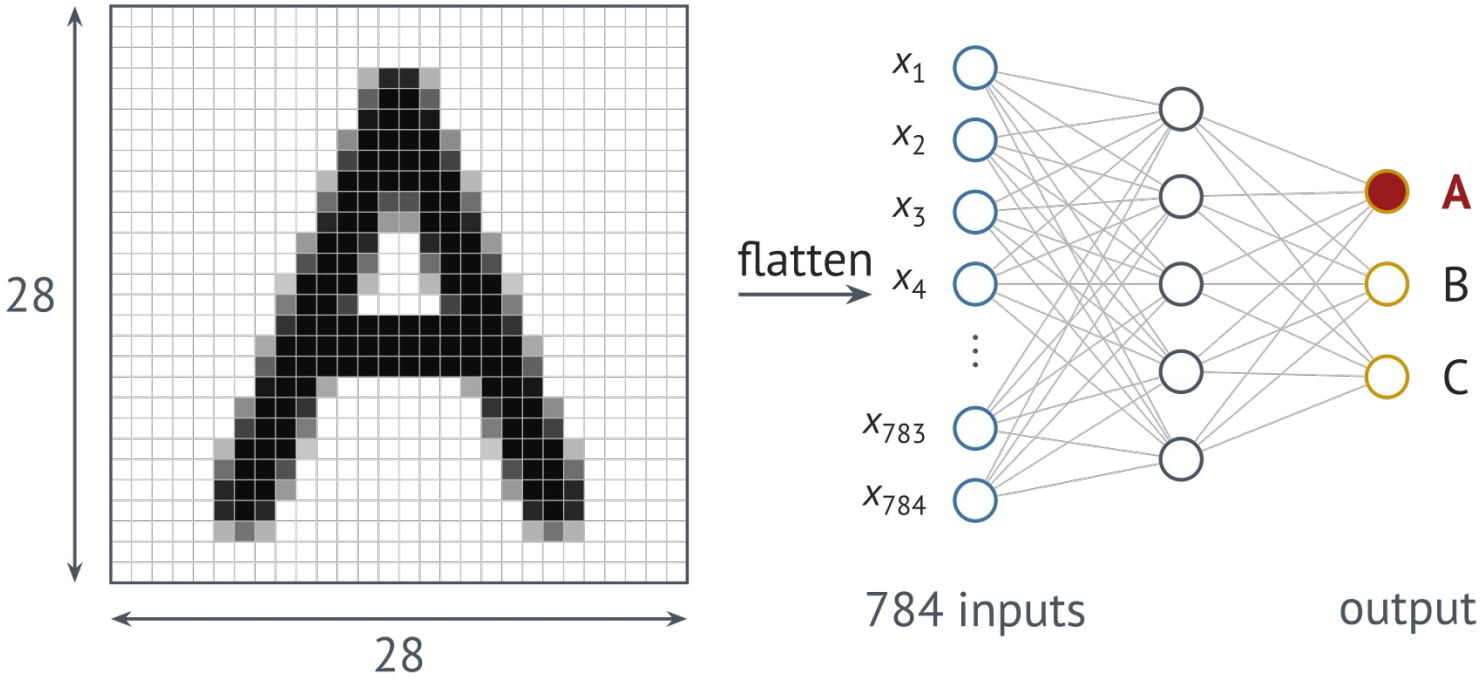



- Neurons 2, 3 and 4 could detect these other patterns; together the 4 images make up a zero
- If all 4 neurons fire, we assume we are seeing a **zero** (a toy example)
- The ‘0’ output neuron puts most weight on N1 to N4 and weakens the others
- With only **4 outputs** the network would need **2 calculations**: find the number, then convert it to binary



# Image processing/classification

A 28 × 28 image is 784 pixel values; the network looks for groups of pixels that form a pattern.

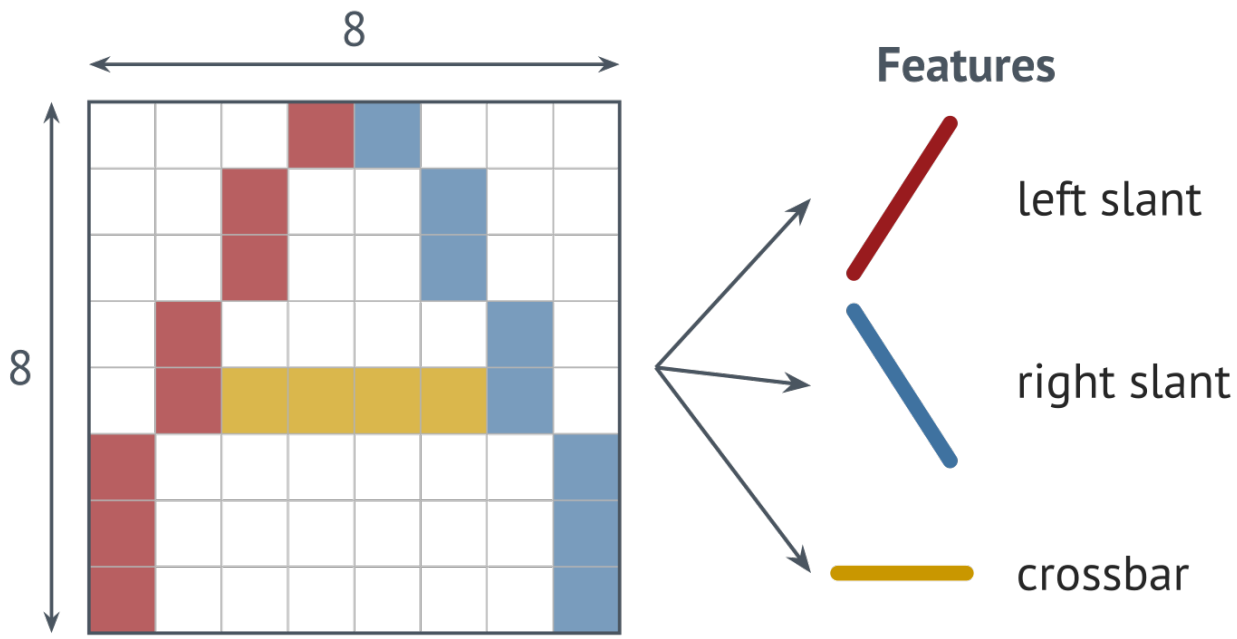



- Each pixel value comes from one photodiode of the image sensor
- 28 × 28 = **784** pixel values, one input each
- **Step 1**: check whether groups of pixels form a pattern, such as the strokes of the A

The handwritten digits of the MNIST lab use the same 28 × 28 format.

# Image processing/classification

Preprocessing shrinks the image to 8 × 8 = 64 pixels; features such as strokes describe the letter.




- **Preprocessing**: shrink 28 × 28 to 8 × 8 = **64** pixels; the shape survives
- **Features**: describe the letter by its parts, two slants and a crossbar
- A few features instead of every pixel: fewer inputs, smaller model, less training data



# Learning how ANNs Learn with your Biological Neural Net

USC University of Southern California

We will use the MNIST dataset: 60,000 training and 10,000 test images of handwritten digits.

We will use the **MNIST** data set:

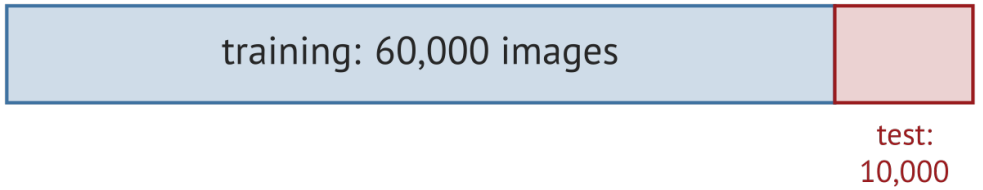
http://yann.lecun.com/exdb/mnist/

- **Two parts**
  - **60,000 training images** (28 × 28 pixels, greyscale)
  - Scanned samples from more than 200 people: ½ government employees, ½ high-school students
  - **10,000 test images**
  - Test data were collected from different people



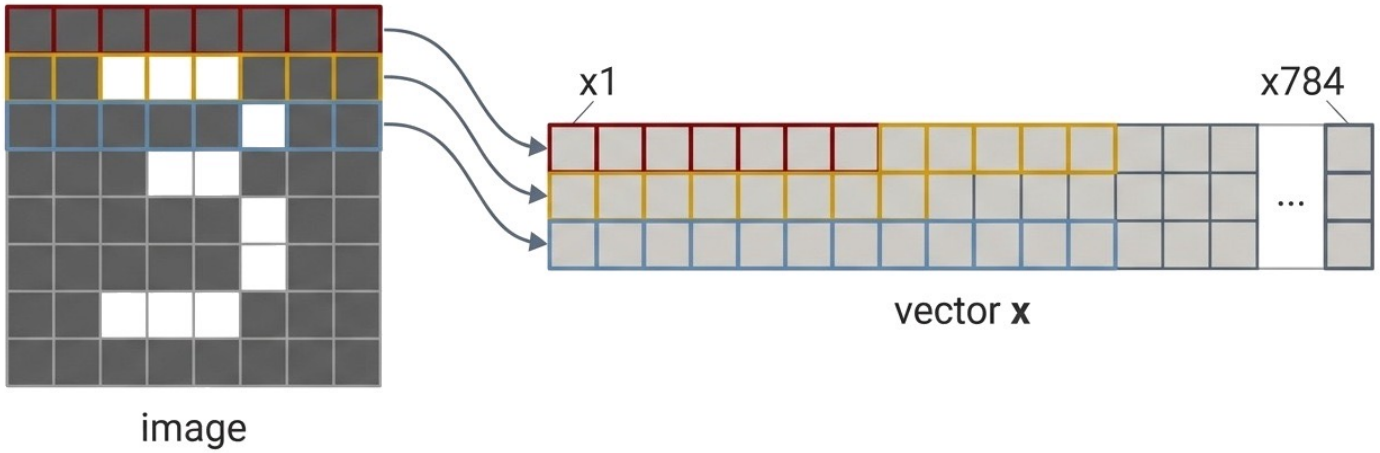
Digits in the MNIST format: white strokes on black, centered in a 28 × 28 frame.

**Why different writers?** A good test score then means the network learned digits, not the handwriting of the people in the training set.



# How to train your artificial neural network

Each training image x comes with a label y(x); training finds weights that make the output match it.




The input: an image read row by row becomes the vector x of 784 values between 0 and 1.

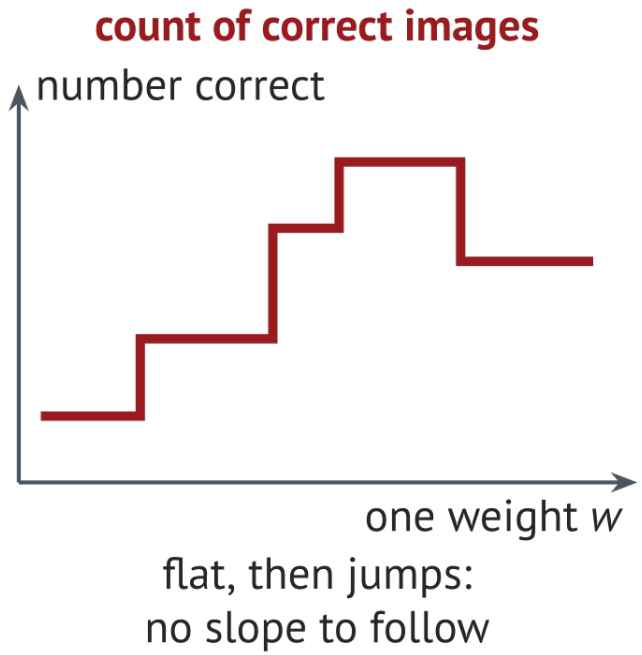


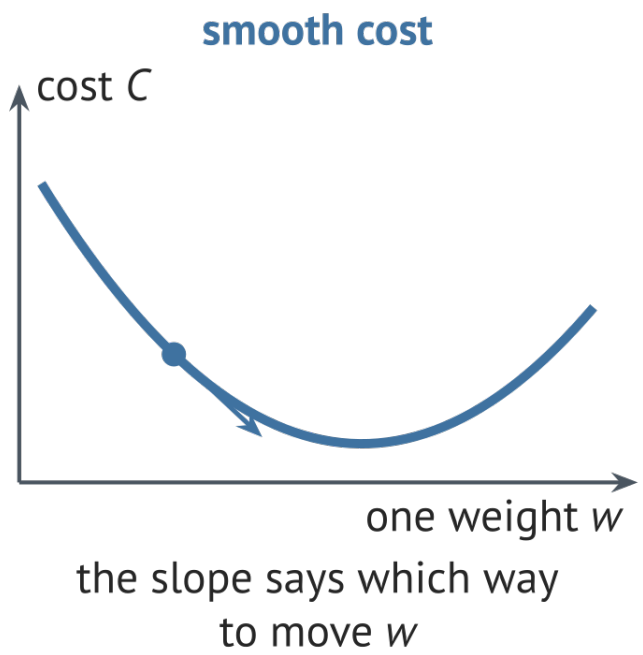

The label for an image of a 1: a 1 in position 1, zeros elsewhere.

- x: the input vector, **784** values between 0 and 1 (greyscale)
- The output is a vector y = y(x) with **10** entries
- If x is an image of a 1: y(x) = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0)
- We need an algorithm that finds the weights and biases so the output is **as close to y(x) as possible**

# How to train your artificial neural network

The count of correct images is what we want, but it is a poor thing to optimize directly.




- We need to decide **what to optimize**
- Obviously, the number of images correctly recognized
- But what if we used that as the objective? C(w, b) = number of correctly classified images
- Could we optimize it easily? **Not really**: it is not a smooth function of w (all the weights) or b (all the biases)

A small weight change usually changes no classification: the count is flat, then jumps.

# How to train your artificial neural network

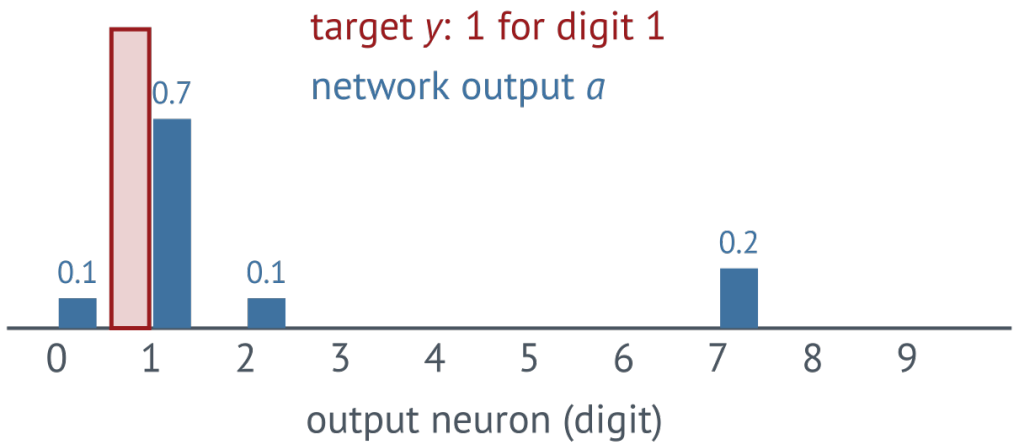
USC University of Southern California

So we define a smooth cost function that measures how far the outputs are from the labels.

**Quadratic cost (mean squared error)**

$$C(\mathbf{w}, \mathbf{b}) = \frac{1}{2n} \sum_{\mathbf{x}} \|\mathbf{y}(\mathbf{x}) - \mathbf{a}(\mathbf{x})\|^2$$

- n training images; a(x) is the network output
- C ≥ 0, and C = 0 only if every output equals its label
- A **smooth** function of every weight and bias
- Training = find the w and b that **minimize C**

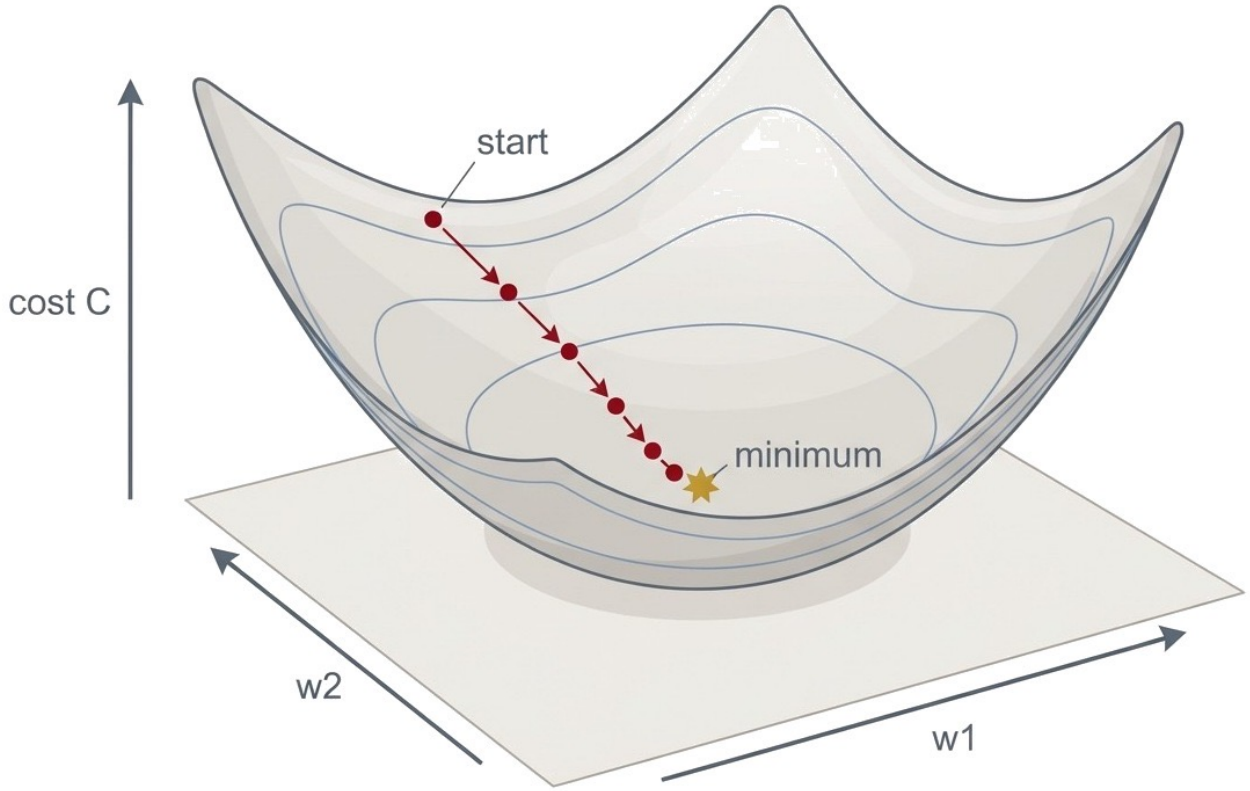



**EXAMPLE**

One image of a 1: ½(0.1² + 0.3² + 0.1² + 0.2²) = ½(0.15) = **0.075**

# Gradient descent

Feel the slope of the cost where you stand and take a small step downhill; repeat.




Each step moves the parameters against the local slope, toward the valley where the cost is low.

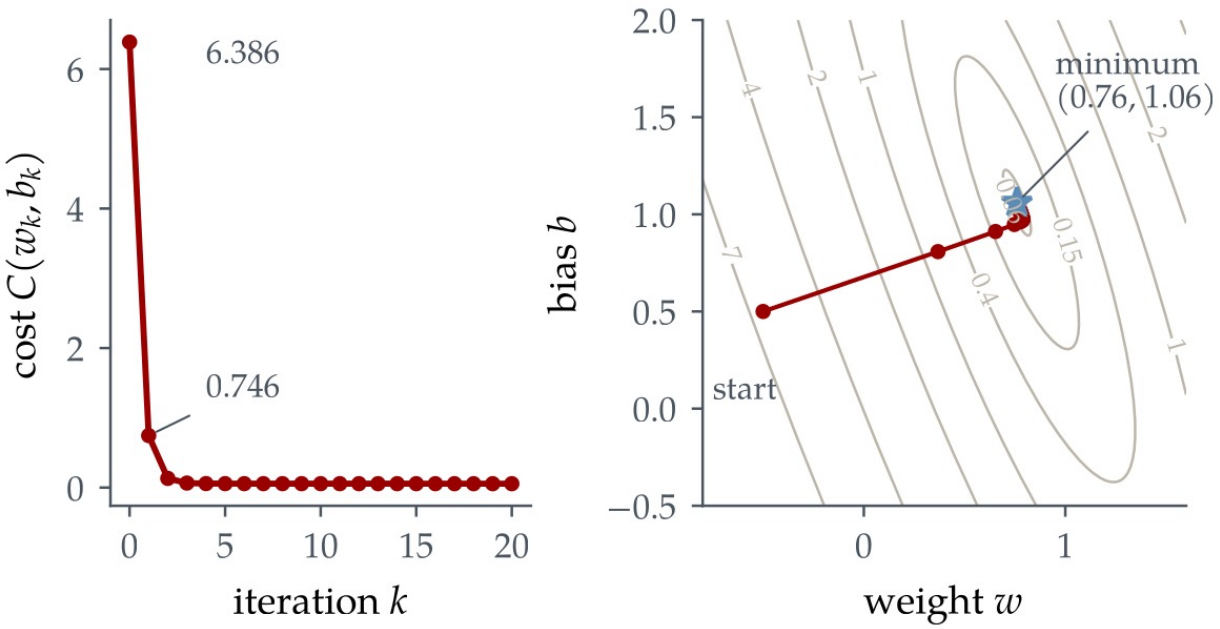
**Gradient descent update**

$$\mathbf{v} \leftarrow \mathbf{v} - \gamma \nabla C$$

- v: all the weights and biases
- ∇C points uphill, so we step the **other way**
- γ is the **learning rate**, the step size
- Each small step lowers the cost:
  $\Delta C \approx -\gamma \, |\nabla C|^2$

# Gradient descent: an example

Fit a line a = wx + b to five points: the first step cuts the cost from 6.39 to 0.75.




Left: the cost per iteration. Right: the path of (w, b) on the contours of C, ending at the minimum.

**Gradients of the quadratic cost**

$$\frac{\partial C}{\partial w} = \frac{1}{n}\sum_i (a_i - y_i)\, x_i, \qquad \frac{\partial C}{\partial b} = \frac{1}{n}\sum_i (a_i - y_i)$$

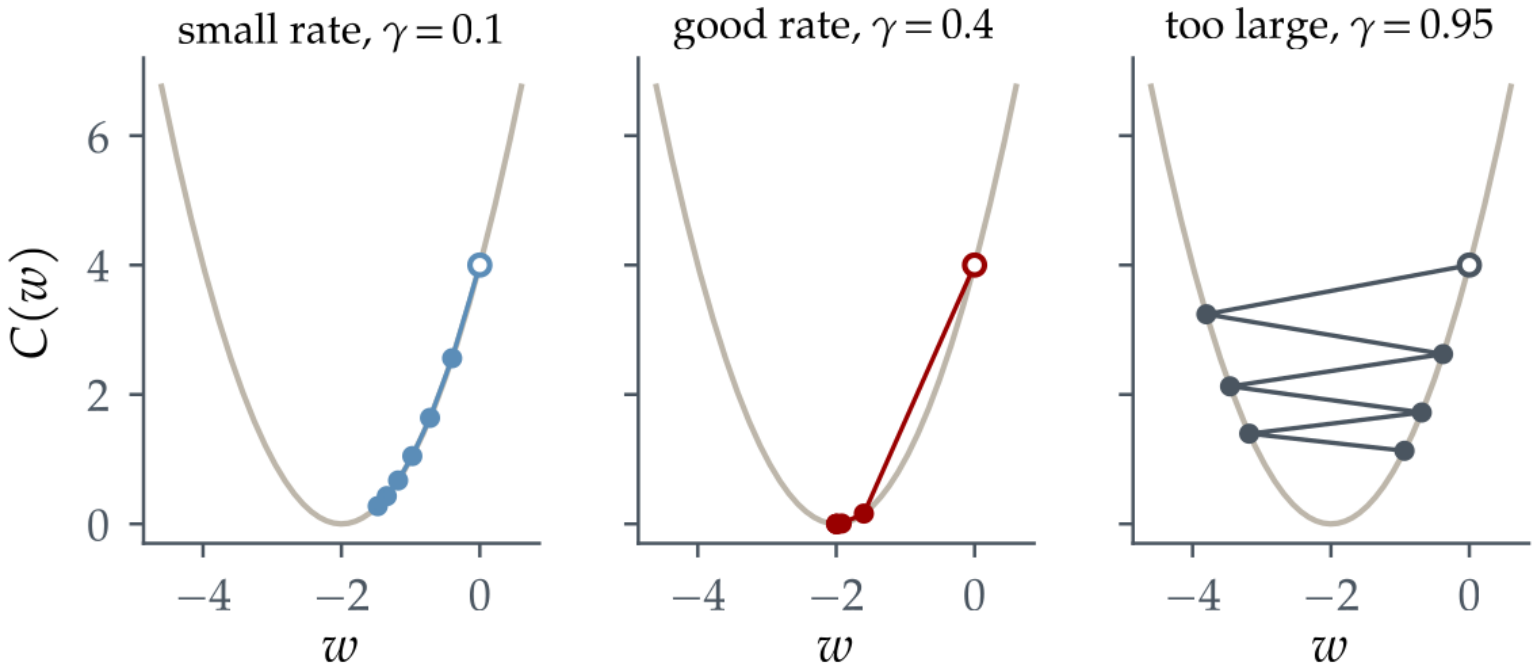
**EXAMPLE**

Data x = (0, 1, 2, 3, 4), y = (1, 2.2, 2.0, 3.6, 4.1); start (w, b) = (−0.5, 0.5), γ = 0.1

1. Errors a − y = (−0.5, −2.2, −2.5, −4.6, −5.6), so C = **6.386**
2. ∂C/∂w = **−8.68** and ∂C/∂b = **−3.08**
3. w = **0.368** and b = **0.808**: the cost falls to C = **0.746**
4. After about 100 steps: (w, b) = (**0.76**, **1.06**), C = 0.0552, the least-squares line

# Choosing the learning rate

Too small a step crawls, a good one gets there fast, and too large a step overshoots the valley.




Six gradient-descent steps on C(w) = (w + 2)² from w = 0 (open circle).

**EXAMPLE**

C(w) = $w^2$ + 4w + 4, from w = 0 with γ = 0.1

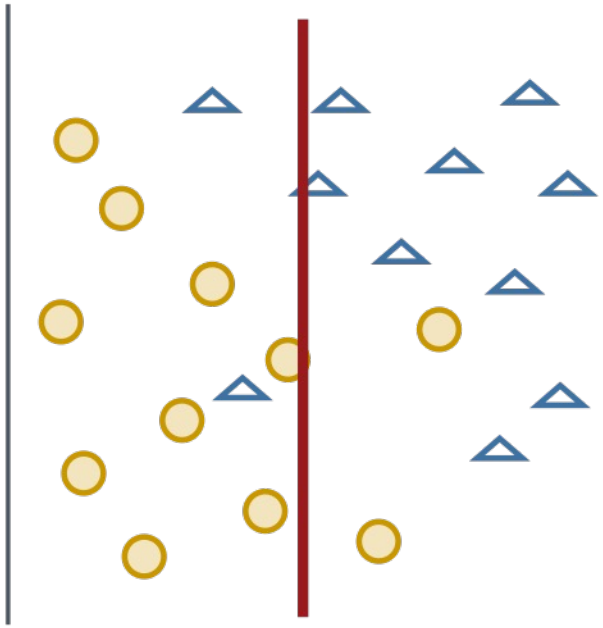
1. dC/dw = 2w + 4 = 4 at w = 0
2. w = 0 − 0.1 × 4 = **−0.4**
3. C drops from 4 to **2.56**; the minimum is at w = −2

$$w_{k+1} + 2 = (1 - 2\gamma)(w_k + 2): \quad \text{converges for } 0 < \gamma < 1$$

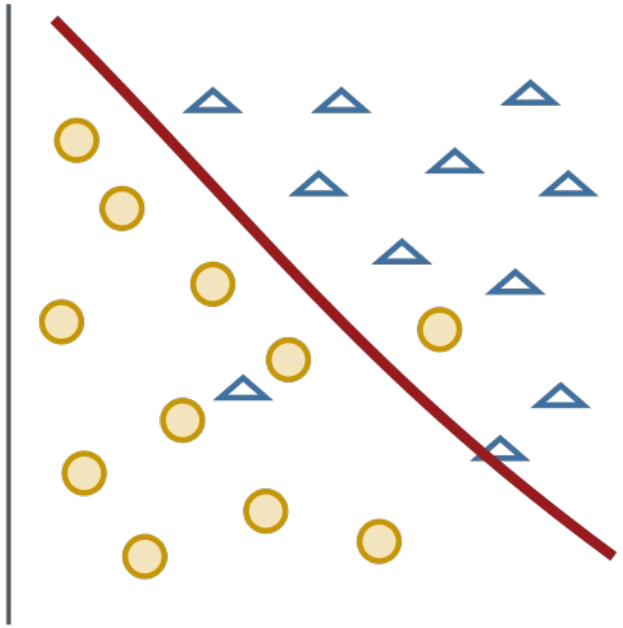
# Model fit

USC University of Southern California

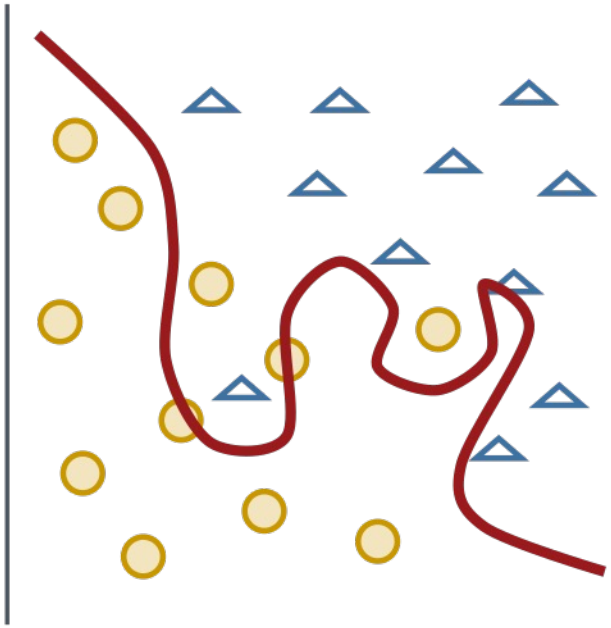
A good model captures the trend and generalizes; too simple underfits, too flexible overfits.



**Underfit**: the model fails to capture the trends in the data



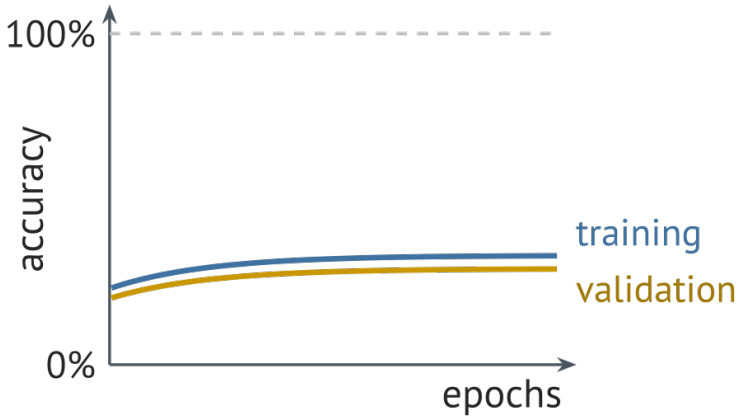
**Good fit**: captures the trends and generalizes to unseen data

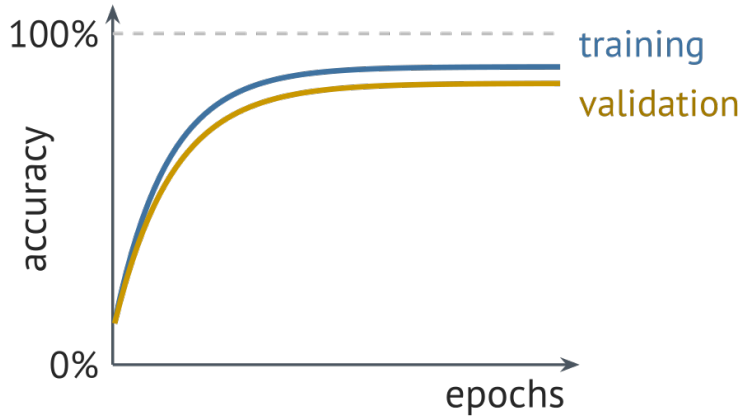


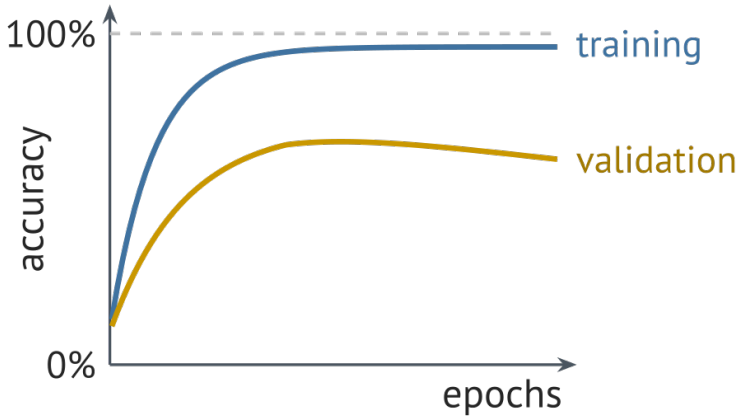
**Overfit**: captures the training data, noise included, but fails on unseen data

# Accuracy vs epoch to understand dataset

Plot training and validation accuracy after every epoch: the gap between the curves diagnoses the model.



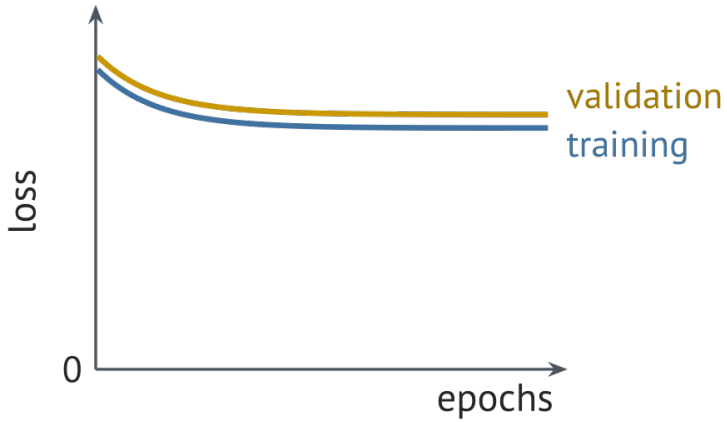

**Underfit**: poor on both training and validation data

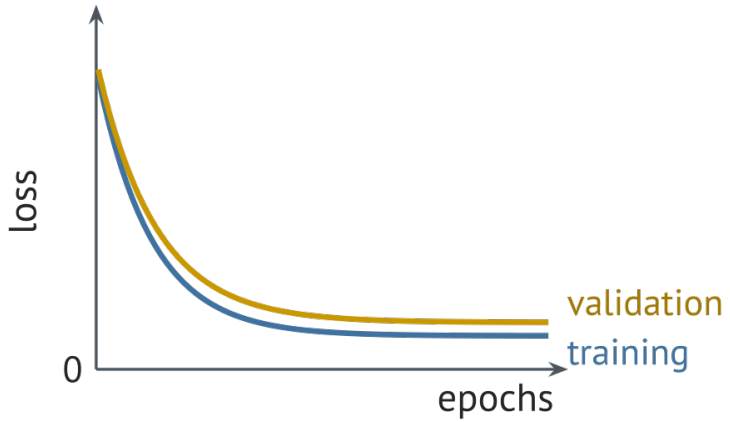
**Good fit**: generalizes well from training to validation data

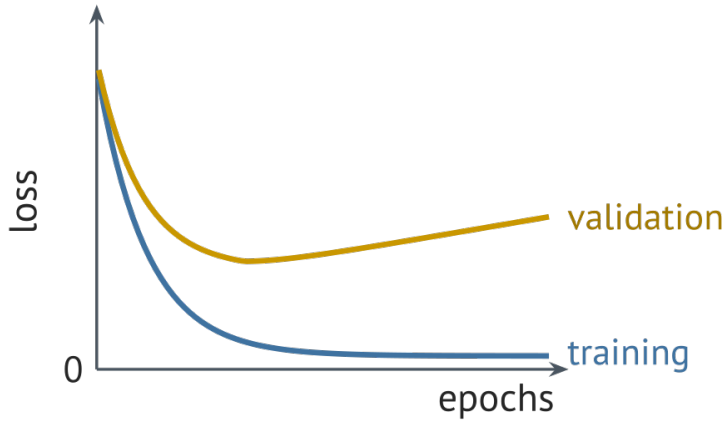
**Overfit**: predicts training data well but fails to generalize to validation data

# Loss vs epoch to understand dataset

The loss (cost) curves tell the same story upside down: low is good, and a growing gap means overfitting.



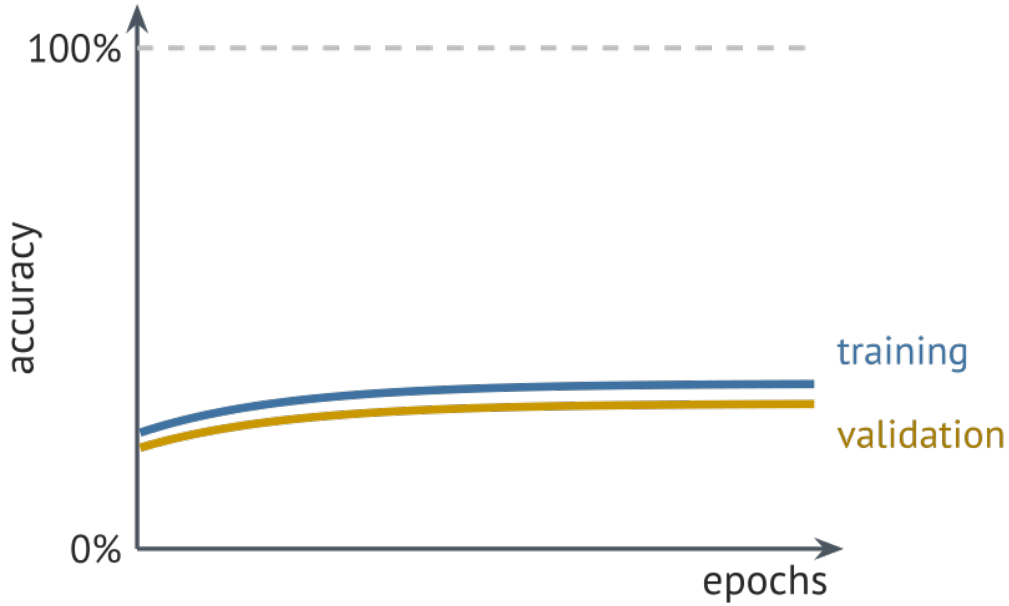

**Underfit**: poor on both training and validation data

**Good fit**: generalizes well from training to validation data

**Overfit**: predicts training data well but fails to generalize to validation data

# Fixing underfit

An underfit model is too weak for the data: give it more information, more training or more capacity.



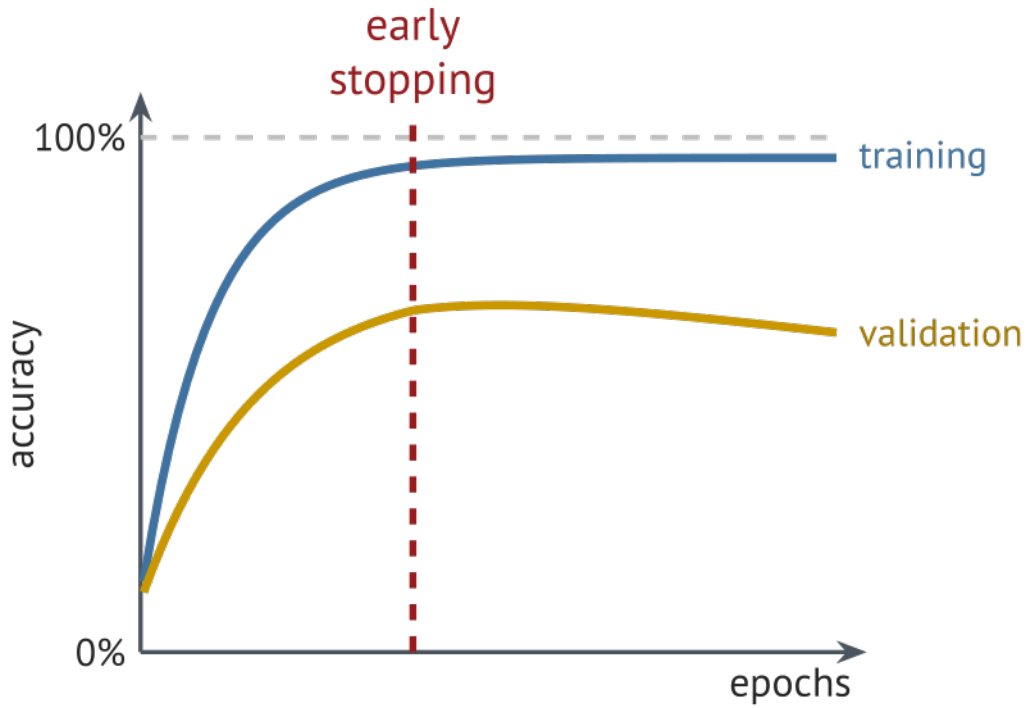

- Get more data
- Try different features or more features
- Train for longer
- Try a more complex model (more layers, more nodes)

**Underfit**: poor on training and validation data

# Fixing overfit

An overfit model memorizes the training set: stop earlier, simplify it, or give it more data.

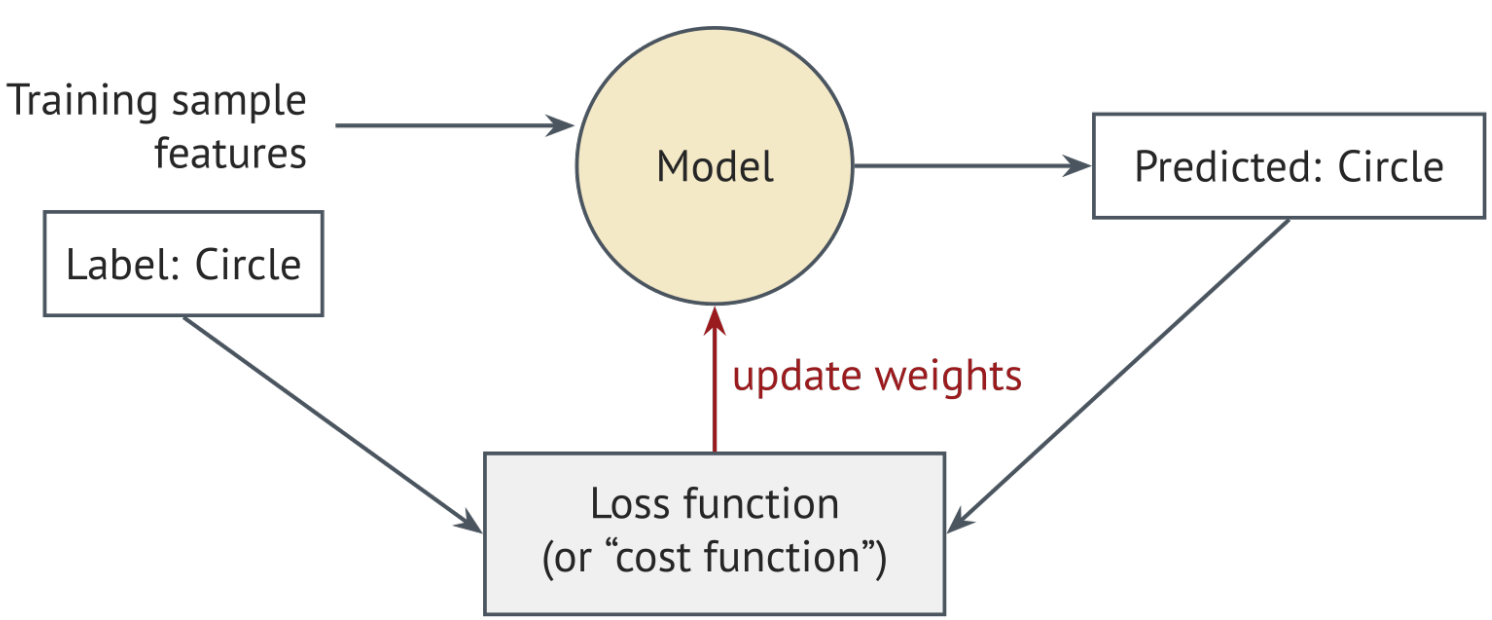
- Get more data
- Early stopping
- Reduce model complexity
- Add regularization terms
- Add dropout layers (for neural networks)

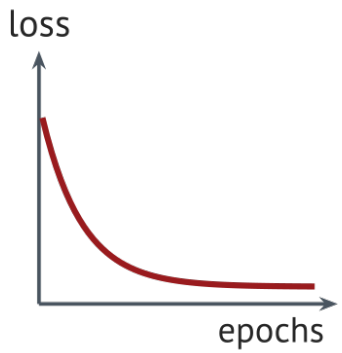


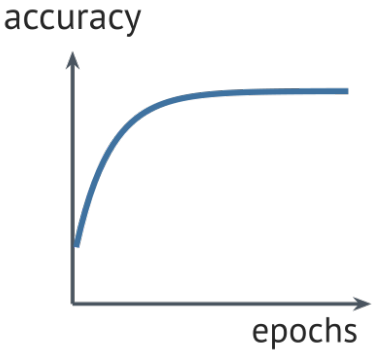

**Overfit**: training data well, validation data poorly

# Model characterization in realtime

During training, each sample's prediction is compared with its label; the loss drives the weight updates.




- For every training sample: predict, compare with the label, compute the loss, nudge the weights
- Watch loss fall and accuracy rise epoch by epoch while training runs