intel AI Academy

NEURAL NETWORKS

LEGAL NOTICES AND DISCLAIMERS

This presentation is for informational purposes only. INTEL MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

This sample source code is released under the Intel Sample Source Code License Agreement.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

INTRO TO NEURAL NETWORKS

NEURAL NETWORKS

A fancy, tunable way to get an *f* when given data and target.

- That is, *f*(data) → tgt

NEURAL NETWORK EXAMPLE: OR LOGIC

A logic gate takes in two Boolean (true/false or 1/0) inputs.

Returns either a 0 or 1, depending on its rule.

The truth table for a logic gate shows the outputs for each combination of inputs.

TRUTH TABLE

For example, let's look at the truth table for an Or-gate:

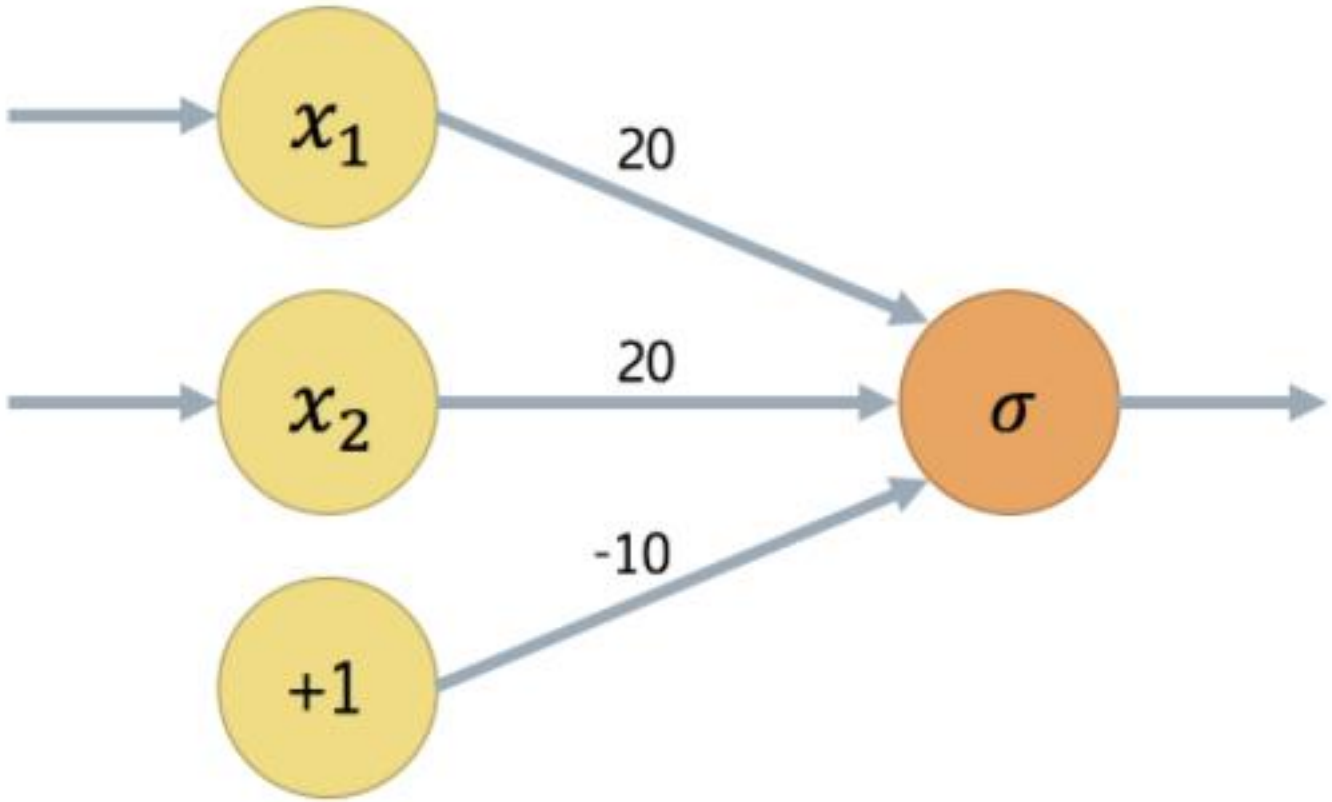
OR gate truth table		
Input		Output
0	0	0
0	1	1
1	0	1
1	1	1

OR AS A NEURON

A neuron that uses the sigmoid activation function outputs a value between (0, 1).

This naturally leads us to think ab

Imagine a neuron that takes in tw

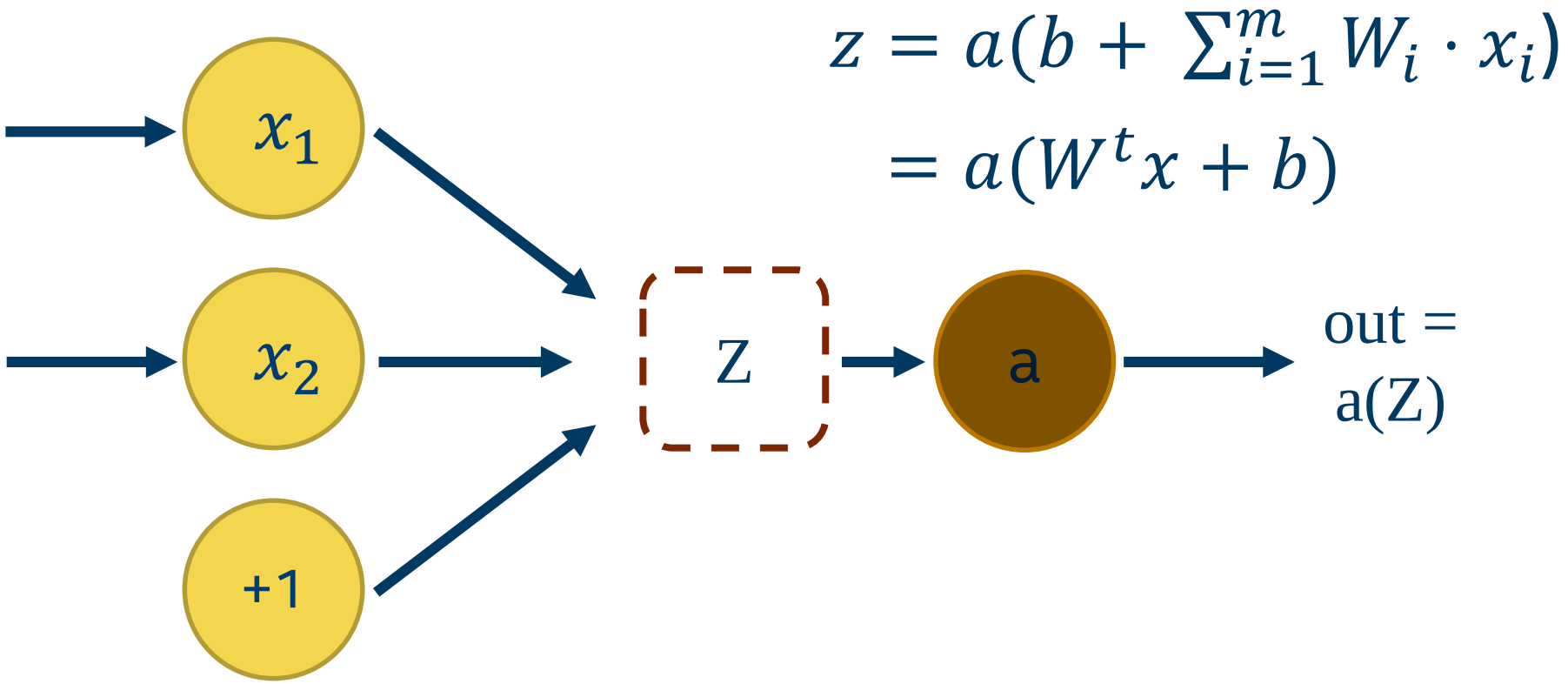


COMPONENTS

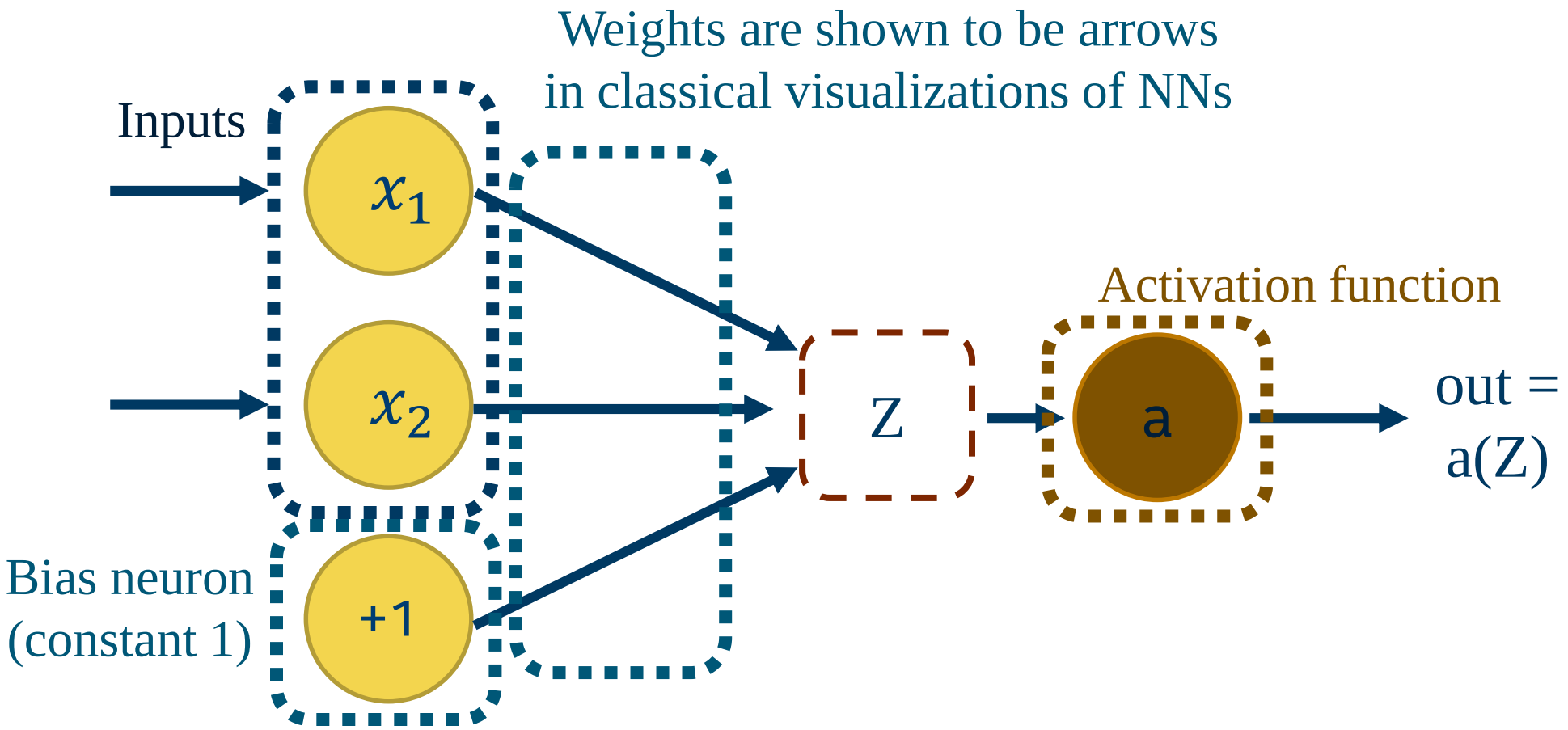
NODES

Nodes are the primitive elements.

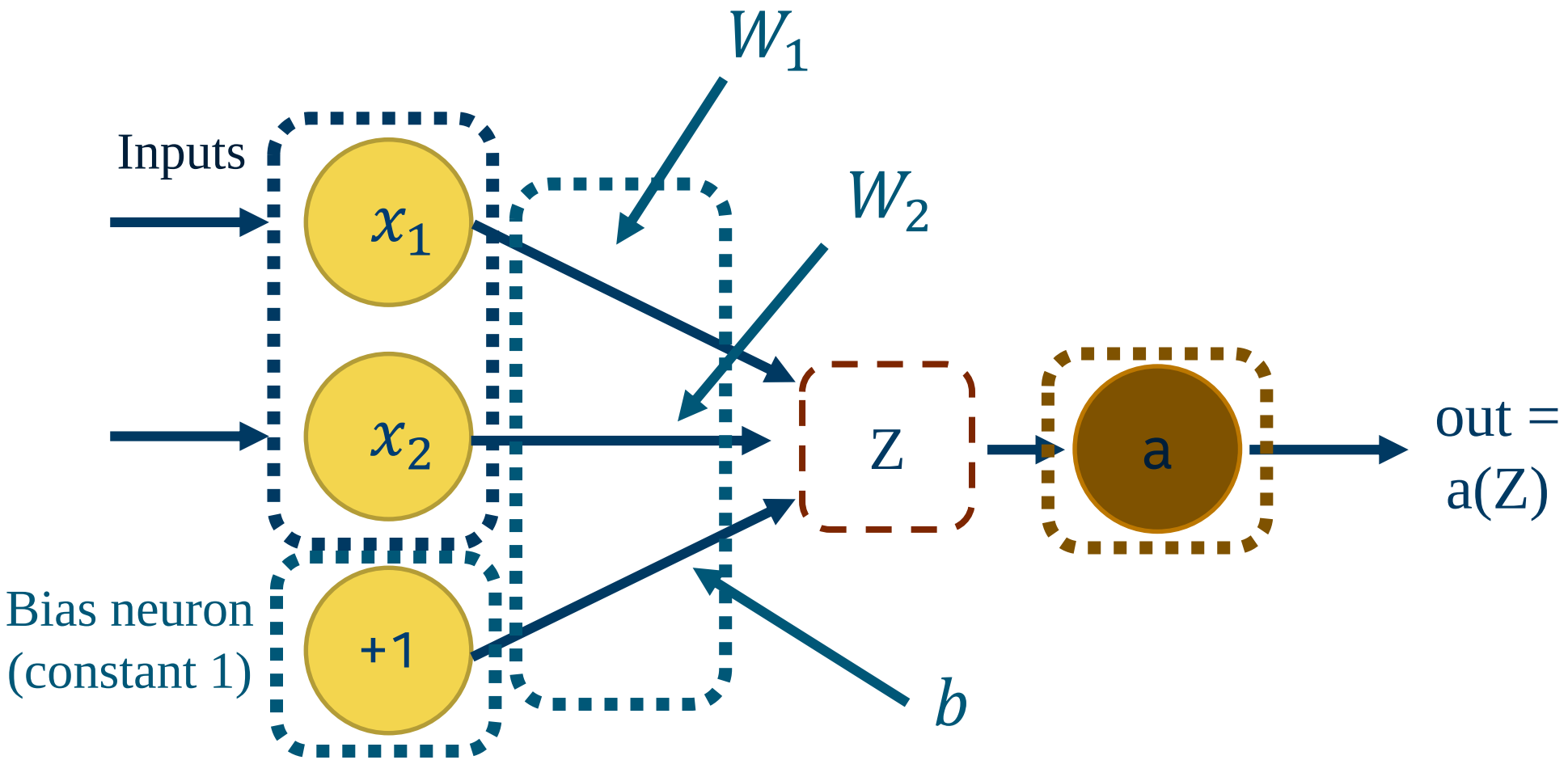
out = activation(f(in) + bias)



CLASSIC VISUALIZATION OF NEURONS



CLASSIC VISUALIZATION OF NEURONS



TRAINING

z is a dot-product between inputs and weights of the node.

- sum-of-squares

We initialize the weights with constants and/or random values.

Learning is the process of finding good weights.

ACTIVATION FUNCTION: SIGMOID

Model inspired by biological neurons.

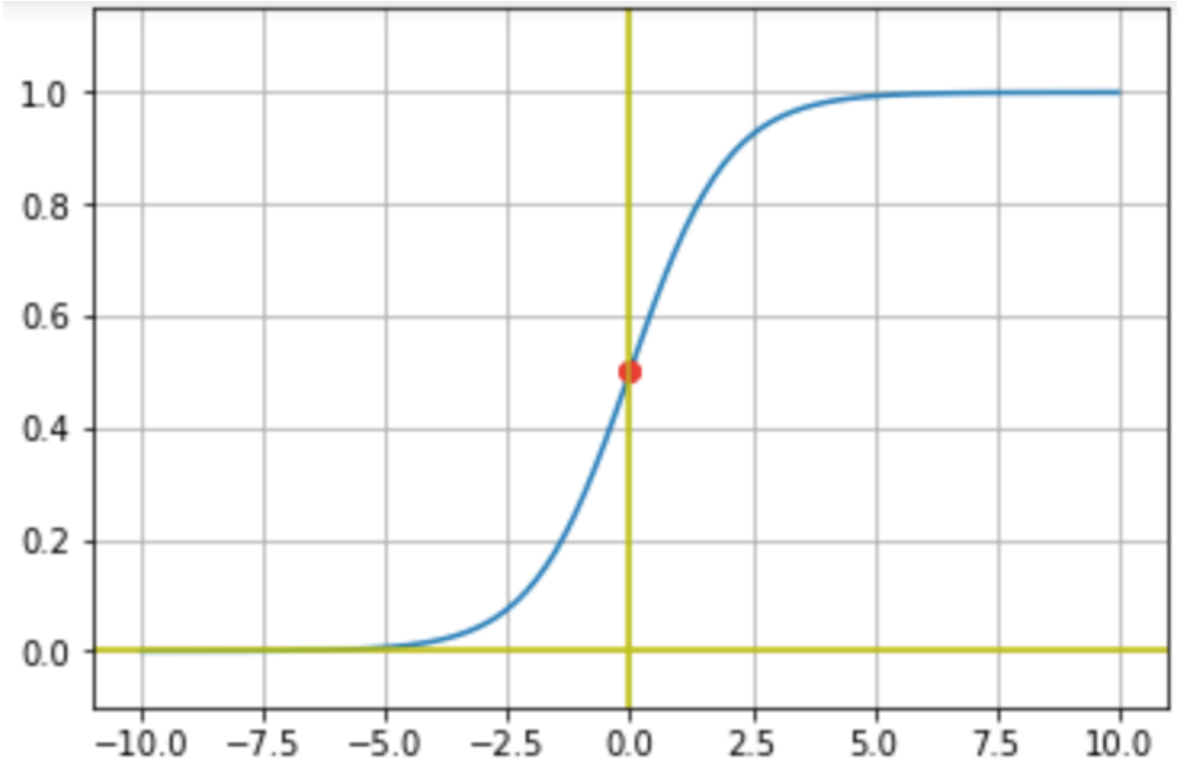
Biological neurons either pass no signal, full signal, or something in between.

Want a function that is like this *and* has an easy derivative.

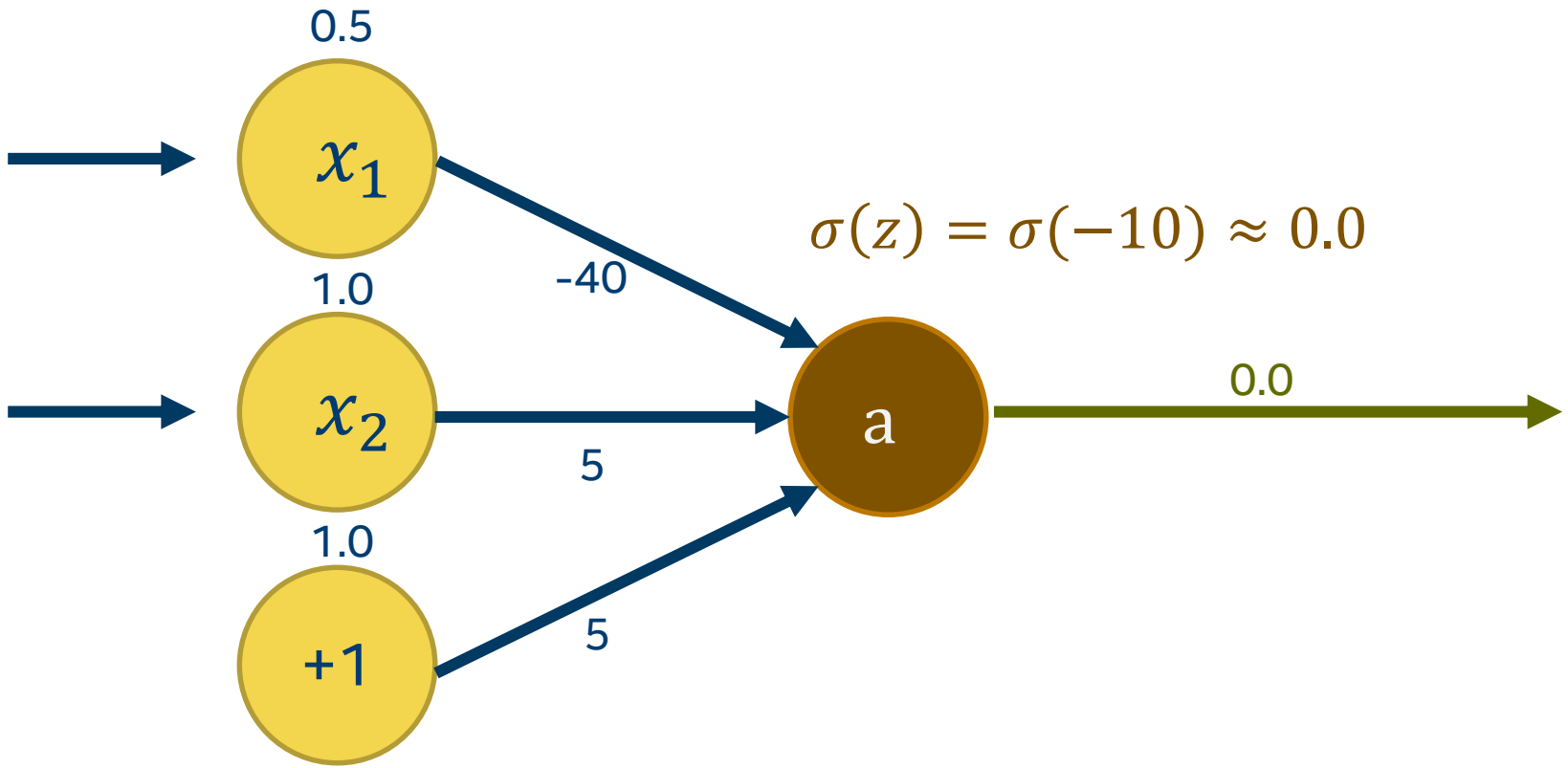
ACTIVATION FUNCTION: SIGMOID

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

- Value at $z \ll 0$? ≈ 0
- Value at $z = 0$? $= 0.5$
- Value at $z \gg 0$? ≈ 1



ACTIVATION FUNCTION: SIGMOID



ACTIVATION FUNCTION: RELU

Many modern networks use rectified linear units (ReLU)

$$ReLU(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

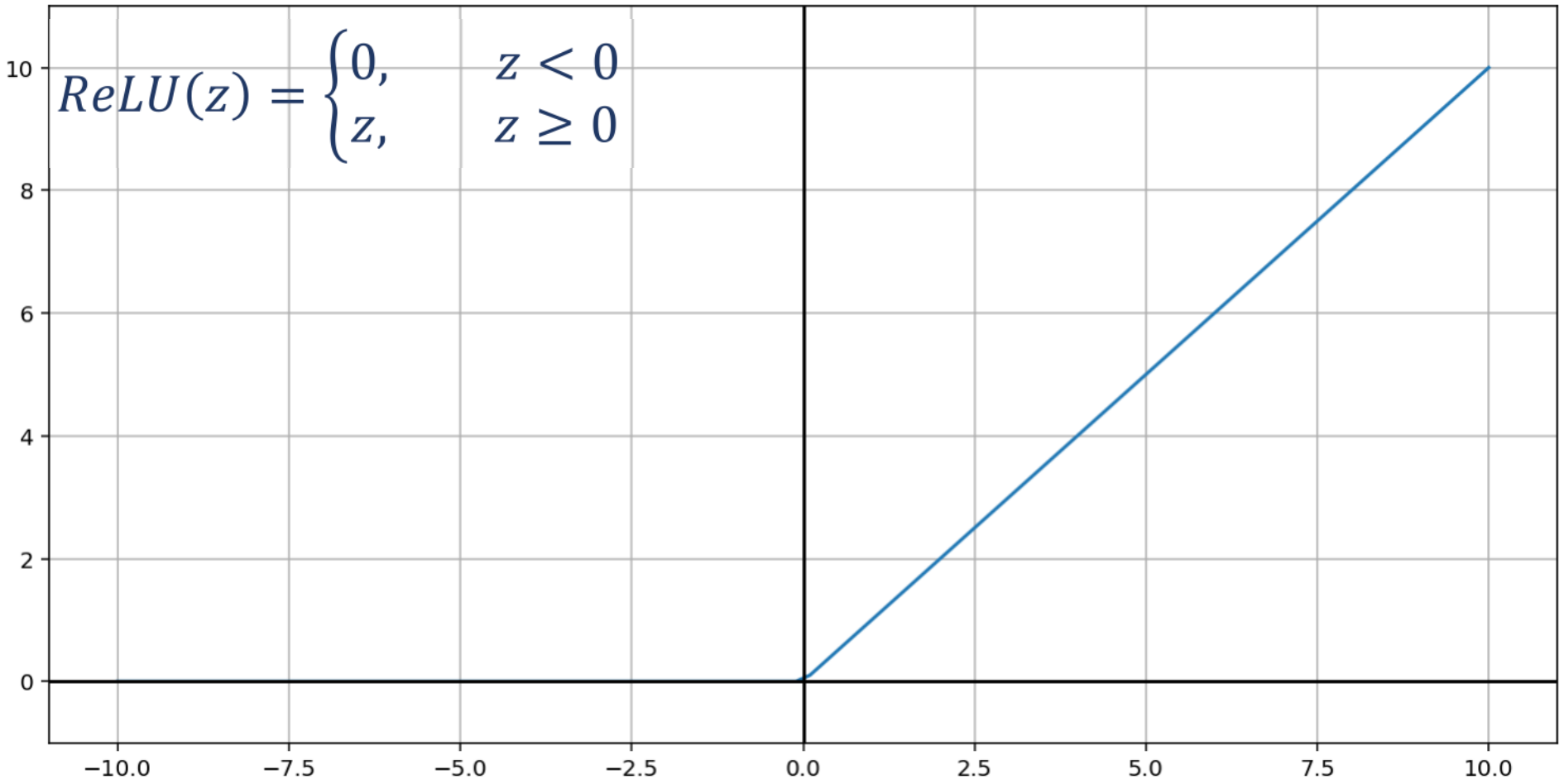
$$= \max(0, z)$$

Value at $z \ll 0? = 0$

Value at $z = 0? = 0.$

Value at $z \gg 0? = z$

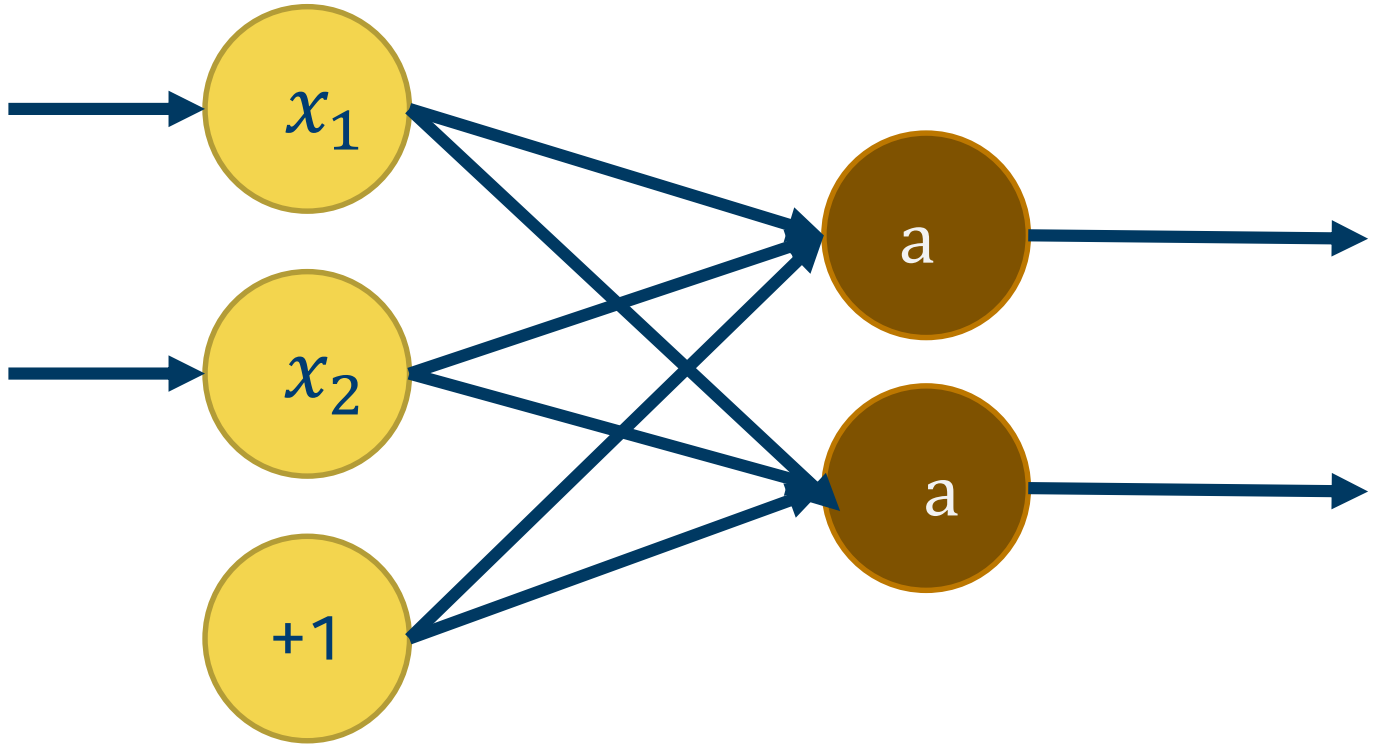
ACTIVATION FUNCTION: RELU



LAYERS AND NETWORKS

Inputs don't need to be limited to passing data into a single neuron.

They can pass data to as many as we like



LAYERS AND NETWORKS

Typically, neurons are grouped into *layers*.

Each neuron in the layer receives input from the same neurons.

Weights are different for each neuron

All neurons in this layer output to the same neurons in a subsequent layer.

LAYERS AND NETWORKS: INPUT/OUTPUT LAYERS

Input layer depends on:

- Form of raw data
- First level of our internal network architecture

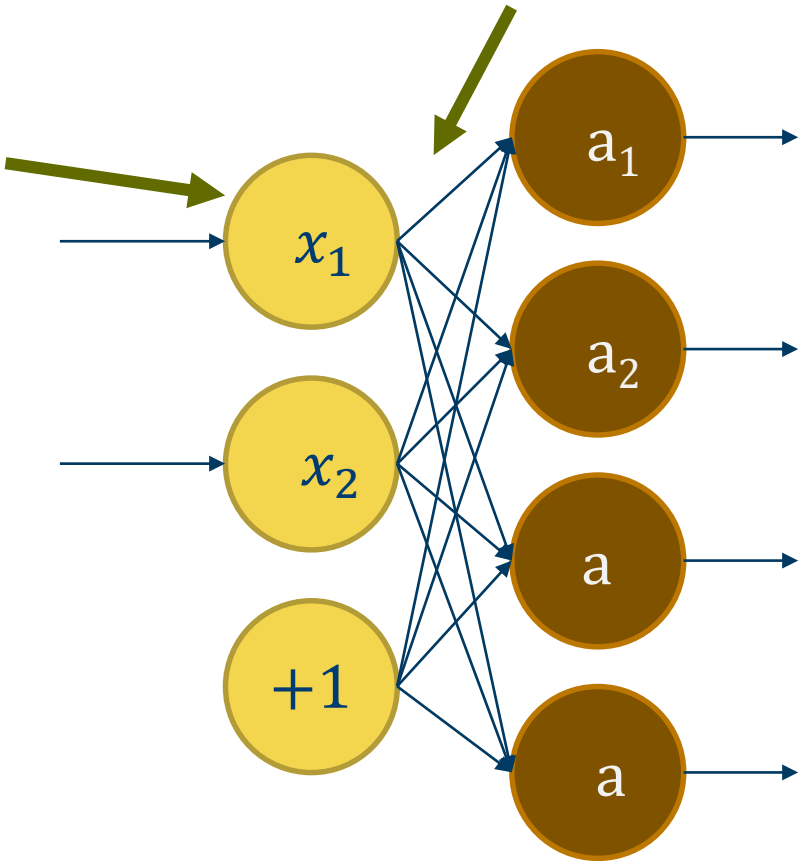
Output layer depends on:

- Last layer of our internal network architecture
- Type of prediction we want to make
    - Regression versus classification

LAYERS AND NETWORKS: INPUT/OUTPUT LAYERS

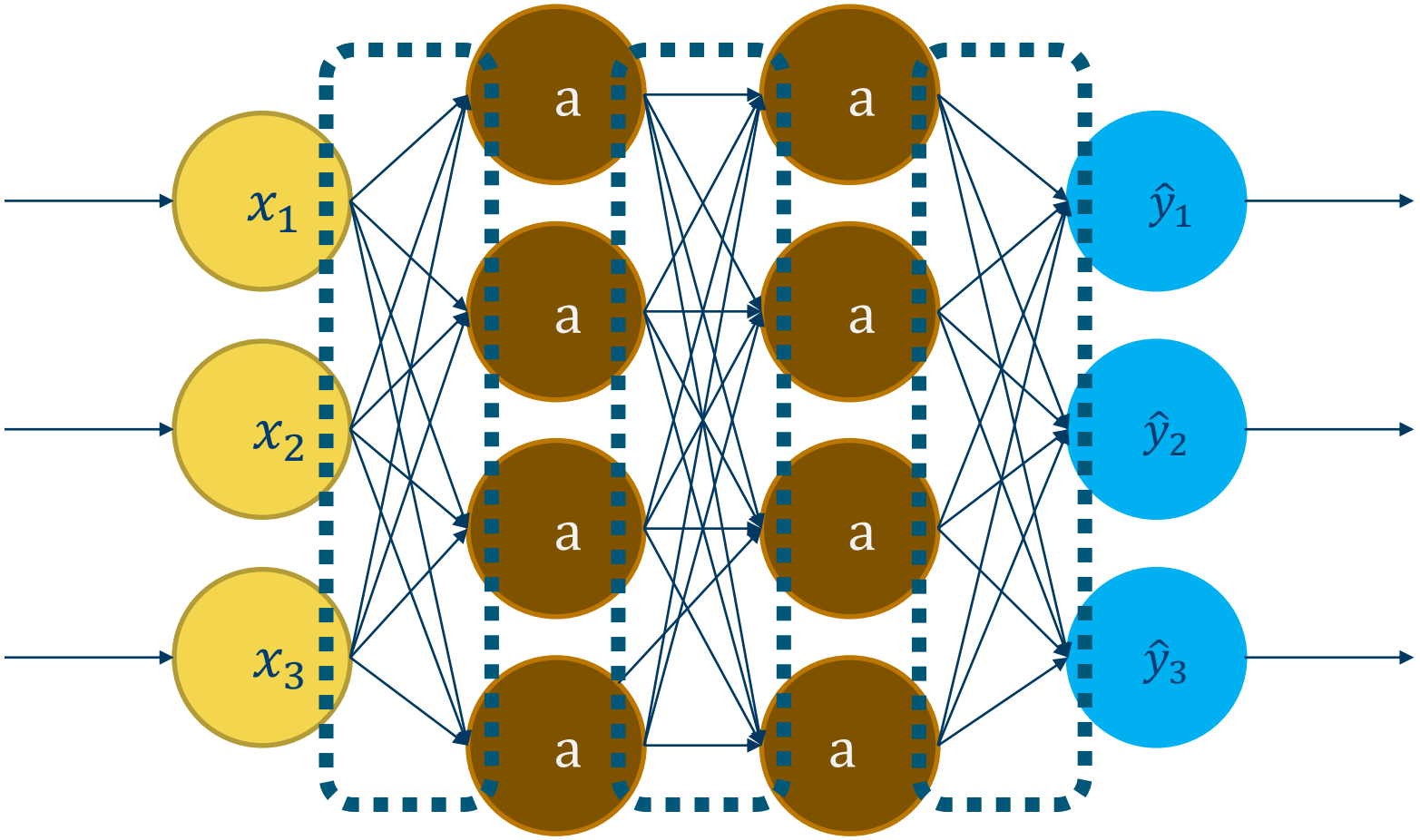
But having different weights mean a_1 and a_2 neurons respond differently.

a_1 and a_2 receive the same x_1 value

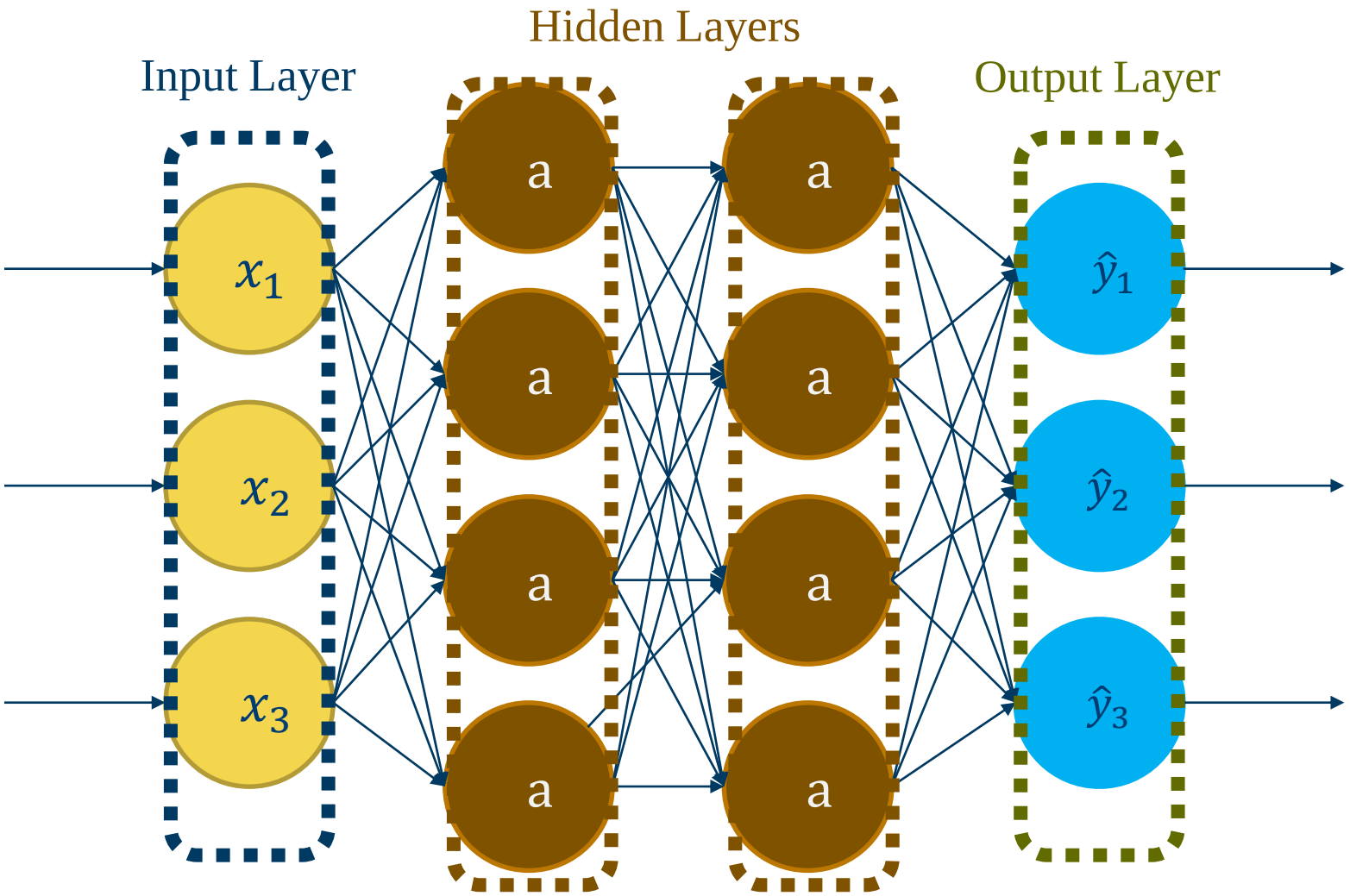


FEED FORWARD NEURAL NETWORK

Weights



FEED FORWARD NEURAL NETWORK



TRAINING

OPTIMIZATION AND LOSS: GRADIENT DESCENT

We will start with the cost function: $J(x) = x^2$

- Cost is what we pay for an error
- For example, an error of -3 gives a cost of 9

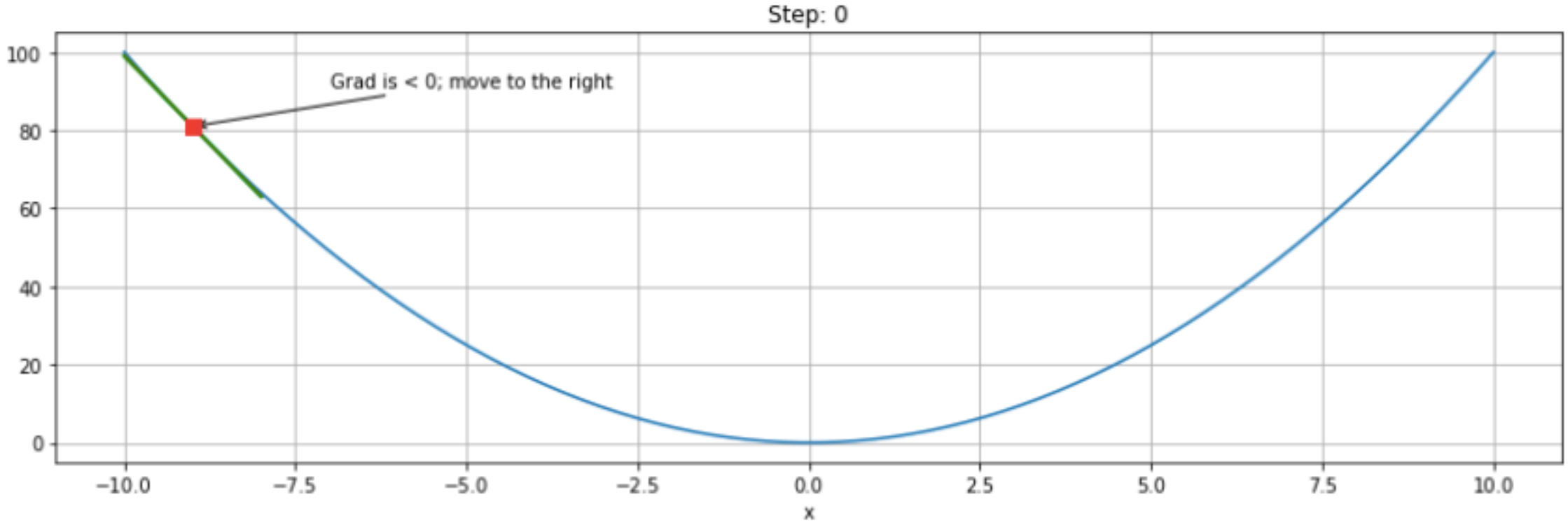
Take the gradient of $x^2 = 2x$.

Select datapoints to generate a gradient slope line.

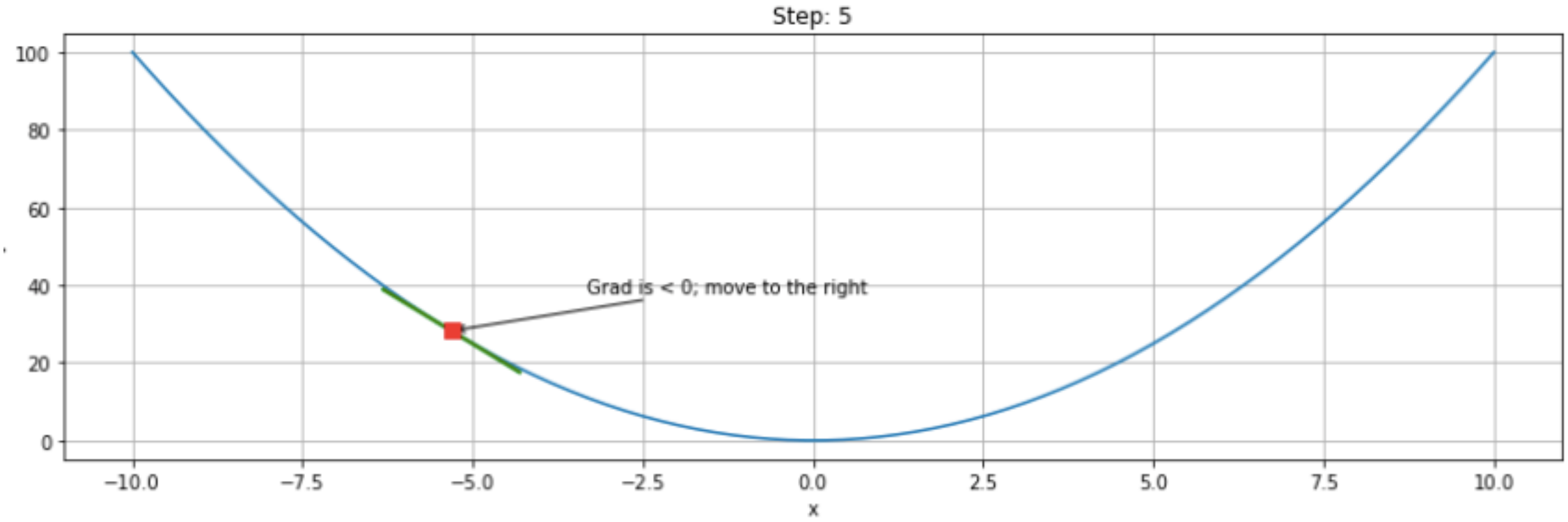
Plot x^2 with a given gradient slope and annotations.

We want the lowest cost.

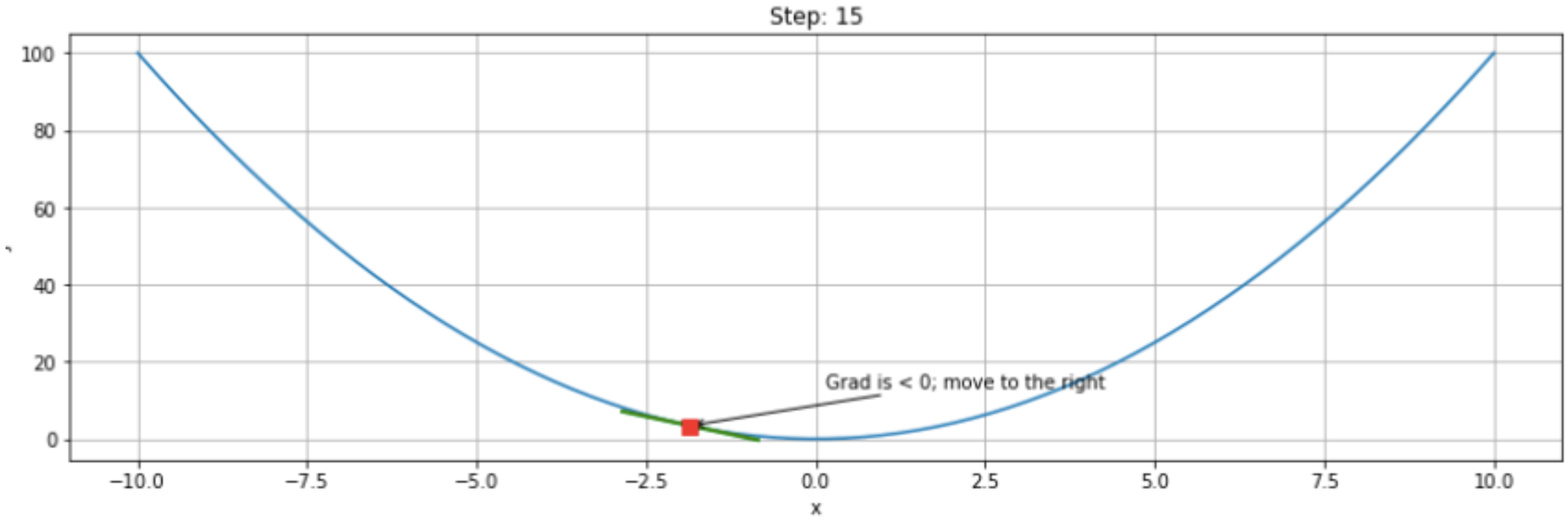
GRADIENT DESCENT: STARTING FROM LEFT SIDE



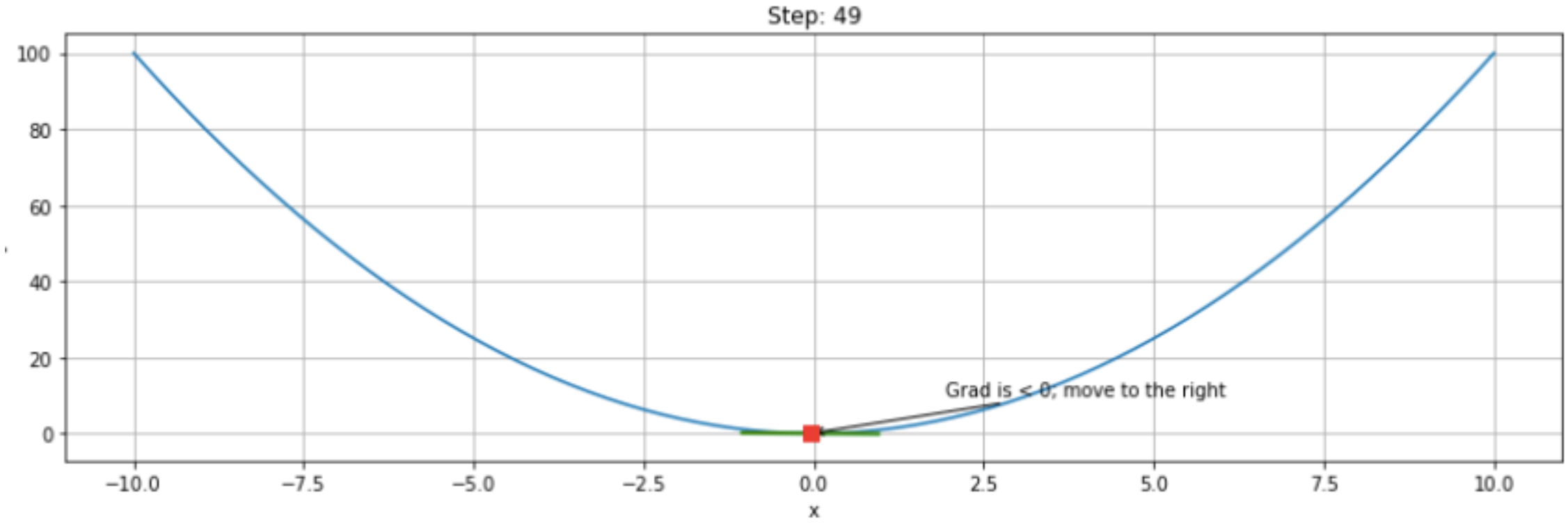
GRADIENT DESCENT: STARTING FROM LEFT SIDE



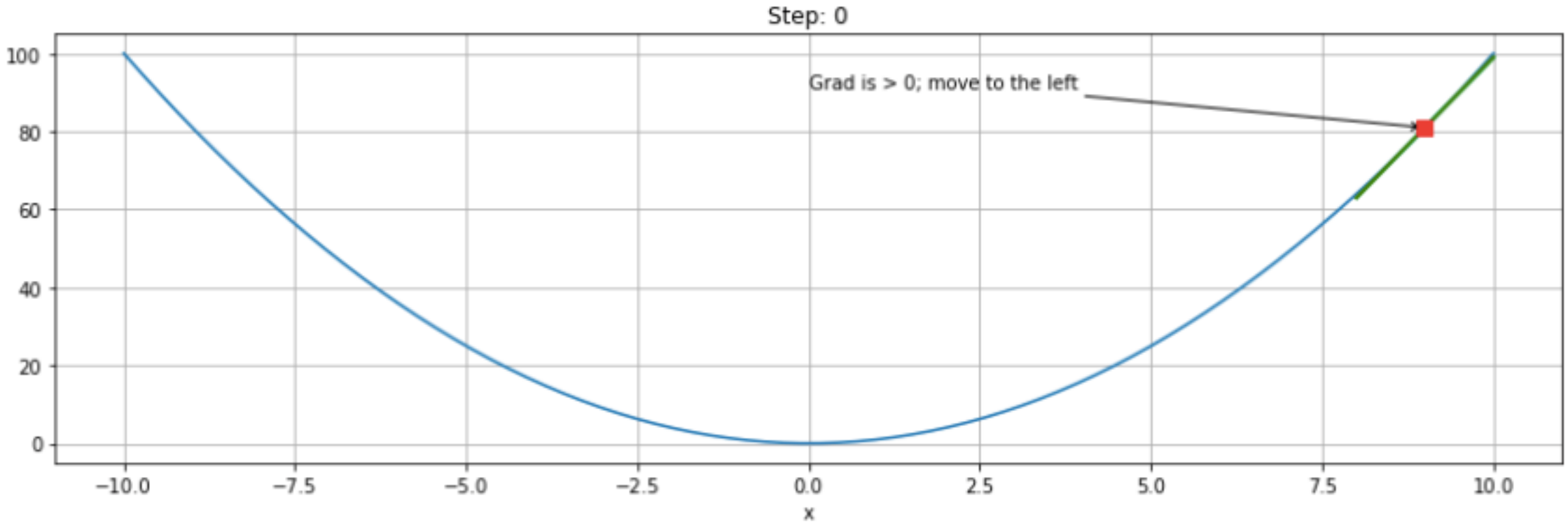
GRADIENT DESCENT: STARTING FROM LEFT SIDE



GRADIENT DESCENT: STARTING FROM LEFT SIDE



GRADIENT DESCENT: STARTING FROM RIGHT SIDE



PROCESS OF GRADIENT DESCENT: MATH

1. Find the gradient with respect to weights over training data.
   - Plug data into our derivative function and sum up over data points

The number we'll use to adjust the weight

$$\Delta W = \sum_{i=1}^{n} \frac{\partial J}{\partial W}(x_i, y_i)$$

$$\frac{\partial J}{\partial W}(x_i, y_i) = \frac{1}{n}\sum_{i=1}^{n} x_i(\hat{y}_i - y_i)$$

Derivative of MSE

PROCESS OF GRADIENT DESCENT: MATH

2. Adjust the weight by subtracting some amount of ΔW.

α (alpha) is known as the *learning rate*

A *hyper-parameter* we choose

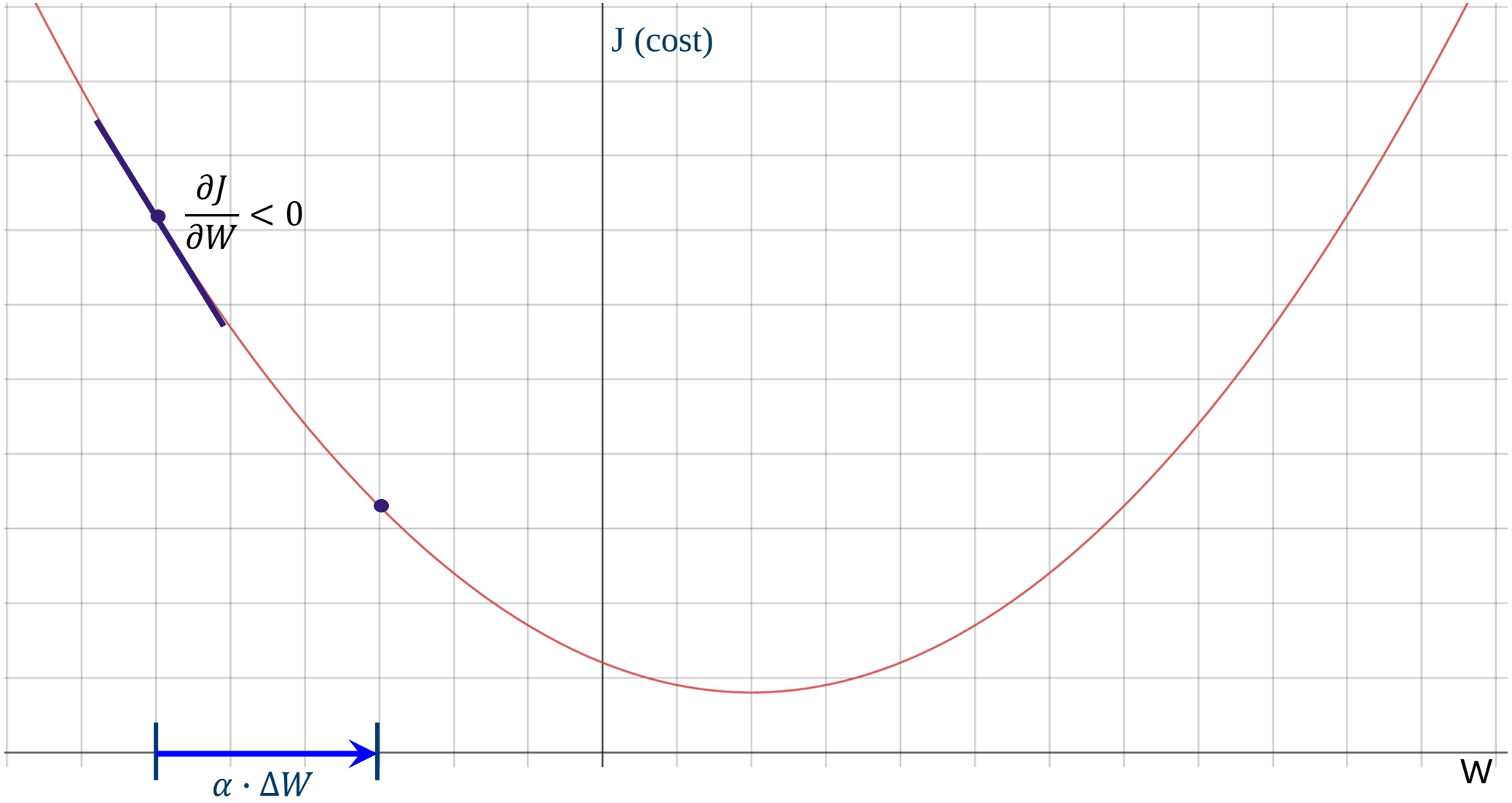
$$W := W - \alpha \cdot \Delta W$$

Minus adjusts W in the correct direction

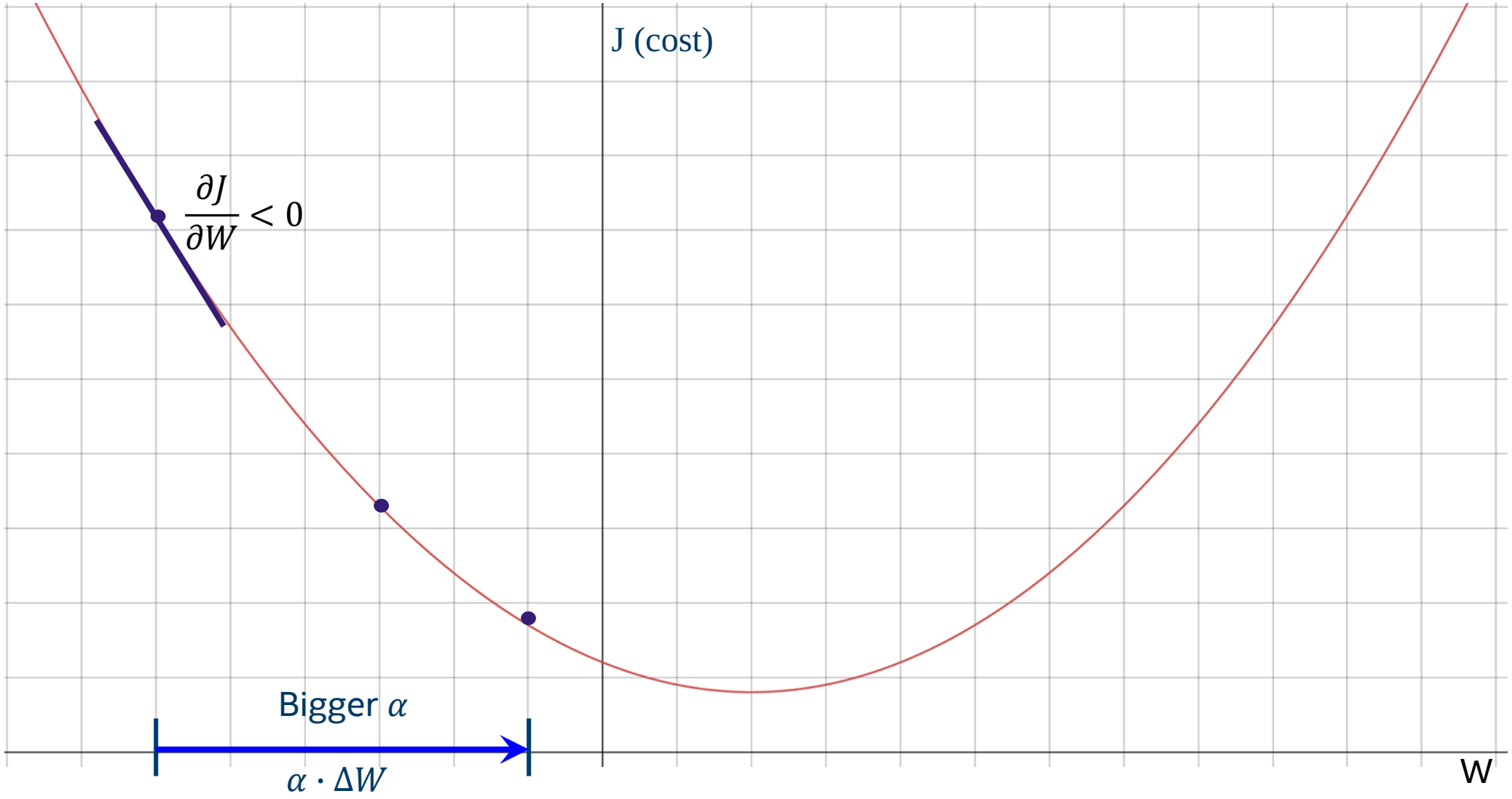
3. Repeat until model is *done training.*

We can also adjust the learning rate as we train

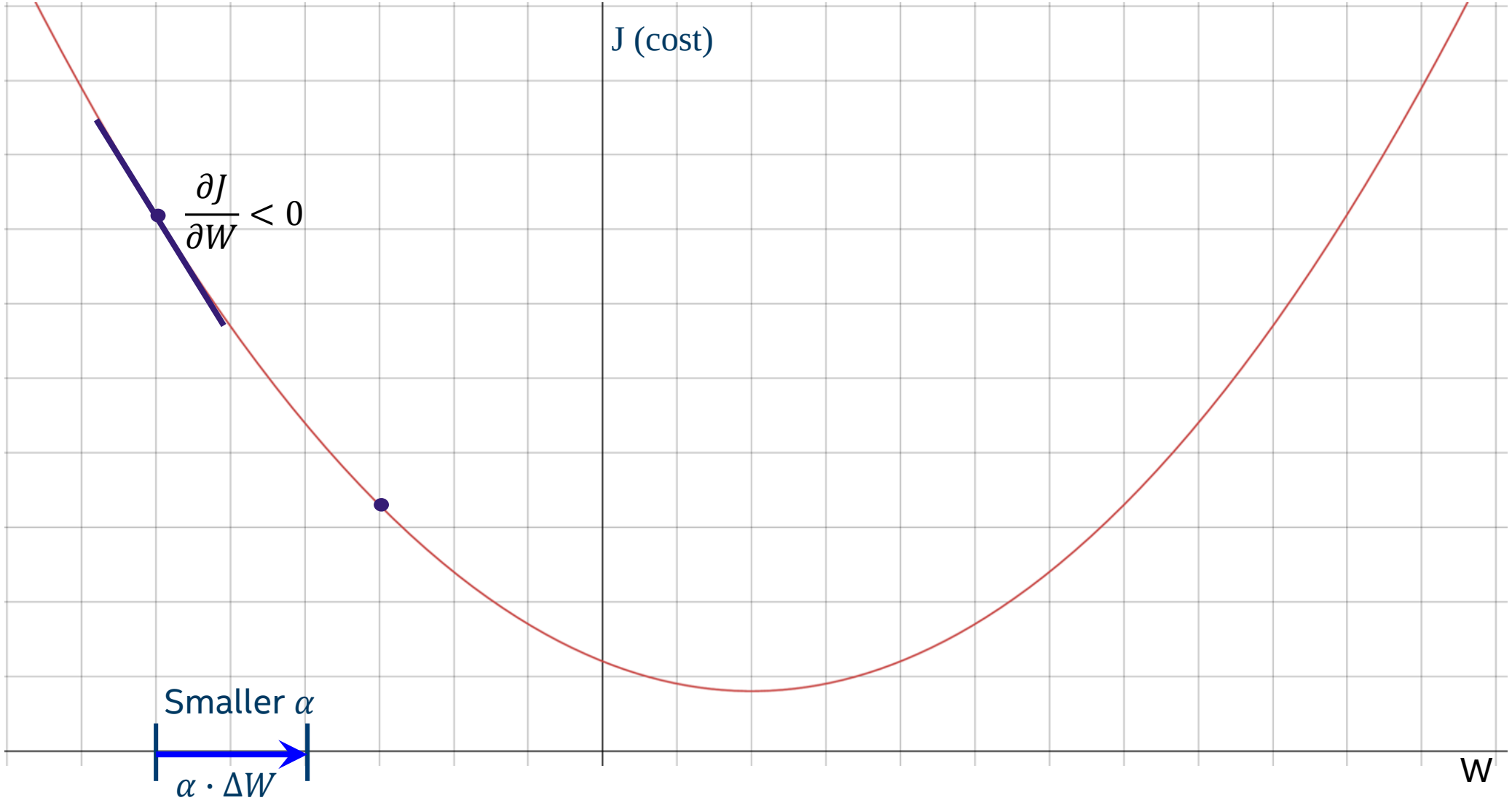
ADJUSTING THE LEARNING RATE



ADJUSTING THE LEARNING RATE



ADJUSTING THE LEARNING RATE



BATCHES

How much data do we use for one training step?

- One training step takes us from *old* network weights to *new* network weights

We could use ALL of the examples at one time.

- Terrible performance -- if it is even possible
- We'll constantly be swapping memory to slow disks

We could use one example at a time.

- But terrible performance
- It doesn't take advantage of caching, vectorized operations, and so on
- We want good data processing size for vectorized operations

BATCHING

How much data do we use for one training step?

- One training step takes us from *old* network weights to *new* network weights

Options

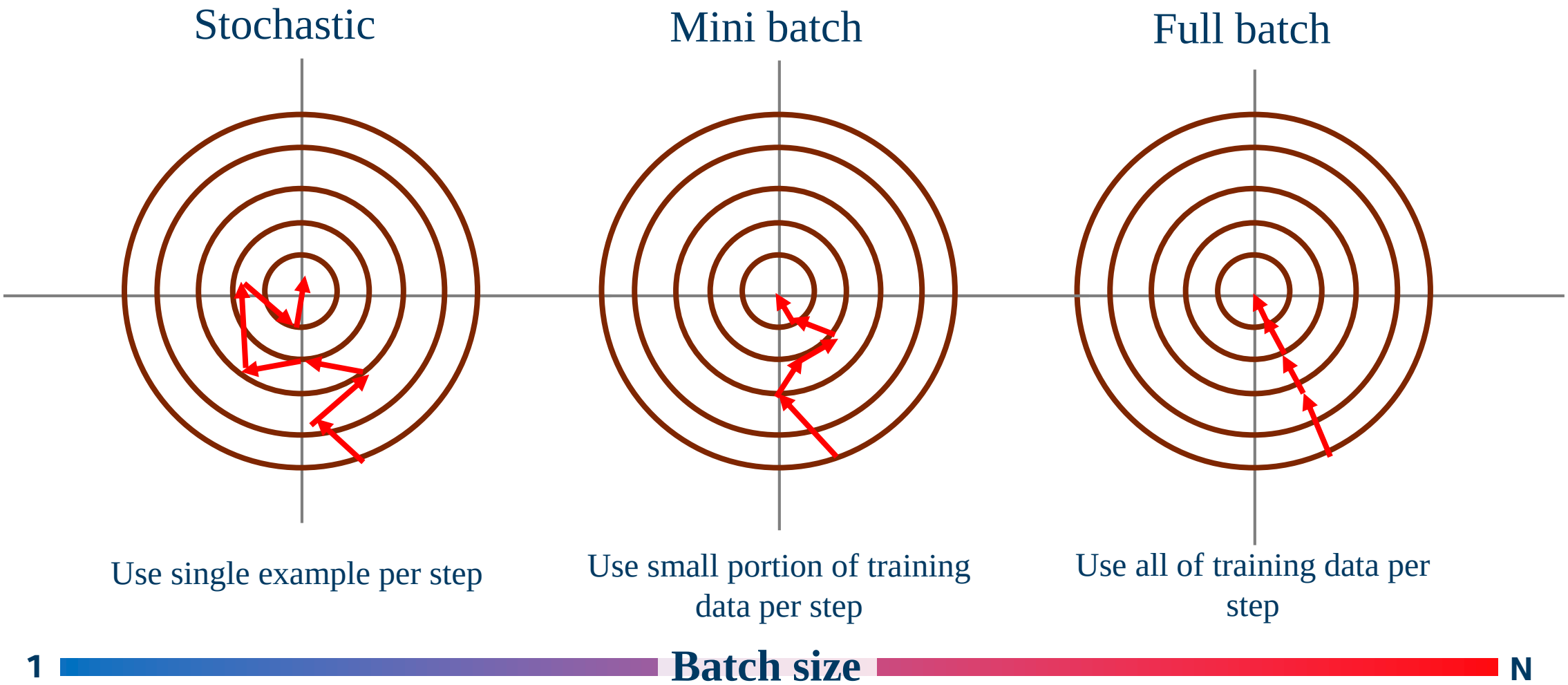
- Full batch
  - Update weights after considering all data in batch
- Mini-batch
  - Update weights after considering part of batch, repeat
  - Approximating the gradient
  - Can help with local minima

BATCHING

Options continued…

- Stochastic gradient descent (SGD)
  - Mini batch with size 1
  - Also called online training
  - Very sporadic, very easy to compute
  - With a big network, performance comes from many weights

COMPARING FULL BATCH, MINI BATCH, AND SGD



EPOCH

One epoch is one pass through the entire dataset.

- Generally, the dataset is too big for system memory.
  - Can't do this all in one go

General measure of the amount of training.

- How many epochs did I perform?

SHUFFLING DATASETS FOR EPOCHS

After each epoch, shuffle the training data.

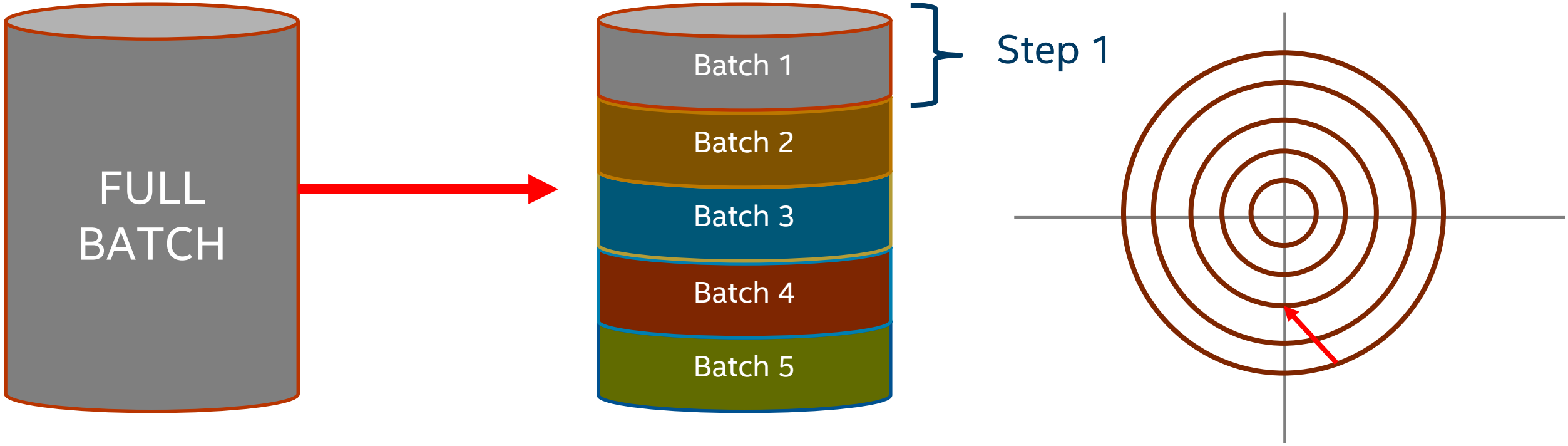
Prevents resampling in the *exact* same way.

- Different epochs sample data in different ways.

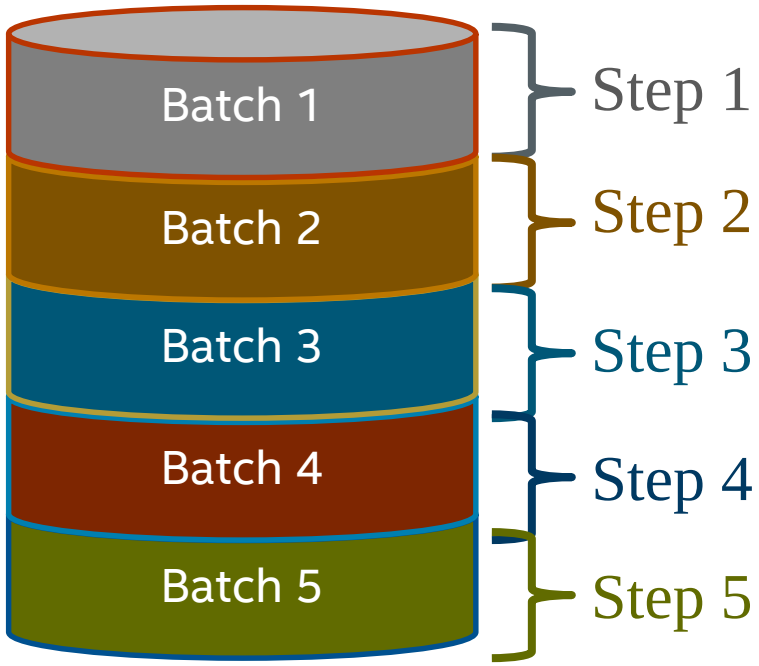
So...

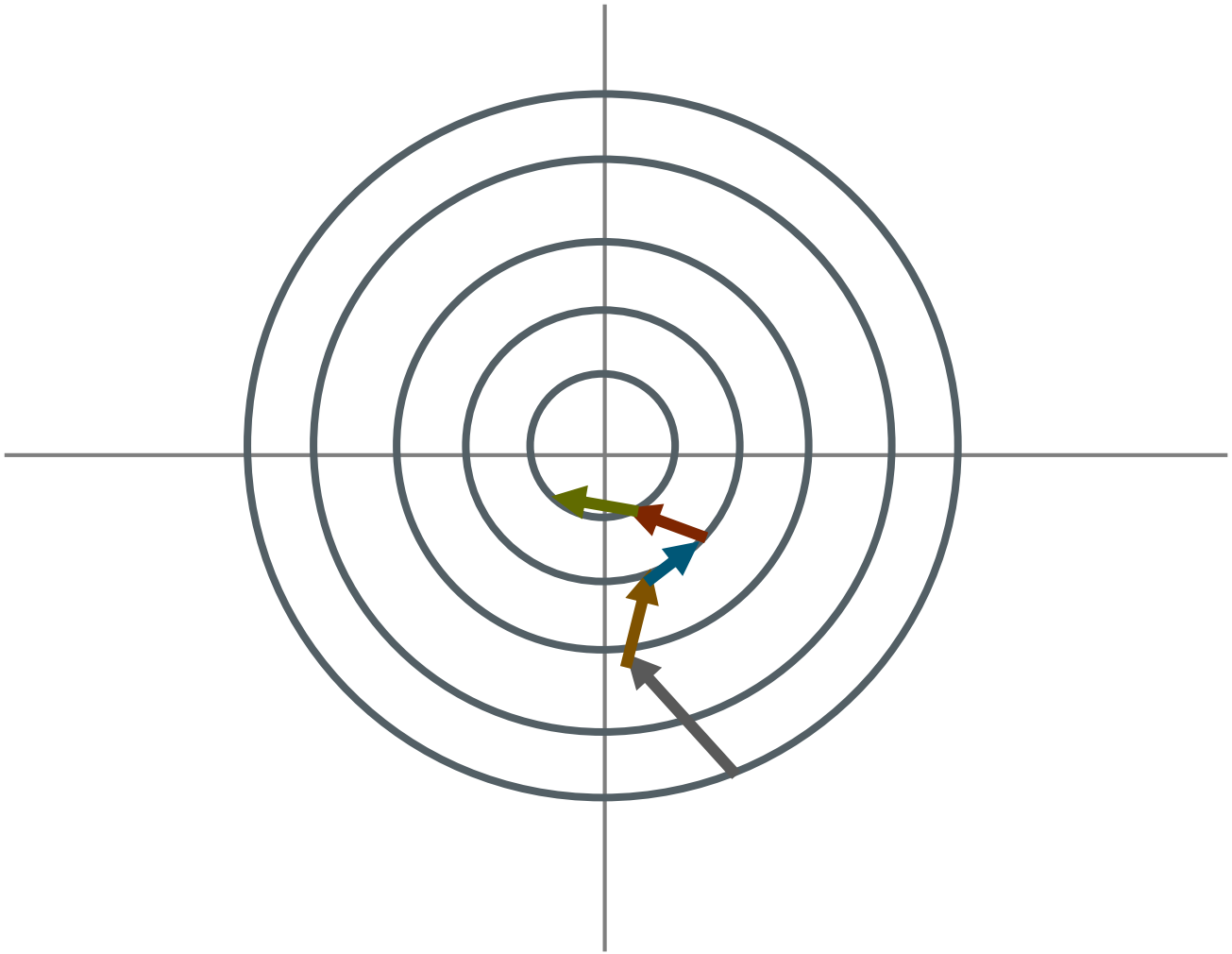
Shuffle, make batches, repeat.

SPLITTING DATA UP INTO BATCHES



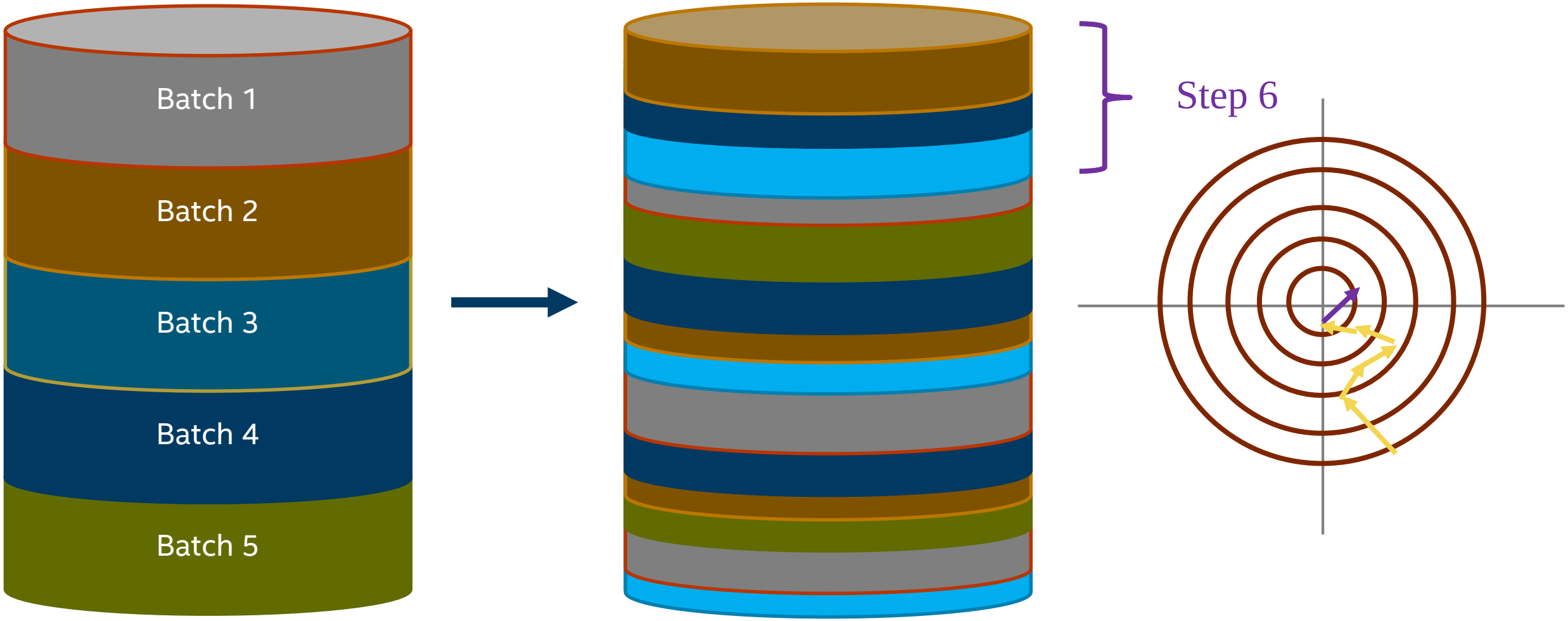
SPLITTING DATA UP INTO BATCHES





First Epoch Completed

SHUFFLE DATA



OVERFITTING

SPECIAL ISSUES WITH OVERFITTING

Very simple neural network architectures can approximate arbitrarily complex functions very well.

- Consequence of universal representation theorem
  - Three layers, finite # nodes → arbitrarily good approximation
  - Although better approximations may require n → *big*

Even simple neural networks are, in some sense, too powerful.

SPECIAL ISSUES WITH OVERFITTING

Many architectures easily overfit data.

- Simply chugging through the data over-and-over leads to overfit.
- Memorizes data but doesn't learn the generality.
- Easily mislead by noise.

Traditionally, we control this by monitoring the performance on a test set.

- As long as it improves, we're good.
- When it starts going the wrong way, we stop.

SPECIAL ISSUES WITH OVERFITTING

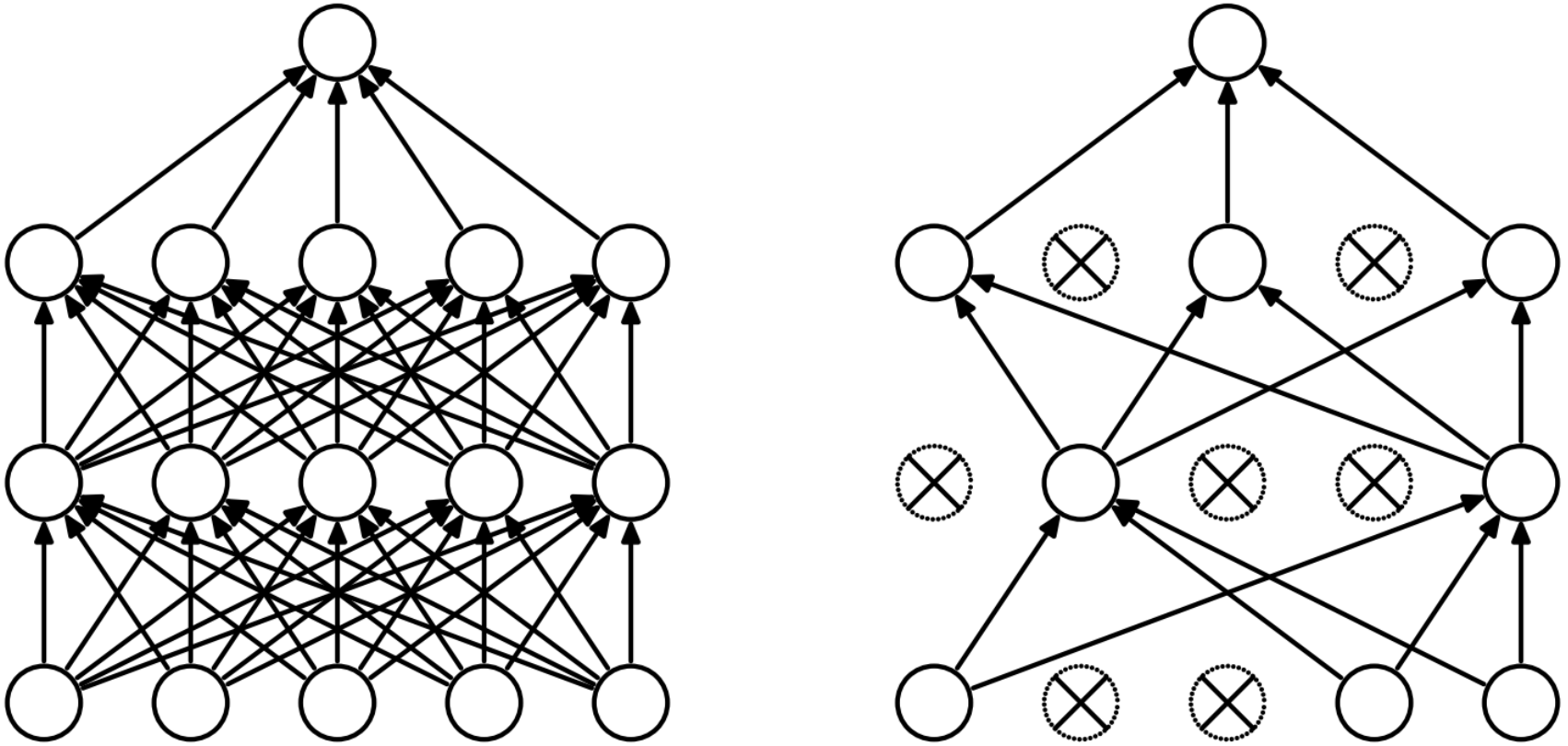
Modern method uses a technique called *dropout*:

- Here we randomly have nodes disappear from the network.
- Everyone else still has to perform.

The overall network has to be more robust.

- Single nodes can't be too important.
- The nodes can't all be highly correlated with one another.
    - *Different nodes must respond to different stimuli*

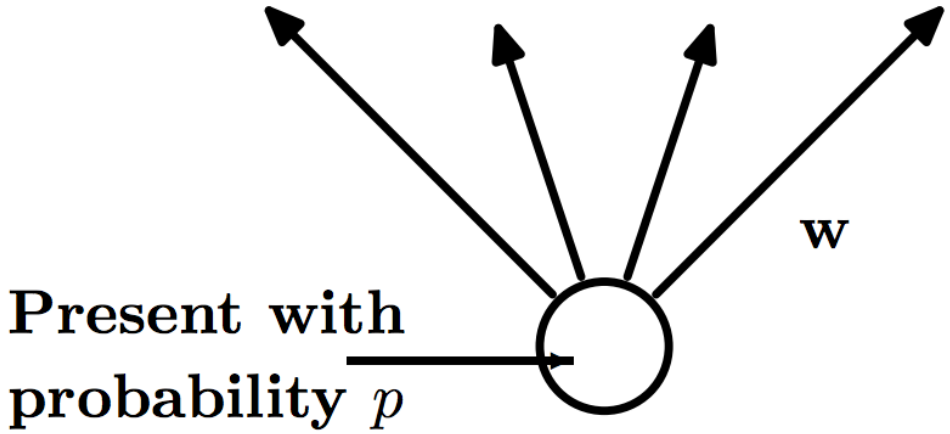
DROPOUT MODEL



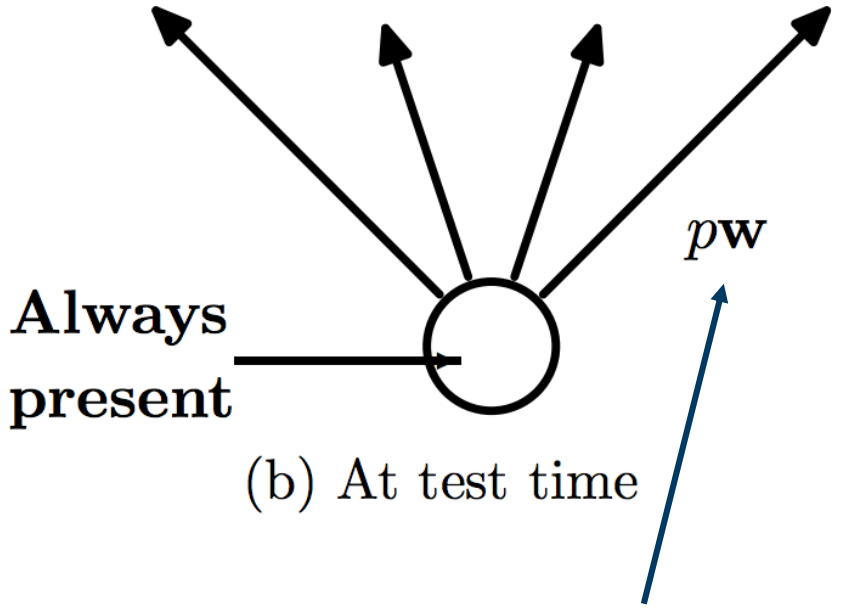
(a) Standard Neural Net

(b) After applying dropout.

KNOCKING OUT AND RESCALING NEURONS



(a) At training time

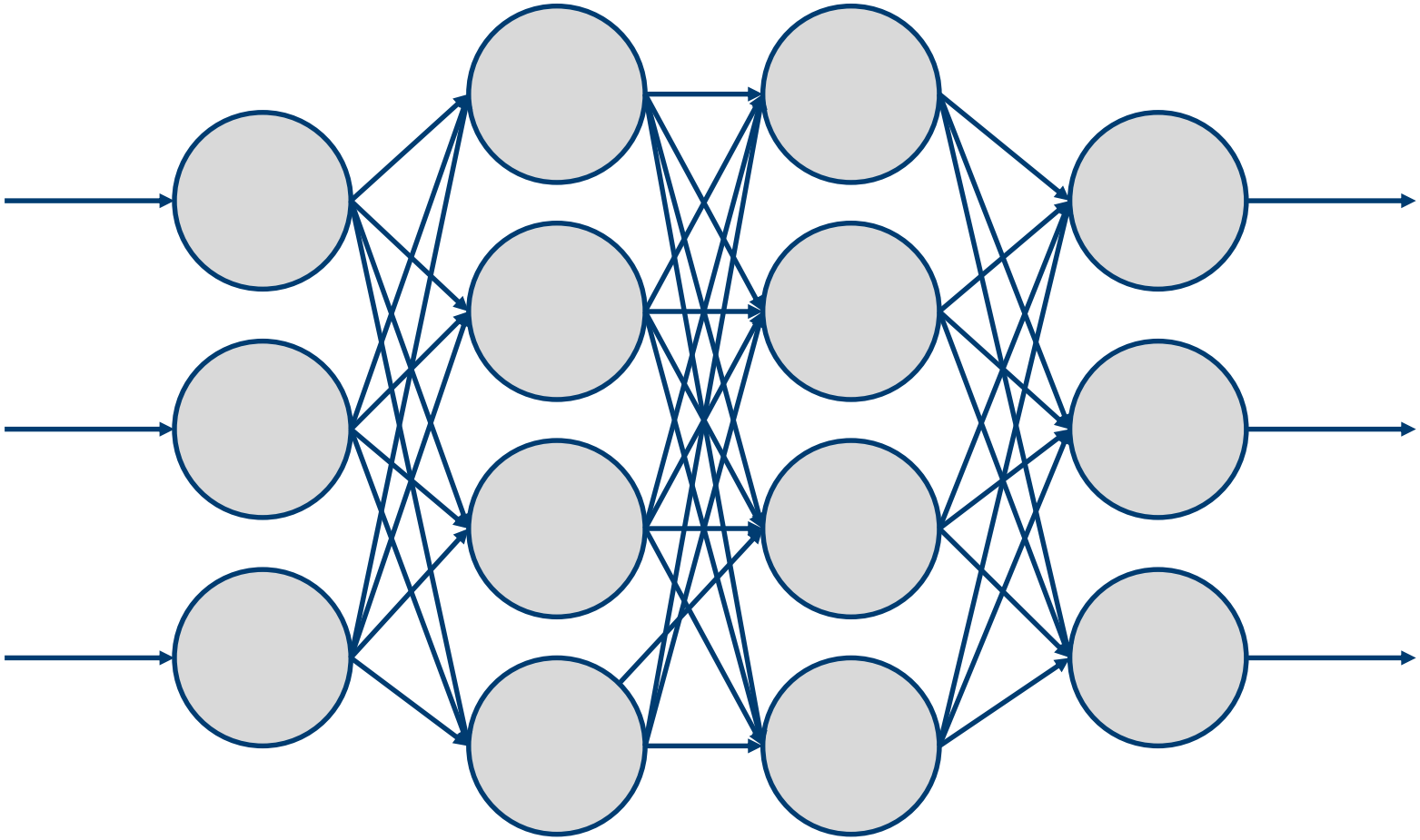


(b) At test time

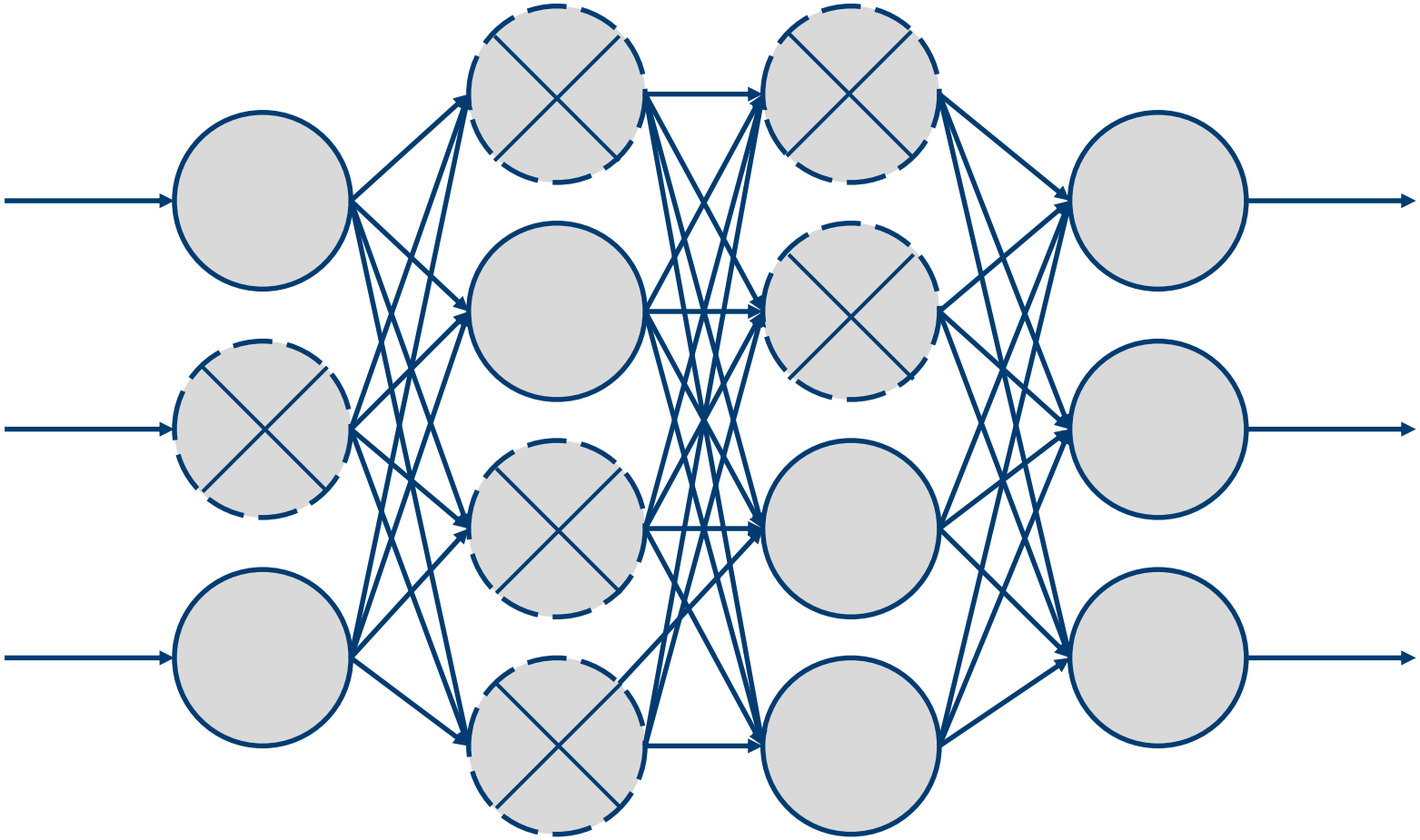
- During training, we randomly drop each neuron with probability $1 - p$.

- When running the model, we scale the outputs of the neuron by p.
- This ensures that the *expected* value of the weights stays the same at run time.

CONCEPT OF A "PSEUDO-ENSEMBLE"

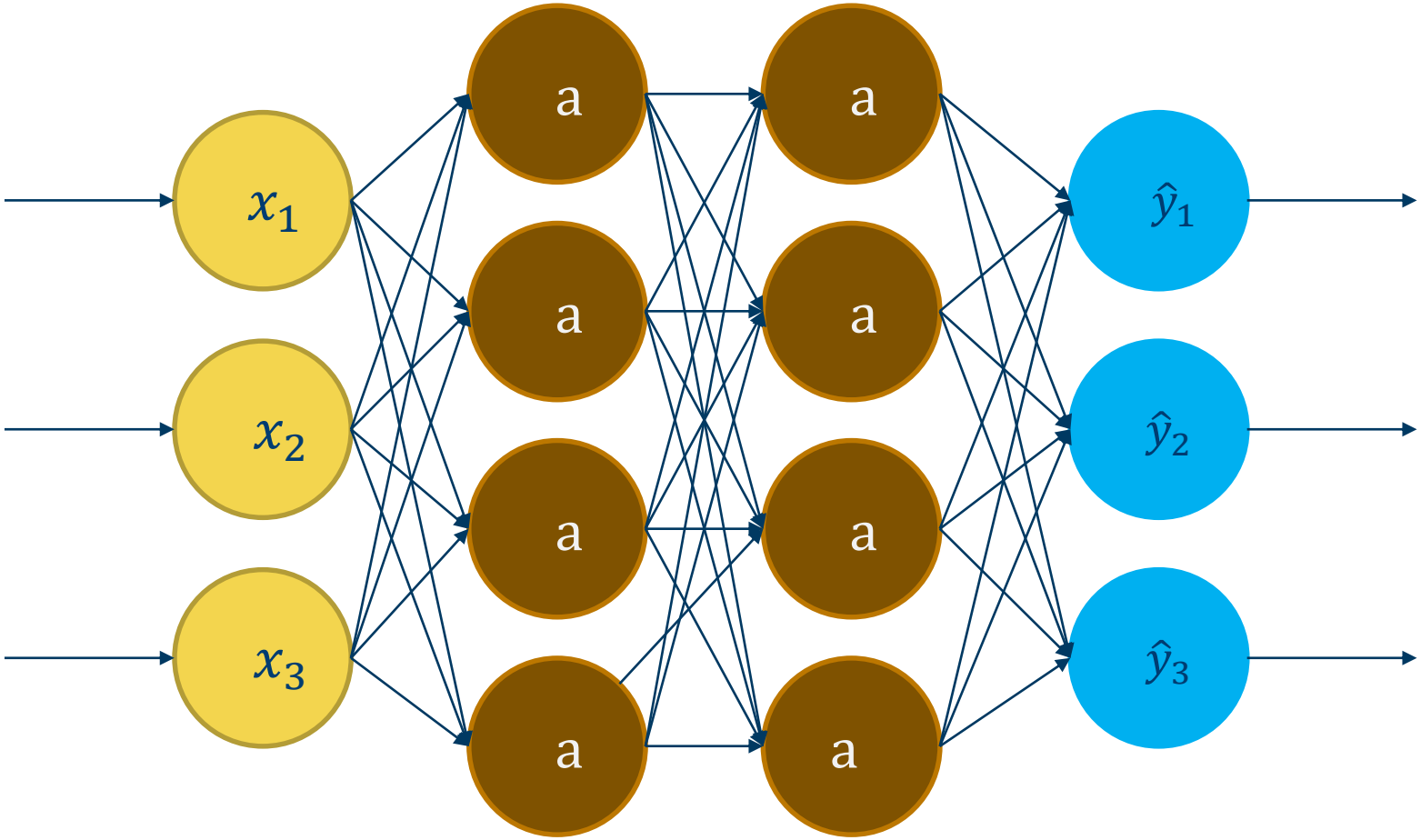


AN EXAMPLE MODEL



ARCHITECTURES

MULTILAYER PERCEPTRON (MLP)



MLP: GENERAL PROCESS

1. Shuffle the data and split between train and test sets
2. Flatten the data
3. Convert class vectors to binary class matrices
4. Generate network architecture
5. Display network architecture
6. Define learning procedure
7. Fit model
8. Evaluate

MLP

Trains a simple MLP with dropout on the MNIST* dataset.

Gets to 98.40 percent test accuracy after 20 epochs.

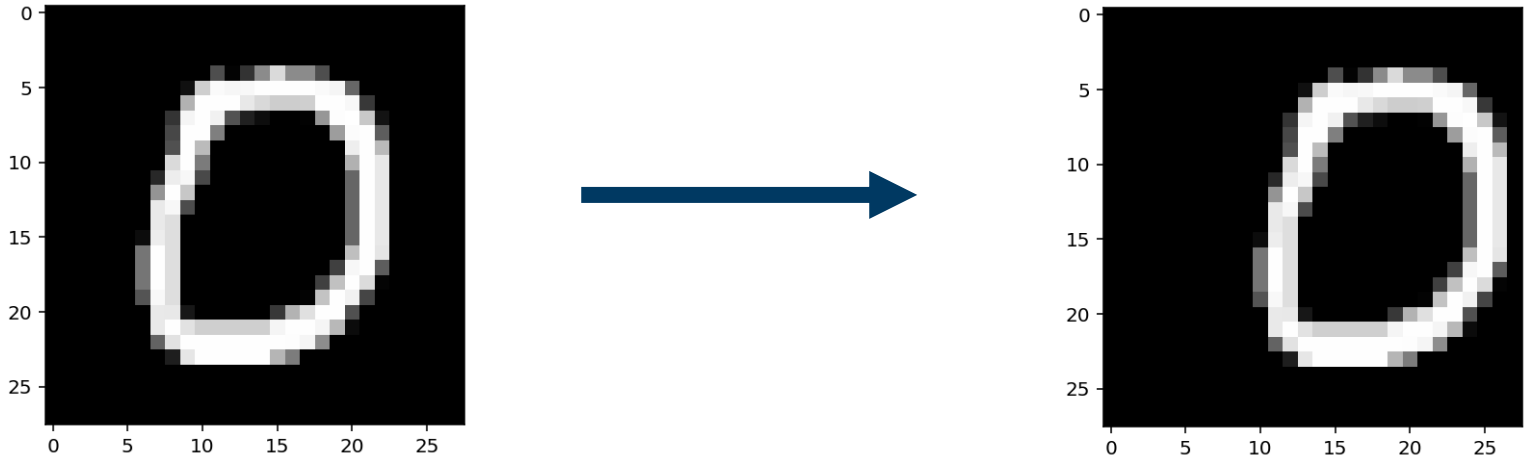
- There is *a lot* of margin for parameter tuning
- 0.2 seconds per epoch on a K520 GPU

CONVOLUTION NEURAL NETWORKS (CNN)

Good to use when you have:

- Translational variance
- Huge number of parameters

We need to train models on translated data



CNN: GENERAL PROCESS

Trains a simple convnet on the MNIST* dataset>

Gets to 99.25 percent test accuracy after 12 epochs.

- There is still a lot of margin for parameter tuning
- 0.16 seconds per epoch on a GRID K520 GPU

CNN

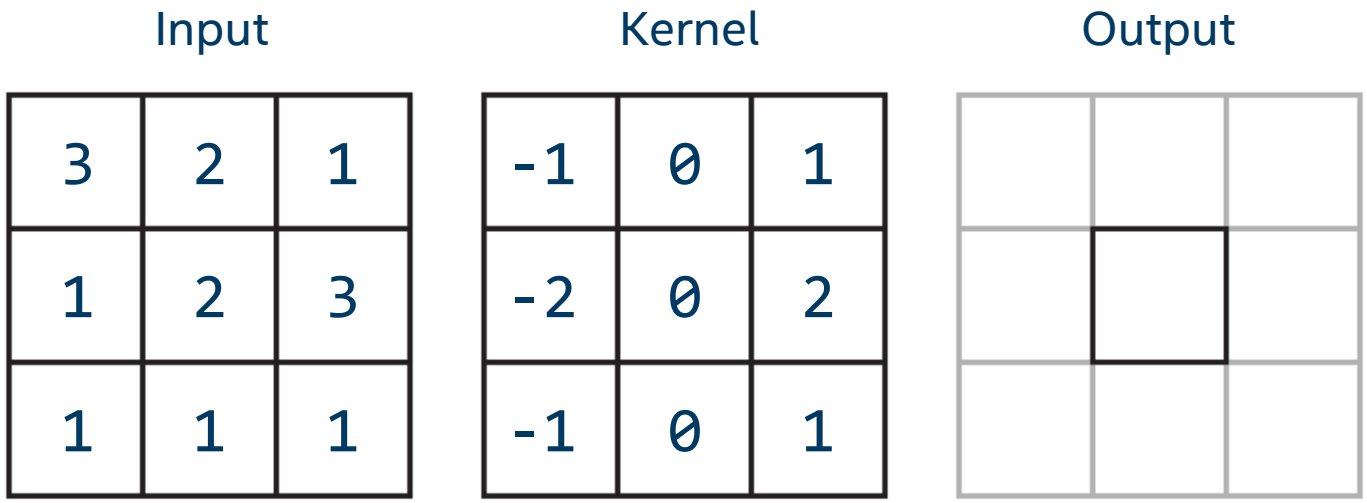
1. Shuffle dataset and split between train and test sets
2. Maintain grid structure of data
    - Add a dimension to account for the single-channel images
3. Convert class vectors to binary class matrices
4. Define architecture
5. Define learning procedure
6. Fit model
7. Evaluate

CNN: KERNELS

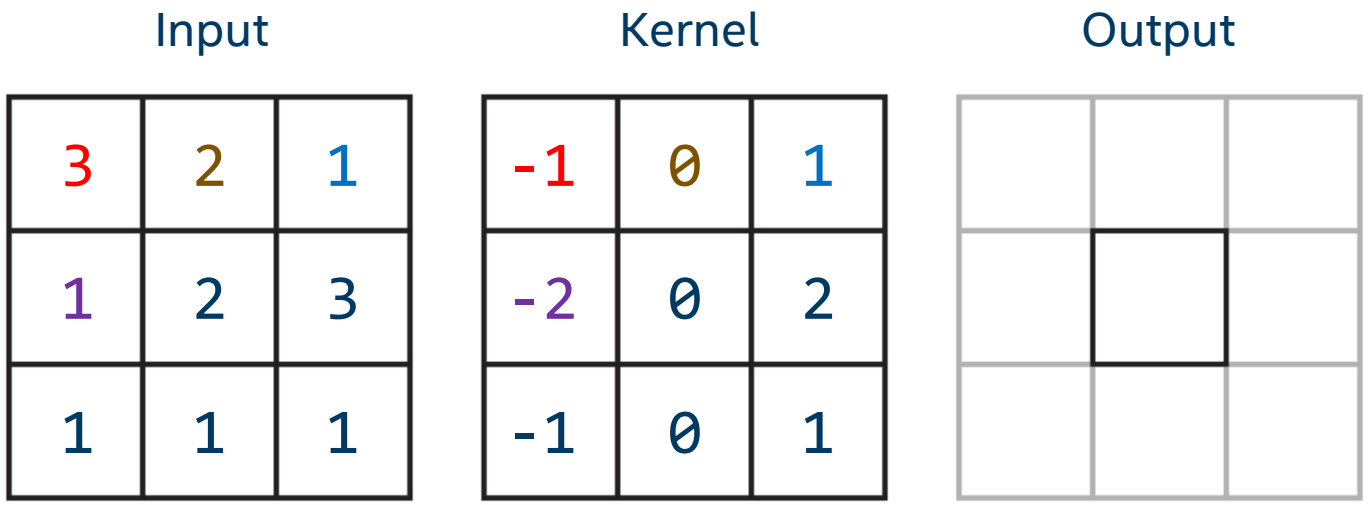
Like our image processing kernels, but we learn their weightings

- Instead of assuming Gaussian, we let the data determine the weights.

Example: 3 x 3

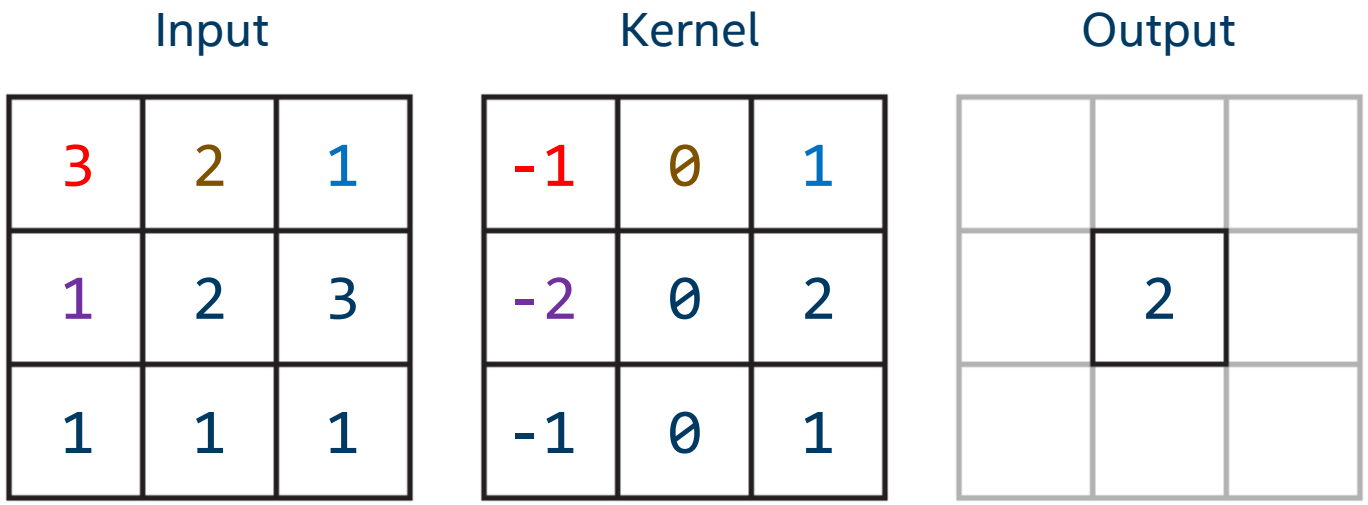


KERNEL MATH



= (3 * -1) + (2 * 0) + (1 * 1) + (1 * -2) … and so on.

KERNEL MATH

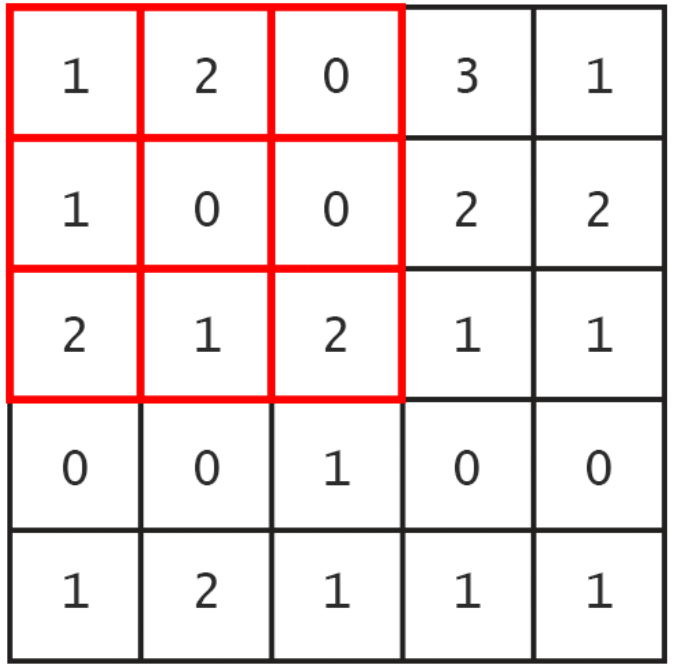


$= (3 * -1) + (2 * 0) + (1 * 1) + (1 * -2)\ (1 \cdot -2) + (2 \cdot 0) + (3 \cdot 2) + (1 \cdot -1) + (1 \cdot 0) + (1 \cdot 1)$

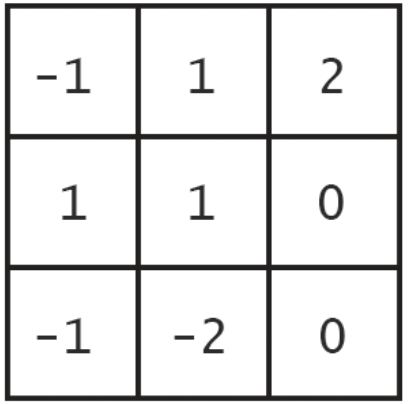
$= -3 + 1 - 2 + 6 - 1 + 1$

$= 2$

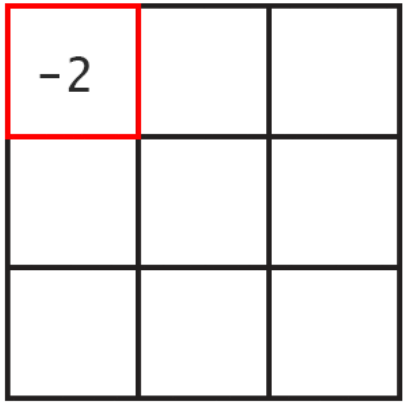
SAME PROCESS, LARGER DATASET



input



kernel



output

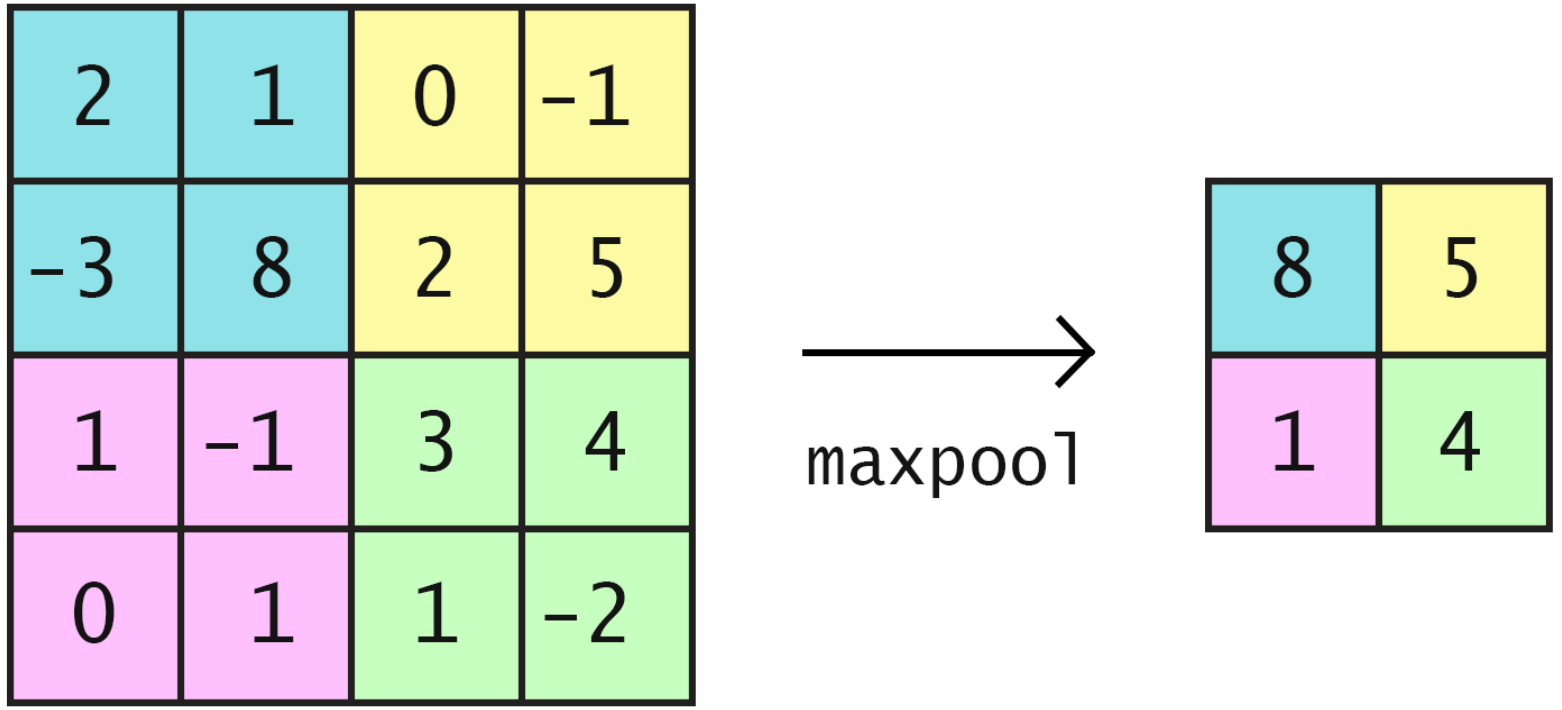
CNN: POOLING LAYERS

Reduce neighboring pixels.

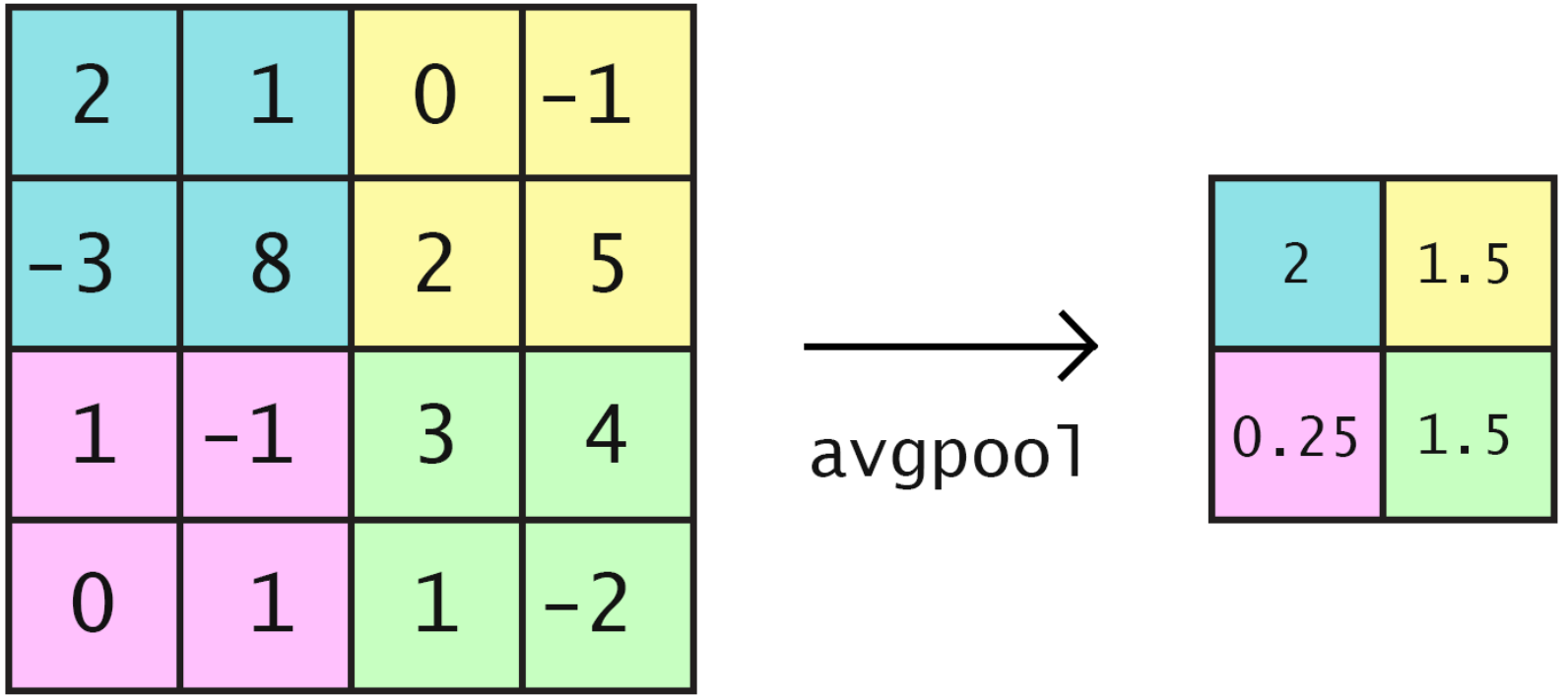
Reduce dimensions of inputs (height and width).

No parameters!

CNN: POOLING LAYERS

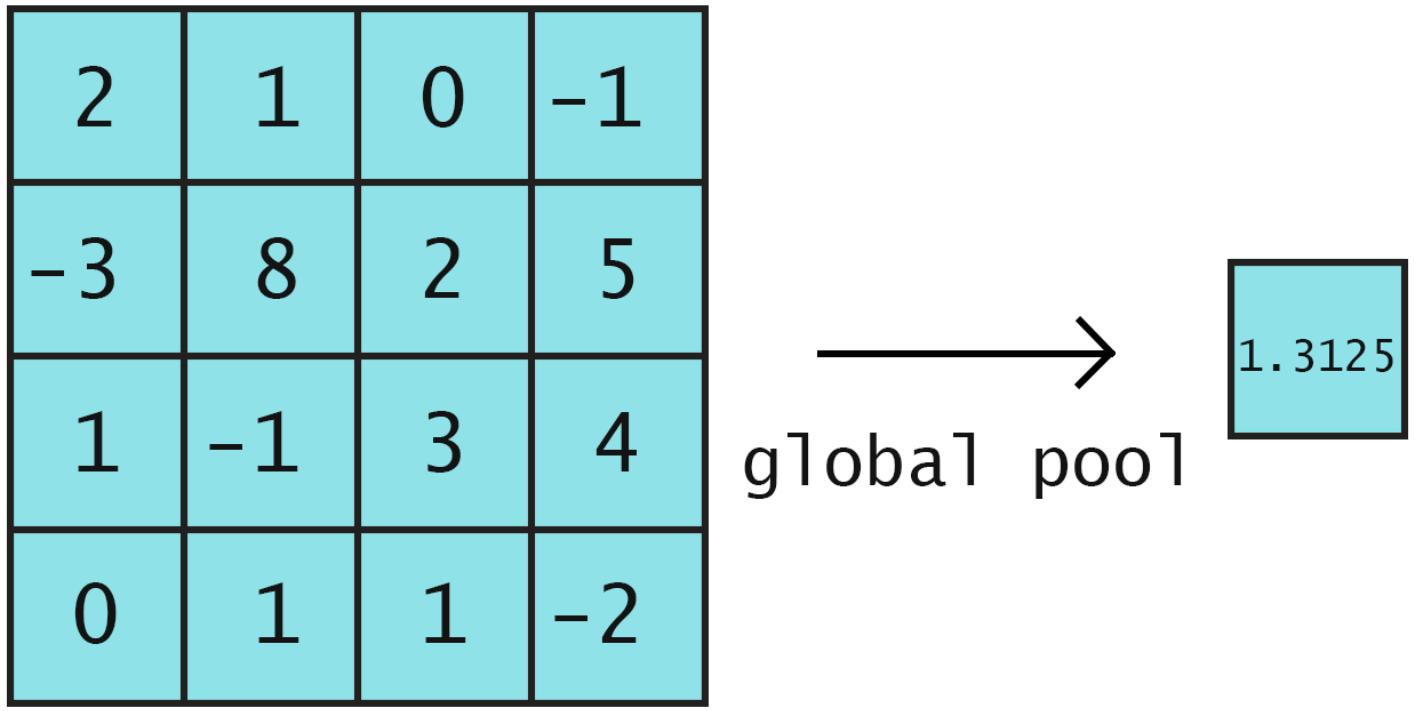


CNN: POOLING LAYERS



CNN: POOLING LAYERS

(Average pool over whole layer)

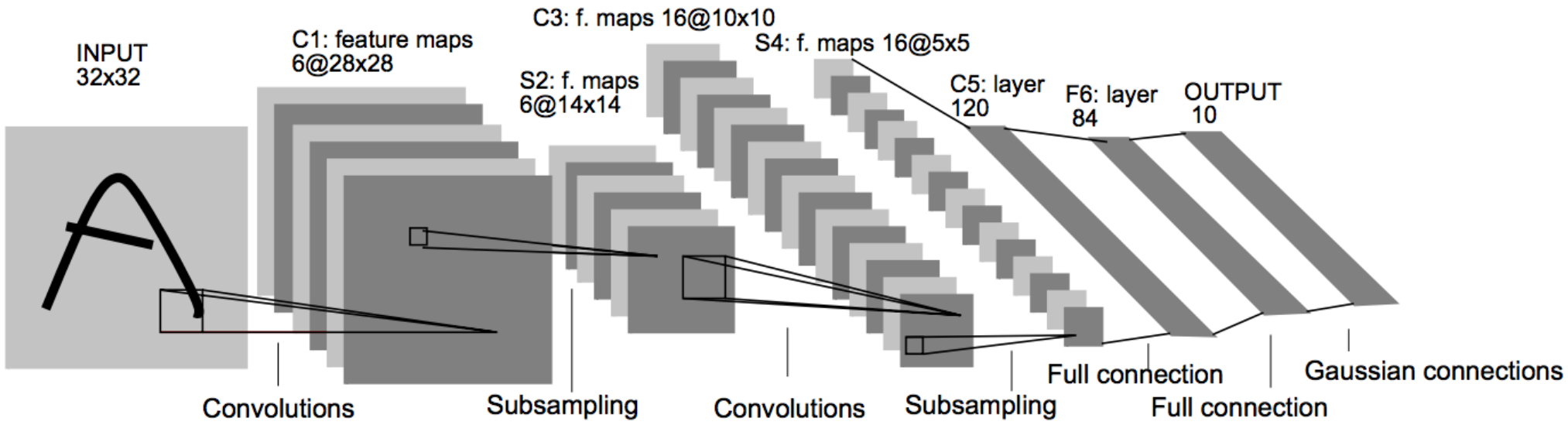


LENET*: EXAMPLE CNN ARCHITECTURE

Use convolutions to learn features on image data.

- Used on the MNIST* dataset

Input: 28 x 28, with two pixels of padding (on all sides)



Convolution size: 5 x 5

LENET*

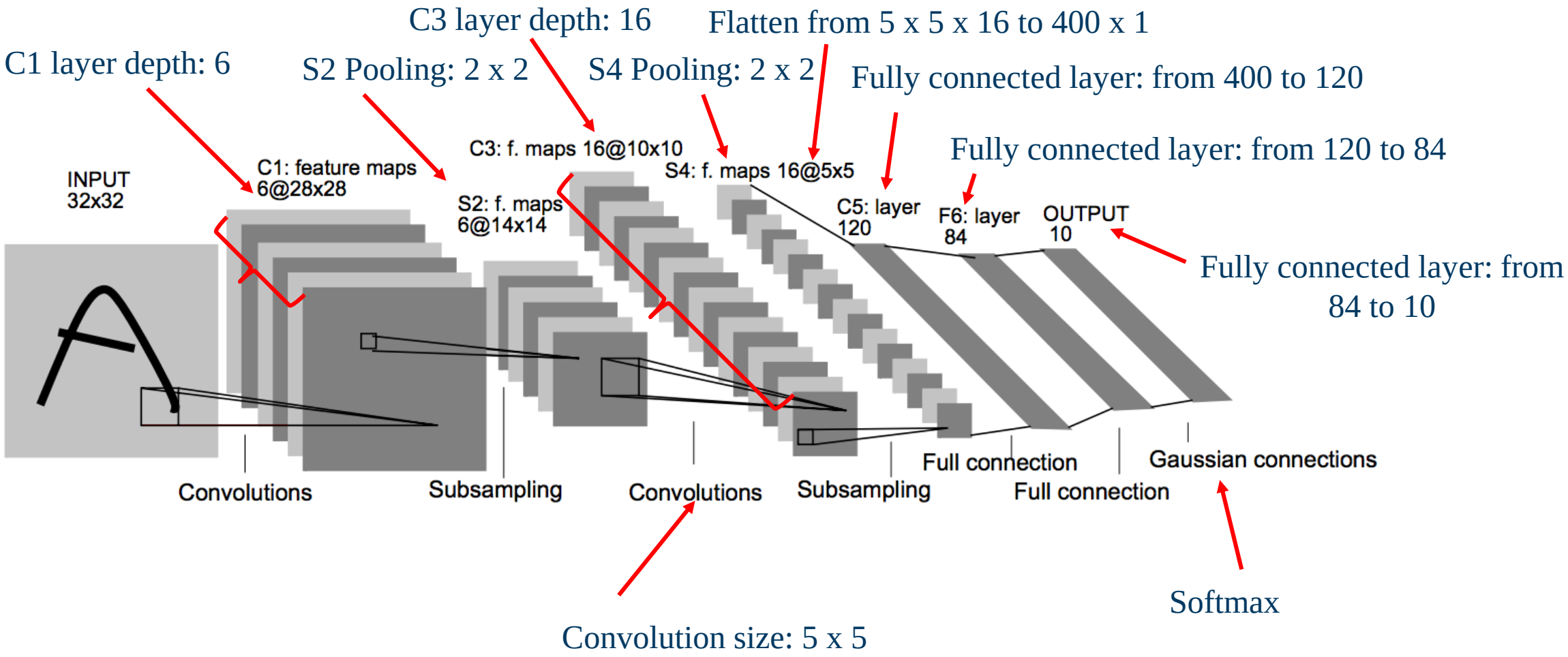

C3 layer depth: 16
Flatten from 5 x 5 x 16 to 400 x 1
C1 layer depth: 6
S2 Pooling: 2 x 2
S4 Pooling: 2 x 2
Fully connected layer: from 400 to 120
Fully connected layer: from 120 to 84
INPUT
32x32
C1: feature maps
6@28x28
S2: f. maps
6@14x14
C3: f. maps 16@10x10
S4: f. maps 16@5x5
C5: layer
120
F6: layer
84
OUTPUT
10
Fully connected layer: from
84 to 10
Convolutions
Subsampling
Convolutions
Subsampling
Full connection
Full connection
Gaussian connections
Softmax
Convolution size: 5 x 5

TABLE DESCRIPTION OF LENET*-5

Layer Name	Parameters
1. Convolution	5 x 5, stride 1, padding 2 (‘SAME’)
2. Max pool	2 x 2, stride 2
3. Convolution	5 x 5, stride 1, padding 2 (‘SAME’)
4. Max pool	2 x 2, stride 2
5. Fully connected (ReLU)	Depth: 120
6. Fully connected (ReLU)	Depth: 84
7. Output (fully connected ReLU)	Depth: 10

WHAT'S THE POINT? COUNT PARAMETERS

```
Conv1:   1*6*5*5 + 6          =    156
Pool2:                             0
Conv3:   6*16*5*5 + 16        =    2416
Pool4:                             0
FC1:     400*120 + 120        =    48120
FC2:     120*84  + 84         =    10164
FC3:     84*10 + 10           =    850
Total:                        =    61706
```

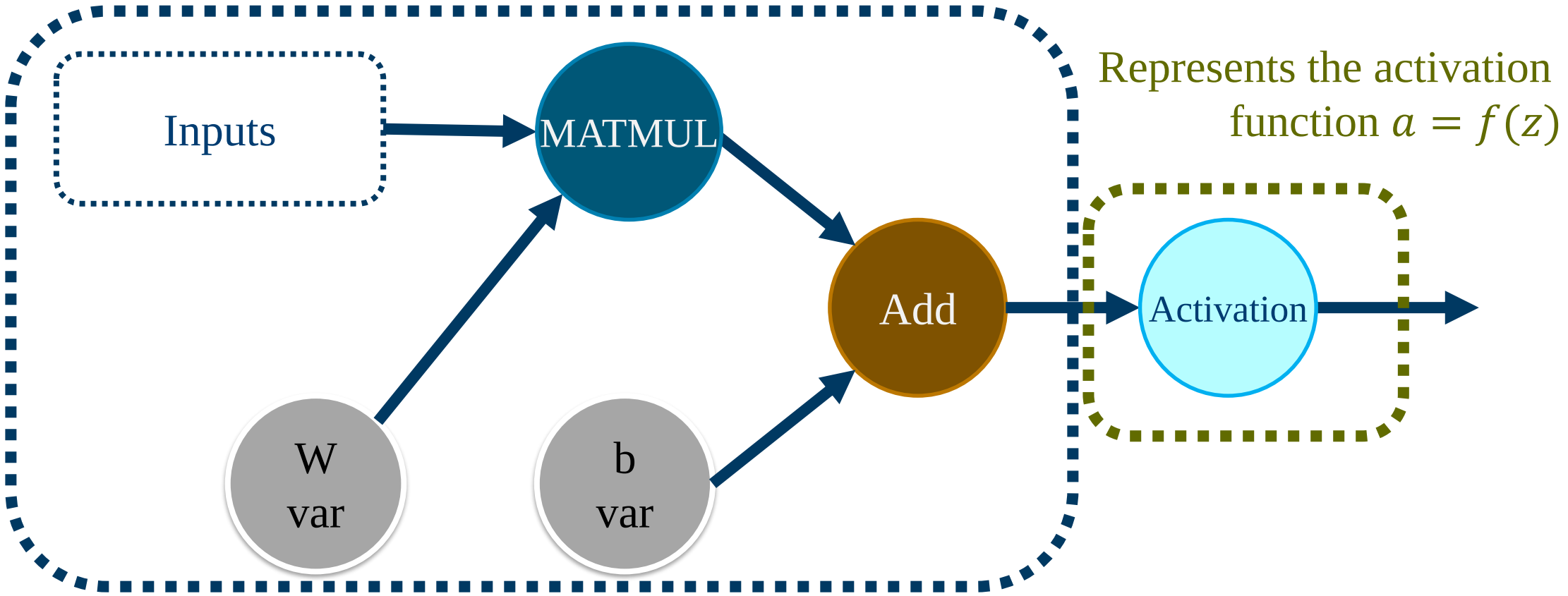
Less than a single FC layer with [1200 x 1200] weights!

WHAT'S THE POINT? CNN LEARNS FEATURES!

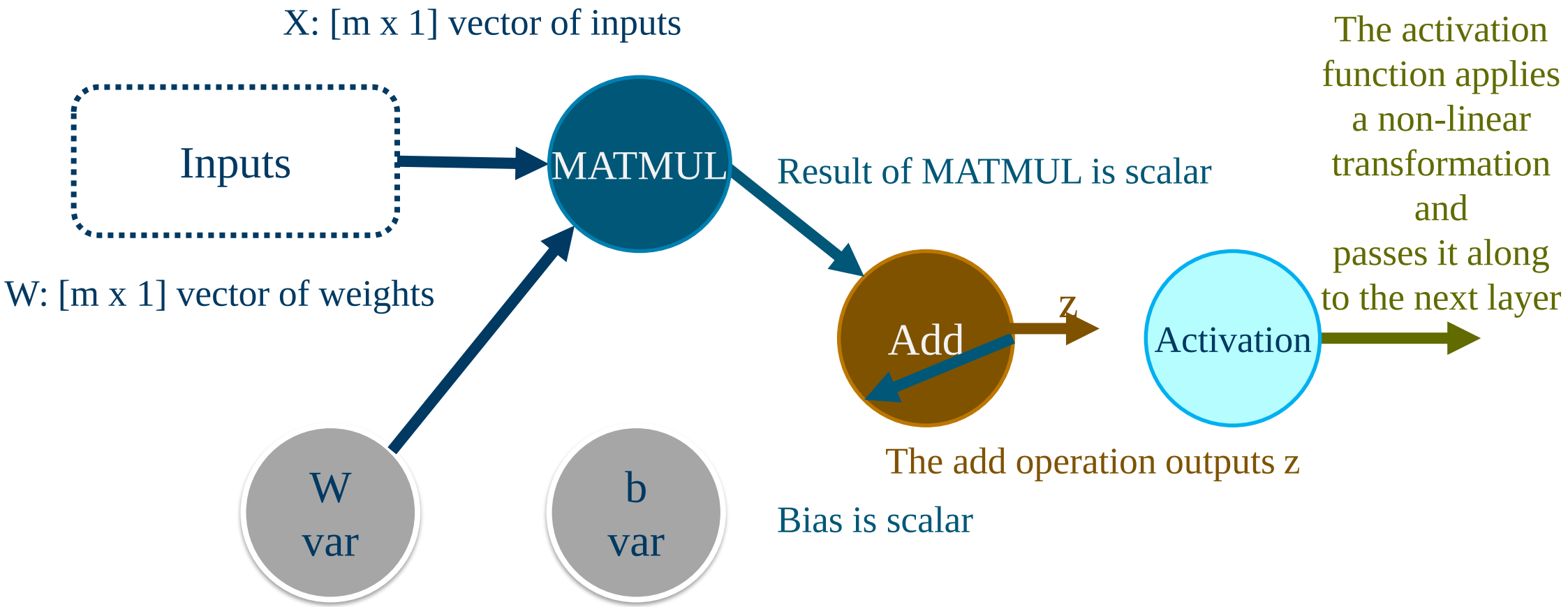
Layers replace manual image processing, transforming, and feature extraction!

- For example, a slightly different architecture called AlexNet has a layer that *essentially* performs Sobel filtering.
- Edge detection as a *layer*
- See:
  - http://cs231n.github.io/assets/cnnvis/filt1.jpeg

NODES


Inputs
MATMUL
Add
Activation
W
var
b
var
Represents the activation function $a = f(z)$

NODES


X: [m x 1] vector of inputs
Inputs
MATMUL
Result of MATMUL is scalar
The activation function applies a non-linear transformation and passes it along to the next layer
W: [m x 1] vector of weights
Add
z
Activation
W
var
b
var
The add operation outputs z
Bias is scalar

BATCHED NODES