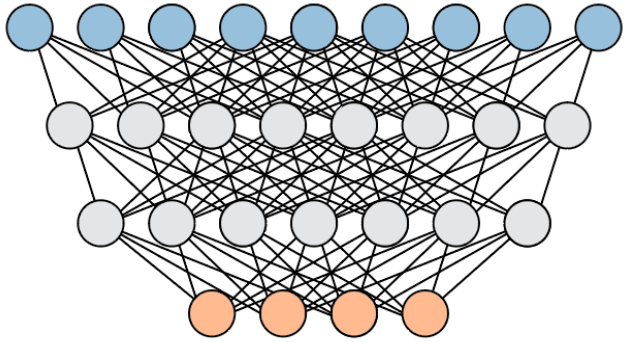


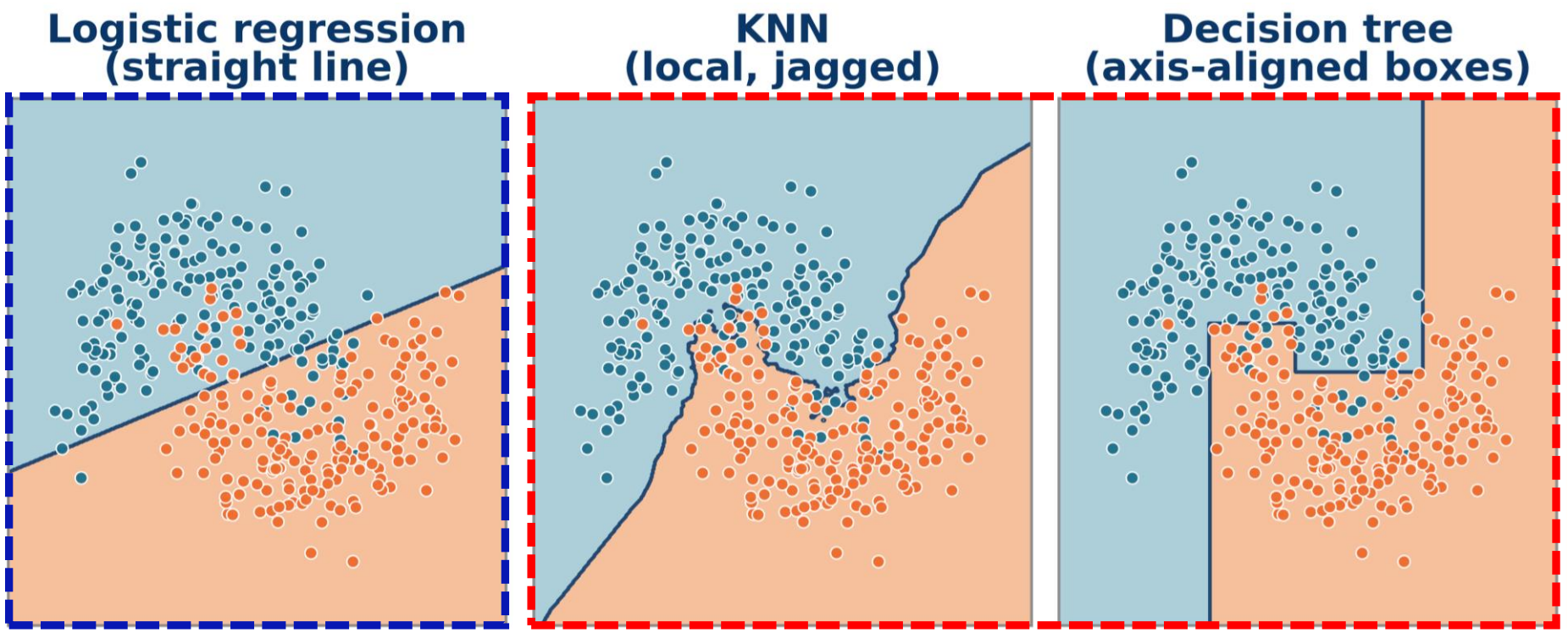
COM SCI M148, Lecture 13:
Neural Networks

Prof. Nathan Szymanski
nszymanski@seas.ucla.edu
Department of Materials Science

UCLA

Samueli
School of Engineering

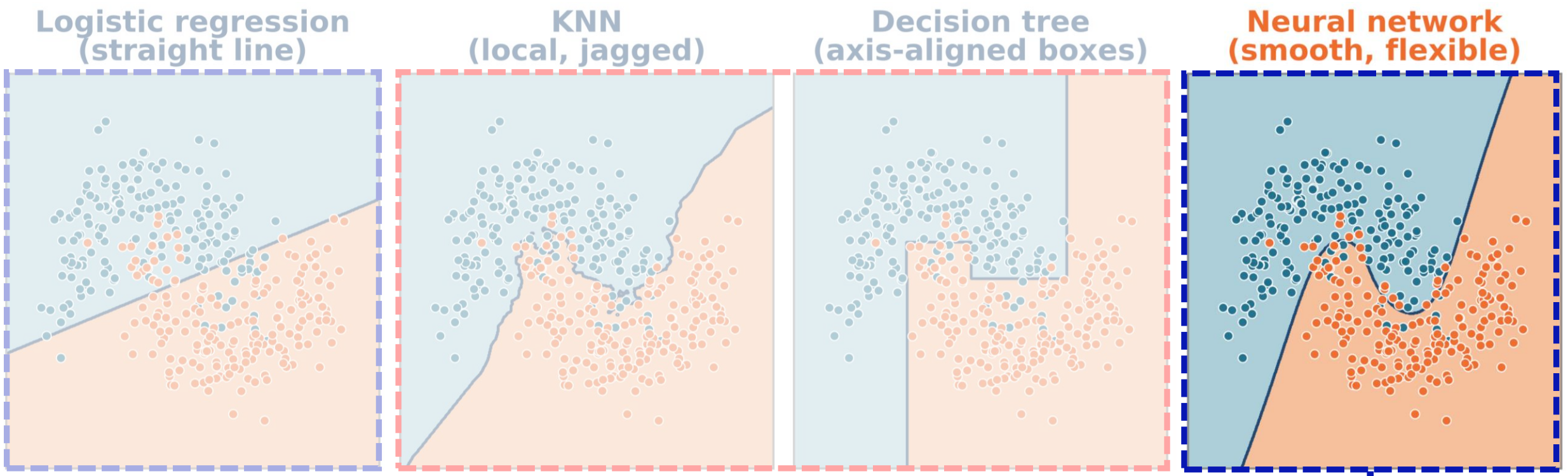
A recap of classification models



Parametric model:
Add interaction terms to make this curved.

Non-parametric models.
We do *not* assume any specific functional form.

A recap of classification models

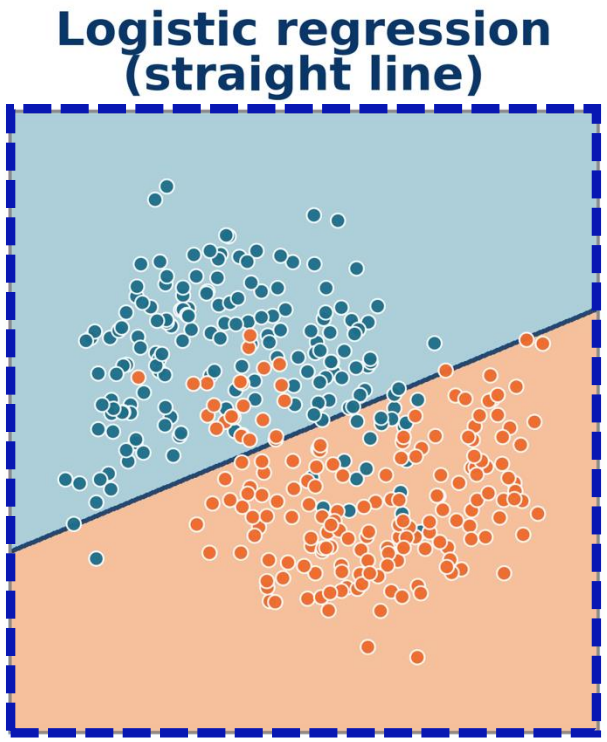


Parametric model:
Add interaction terms to make this curved.

Non-parametric models.
We do *not* assume any specific functional form.

Neural networks are parametric, but unlike logistic regression, neural networks can **automatically learn complex features from the data – without hand crafting.**

Recap: Logistic regression (a precursor to NNs)



$$P(y = 1|x) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

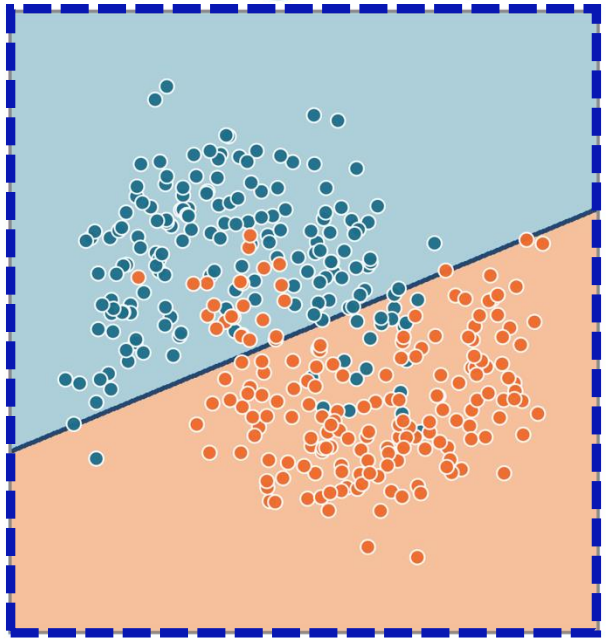
The probability that x belongs to class 1 is equal to the **sigmoid of z**

$$z = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$$

z is a weighted sum of the input features + an offset

Recap: Logistic regression (a precursor to NNs)

Logistic regression (straight line)



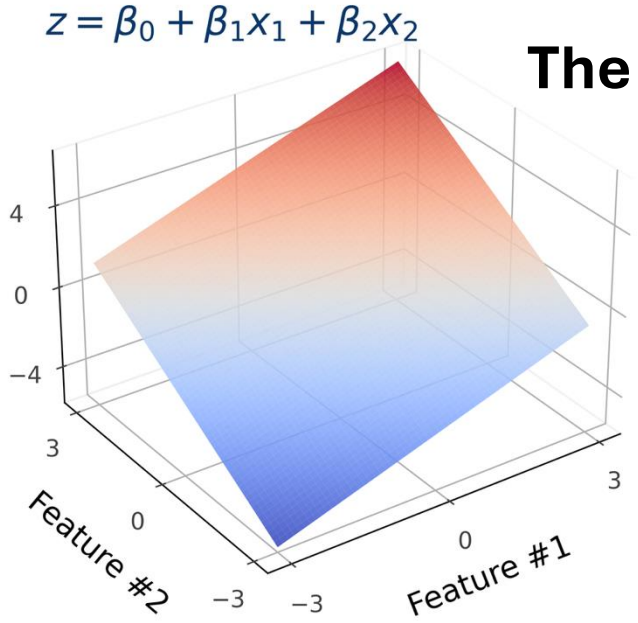
$$P(y = 1|x) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

The probability that x belongs to class 1 is equal to the **sigmoid of z**

$$z = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$$

z is a weighted sum of the input features + an offset

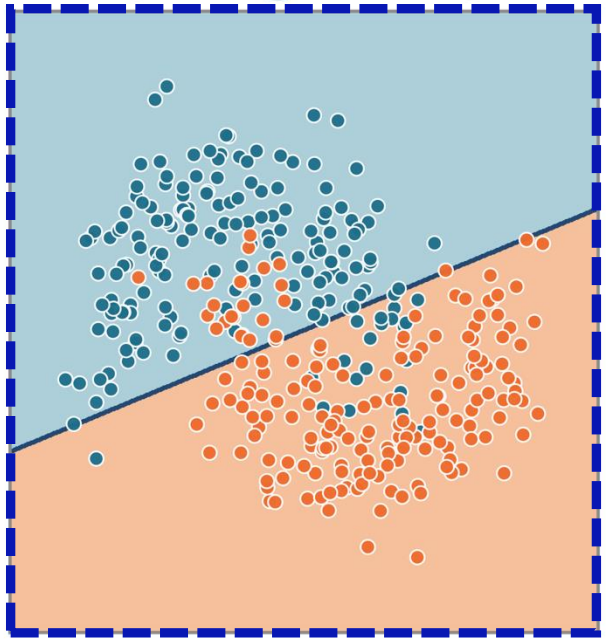
A **linear mapping** ($x_i \rightarrow z$)



The log-odds

Recap: Logistic regression (a precursor to NNs)

Logistic regression (straight line)



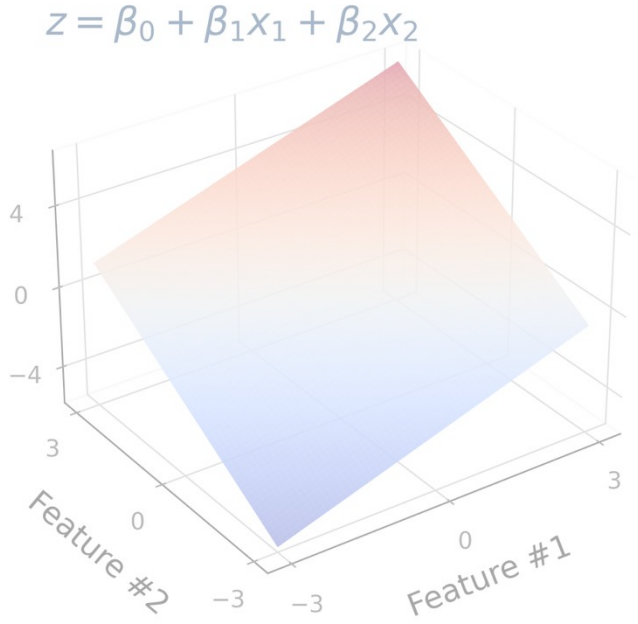
$$P(y = 1|x) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

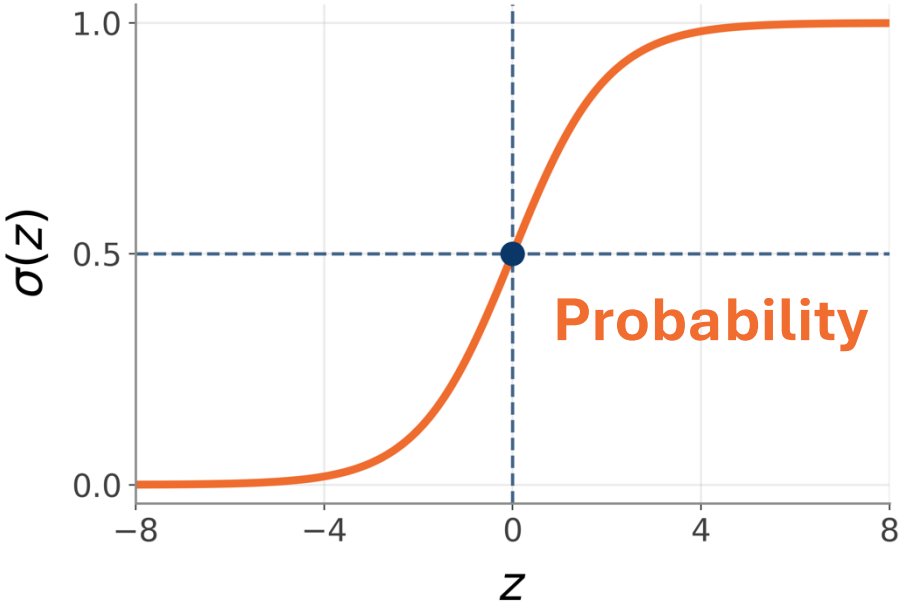
The probability that x belongs to class 1 is equal to the **sigmoid of z**

$$z = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$$

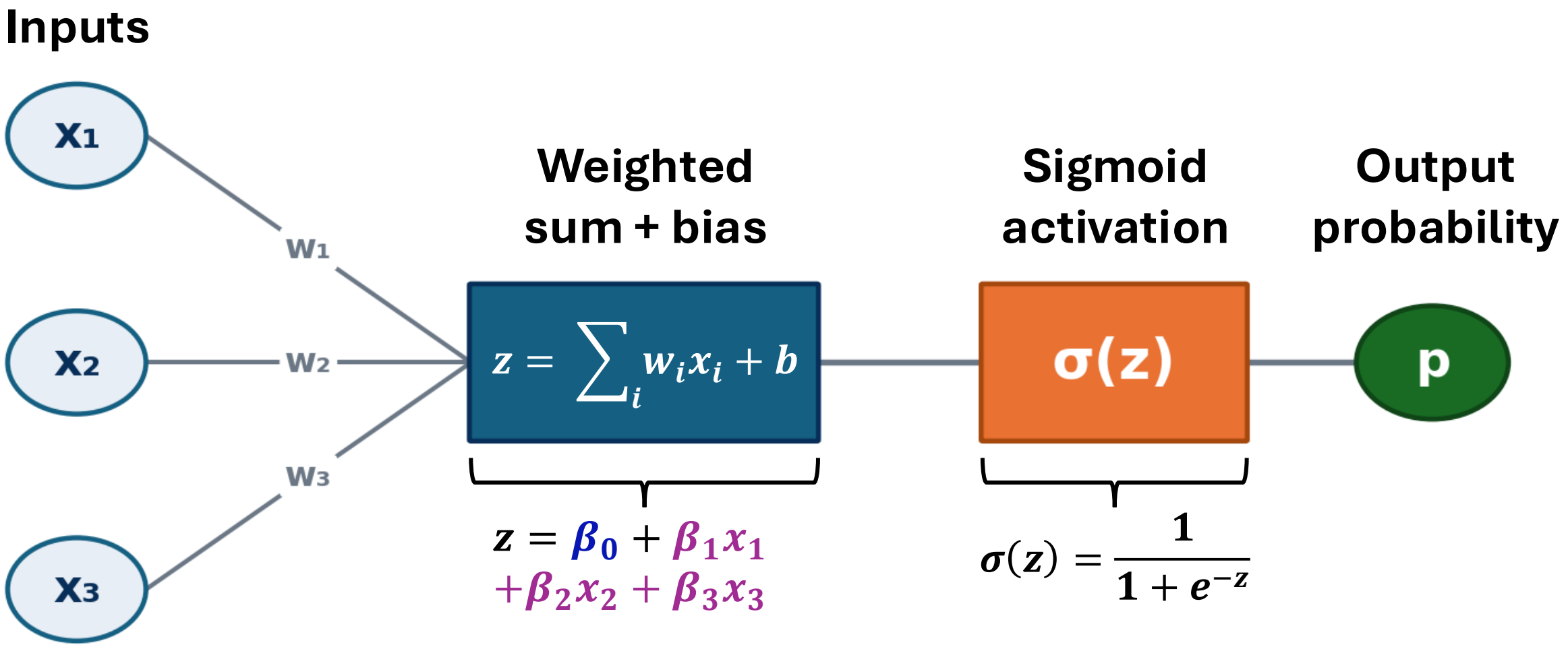
z is a weighted sum of the input features + an offset

A **linear mapping** ($x_i \rightarrow z$) followed by a **non-linear mapping** ($z \rightarrow P$)



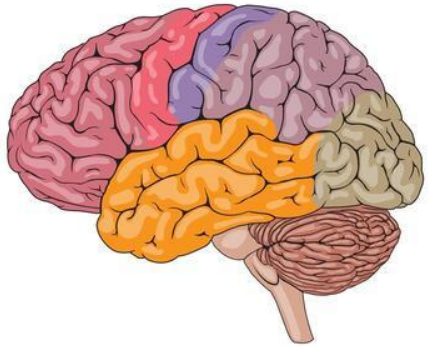


Logistic regression as a *neuron*



Rewrite the **coefficients (β_i) as weights (w_i)** and **β_0 as the bias (b)**

Inspiration from biology: The neuron



Dendrites

Axon

Nucleus

Signal in → Signal out

Inputs

x0=1

x1

x2

...

xn

b

w1

w2

wn

z

f

y

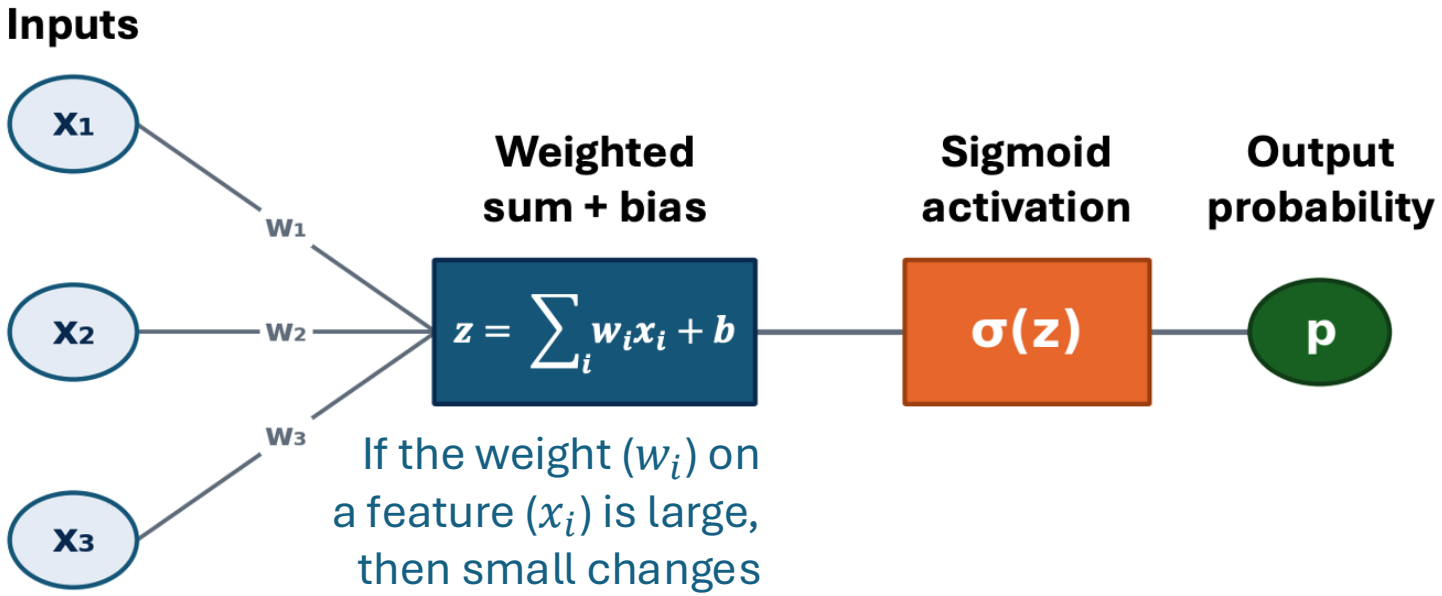
Output

Linear function

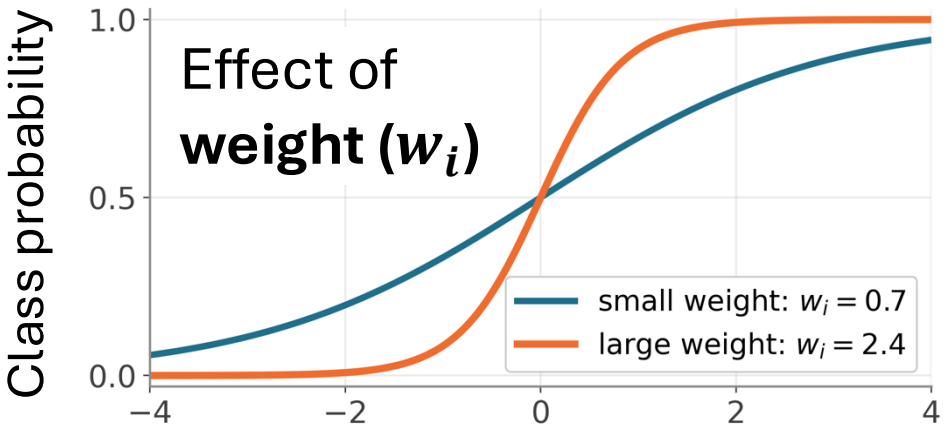
Activation function

Both operations written as one neuron

What do the weights (w_i) and bias (b) do?



If the weight (w_i) on a feature (x_i) is large, then small changes in that feature lead to large changes in the probability

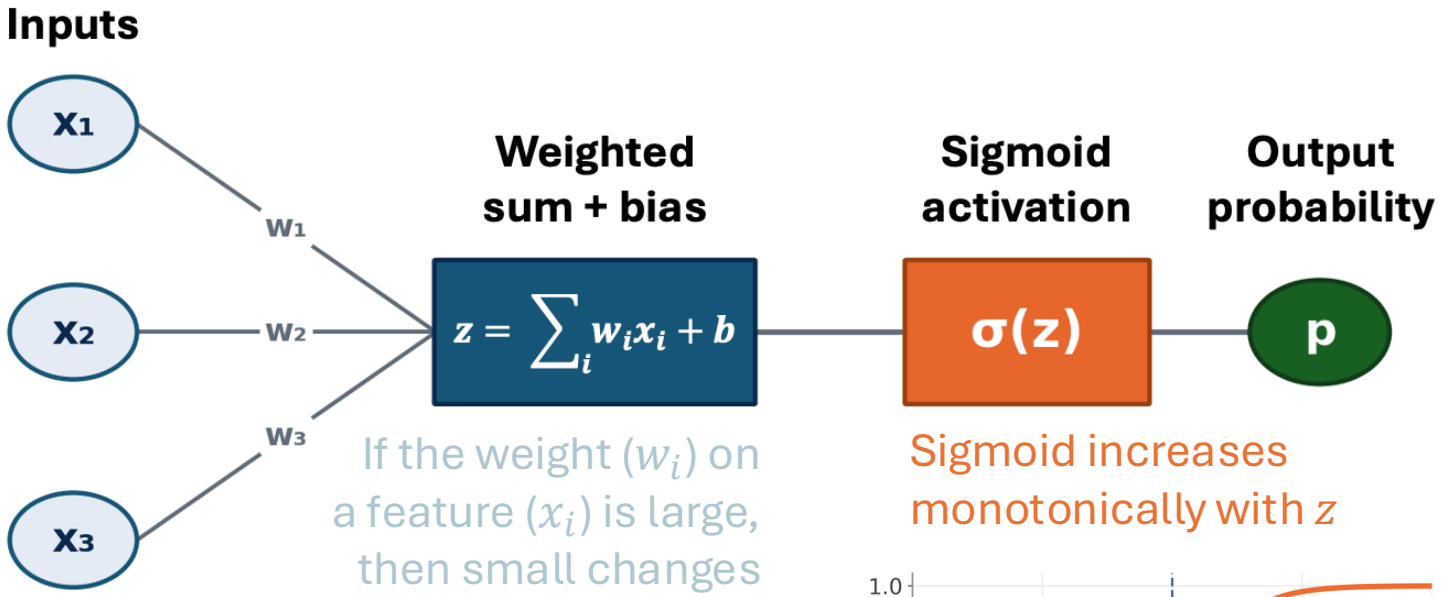


Just like the coefficients (β_i) in logistic regression

Large w_i → big effect

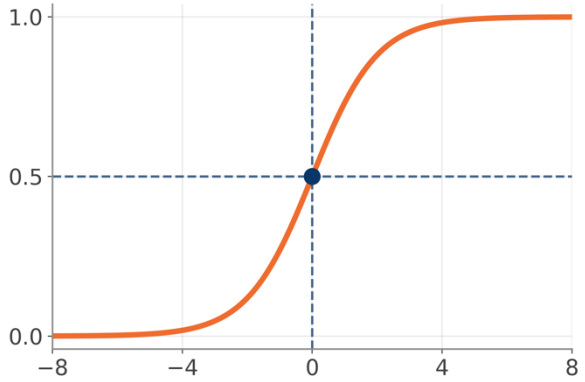
Weights determine **how strongly** and **in which direction** each feature changes the class probability

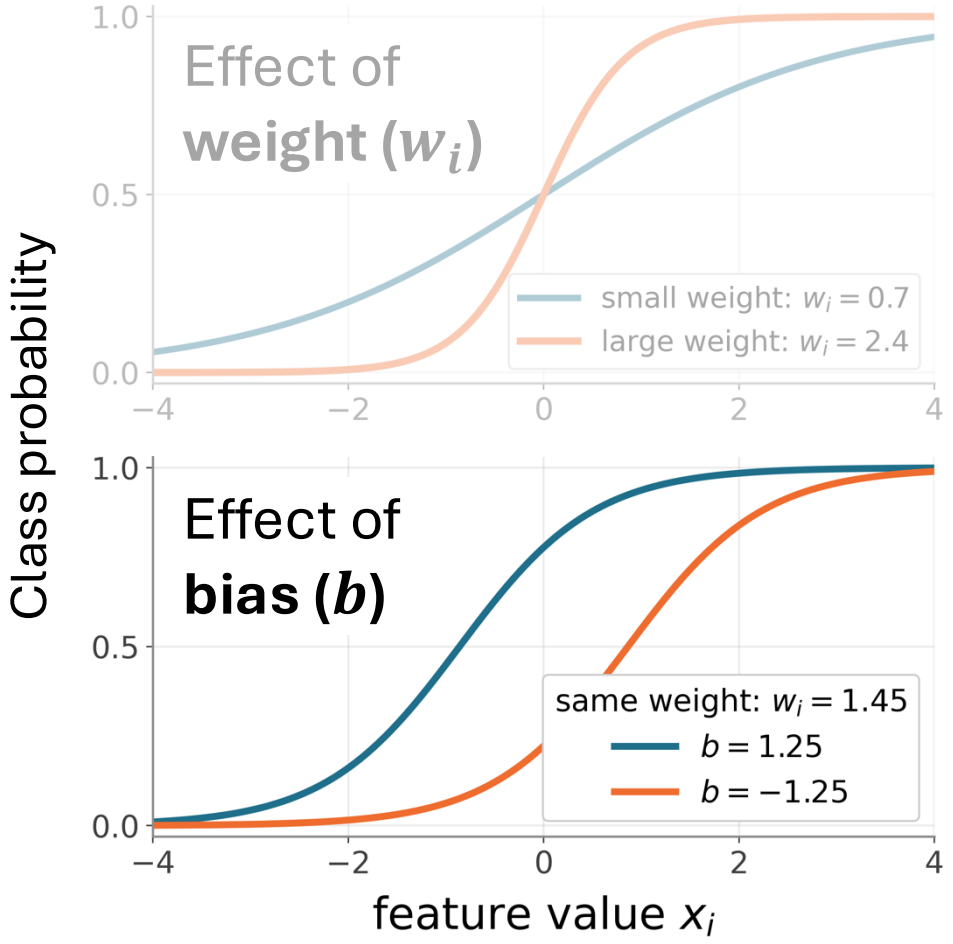
What do the weights (w_i) and bias (b) do?



If the weight (w_i) on a feature (x_i) is large, then small changes in that feature lead to large changes in the probability

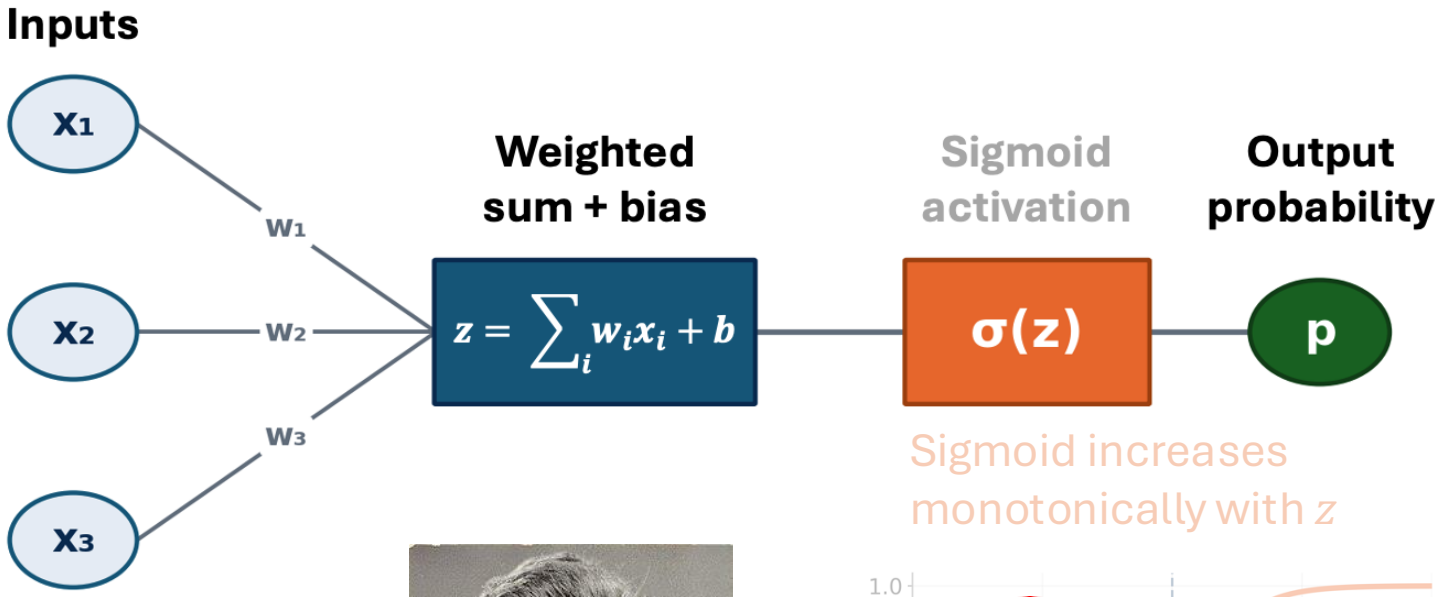
Sigmoid increases monotonically with z



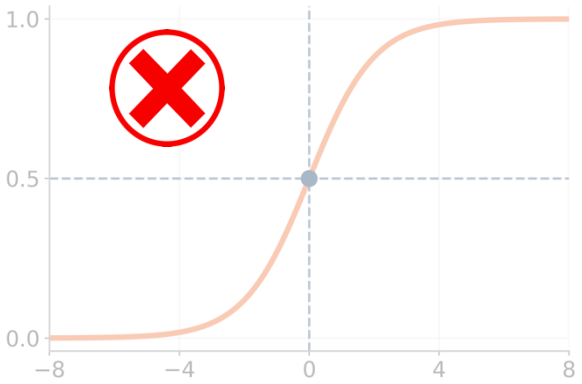


Weights determine **how strongly** and **in which direction** each feature changes the class probability, the **bias sets the neuron's baseline tendency toward one class.**

A *perceptron* = 1 neuron with a step function



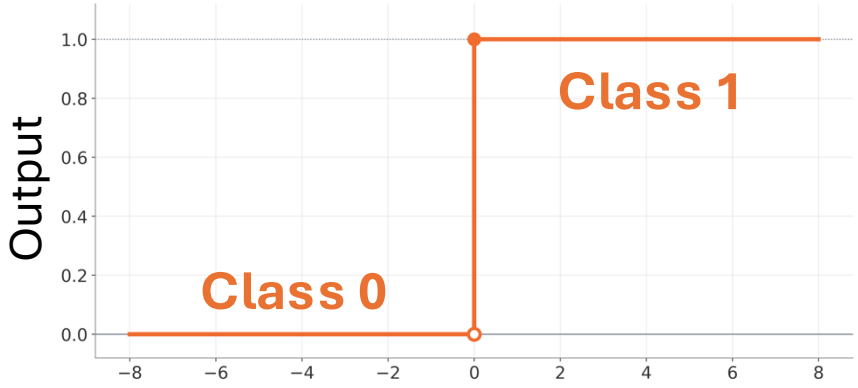
Sigmoid increases monotonically with z





Suppose we don't care about probability; we just want a class output (0 or 1). We can replace sigmoid with a **step function:**

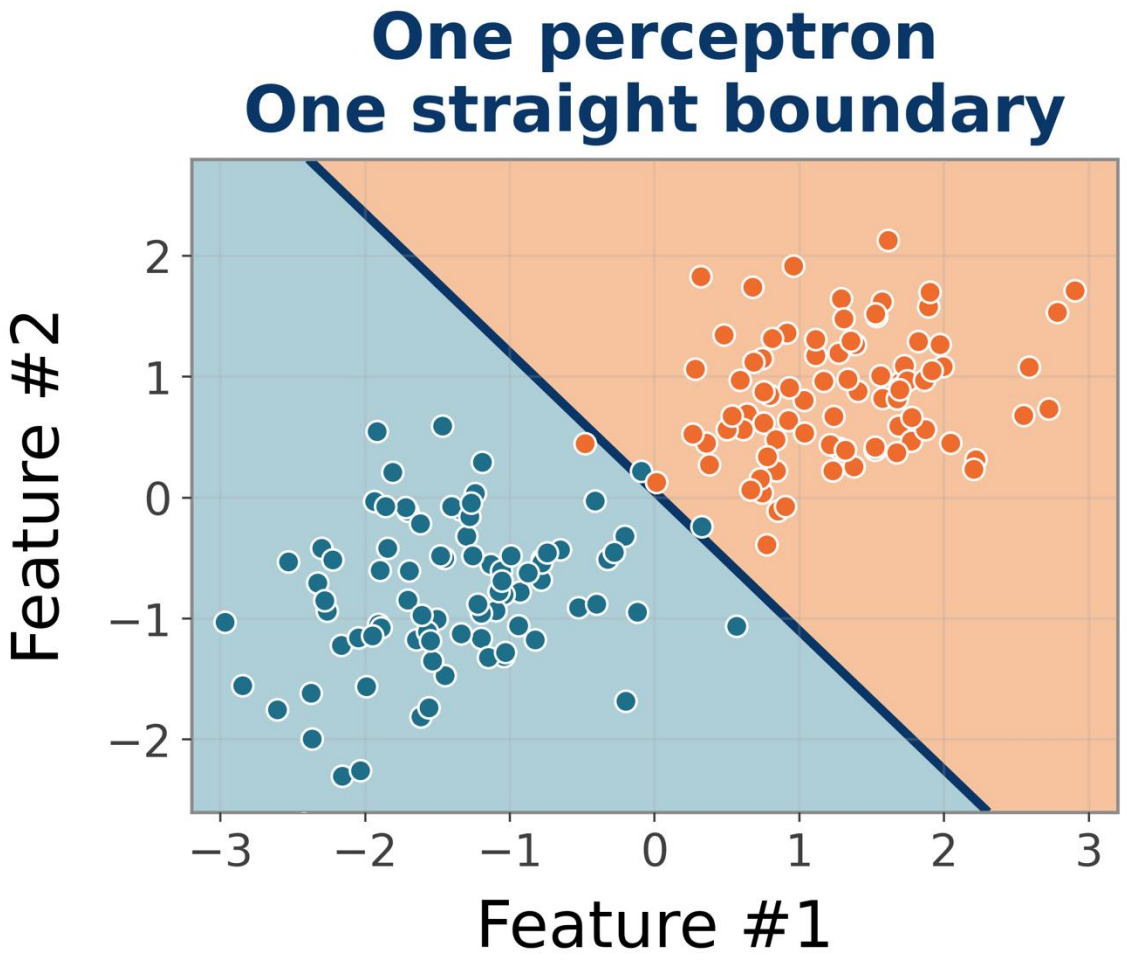
$$y = \begin{cases} 1, z \geq 0 \\ 0, z < 0 \end{cases}$$

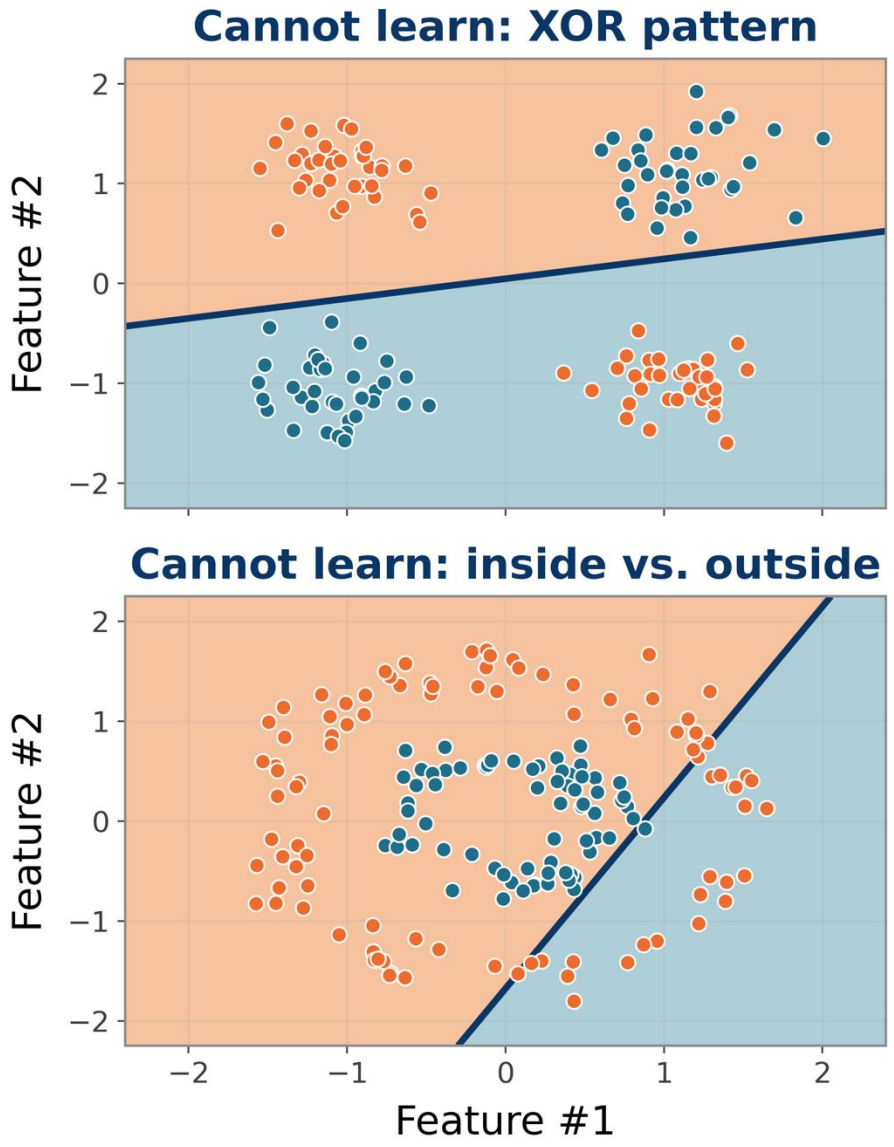


$$z = \sum_i w_i x_i + b$$

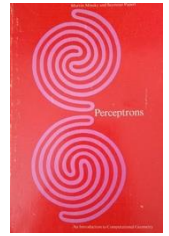
This was the idea behind the original ***perceptron***, created by **Frank Rosenblatt** – the "father of deep learning" – in 1957

What can be learned with one perceptron?









1969 book

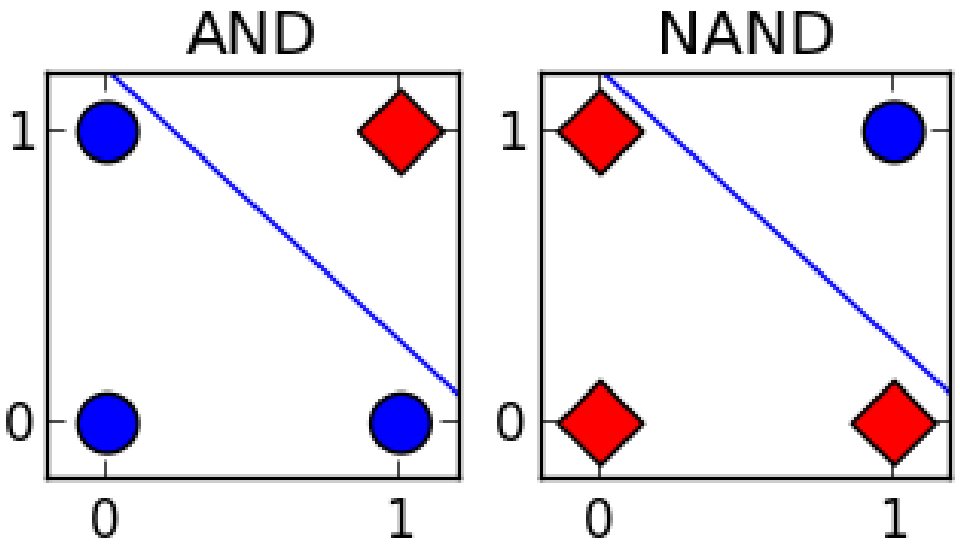
What can be learned with one perceptron?

Logic gates for computing:

AND: Outputs 1 when **both inputs are 1**

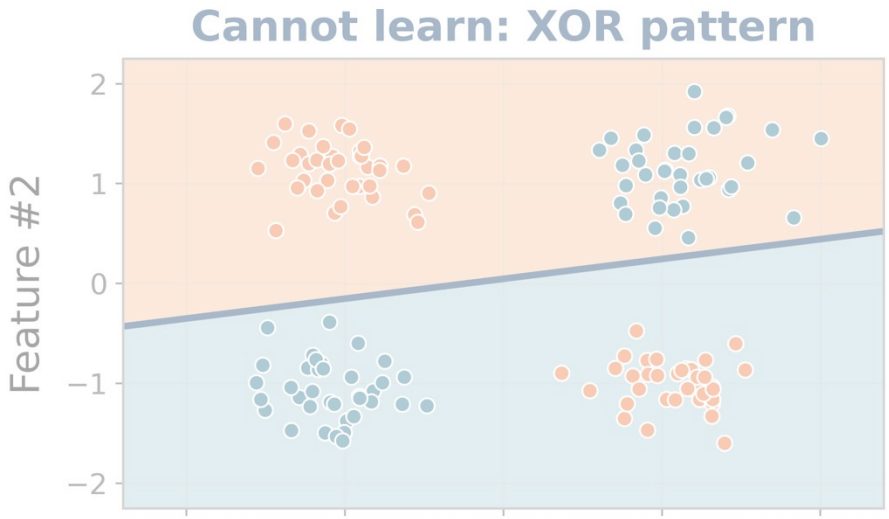
NAND: Outputs 0 when **both inputs are 1**





= Output 1

= Output 0







1969 book

What can be learned with one perceptron?

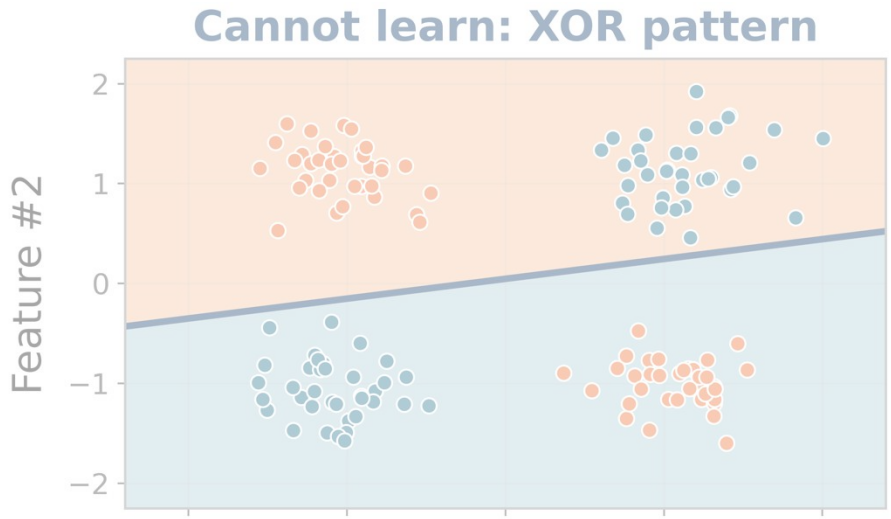
Logic gates for computing:

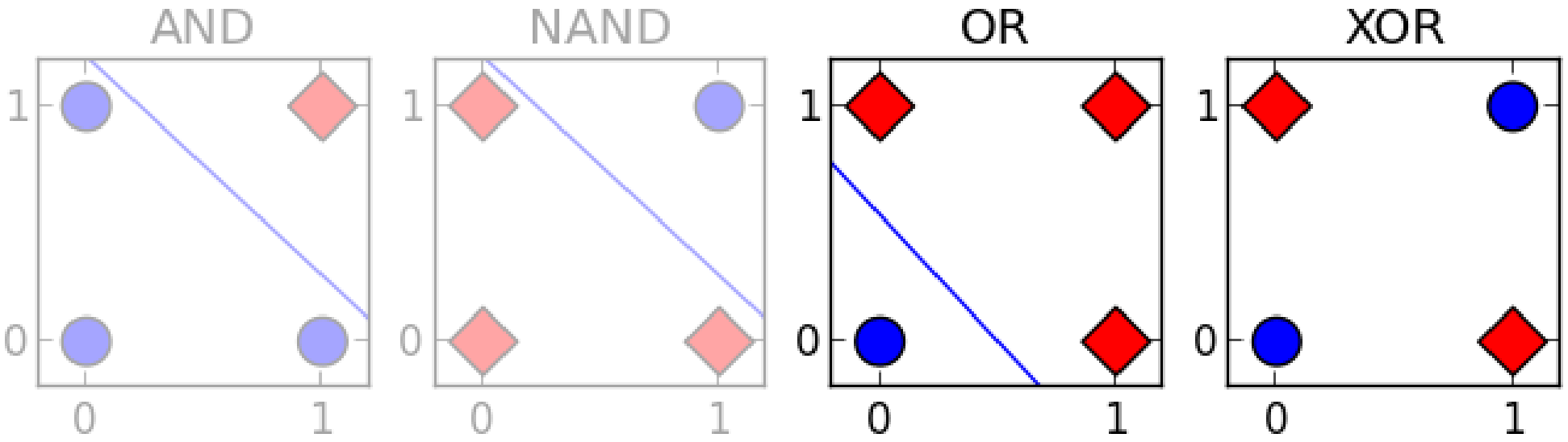
AND: Outputs 1 when **both inputs are 1**

NAND: Outputs 0 when **both inputs are 1**

OR: Outputs 1 when **at least one input is 1**

XOR: Outputs 1 when inputs are **different (exclusive OR)**



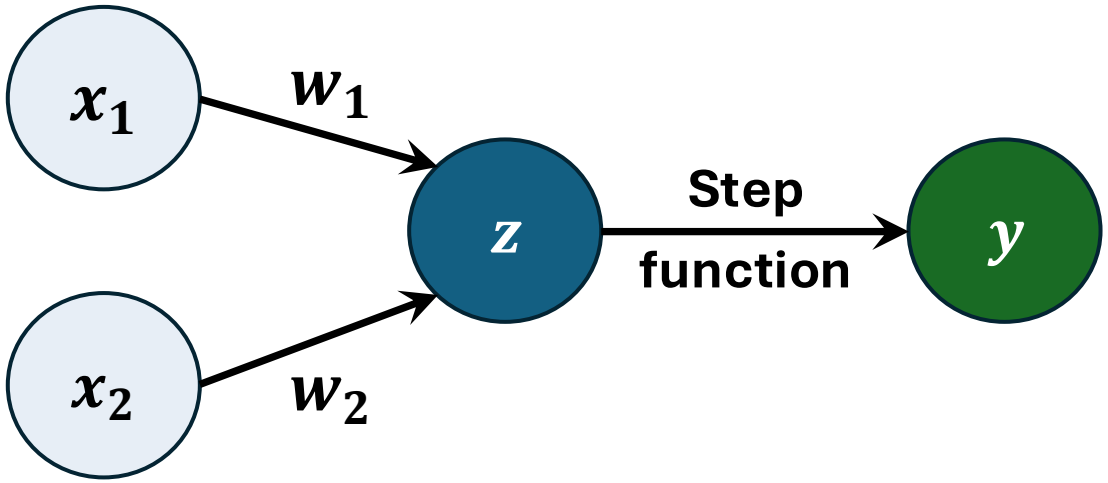






1969 book

One perceptron: AND, NAND, OR



We can solve AND, NAND, OR problems with just one perceptron by setting weights and bias appropriately

AND: $y = \text{step}(x_1 + x_2 - 1.5) = 1$ only if x_1 and x_2 are both 1

NAND: $y = \text{step}(-x_1 - x_2 + 1.5) = 1$ only if x_1 and x_2 are both *not* 1

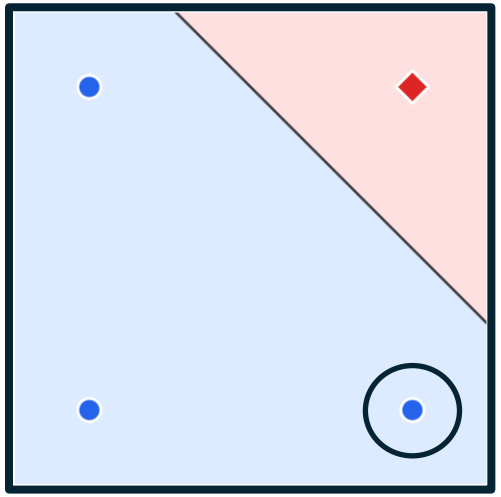
OR: $y = \text{step}(x_1 + x_2 - 0.5) = 1$ if either x_1 or x_2 is 1

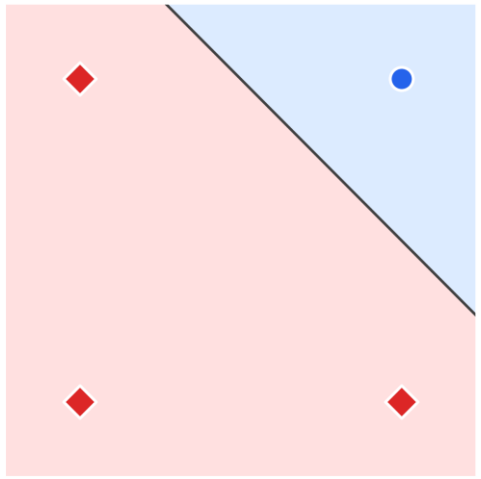
XOR: No possible weights to give 1 only if x_1 or x_2 are different…

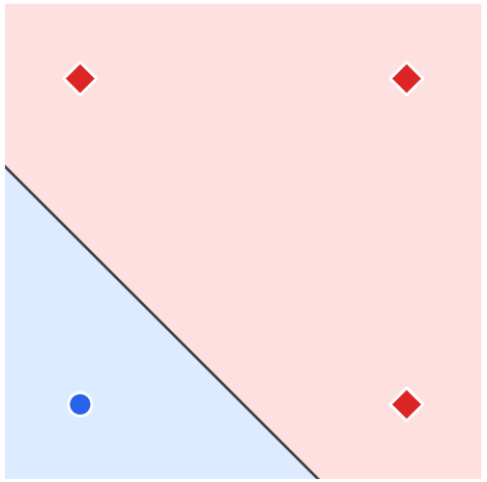
One perceptron: AND, NAND, OR

Example:

$$\mathbf{1 + 0 - 1.5 = -0.5}$$
$$\mathbf{step(-0.5) = 0}$$







AND: $y = \text{step}(x_1 + x_2 - 1.5) = 1$ only if x_1 and x_2 are both 1

NAND: $y = \text{step}(-x_1 - x_2 + 1.5) = 1$ only if x_1 and x_2 are both *not* 1

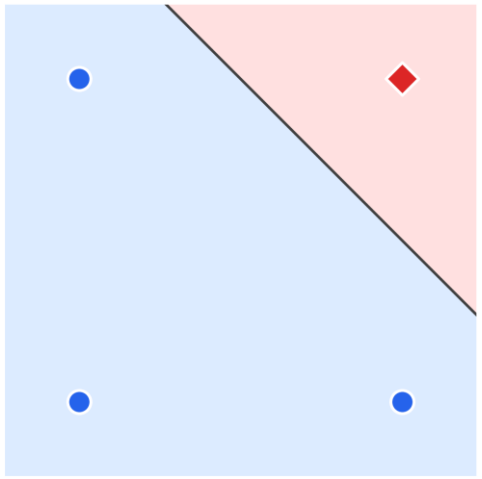
OR: $y = \text{step}(x_1 + x_2 - 0.5) = 1$ if either x_1 or x_2 is 1

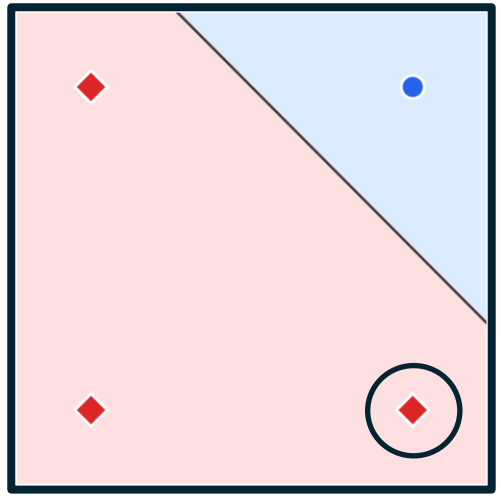
XOR: No possible weights to give 1 only if x_1 or x_2 are different…

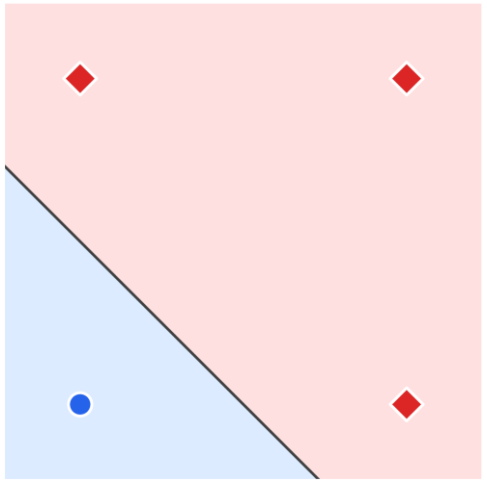
One perceptron: AND, NAND, OR

Example:

$$-1 - 0 + 1.5 = 0.5$$
$$\text{step}(0.5) = 1$$







AND: $y = \text{step}(x_1 + x_2 - 1.5) = 1$ only if x_1 and x_2 are both 1

NAND: $y = \text{step}(-x_1 - x_2 + 1.5) = 1$ only if x_1 and x_2 are both *not* 1

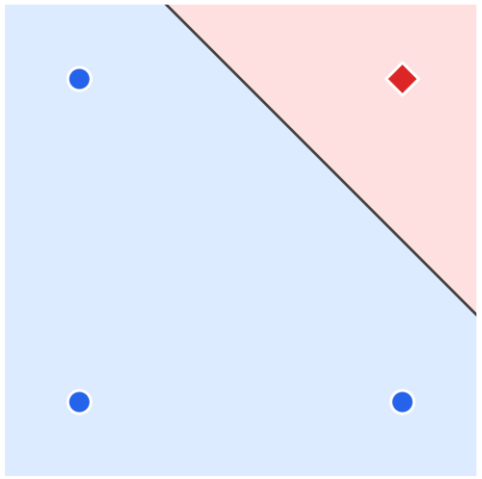
OR: $y = \text{step}(x_1 + x_2 - 0.5) = 1$ if either x_1 or x_2 is 1

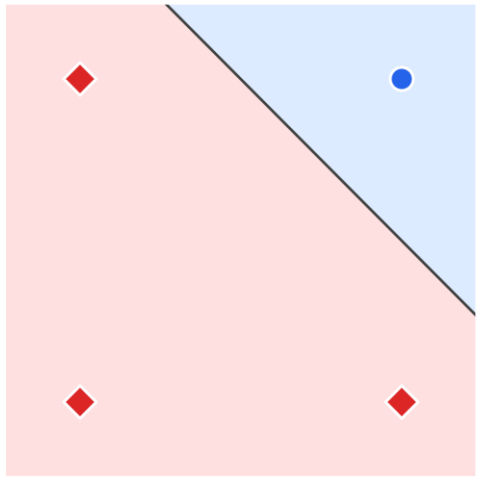
XOR: No possible weights to give 1 only if x_1 or x_2 are different…

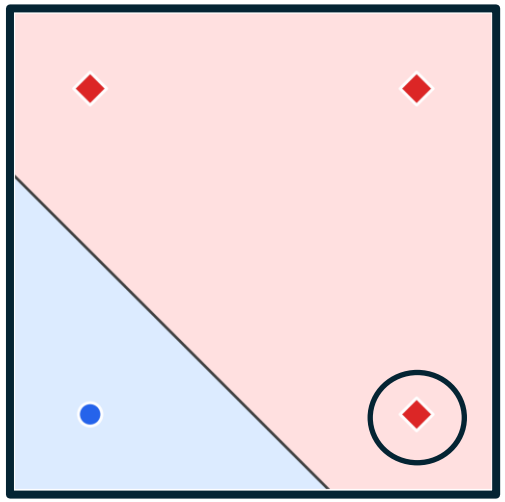
One perceptron: AND, NAND, OR

Example:

$$\mathbf{1 + 0 - 0.5}$$
$$\mathbf{= 0.5}$$
$$\mathbf{step(0.5) = 1}$$







AND: $y = \text{step}(x_1 + x_2 - 1.5) = 1$ only if x_1 and x_2 are both 1

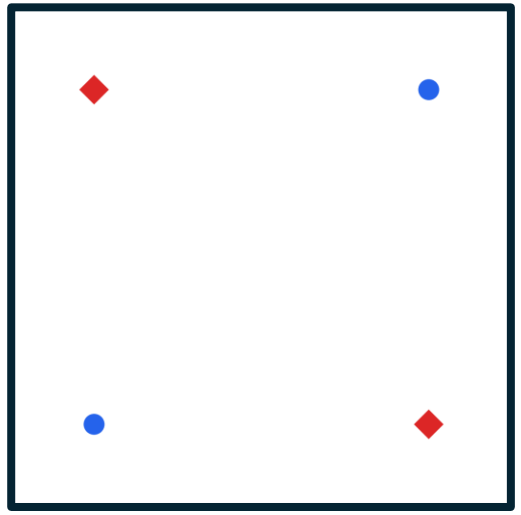
NAND: $y = \text{step}(-x_1 - x_2 + 1.5) = 1$ only if x_1 and x_2 are both *not* 1

OR: $y = \text{step}(x_1 + x_2 - 0.5) = 1$ if either x_1 or x_2 is 1

XOR: No possible weights to give 1 only if x_1 or x_2 are different…

One perceptron: AND, NAND, OR

There is **no single, straight line** we can draw to separate these!



?

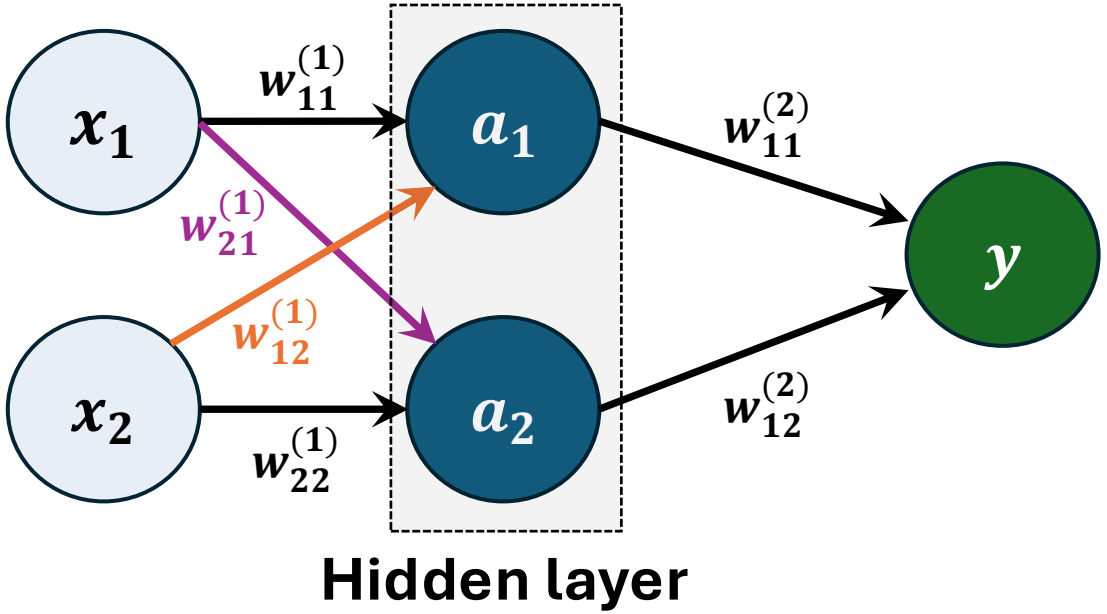
AND: $y = \text{step}(x_1 + x_2 - 1.5) = 1$ only if x_1 and x_2 are both 1

NAND: $y = \text{step}(-x_1 - x_2 + 1.5) = 1$ only if x_1 and x_2 are both *not* 1

OR: $y = \text{step}(x_1 + x_2 - 0.5) = 1$ if either x_1 or x_2 is 1

XOR: No possible weights to give 1 only if x_1 or x_2 are different…

Solution: Two perceptrons in a “hidden layer”



Implicit to the perceptrons in the hidden layer, and the output, is ***activation*** in the form of a step function:

$$a_1 = \mathbf{step}(z_1), \qquad a_2 = \mathbf{step}(z_2)$$

$$y = \mathbf{step}(w_{11}^{(2)} a_1 + w_{12}^{(2)} a_2)$$

We can now solve the **XOR problem** as follows:

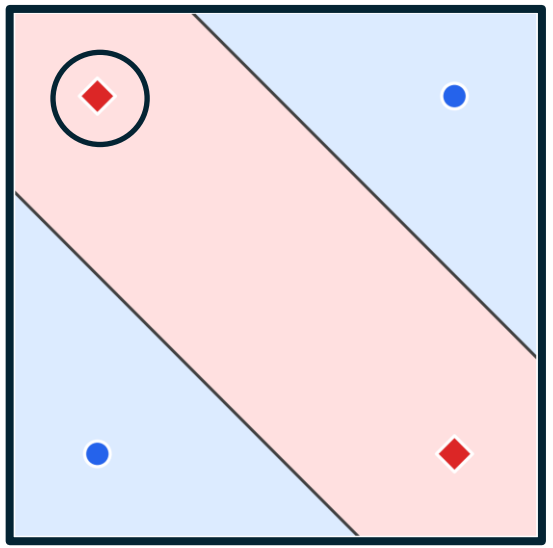
First hidden neuron computes **OR:** $a_1 = \mathbf{step}(x_1 + x_2 - 0.5)$

Second hidden neuron computes **NAND:** $a_2 = \mathbf{step}(-x_1 - x_2 + 1.5)$

The output neuron computes **AND of those:** $y = \mathbf{step}(a_1 + a_2 - 1.5)$

Solution: Two perceptrons in a “hidden layer”

x_1	x_2	a_1	a_2	y
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0



$\text{step}(0 + 1 - 0.5) = 1,$

$\text{step}(-0 - 1 + 1.5) = 1,$

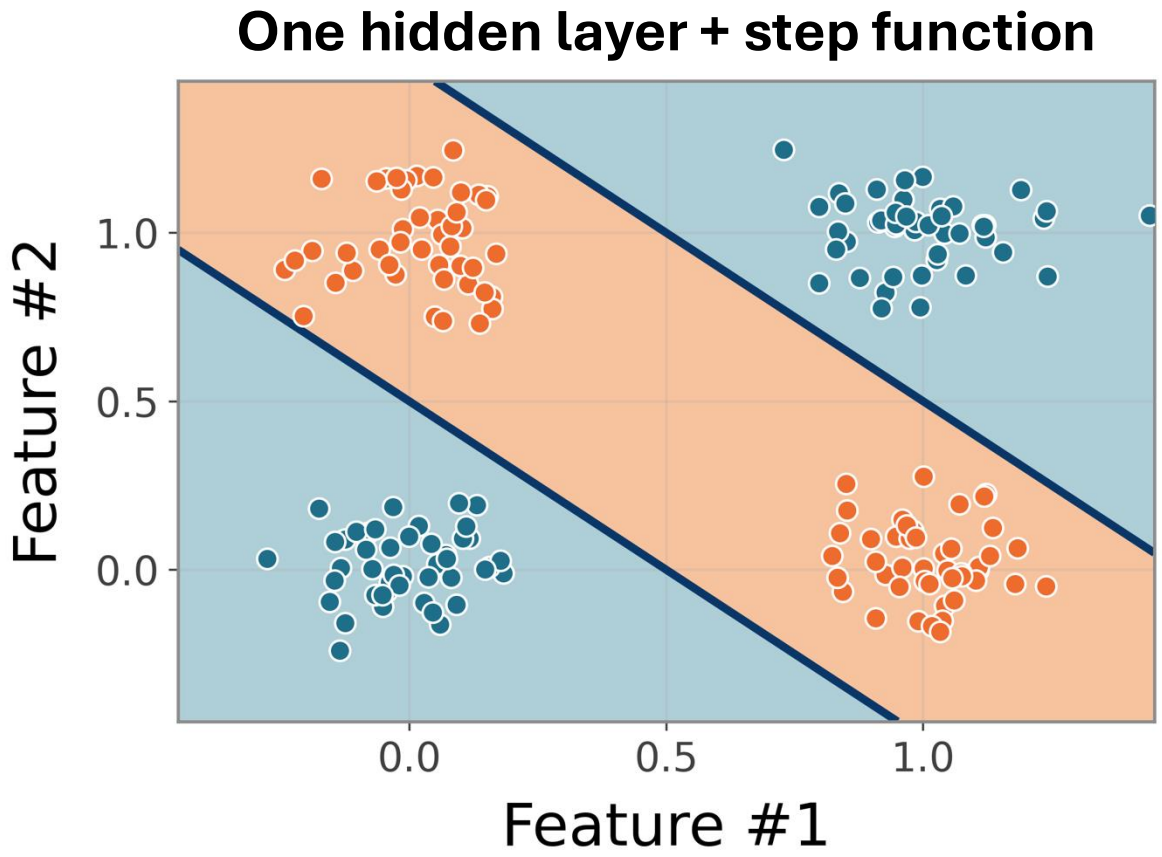
$\text{step}(1 + 1 - 1.5) = 1$

First hidden neuron computes **OR:** $\boldsymbol{a_1 = \text{step}(x_1 + x_2 - 0.5)}$

Second hidden neuron computes **NAND:** $\boldsymbol{a_2 = \text{step}(-x_1 - x_2 + 1.5)}$

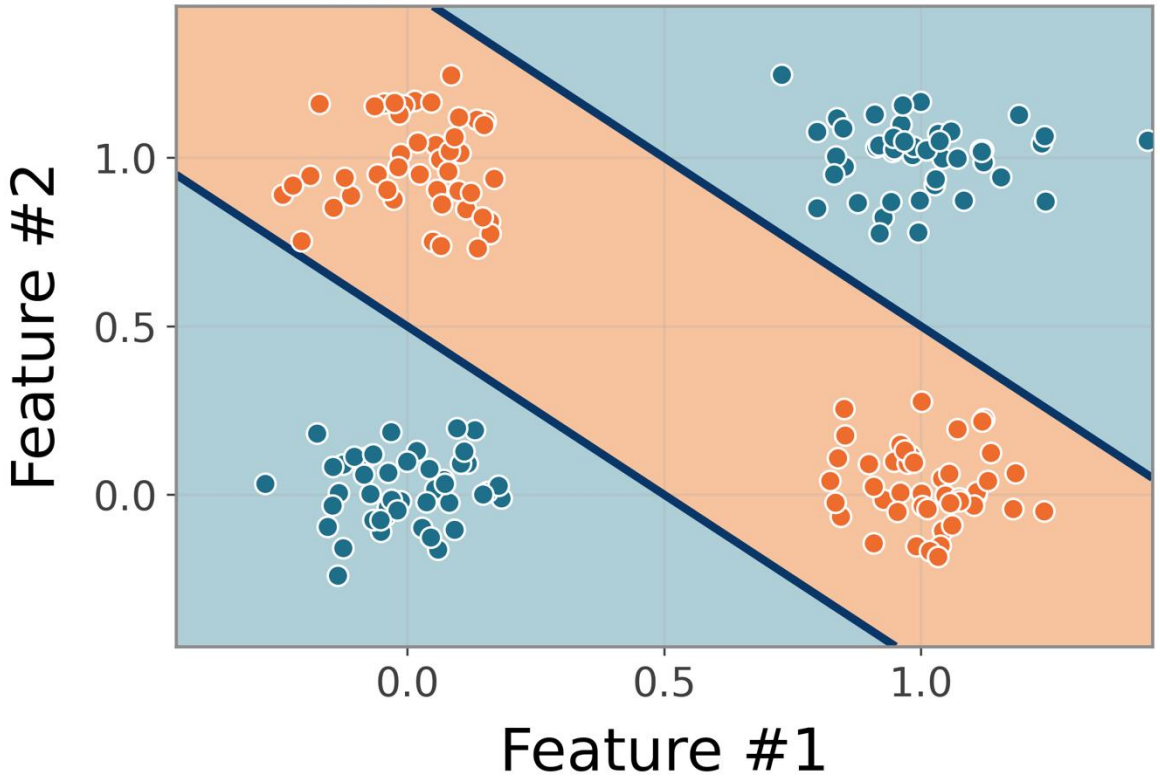
The output neuron computes **AND of those:** $\boldsymbol{y = \text{step}(a_1 + a_2 - 1.5)}$

Decision maps from a *multilayer perceptron*

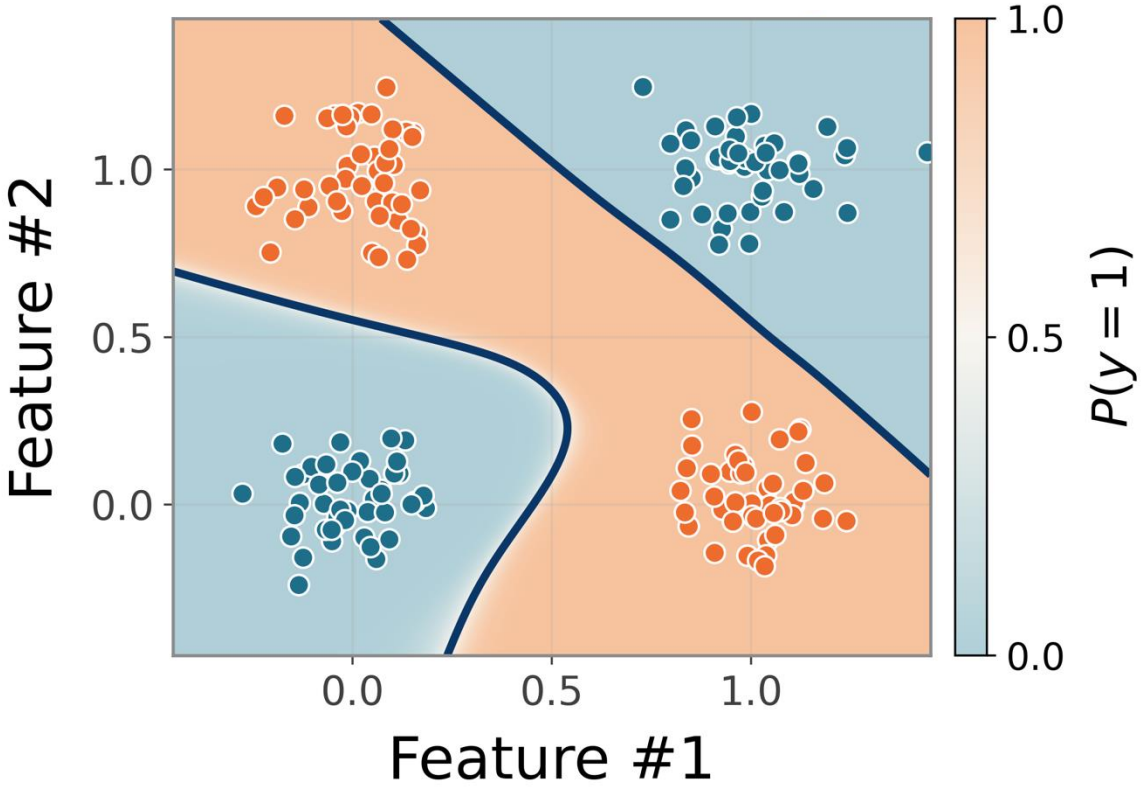


Decision maps from a *multilayer perceptron*

One hidden layer + step function



One hidden layer + sigmoid

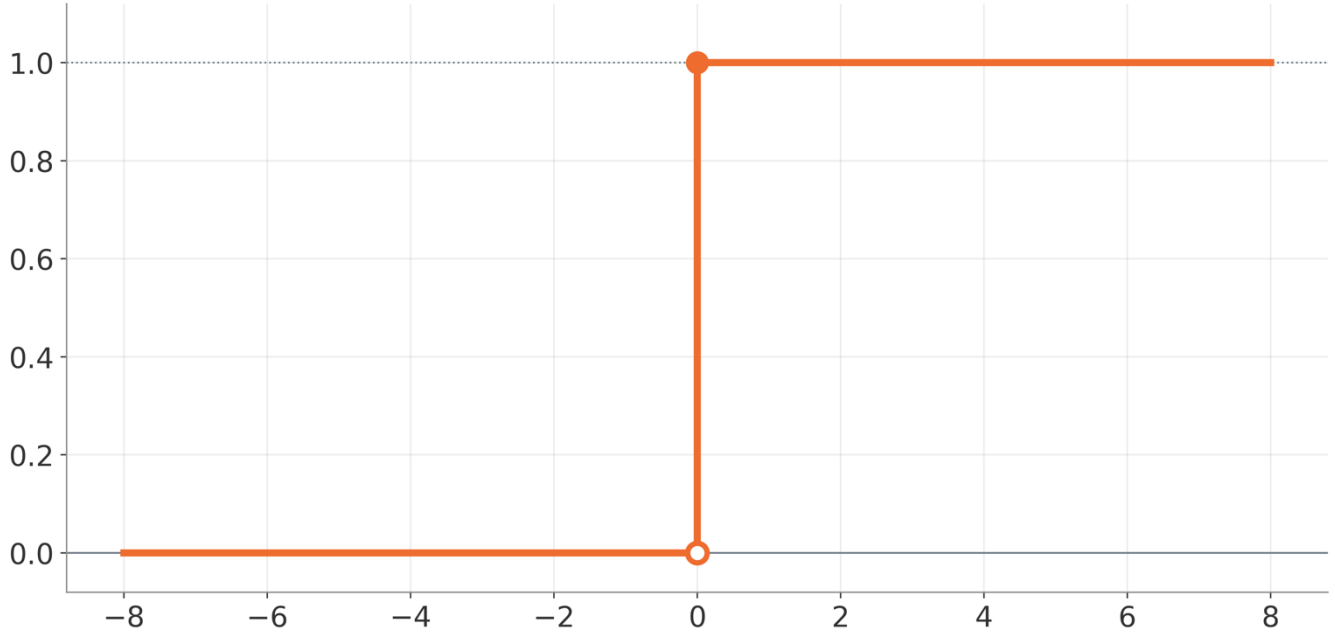


Sigmoid activations in hidden layers allow the network to build **nonlinear features.**

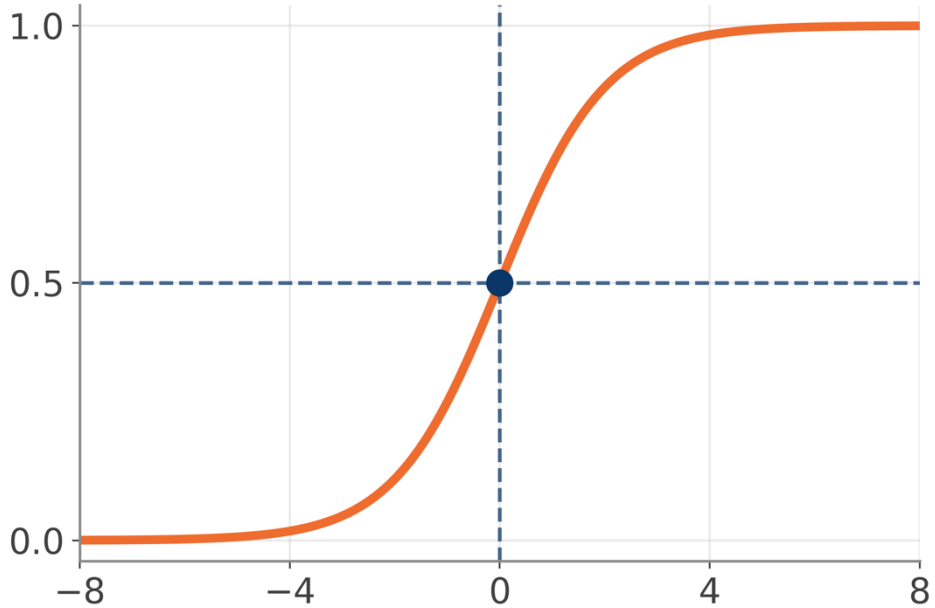
A sigmoid activation on the **output** gives **soft decision boundaries** (probability)

The importance of non-linearity

Step function



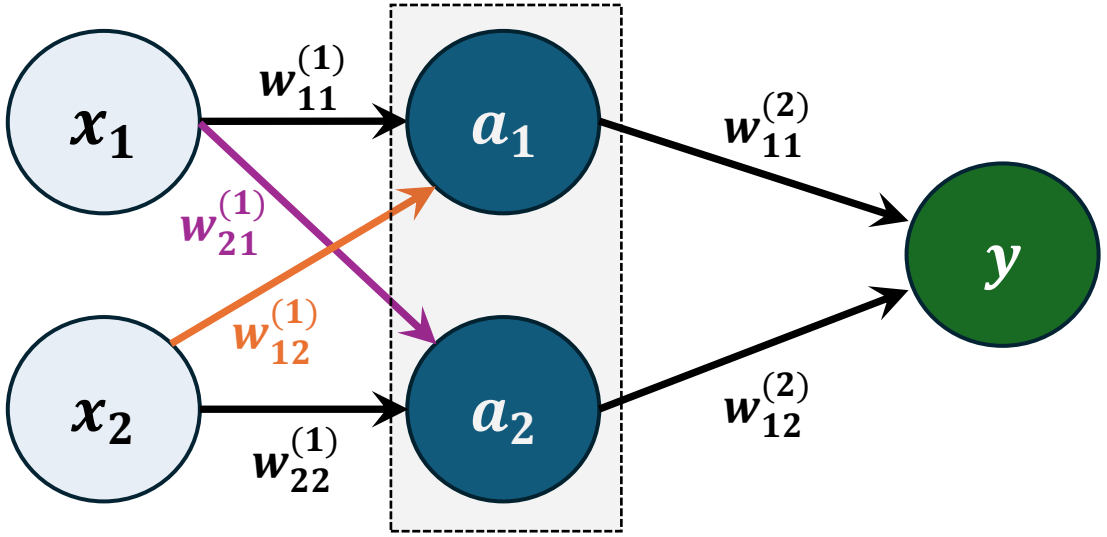
Sigmoid



It is critical that the activation function have some **non-linearity**...Even if just at a point!

Otherwise, composing linear layers still produces only a linear model!

Without non-linear activation



Without step functions:

$$a_1 = x_1 + x_2 - 0.5$$

$$a_2 = -x_1 - x_2 + 1.5$$

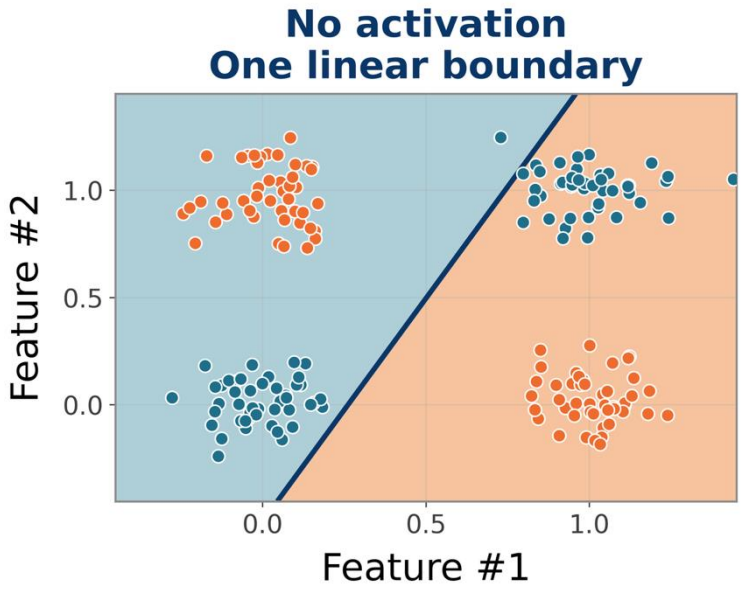
No activation

$$y = a_1 + a_2 - 1.5$$

$$y = (x_1 + x_2 - 0.5) + (-x_1 - x_2 + 1.5) - 1.5$$

$$y = -0.5$$

Here is a plot with "optimal" weights, no non-linear activation



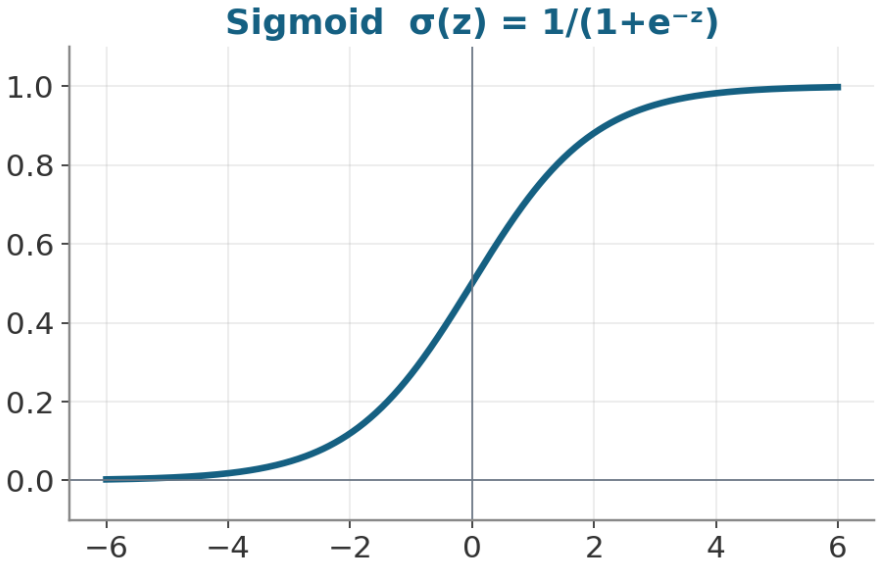
x_i may not always cancel, but we will ***always*** **get a linear function if no activation functions are used.**

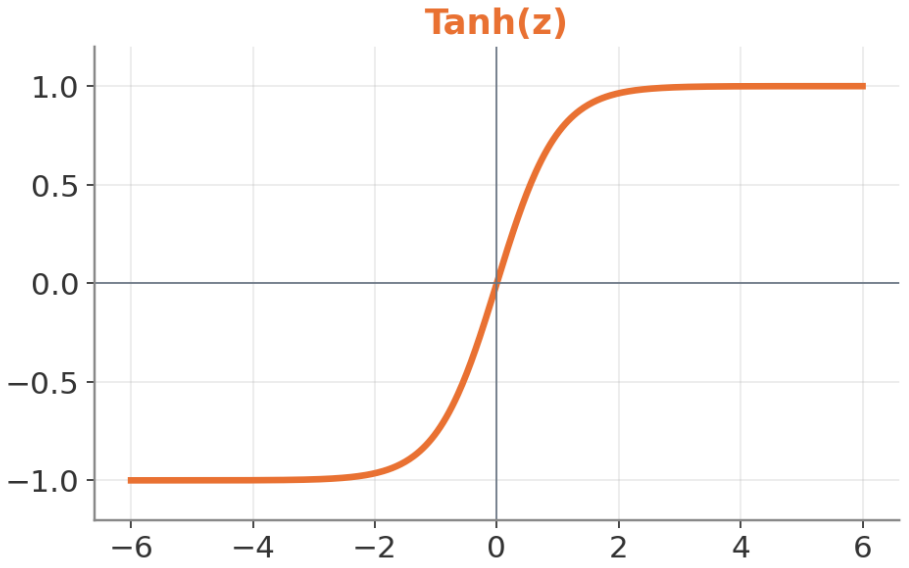
Questions?

Activation Functions

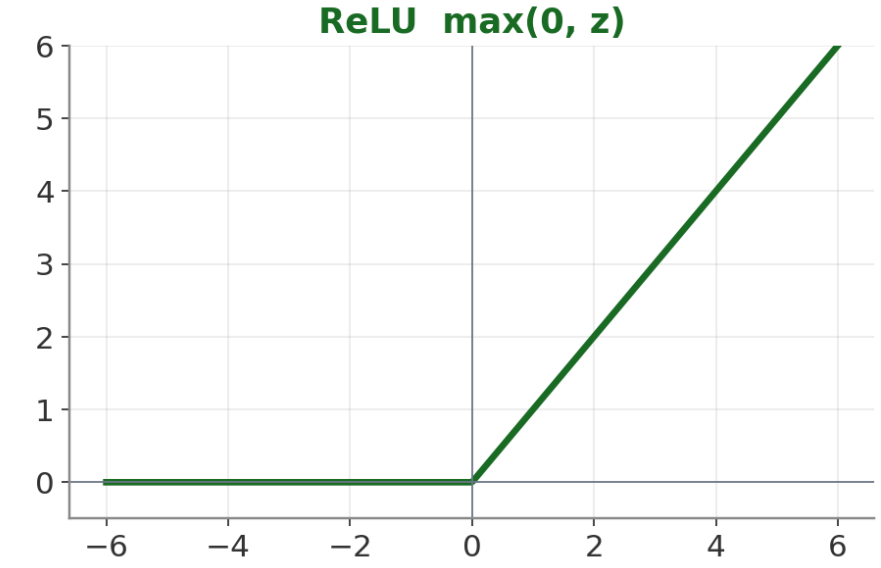
It is these **non-linear** functions that make neural networks so flexible!

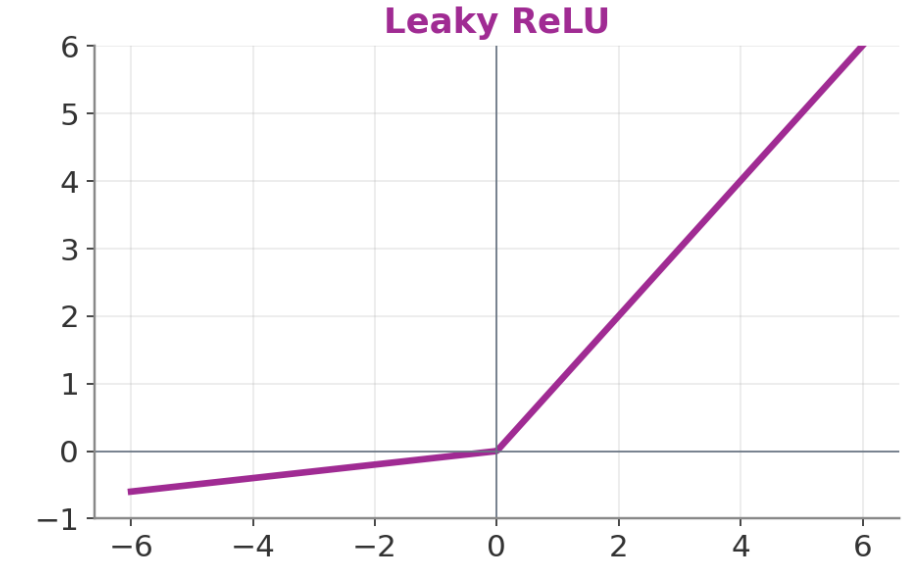
Common activation functions





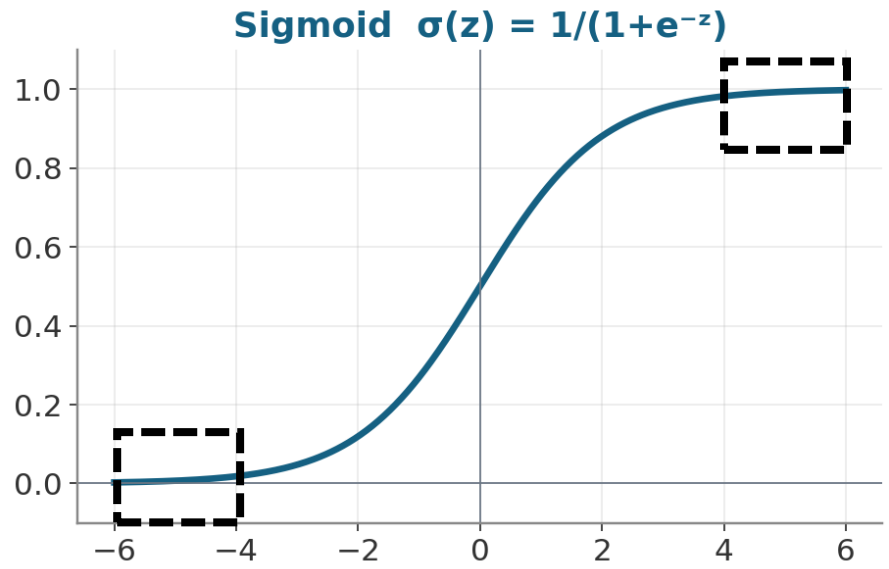
In contrast to sigmoid, **tanh** output ranges **from −1 to 1**

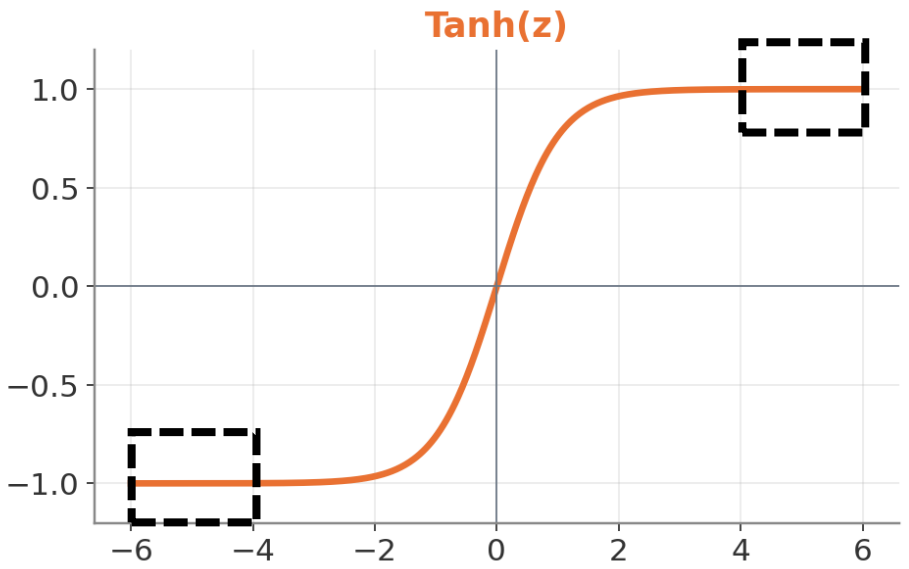


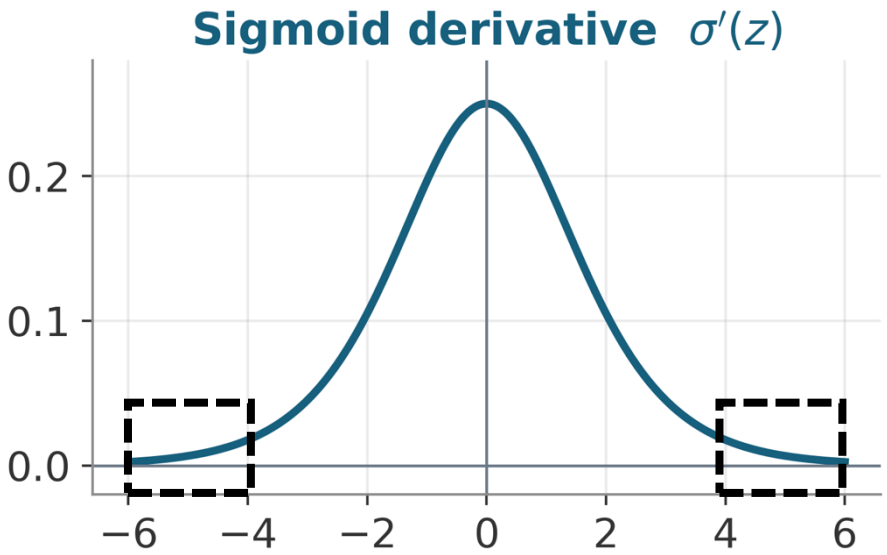


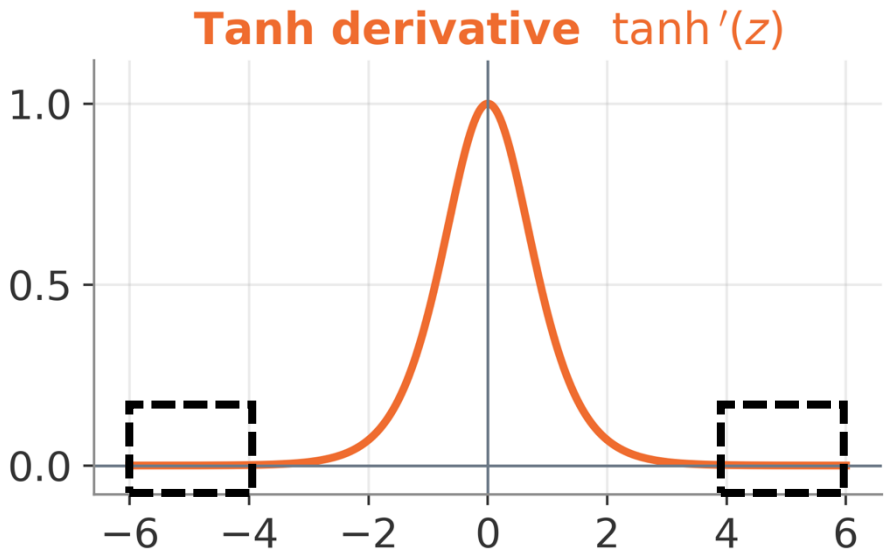
In contrast to ReLU, leaky ReLU allows a small, nonzero output for negative inputs

The “vanishing gradient” problem





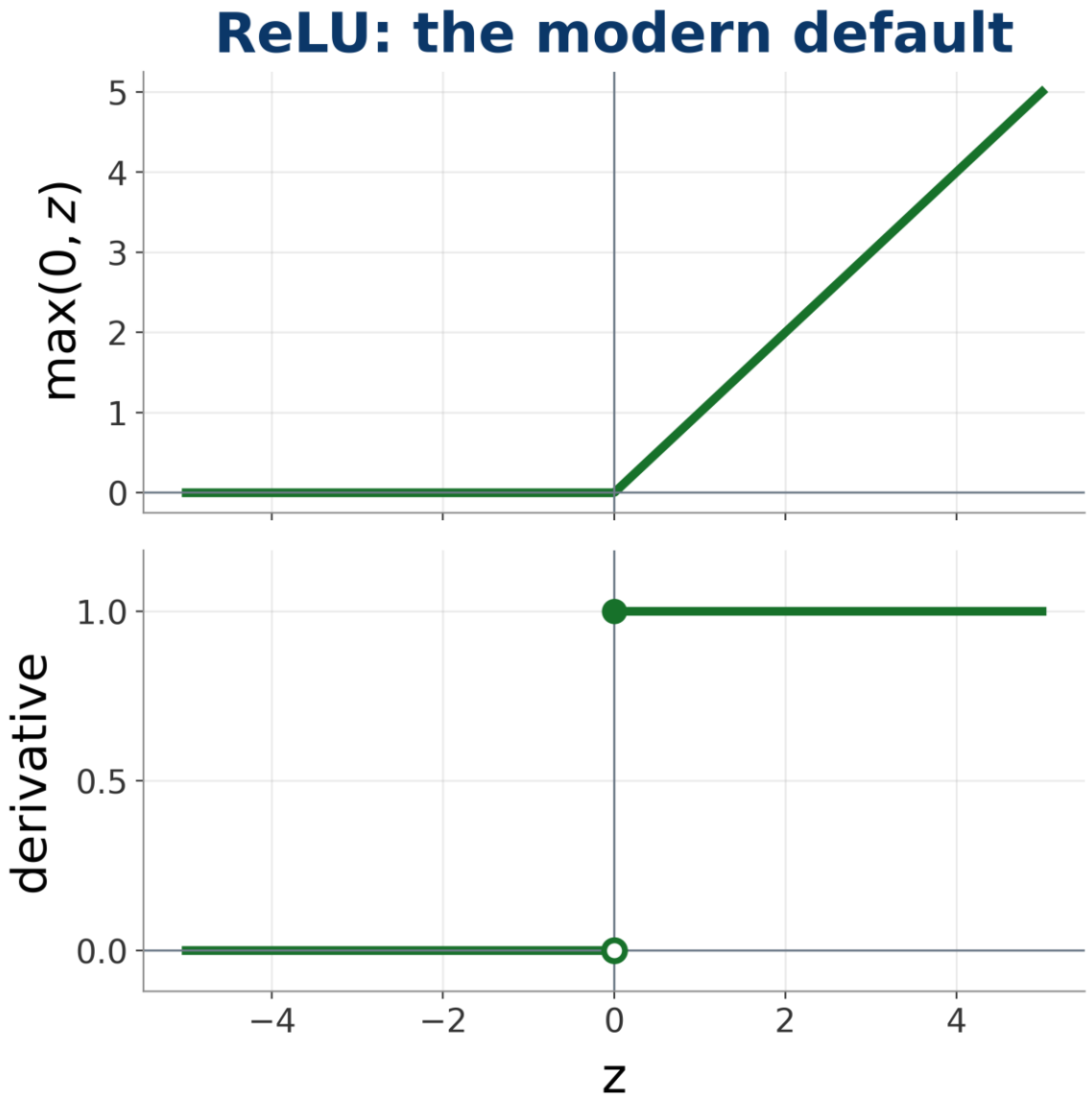




For very large positive or negative inputs, sigmoid and tanh become nearly flat. Their derivatives (which we will use for optimization) are then close to zero.

We’ll come back to this next lecture…

ReLU avoids the vanishing gradient problem



Rectified Linear Unit (ReLU):

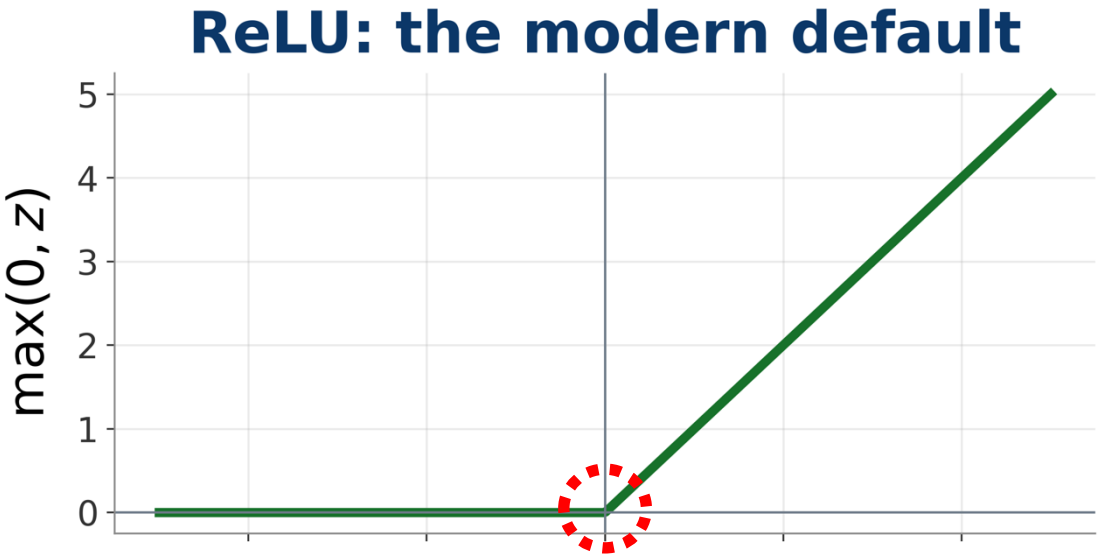
$$\mathbf{ReLU} = \mathbf{max}(\mathbf{0}, \boldsymbol{z}) = \begin{cases} \boldsymbol{z} \textbf{ if } \boldsymbol{z} \geq \mathbf{0} \\ \mathbf{0} \textbf{ if } \boldsymbol{z} < \mathbf{0} \end{cases}$$

- This is a simple, widely used default for hidden layers
- The derivative is easy to compute:

$$\frac{d}{dz} ReLU(z) = \begin{cases} \mathbf{1} \textbf{ if } \boldsymbol{z} > \mathbf{0} \\ \mathbf{0} \textbf{ if } \boldsymbol{z} < \mathbf{0} \end{cases}$$

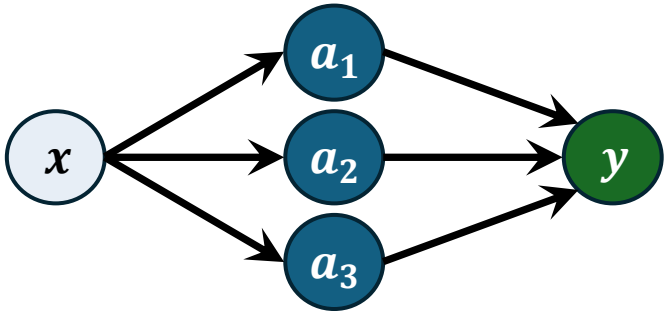
- No vanishing gradient at $\boldsymbol{z} > \mathbf{0}$
- $\boldsymbol{z} < \mathbf{0}$ creates an “inactive” neuron

ReLU → piecewise linear decision boundaries

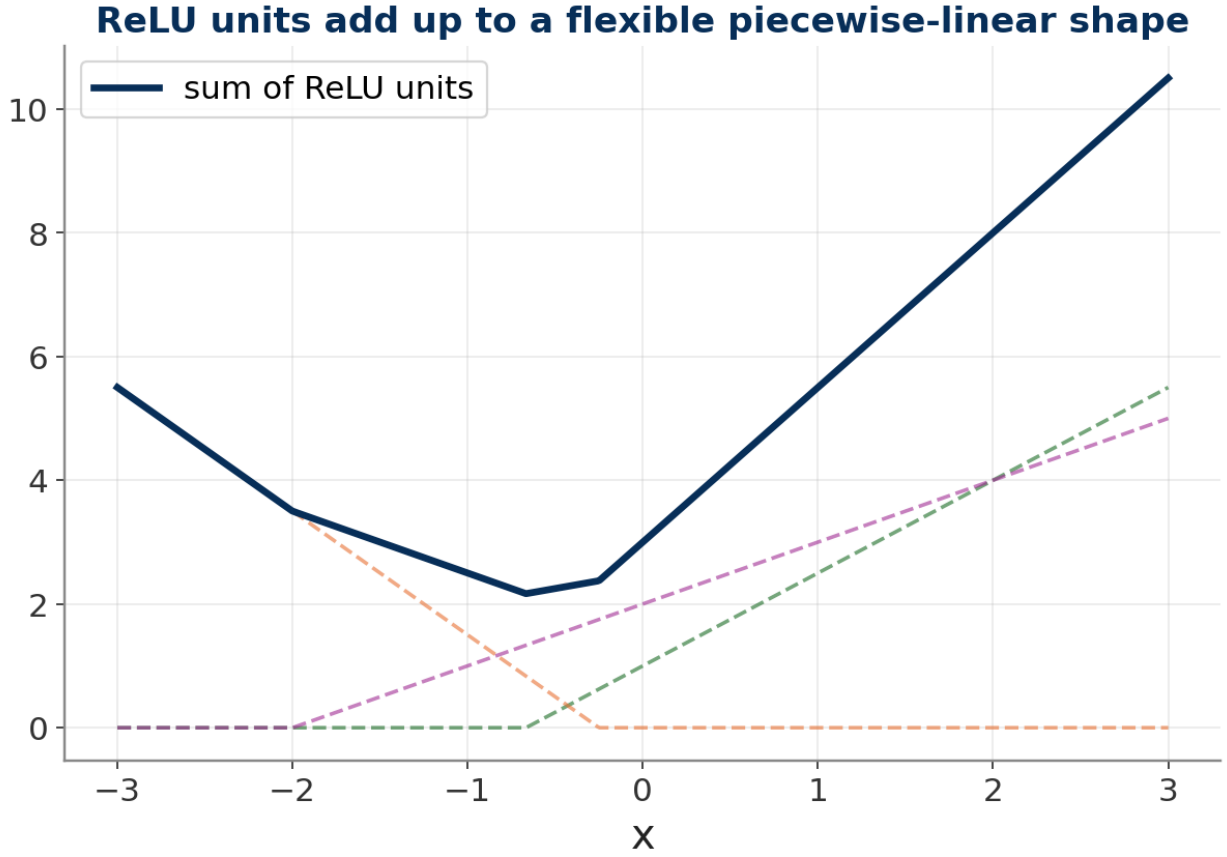


It is this one “kink” that allows us to learn complex, **non-linear*** decision boundaries!

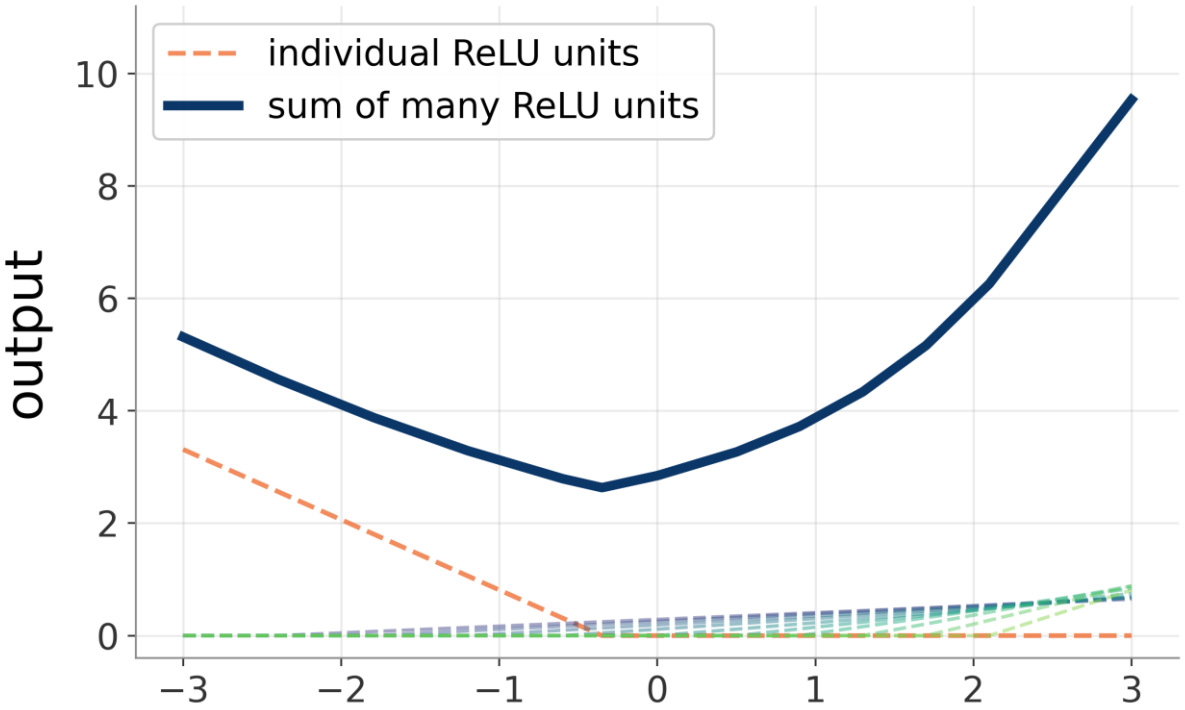
***Piecewise linear**

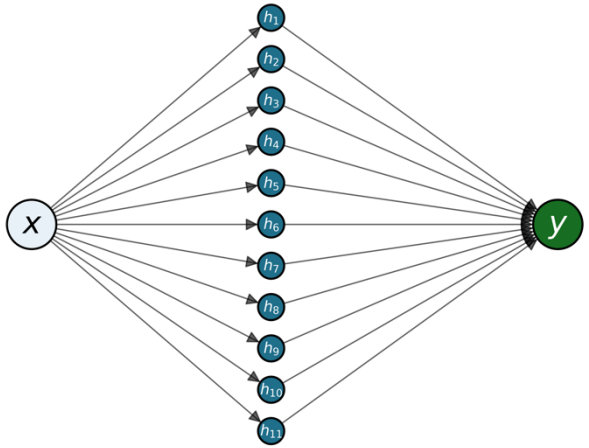


1 hidden layer
3 neurons
ReLU

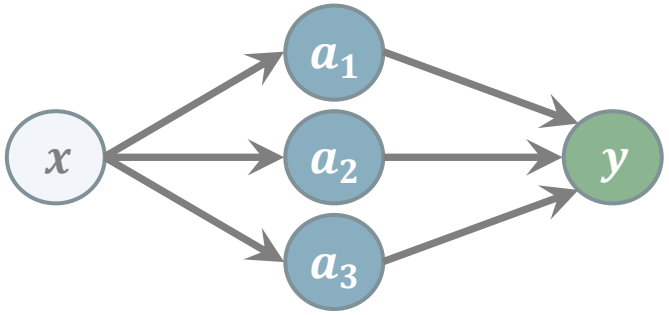


More hidden neurons ≈ smoother boundary



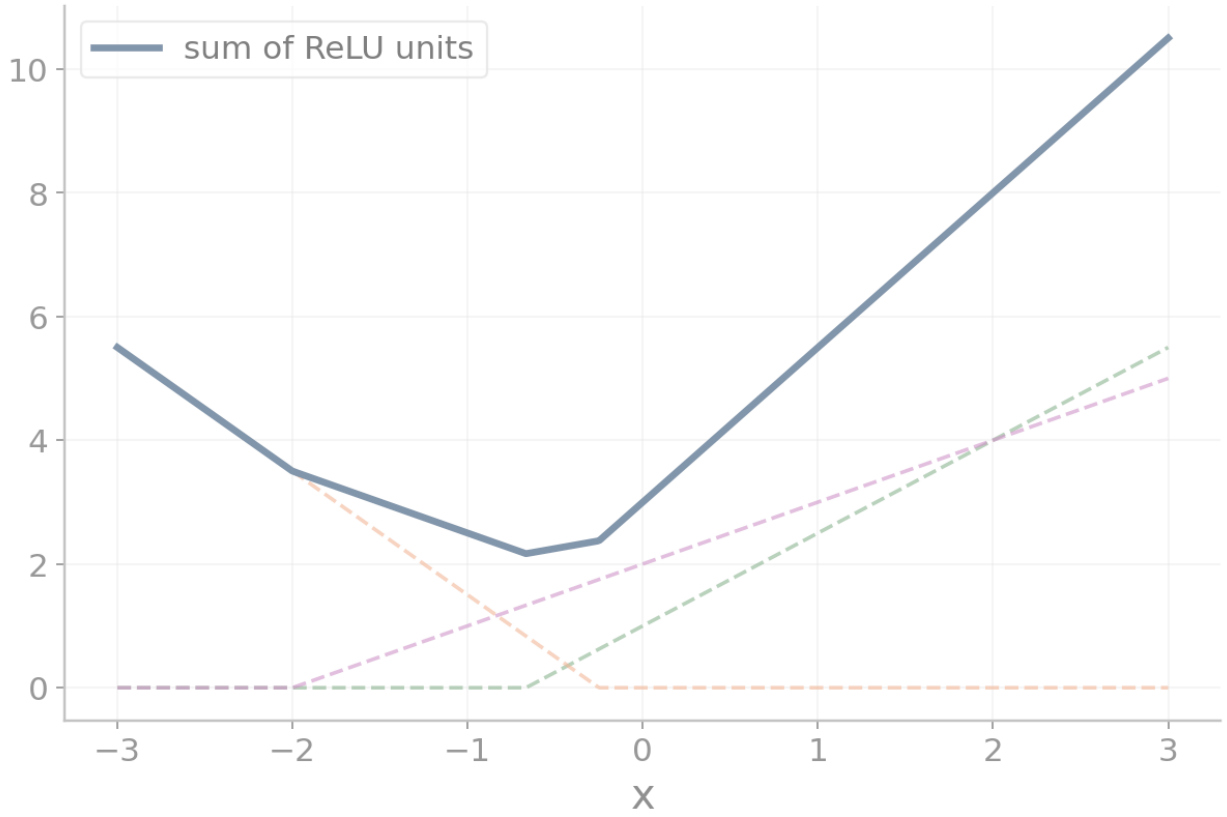


1 hidden layer
11 neurons
ReLU

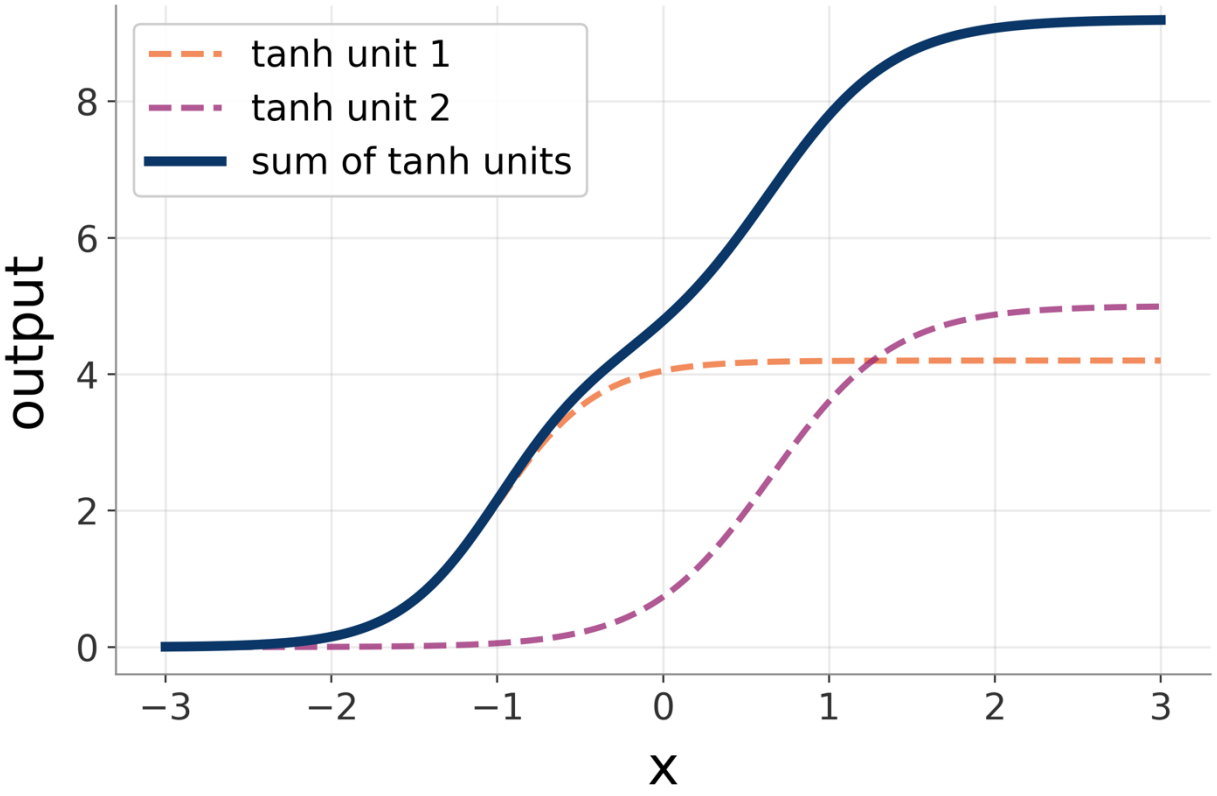


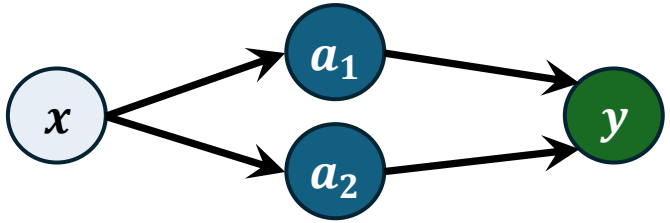
1 hidden layer
3 neurons
ReLU

ReLU units add up to a flexible piecewise-linear shape

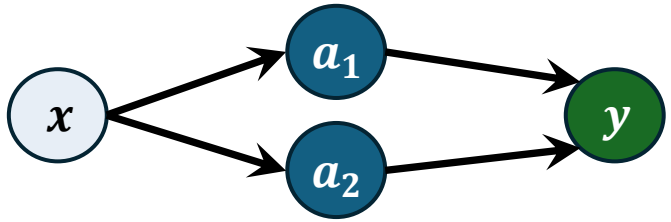


Sigmoid and tanh are naturally smooth

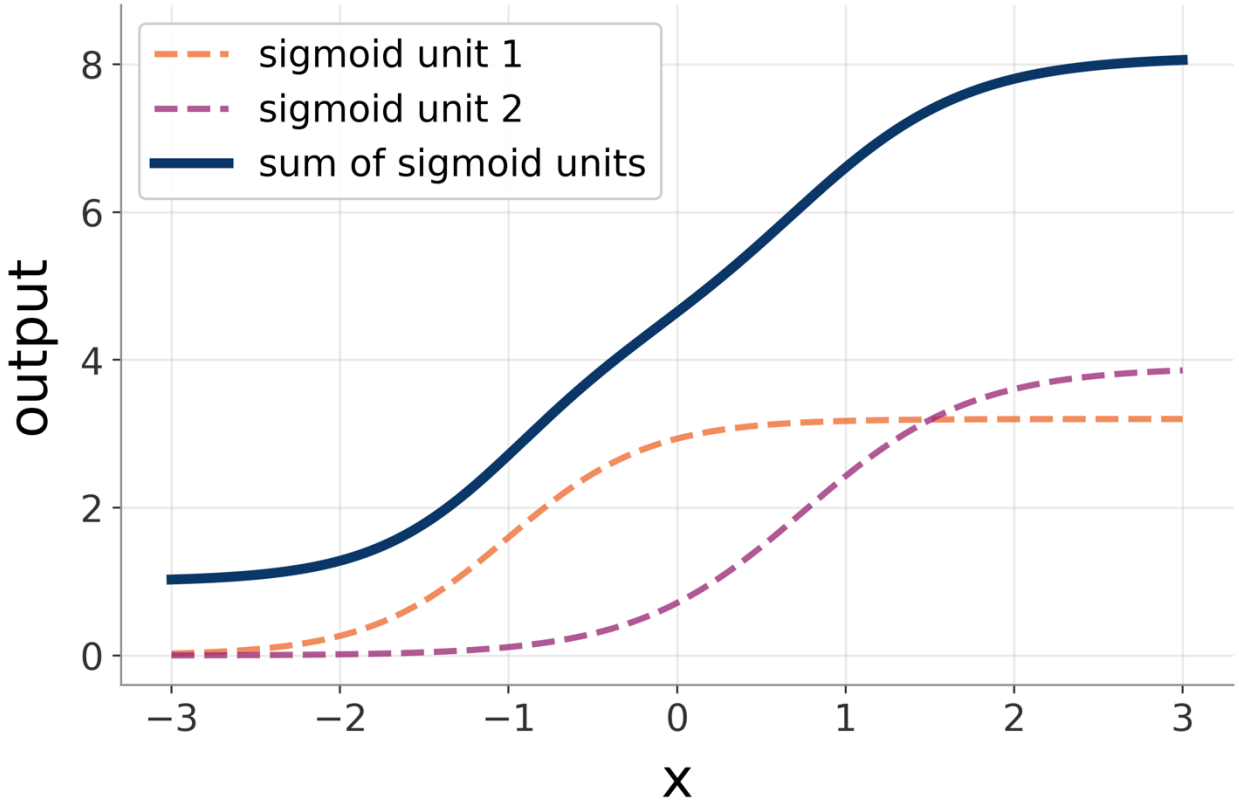




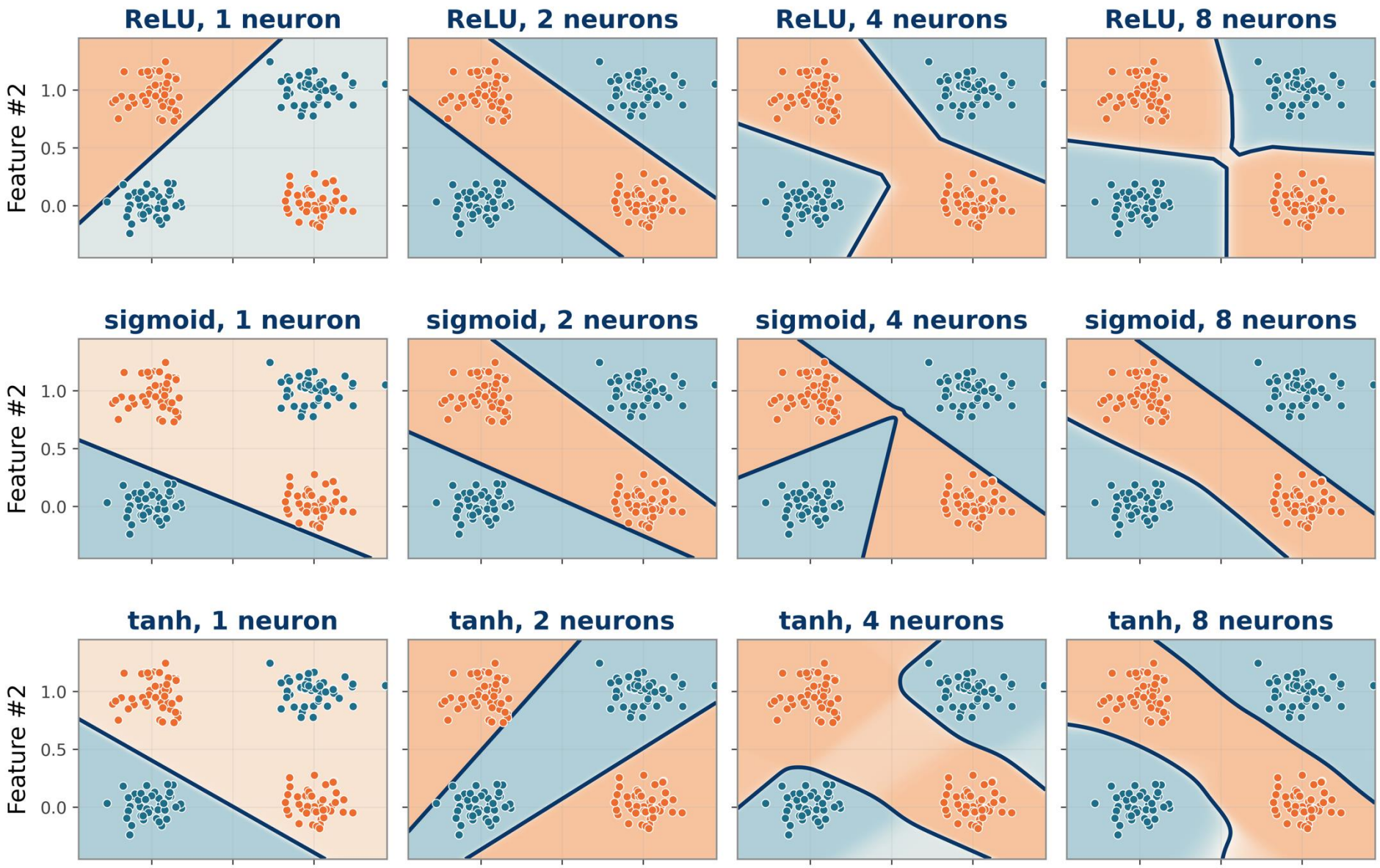
1 hidden layer
2 neurons
Tanh



1 hidden layer
2 neurons
Sigmoid



Effects of neurons and activation functions

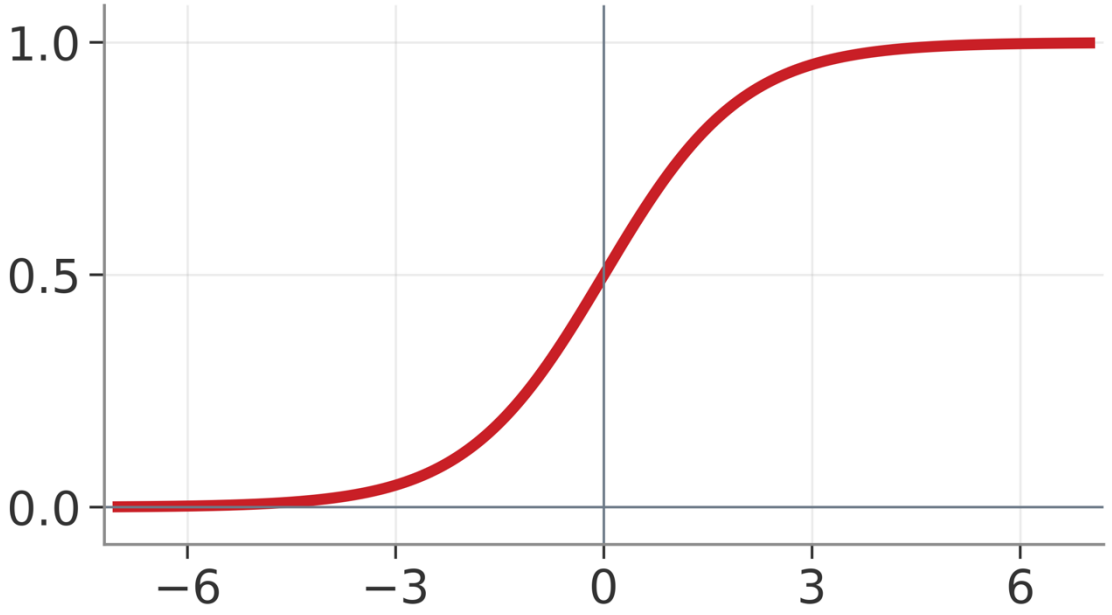


Here we change which activation function is applied to the hidden layer, while **sigmoid is always used on the final output**

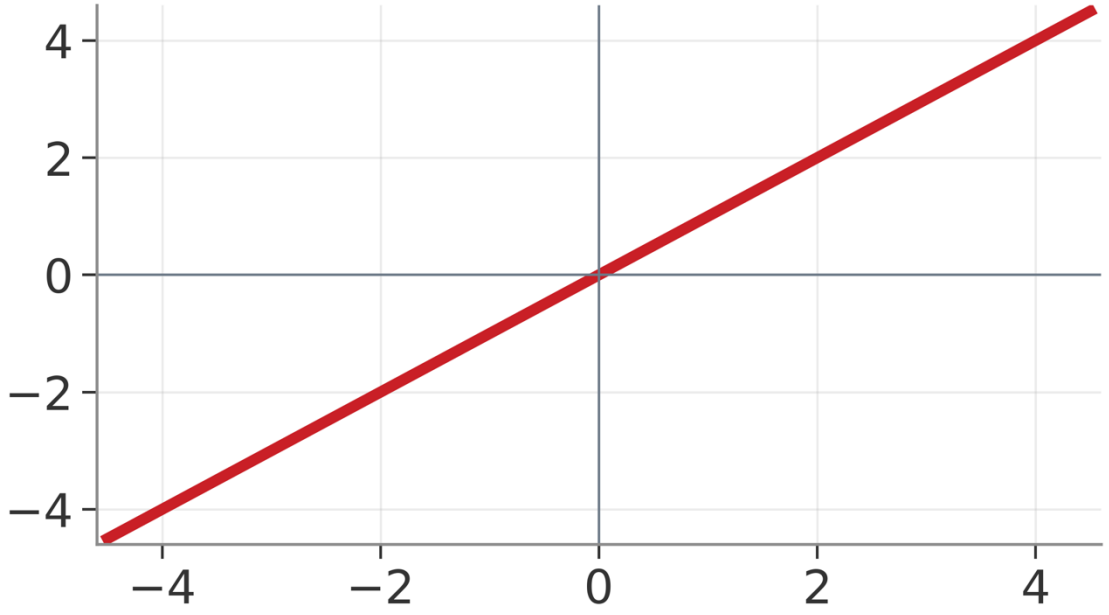
Sigmoid ranges from 0 to 1, which is natural for *class probability*

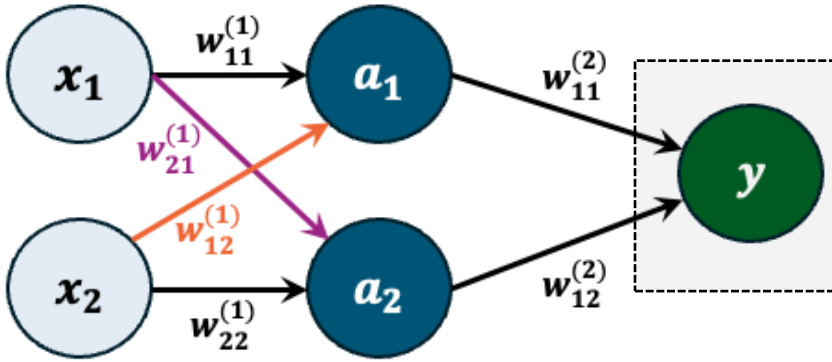
Activation *on the output*

Binary classification
sigmoid output



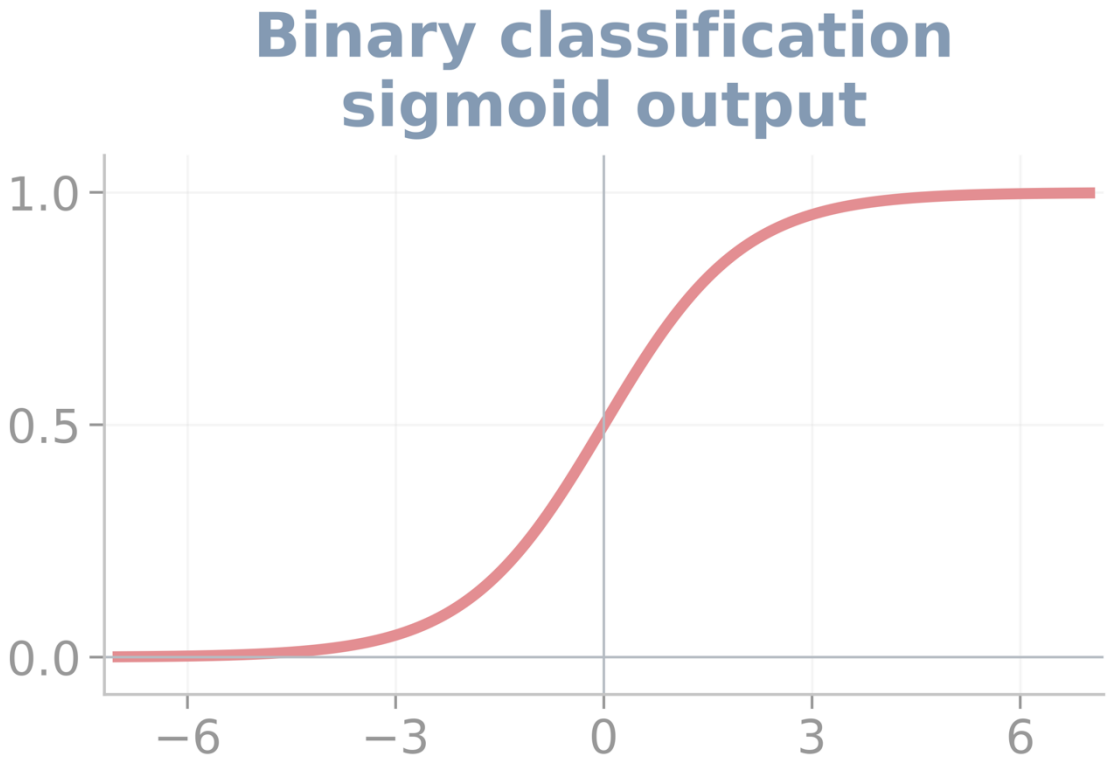
Regression
linear output

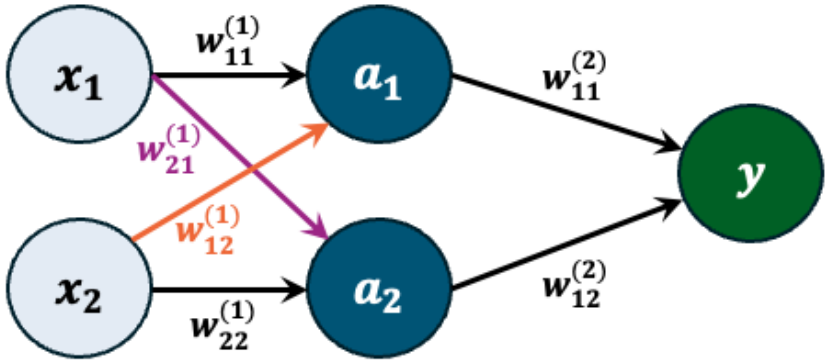


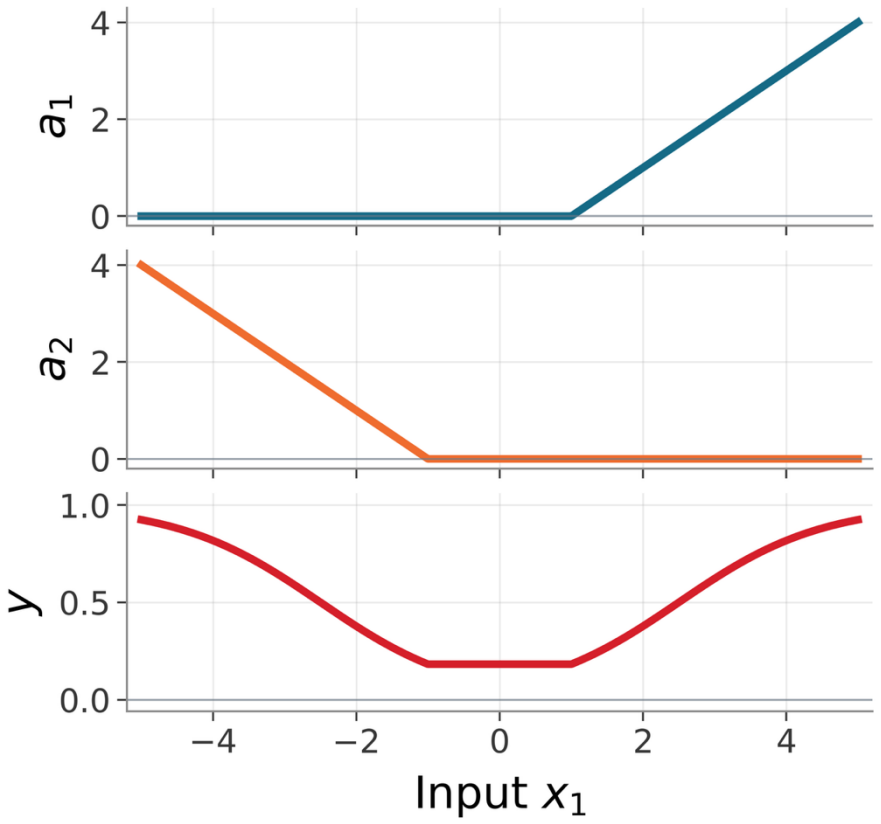


$$y = \text{activation}(w_{11}^{(2)} a_1 + w_{12}^{(2)} a_2 + b)$$

Activation *on the output* for binary classification







1st hidden neuron

2nd hidden neuron

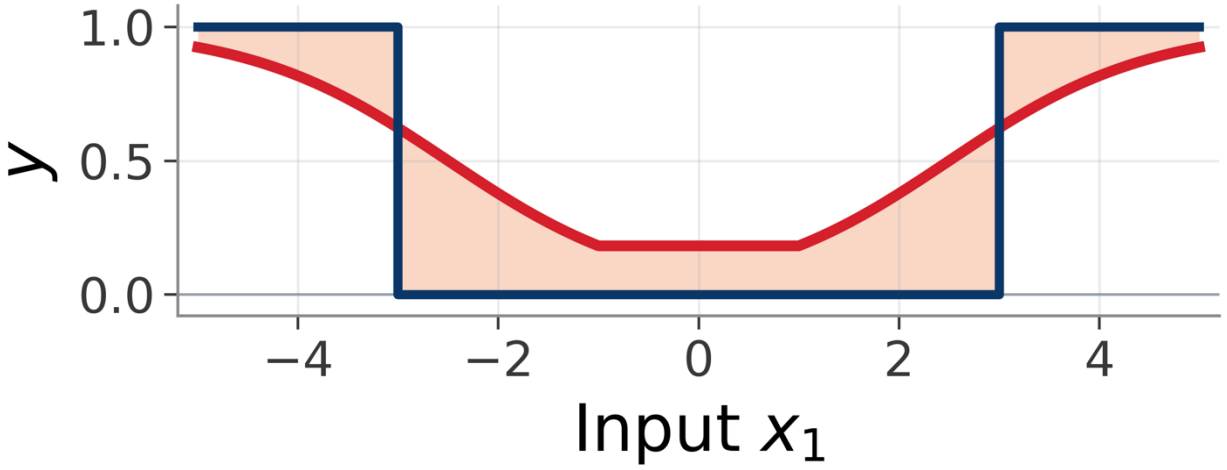
Activated output

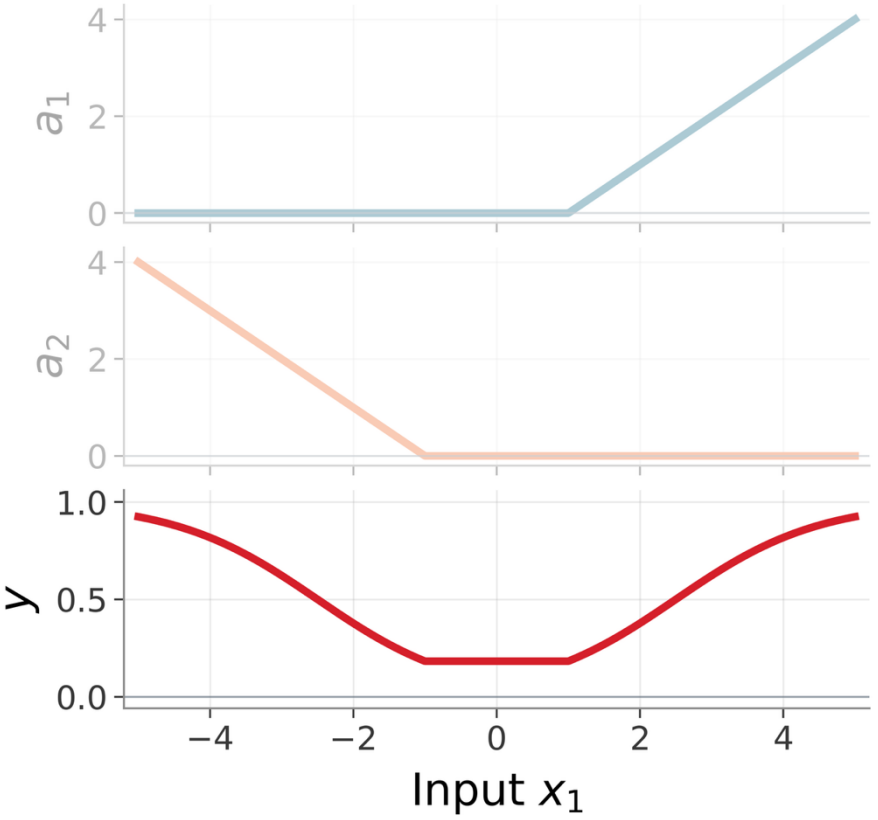
$$y = \text{sigmoid}(w_{11}^{(2)} a_1 + w_{12}^{(2)} a_2 + b)$$

Bounded from 0 to 1

This can be anything! It's *not* bounded

Loss for binary classification





1st hidden neuron

2nd hidden neuron

Activated output

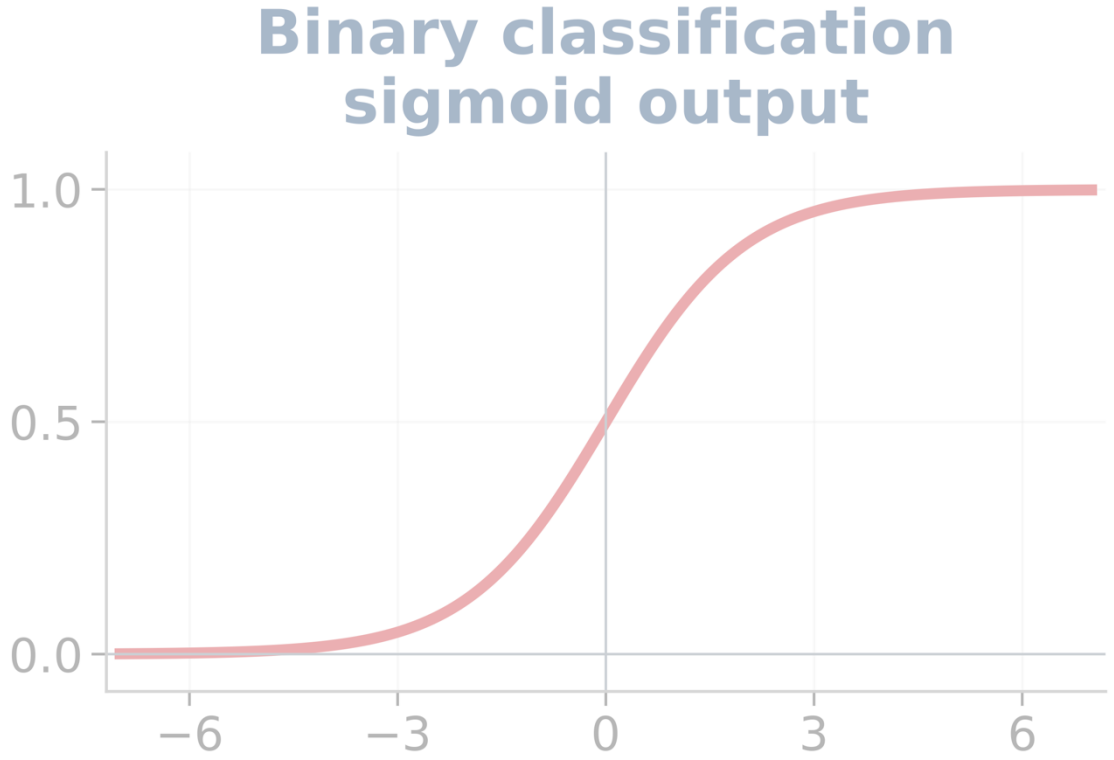
Choose weights by minimizing the *binary cross-entropy*

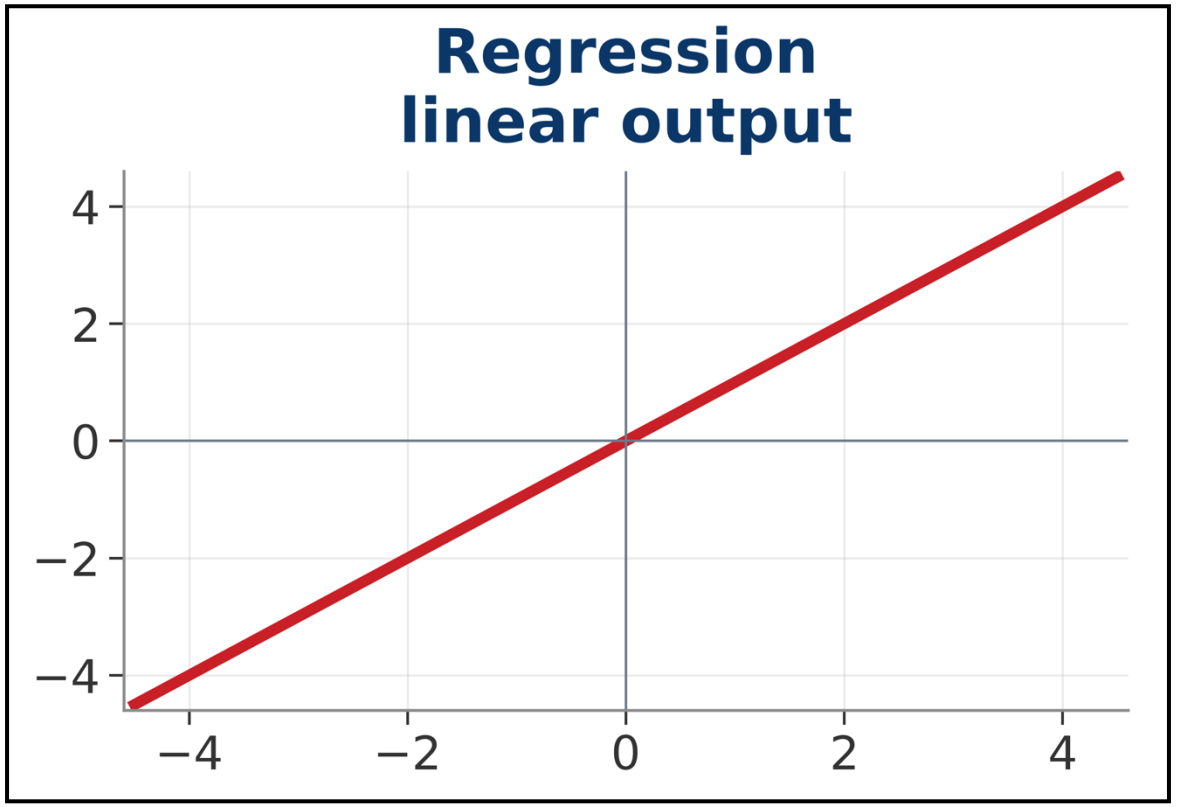
$$-\frac{1}{n}\sum_{i=1}^{n} y_i \log(p_i) + (1-y_i)\log(1-p_i)$$

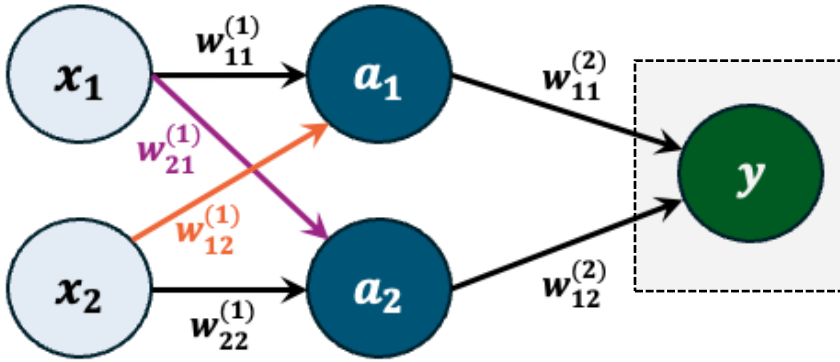
$$y = \text{sigmoid}(w_{11}^{(2)} a_1 + w_{12}^{(2)} a_2 + b)$$

Training is a matter of learning weights and bias

Activation *on the output*







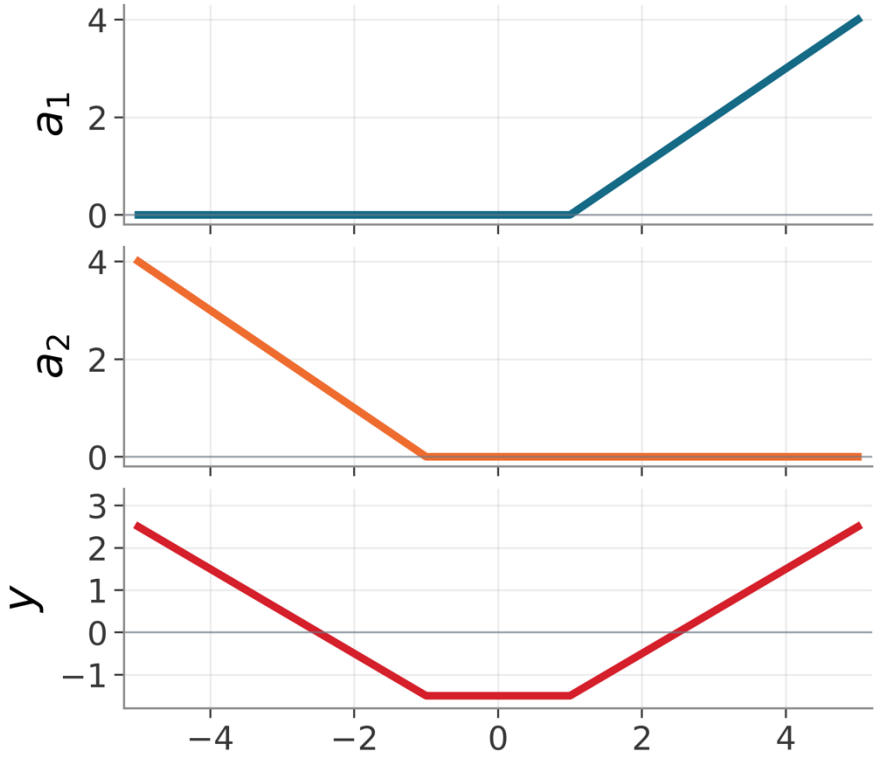
$$y = \text{activation}(w_{11}^{(2)} a_1 + w_{12}^{(2)} a_2 + b)$$

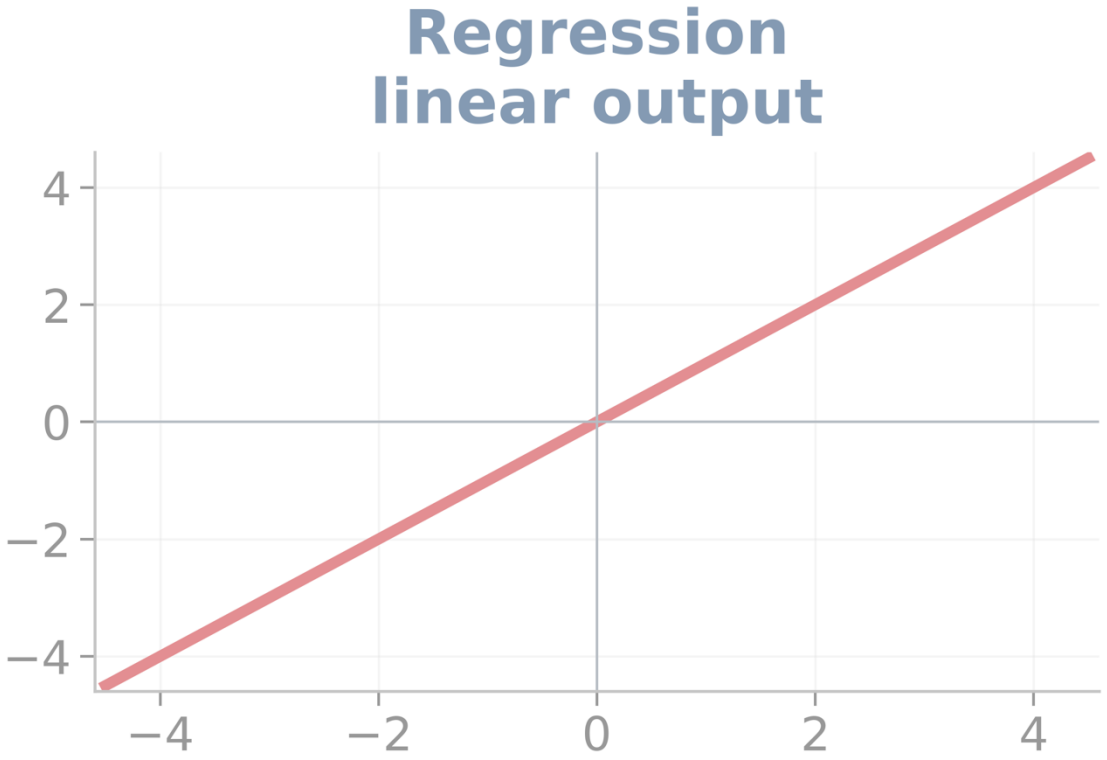
Activation *on the output* for regression

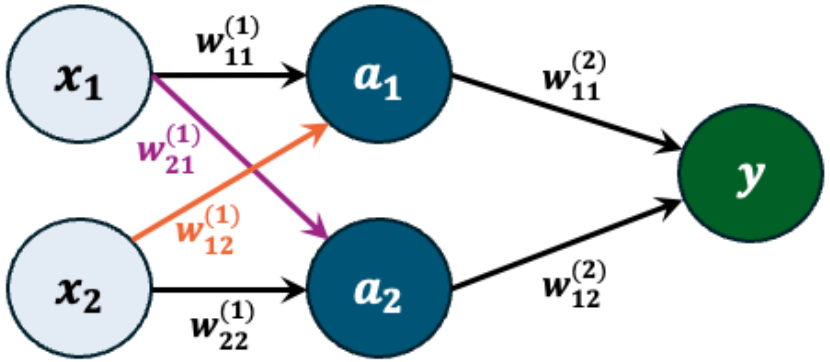
1st hidden neuron

2nd hidden neuron

Activated output







$$y = \mathbf{1}(w_{11}^{(2)} a_1 + w_{12}^{(2)} a_2 + b)$$

This can be anything!

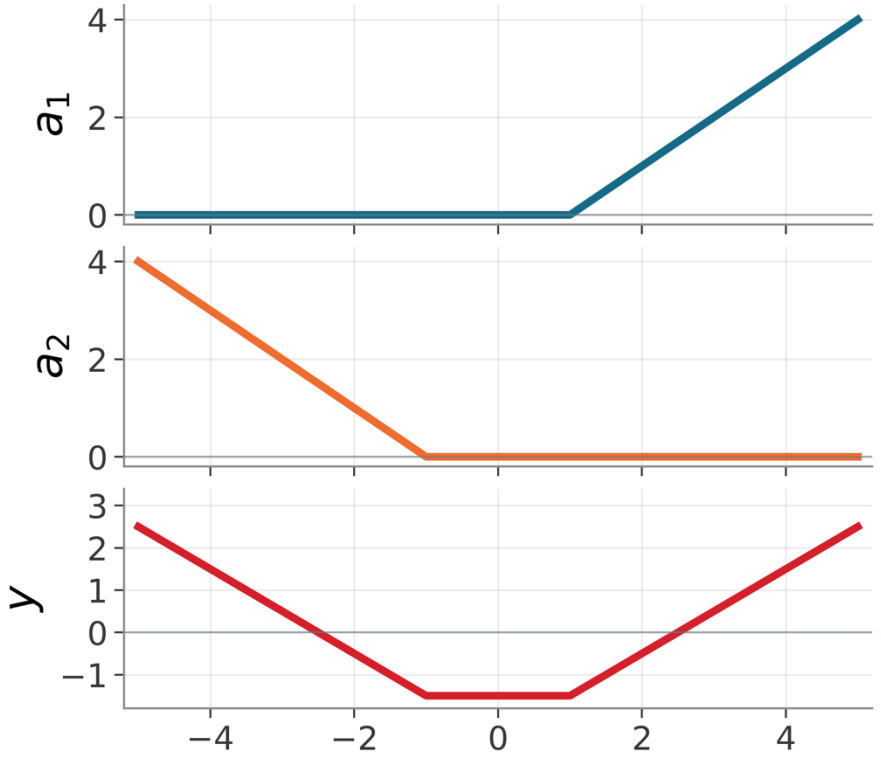
It's *not* bounded

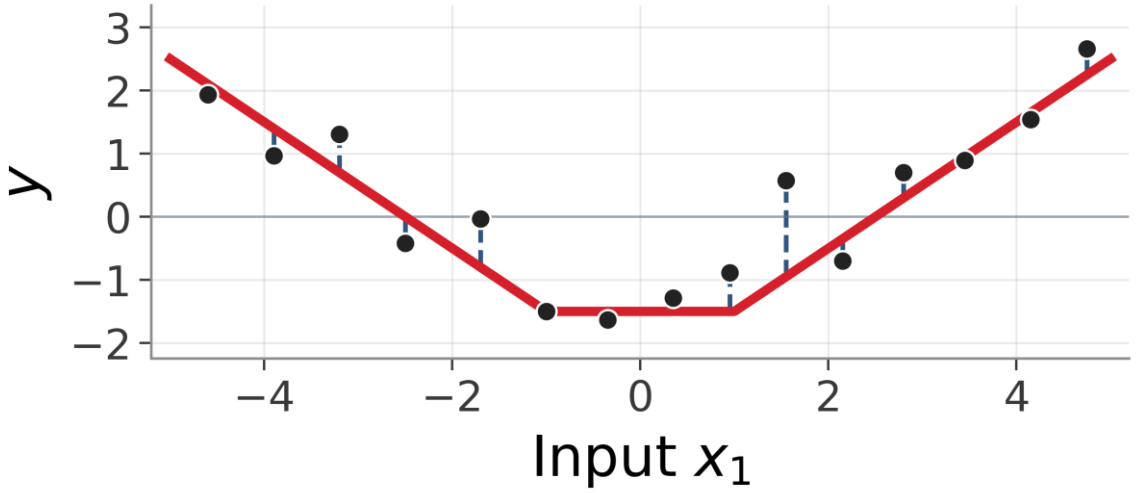
Loss for regression

1st hidden neuron

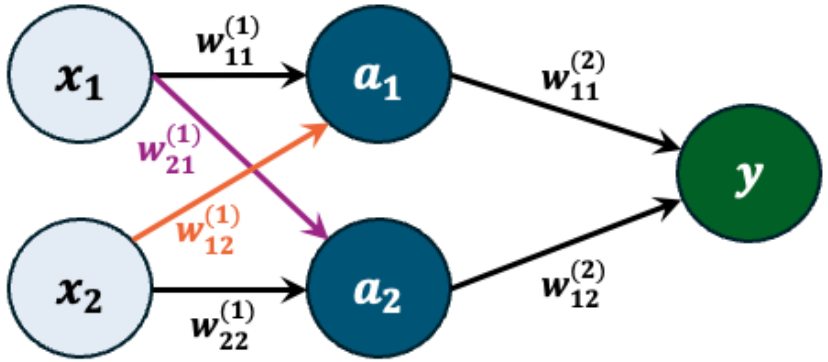
2nd hidden neuron

Activated output





Choose weights by minimizing the *Mean Squared Error* (MSE)

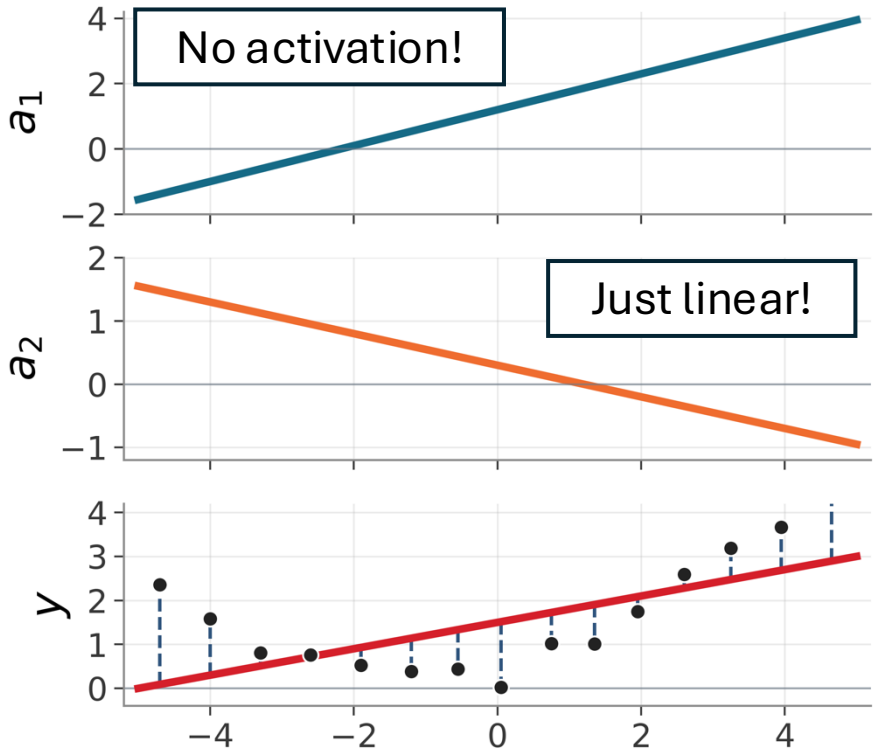


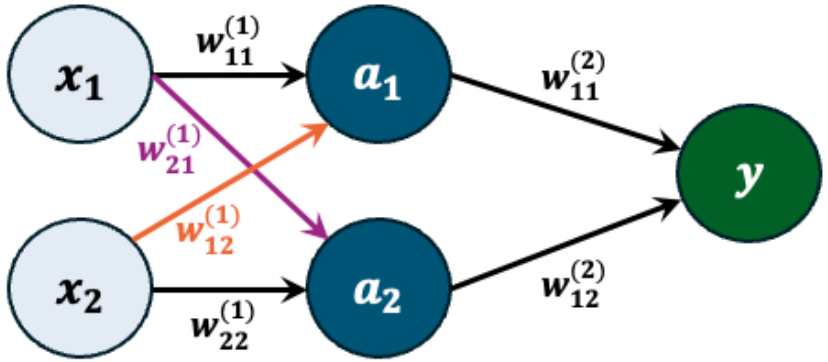
$$y = 1(w_{11}^{(2)} a_1 + w_{12}^{(2)} a_2 + b)$$

Training is a matter of learning weights and bias

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Without non-linear activation: Least-squares regression!





One hidden layer with two neurons

$$y = w_{11}^{(2)} a_1 + w_{12}^{(2)} a_2 + b^{(2)}$$

$$w_{11}^{(2)}\left(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + b_1^{(1)}\right) + w_{12}^{(2)}\left(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + b_2^{(1)}\right) + b^{(2)}$$

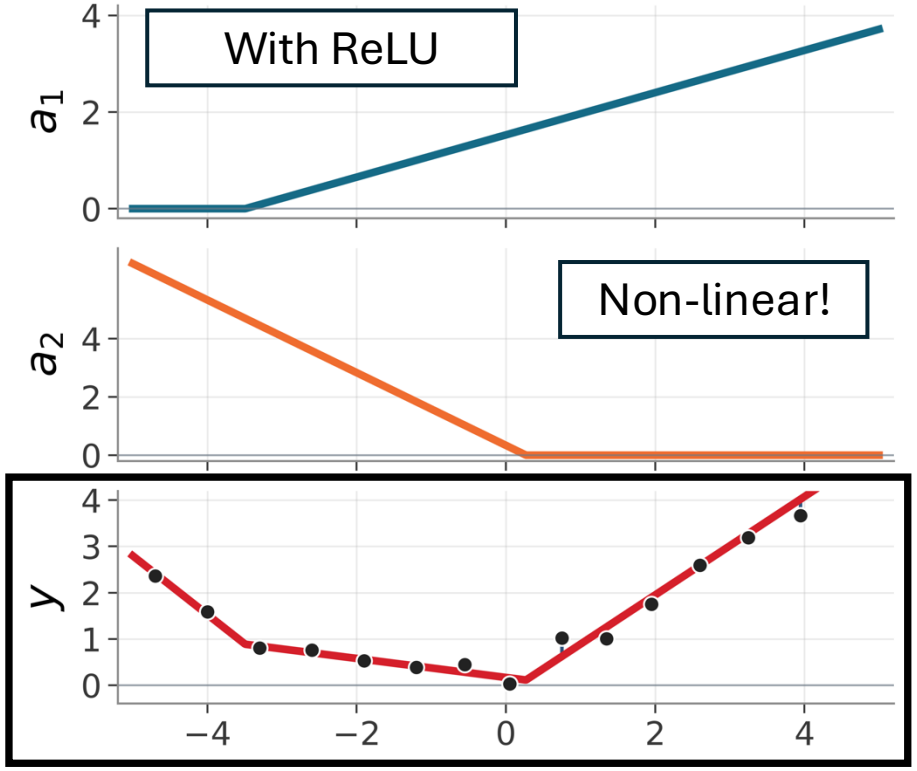
$$\left(w_{11}^{(2)}w_{11}^{(1)} + w_{12}^{(2)}w_{21}^{(1)}\right)x_1 + \left(w_{11}^{(2)}w_{12}^{(1)} + w_{12}^{(2)}w_{22}^{(1)}\right)x_2 + \left(b_1^{(1)} + b_2^{(1)} + b^{(2)}\right)$$

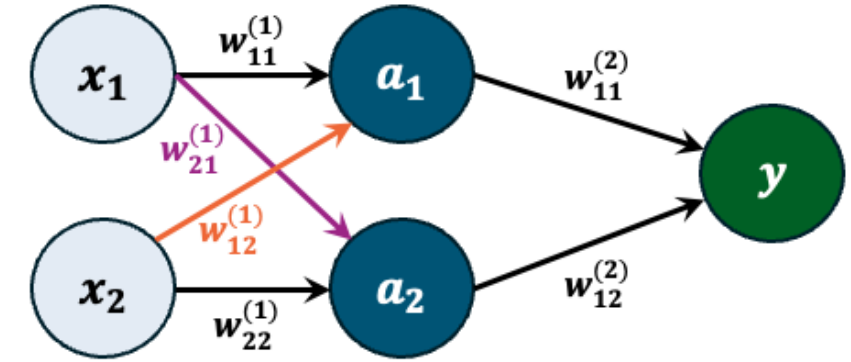
$$y = \beta_1 x_1 + \beta_2 x_2 + \beta_0$$

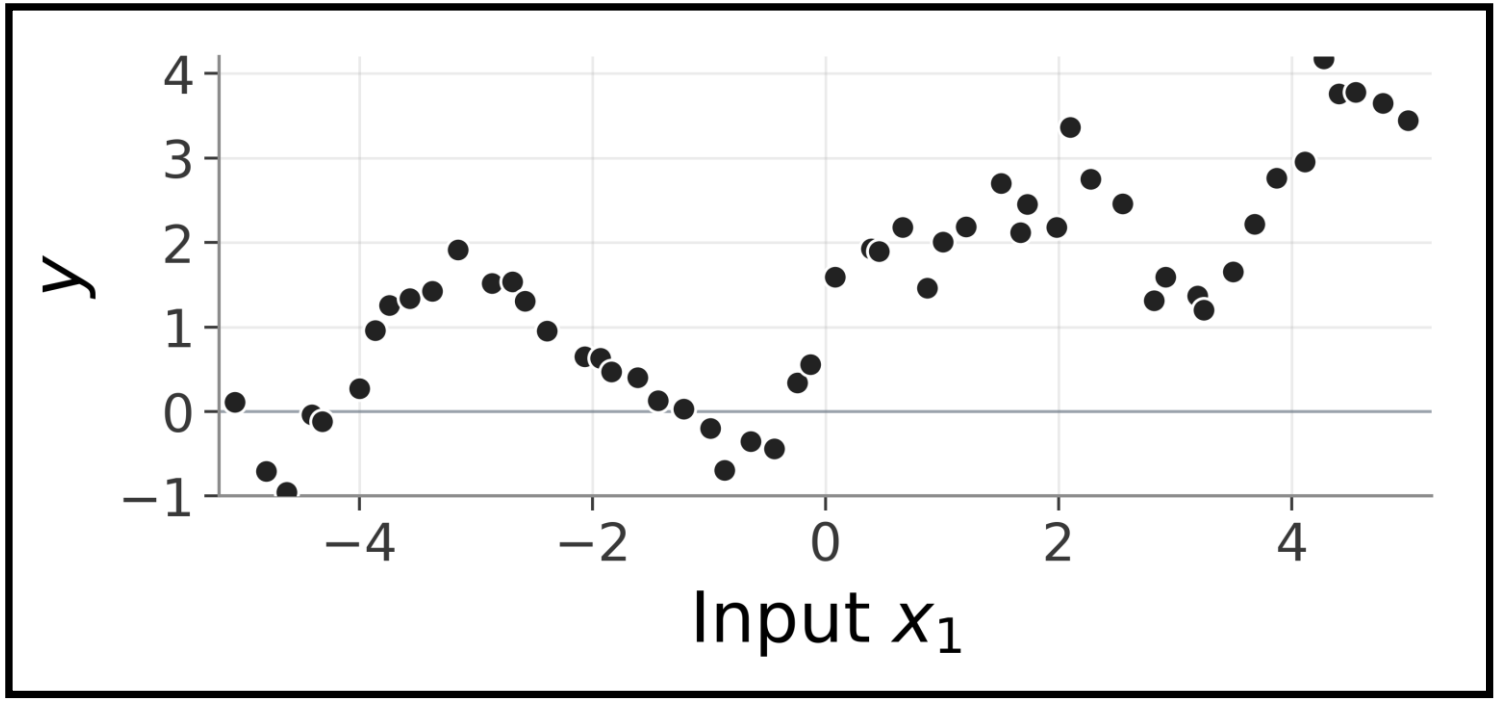
If we choose weights in this neural network by minimizing MSE, it brings us right back to least-squares linear regression!

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

With non-linear activation: Complex functions

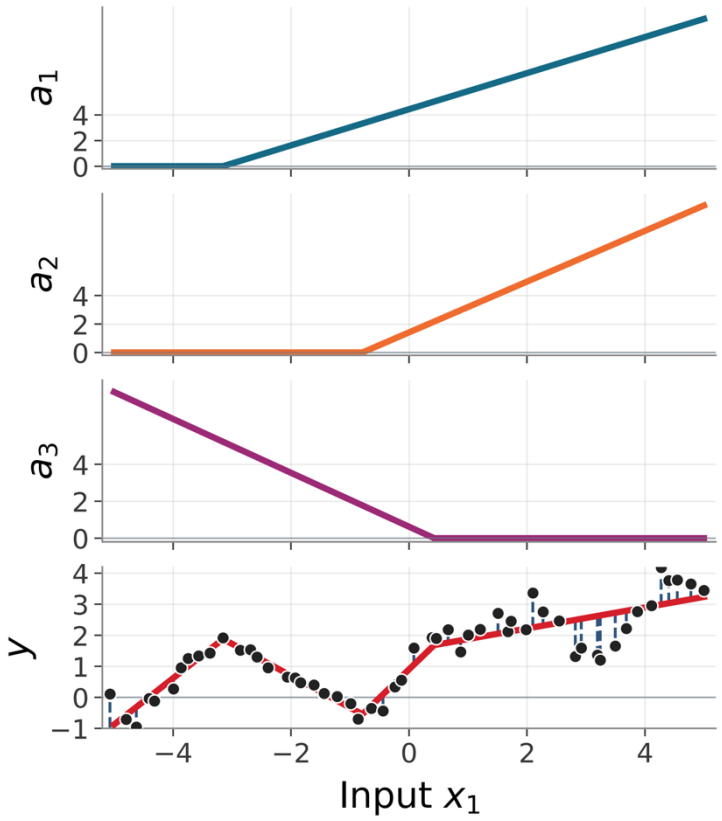


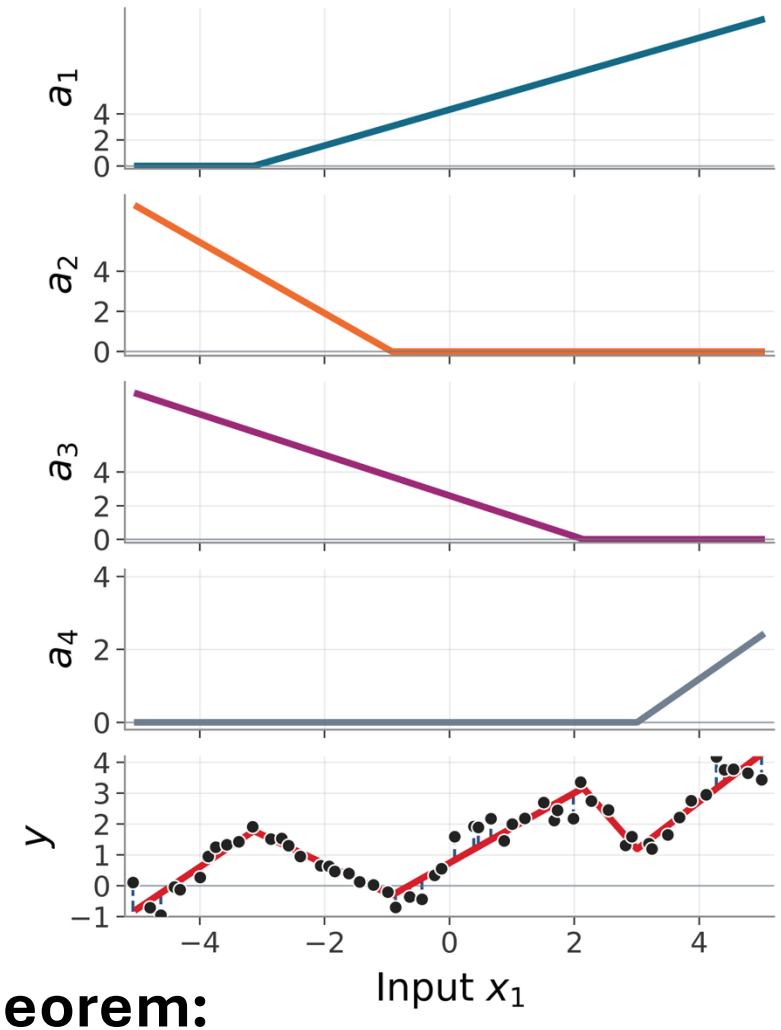


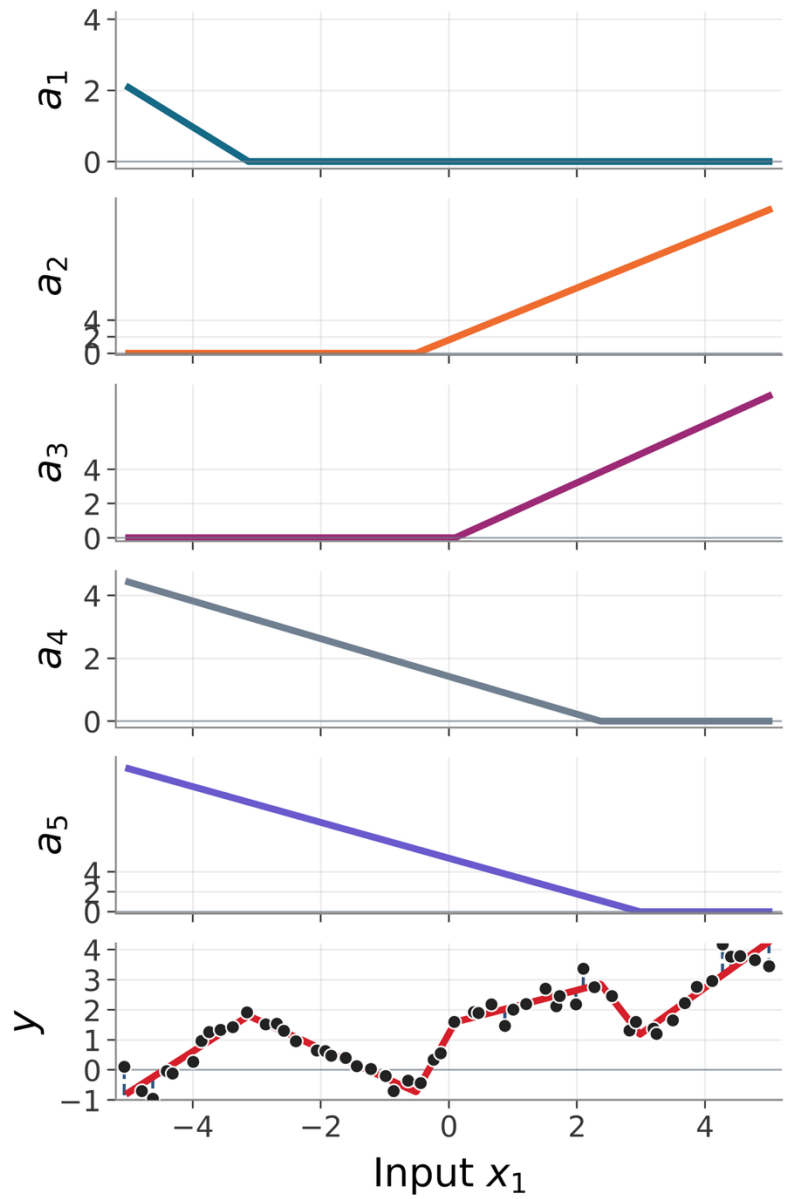


How "crazy" of a function can we model with neural networks?

Enough hidden units can fit *any* curve



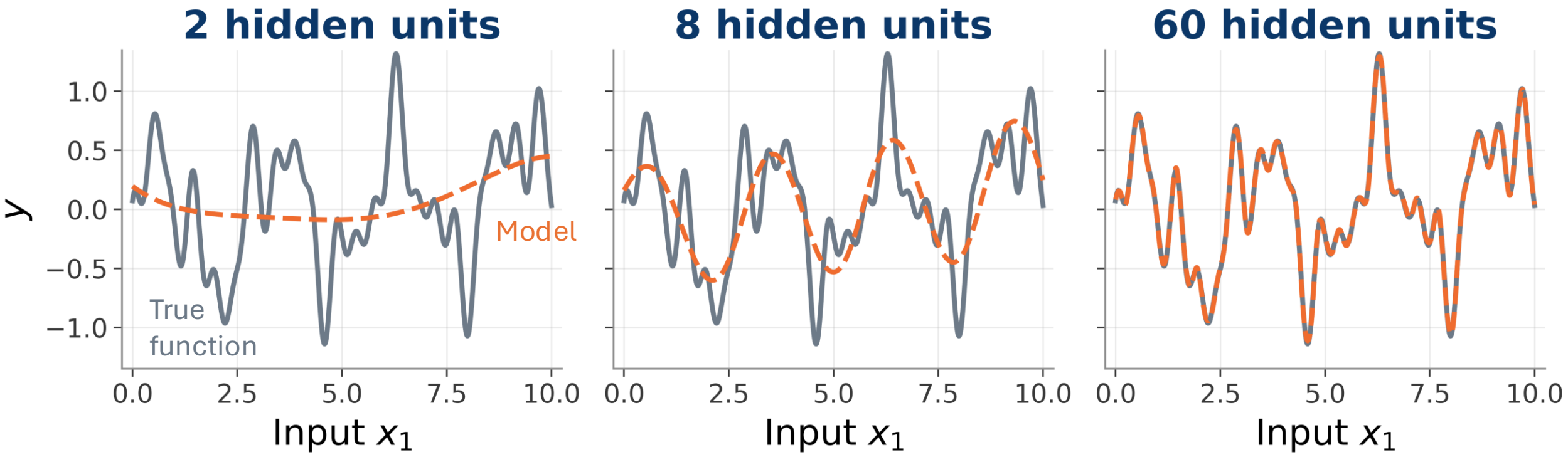




Universal approximation theorem:
A neural network can approximate any continuous function if it has enough neurons.

Universal approximation holds for any non-linear activation

The model in the below example used **tanh activation**



More neurons in the hidden layer → more flexibility in the function we can learn

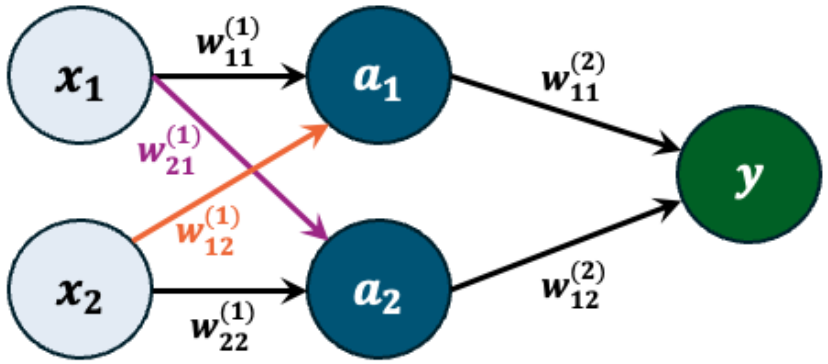
But as always, we must consider the **risk of overfitting**

Questions?

Network design based on task

Task	Output activation	Output shape	Loss
Regression	Linear (identity)	1 real value	Squared error (MSE)
Binary classification	Sigmoid	1 probability	Binary cross-entropy

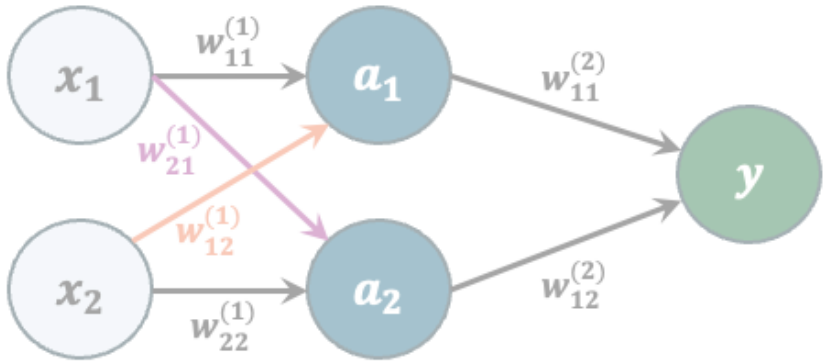
What about multiclass?

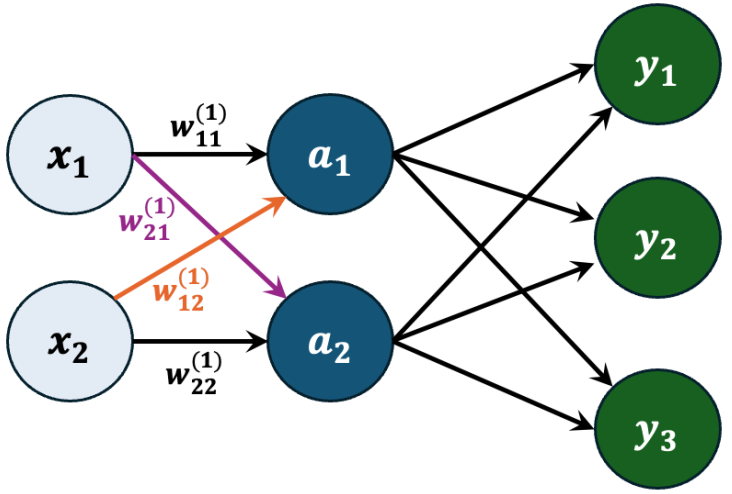


Just one output neuron

Network design based on task

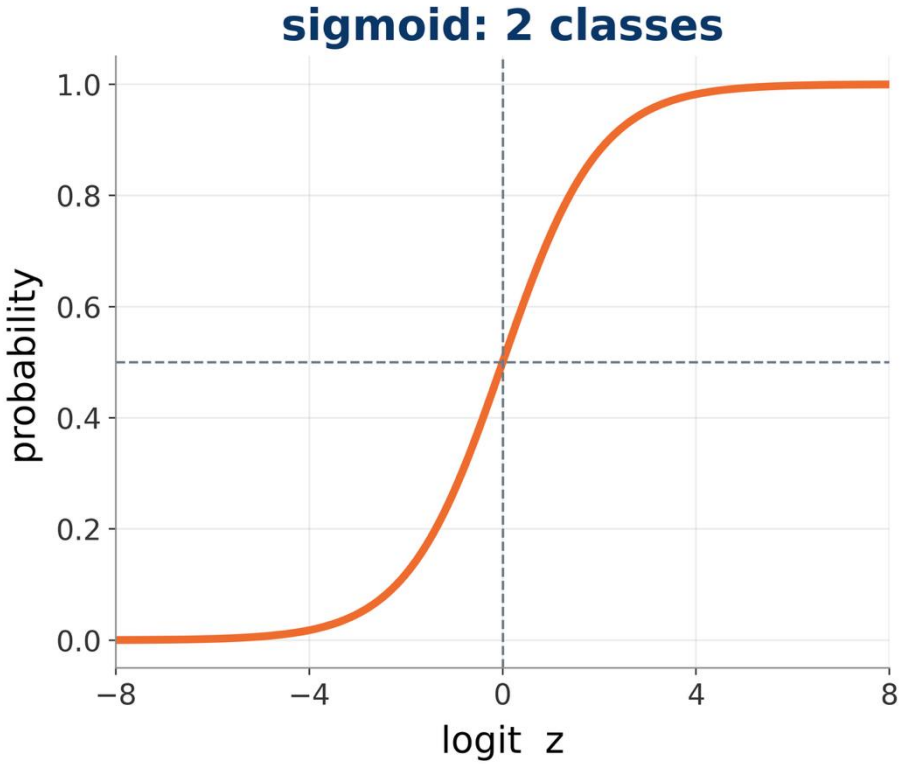
Task	Output activation	Output shape	Loss
Regression	Linear (identity)	1 real value	Squared error (MSE)
Binary classification	Sigmoid	1 probability	Binary cross-entropy
Multiclass (K)	Softmax	K probs, sum = 1	Cross-entropy

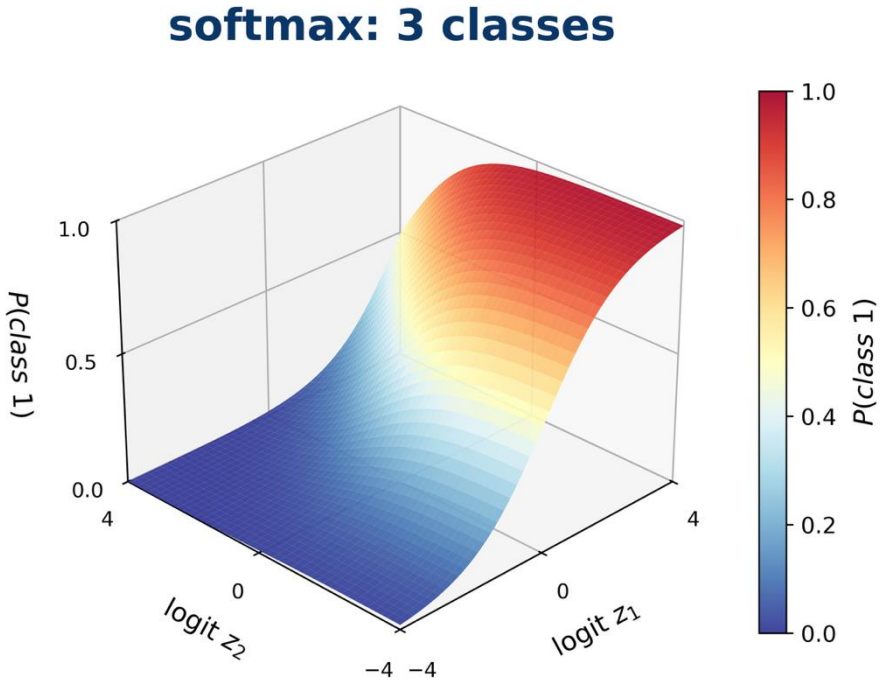


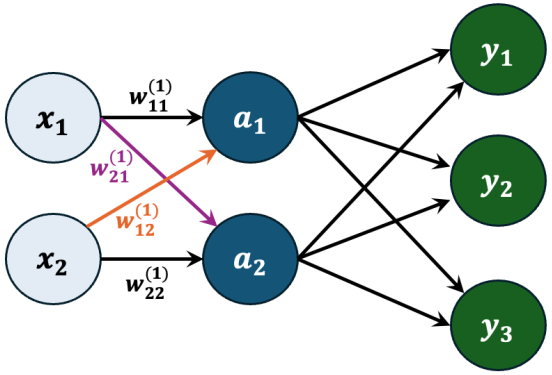


Each output neuron is a distinct class

Recall: The softmax function





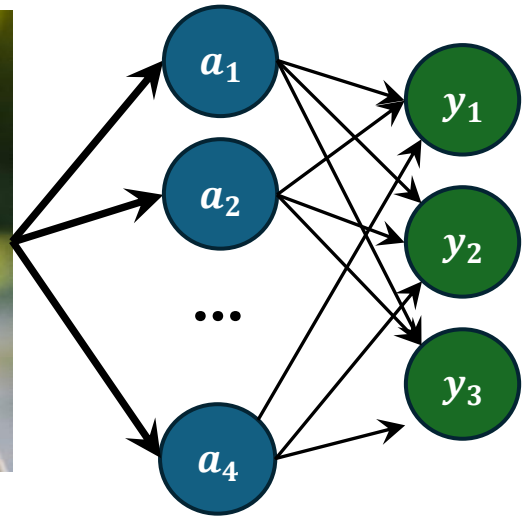


$$P(y = k) = \frac{e^{y_k}}{\sum_{i=1}^{K} e^{y_i}}$$

Softmax ensures probabilities sum to 1



In this case, each pixel is an input neuron



Cat – raw output 1.2

Dog – raw output 5.7

Fox – raw output 4.9

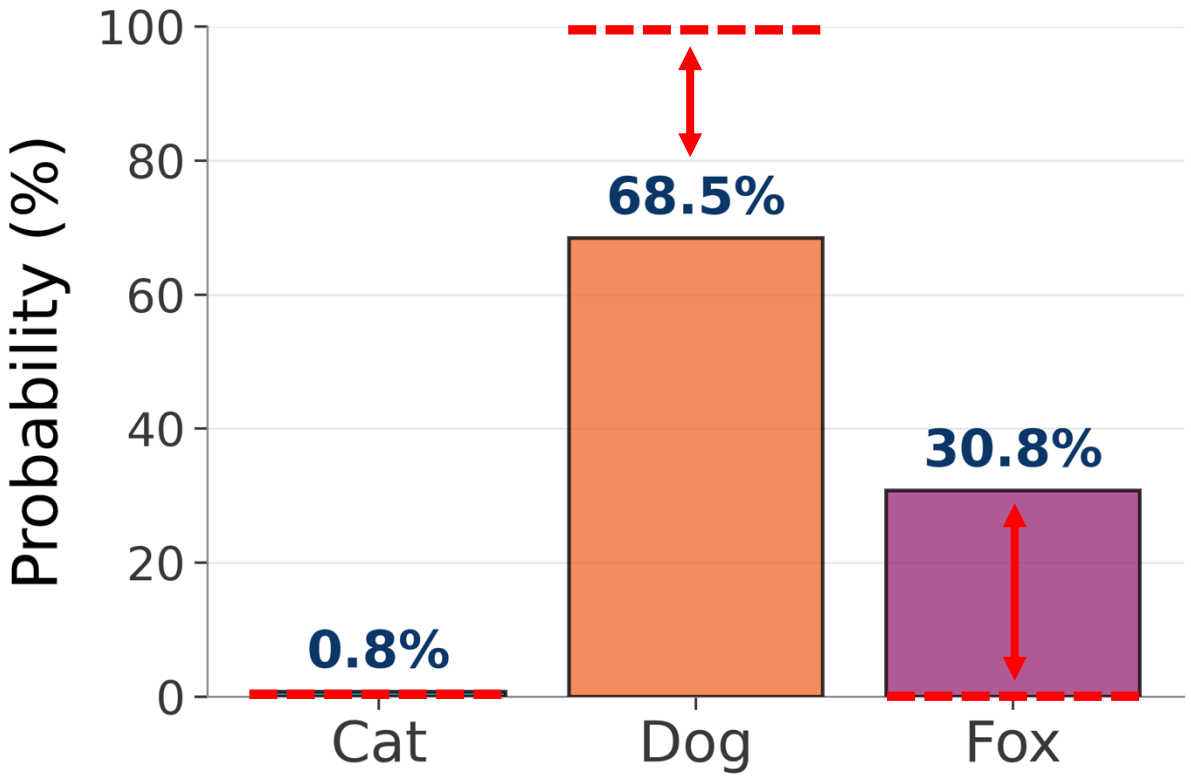
→ ***Softmax***

$$P(y = \text{dog}) = \frac{e^{5.7}}{e^{1.2} + e^{5.7} + e^{4.9}} \approx 0.685$$

$$P(y = \text{cat}) = \frac{e^{1.2}}{e^{1.2} + e^{5.7} + e^{4.9}} \approx 0.008$$

$$P(y = \text{cat}) = \frac{e^{4.9}}{e^{1.2} + e^{5.7} + e^{4.9}} \approx 0.308$$

Optimize weights using a cross-entropy loss



For one sample (*e.g.*, the dog photo):

$$-(0 \cdot \log(0.008) + 1 \cdot \log(0.685) + 0 \cdot \log(0.308))$$

Cat **Dog** **Fox**

$$\approx \mathbf{0.378}$$

This will be smallest when the high probability aligns with the true class

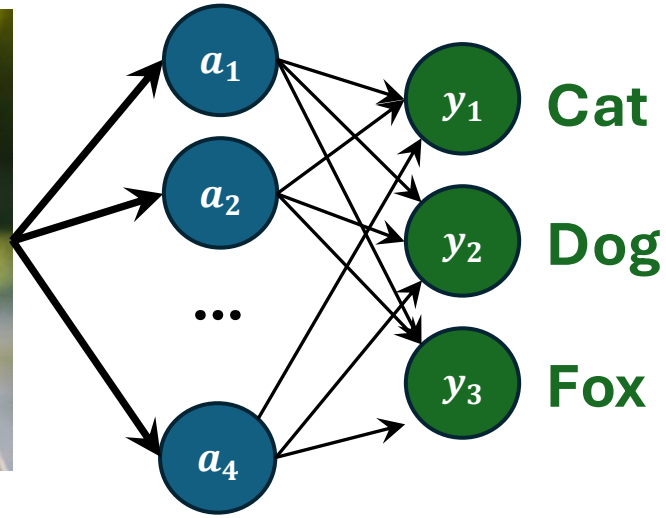
For example: $-1 \cdot \log(\mathbf{0.90}) \approx 0.105$

$$J(\boldsymbol{\beta}) == -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} \log(p_{ik})$$

We want to minimize the **multiclass cross-entropy**

How feasible is this modeling task?





48×48×3 (pixels RGB) input

One hidden layer, 128 neurons

→ **49% classification accuracy**

kaggle



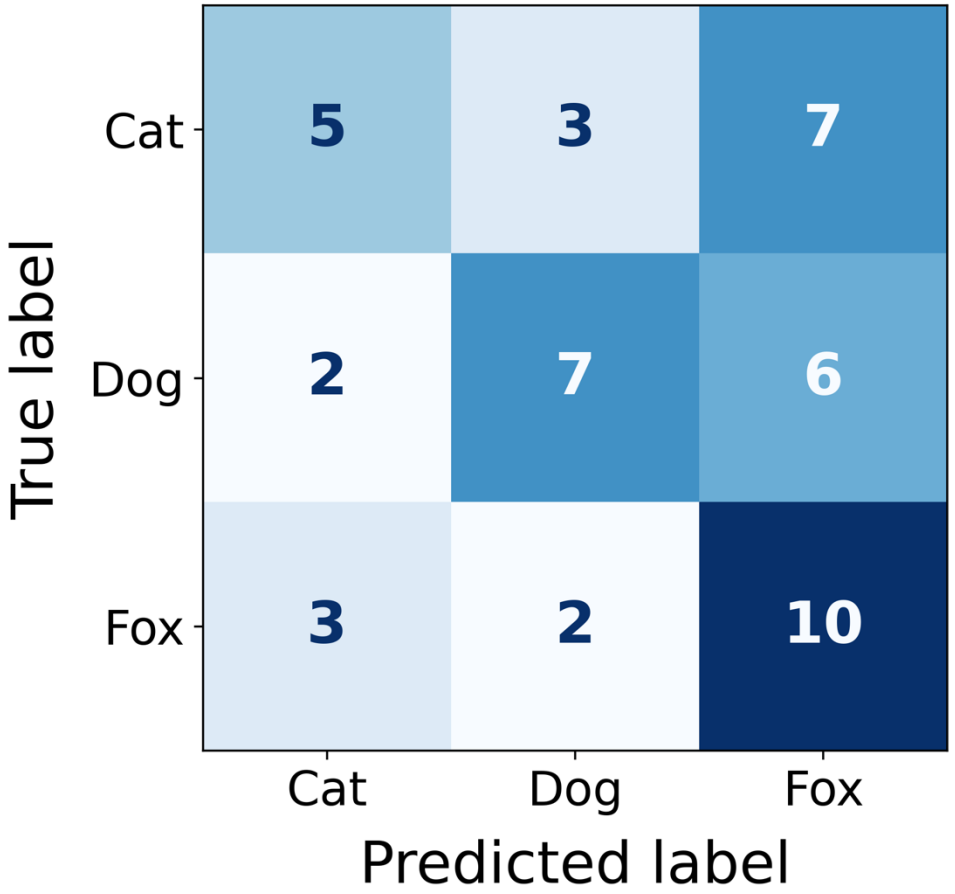
180 images



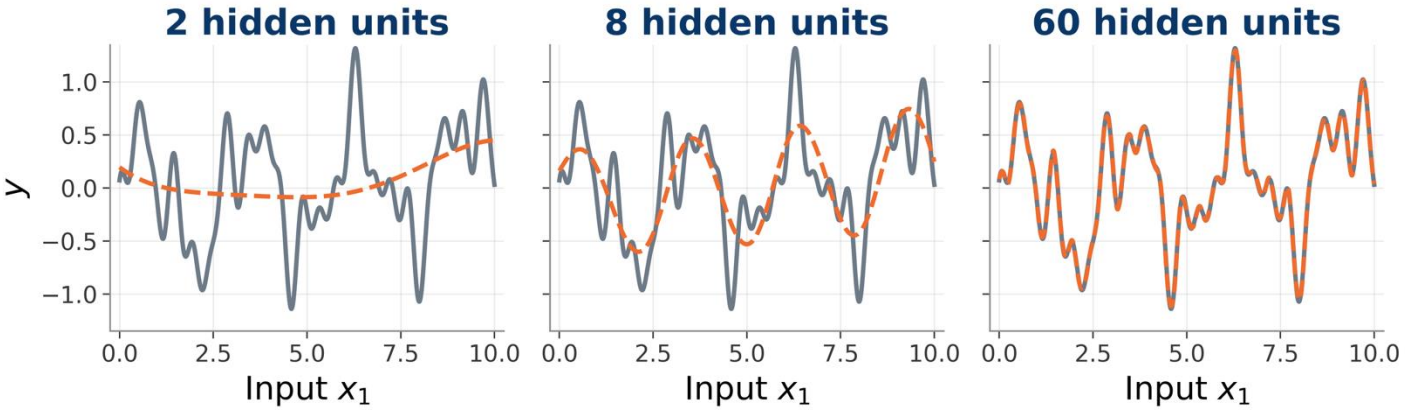








How feasible is this modeling task?



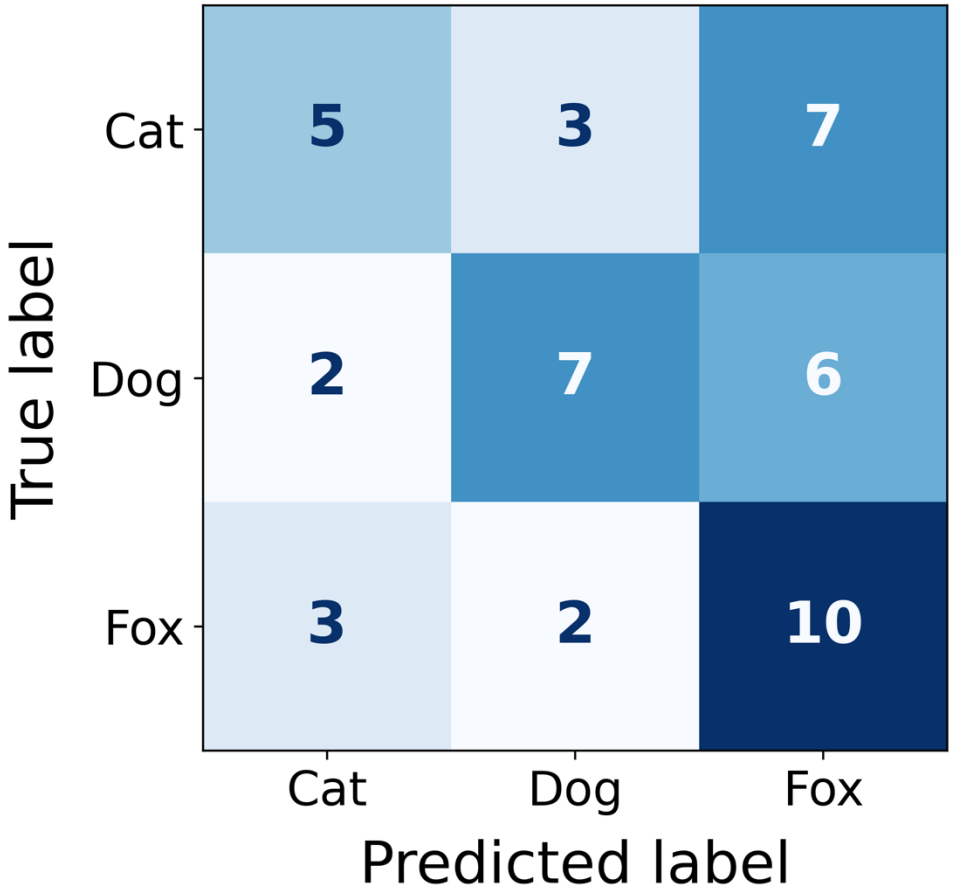
Remember the universal approximation theorem…With enough hidden neurons, we can learn anything, right?
(curves *and* decision boundaries)

Let's try adding **more neurons** to our hidden layer and see what happens…

48×48×3 (pixels RGB) input

One hidden layer, 128 neurons

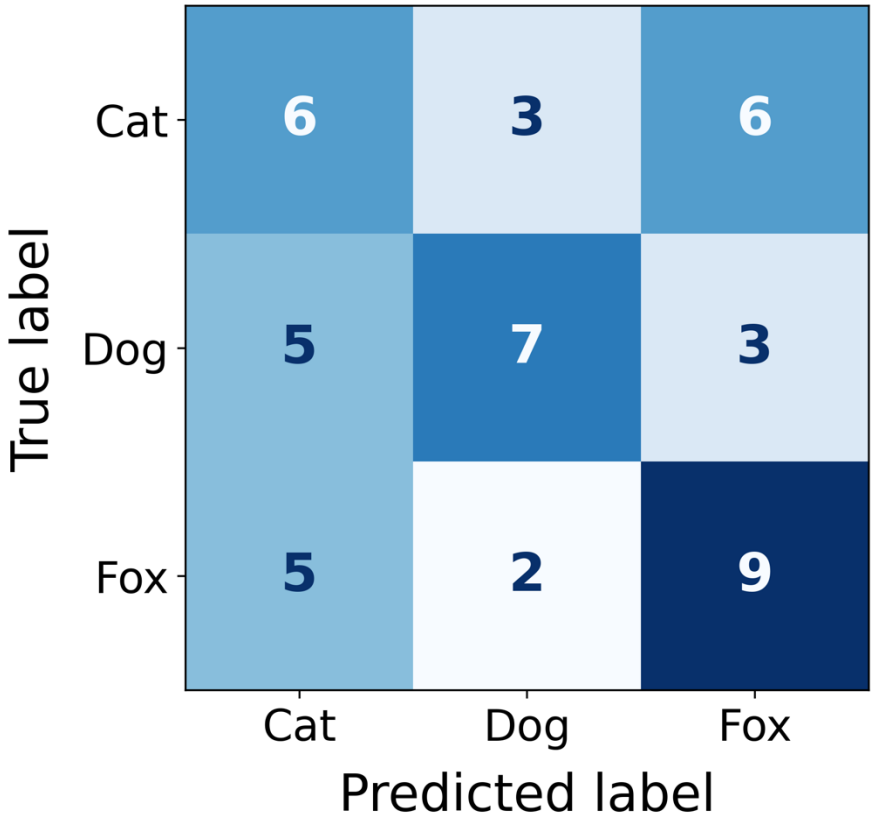
→ **49% classification accuracy**



Results from 1 hidden layer, many neurons

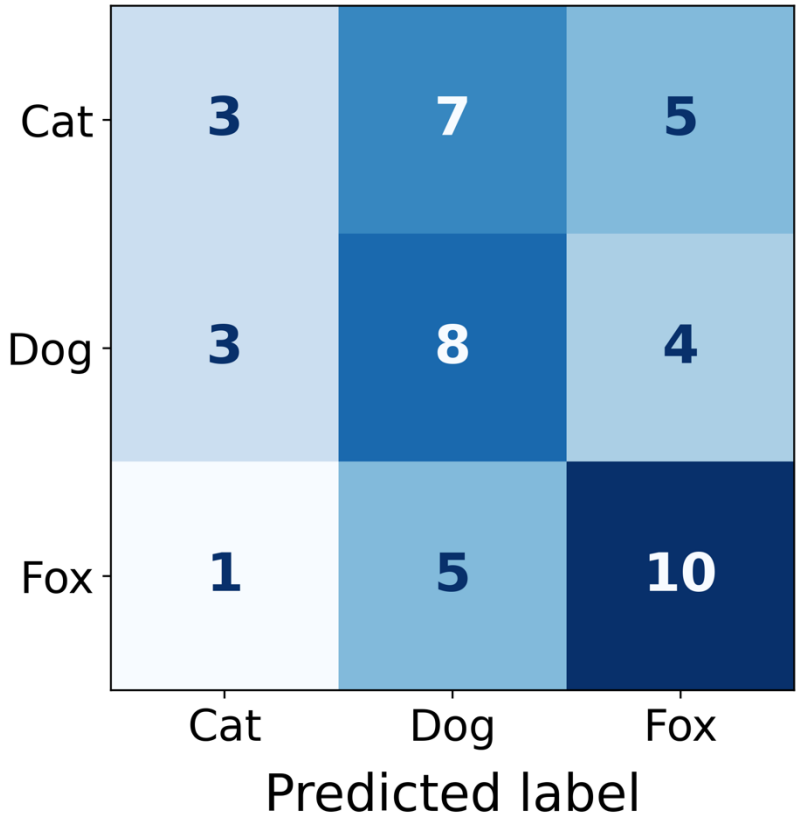
256 neurons

48% accuracy



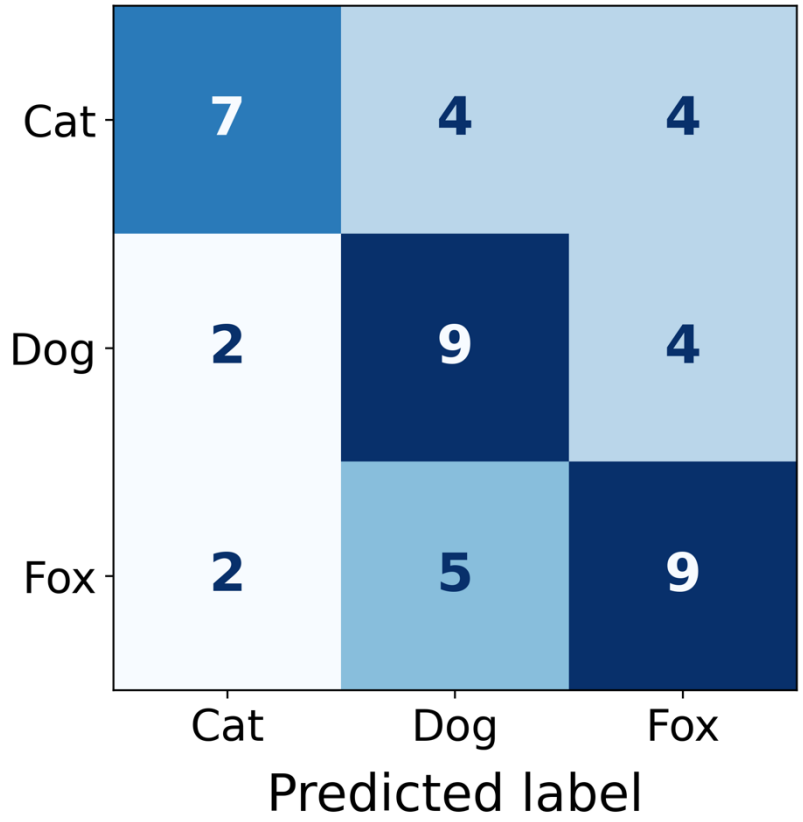
512 neurons

46% accuracy



1,024 neurons

54% accuracy



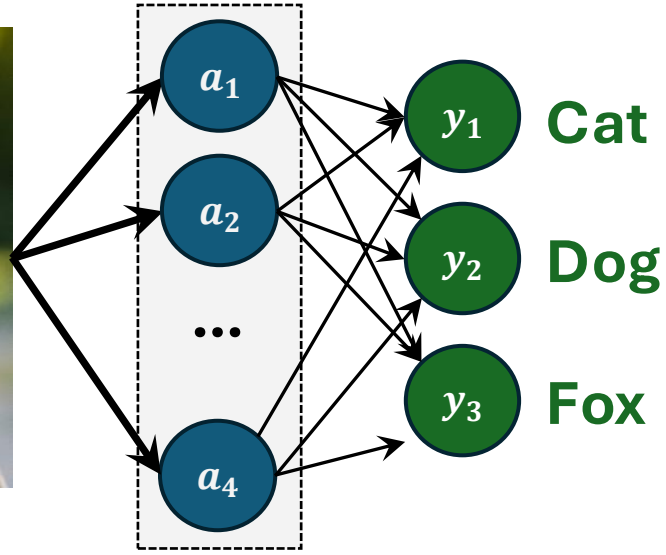
Not a very strong improvement…How else can we increase complexity?

Deep Neural Networks

What happens if we include **more than one hidden layer**?

Introducing more than one hidden layer

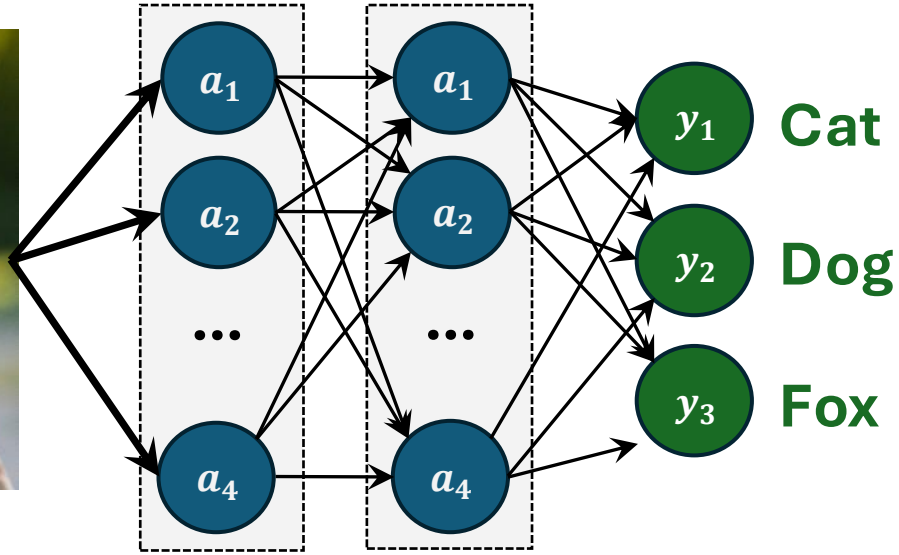




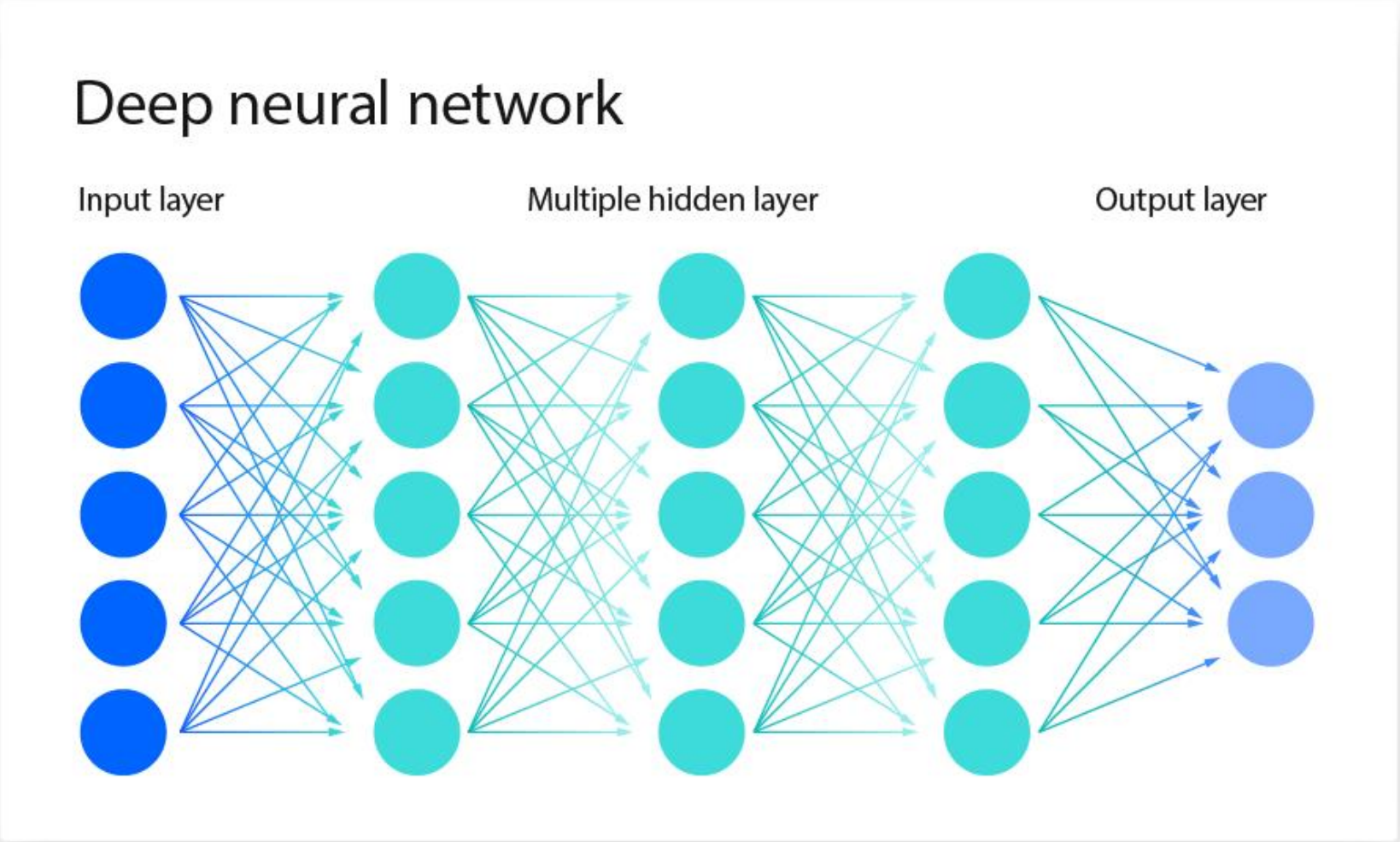
A “shallow” neural network has just one hidden layer separating the input from the output

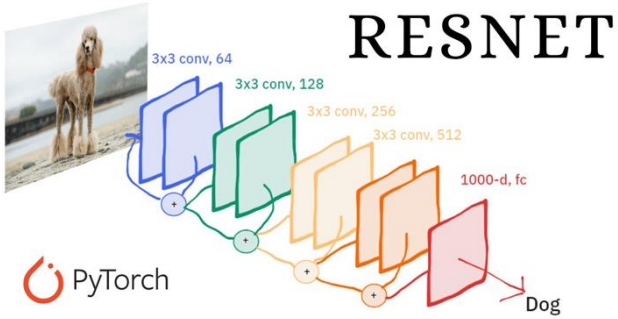
A **“deep”** neural network has **> 1 hidden layers** separating the input from the output





Modern neural networks often have *many* layers!

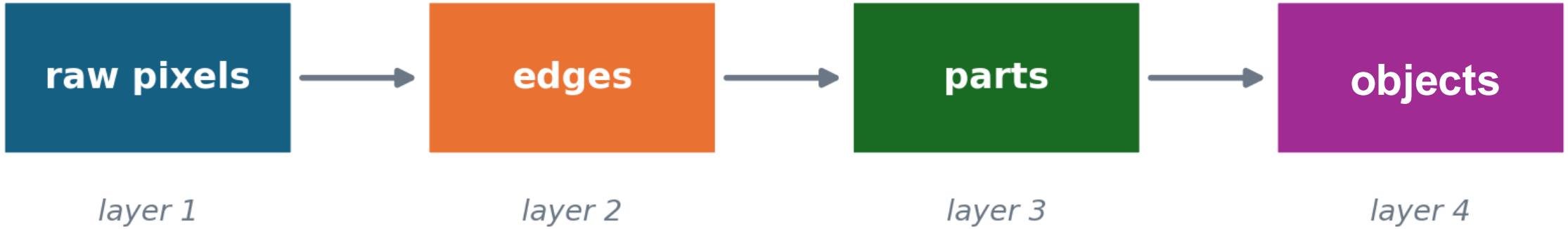


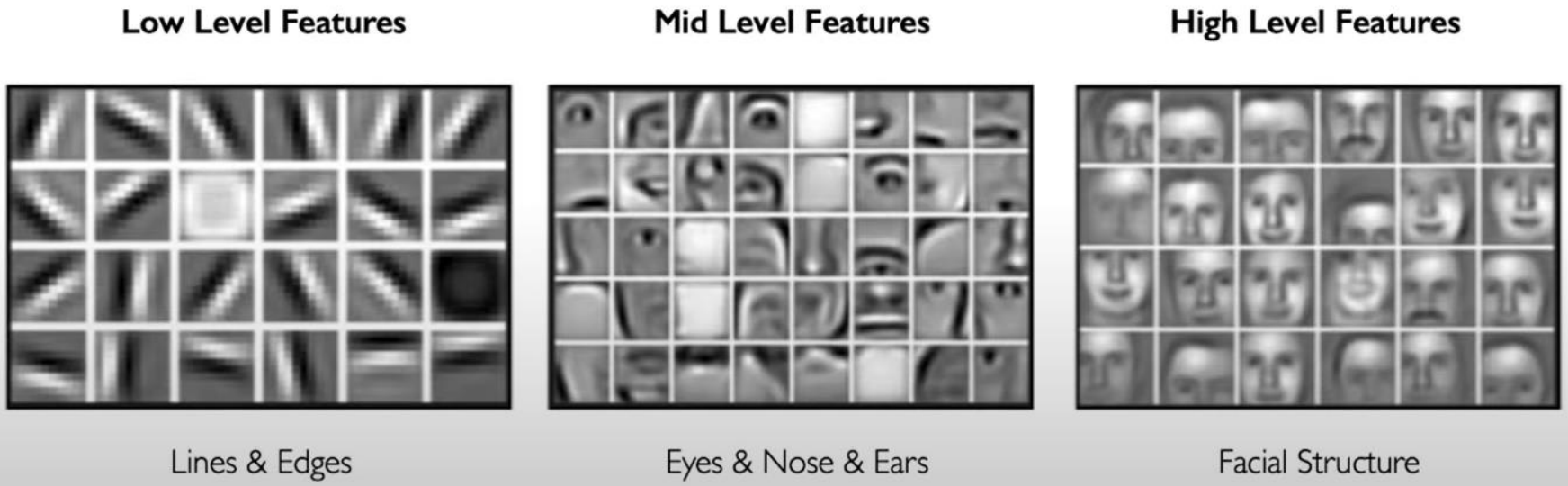


For example, ResNet is a common model architecture for image classification that uses up to **hundreds of hidden layers** (though, conv…)

The benefit of using multiple hidden layers

A single hidden layer, wide enough, can fit **any continuous function**.
But in practice, **deeper** networks learn more easily than very wide ones.

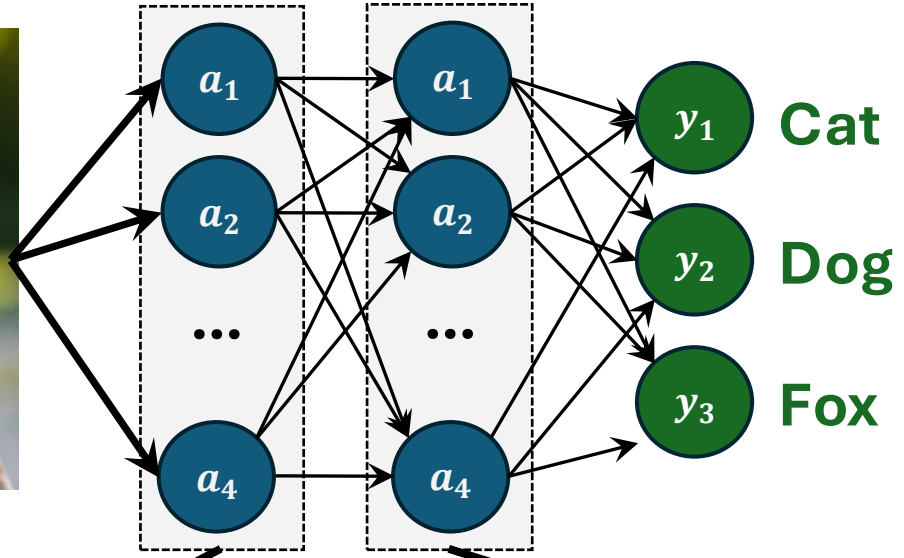




Each layer builds **new features** from the last.
Deeper layers leads to **richer representations**.

Back to our classification example



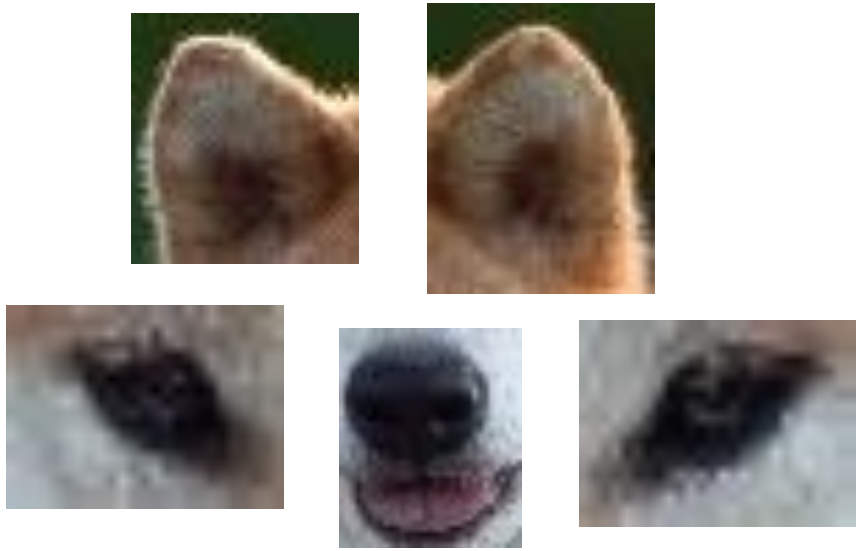




Output layer: which shapes correspond to which animal?



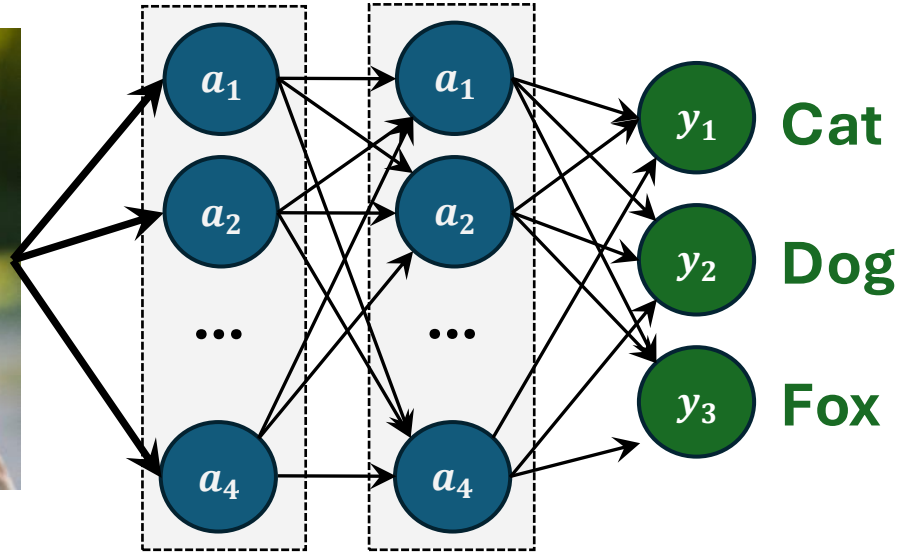
Layer #1: simple tasks like edge detection (light/dark pixels)



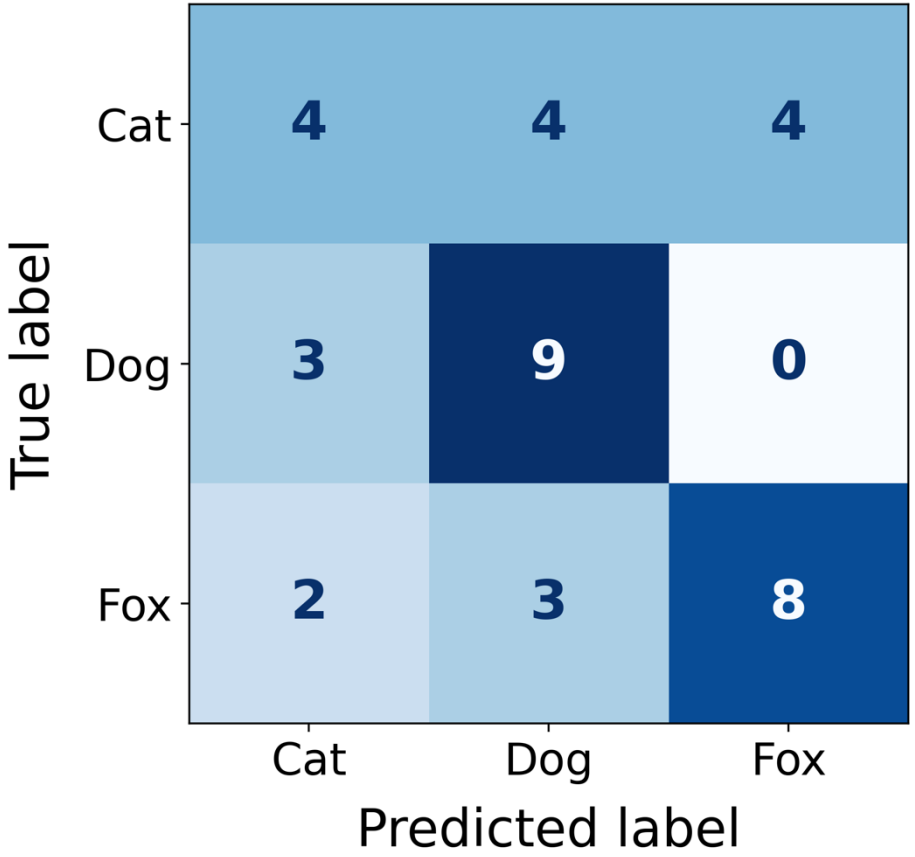
Layer #2: how do edges combine to form shapes?

Back to our classification example





With **3×128-neuron layers,** we get **57% accuracy**



Recall: one layer with 1,024 neurons gave only **54% accuracy, d**espite having *more* neurons than our current model!

Note: 57% accuracy is still not great. In practice, **convolutional filters** are used for image analysis

Convolutional neural networks (CNNs)

"Sliding window"

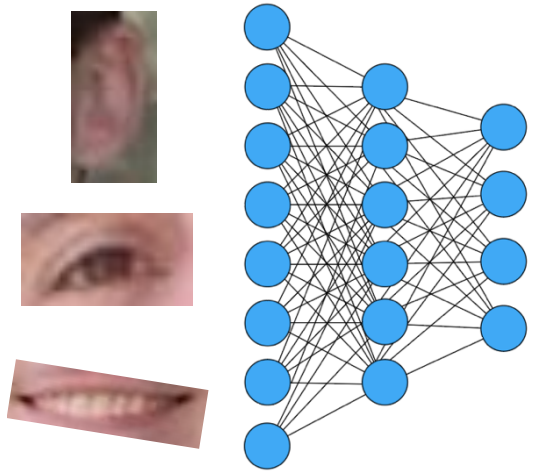


4032×3024 ~ **12 million pixels!**

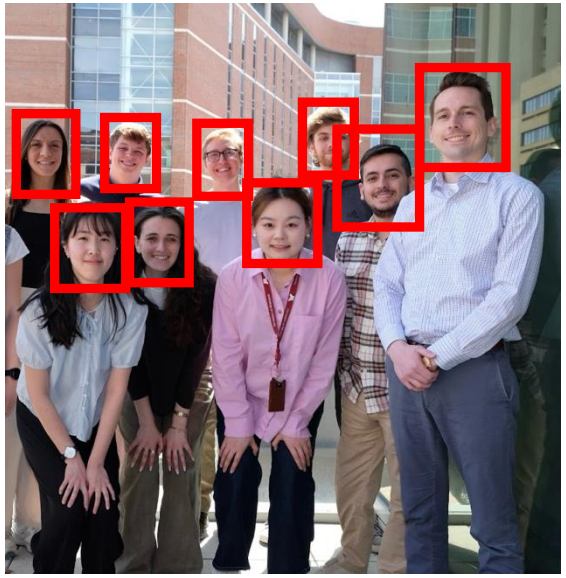
$$\begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

Convolution

Edges,
corners, *etc.*



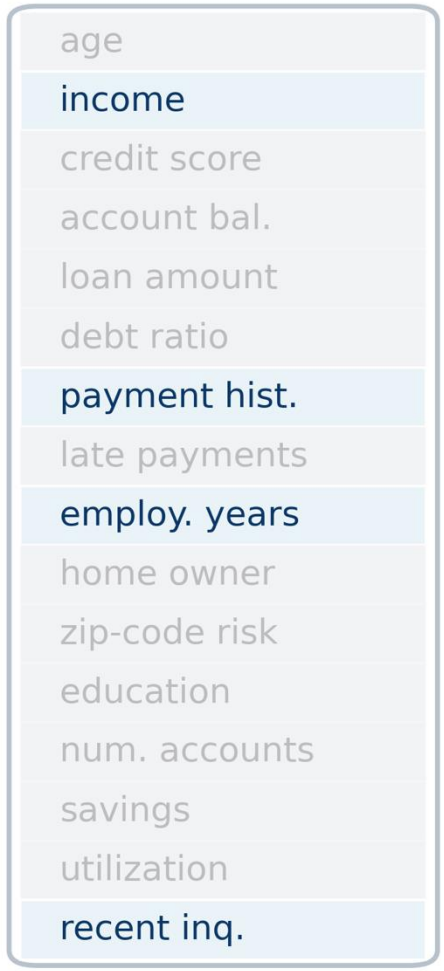
Faces identified



Convolutional filters **can detect the same feature anywhere in an image – they are *translationally equivariant* –** making the model less sensitive to *where* the feature appears.

“Vanilla” neural networks work better on tabular data

Tabular data = rows and columns, with **no inherent spatial relationship** between neighboring features like pixels have in an image.



EX #1: Personal finance data

EX #2: Study data

Hrs Studied	Took prerequisite?	Passed?
2	N	N
6	Y	Y
3	Y	Y
…	…	…

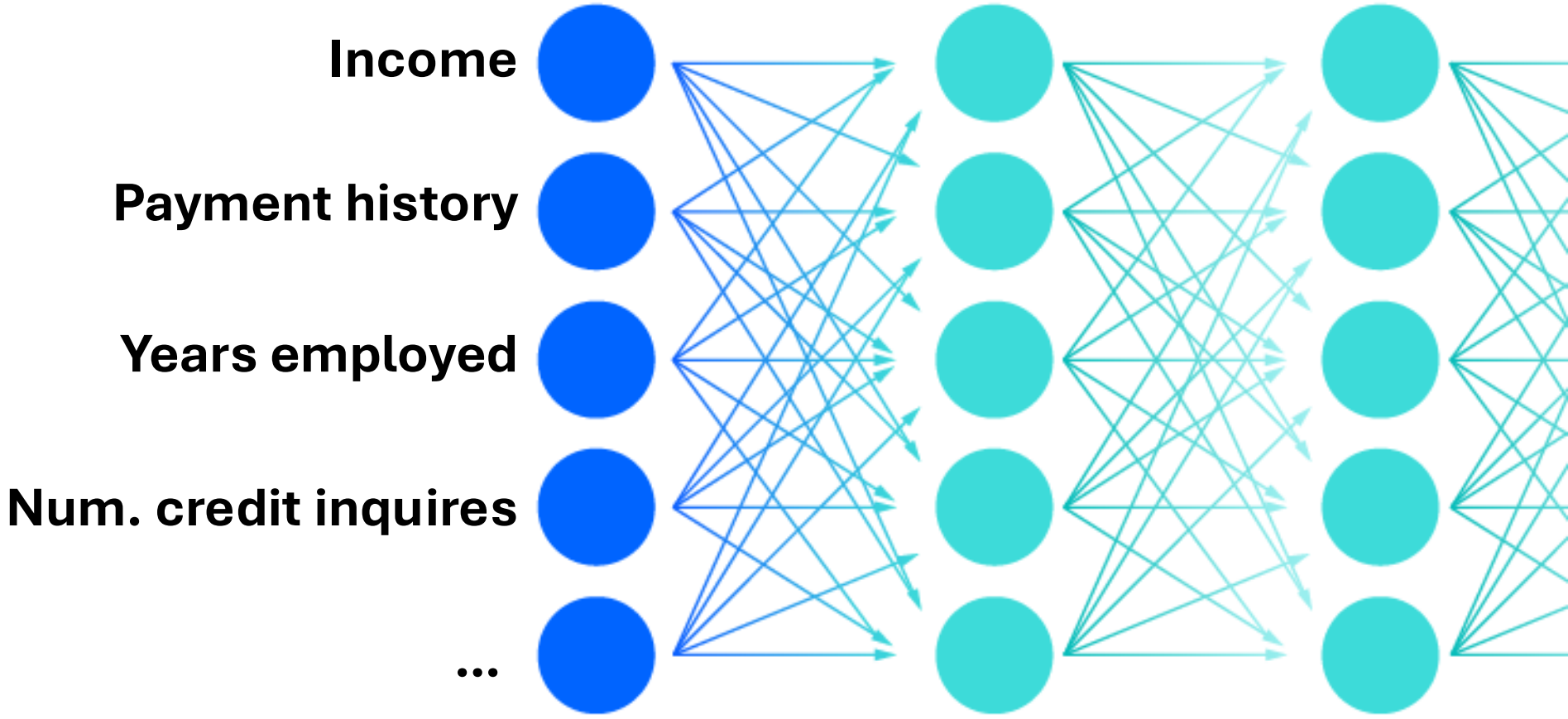
Size (1000 sq ft)	Sale price ($1000)	Home type
1.99	605	Town.
2.08	645	Town.
2.04	670	Fam.
…	…	…

EX #3: Housing sales data

“Vanilla” neural networks work better on tabular data

Each input neuron represents one input feature

- age
- income
- credit score
- account bal.
- loan amount
- debt ratio
- payment hist.
- late payments
- employ. years
- home owner
- zip-code risk
- education
- num. accounts
- savings
- utilization
- recent inq.



“Vanilla” neural networks work better on tabular data

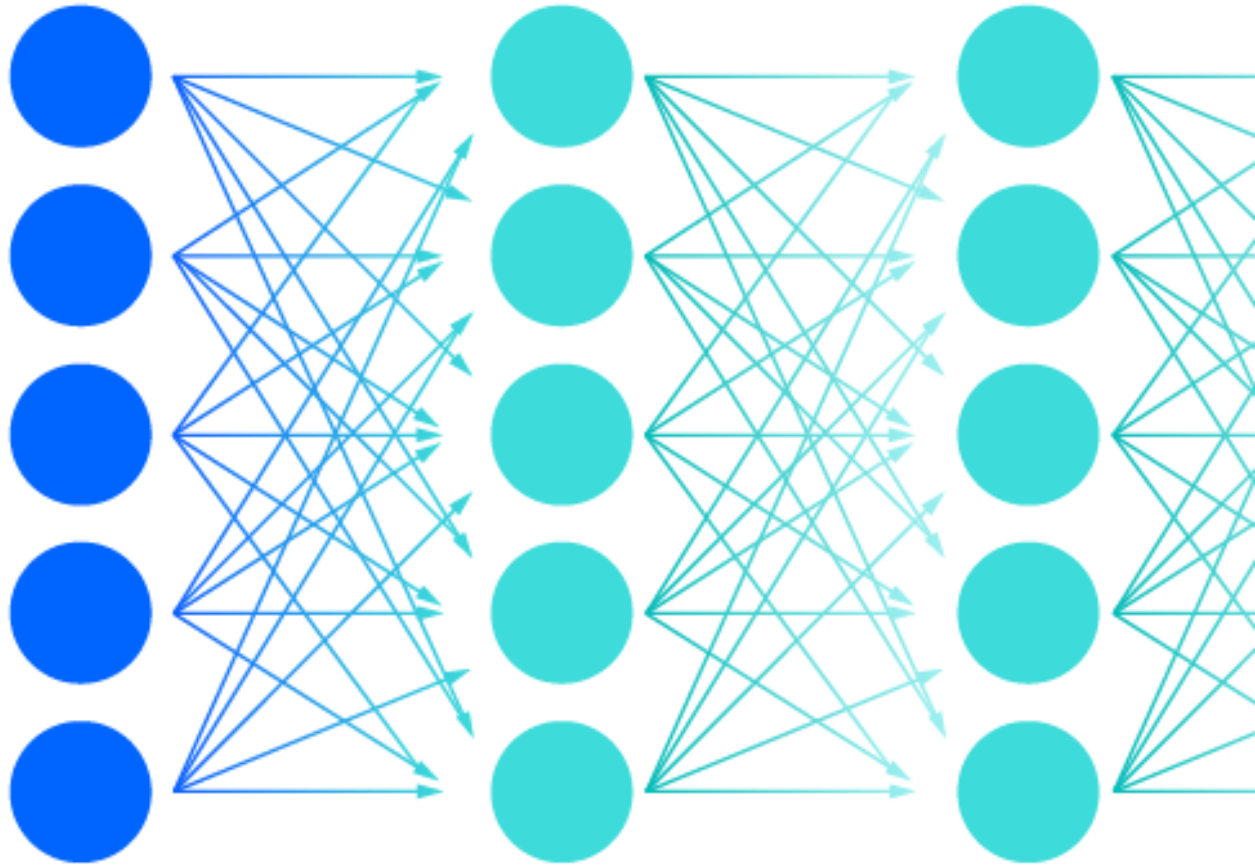
Neural networks can also handle *mixed* data types

Hrs Studied	Took prerequisite?	Passed?
2	N	N
6	Y	Y
3	Y	Y
…	…	…

Hours studied (a number)

Took prereq: Y (1) or N (0)

Passed: Y (1) or N (0)



“Vanilla” neural networks work better on tabular data

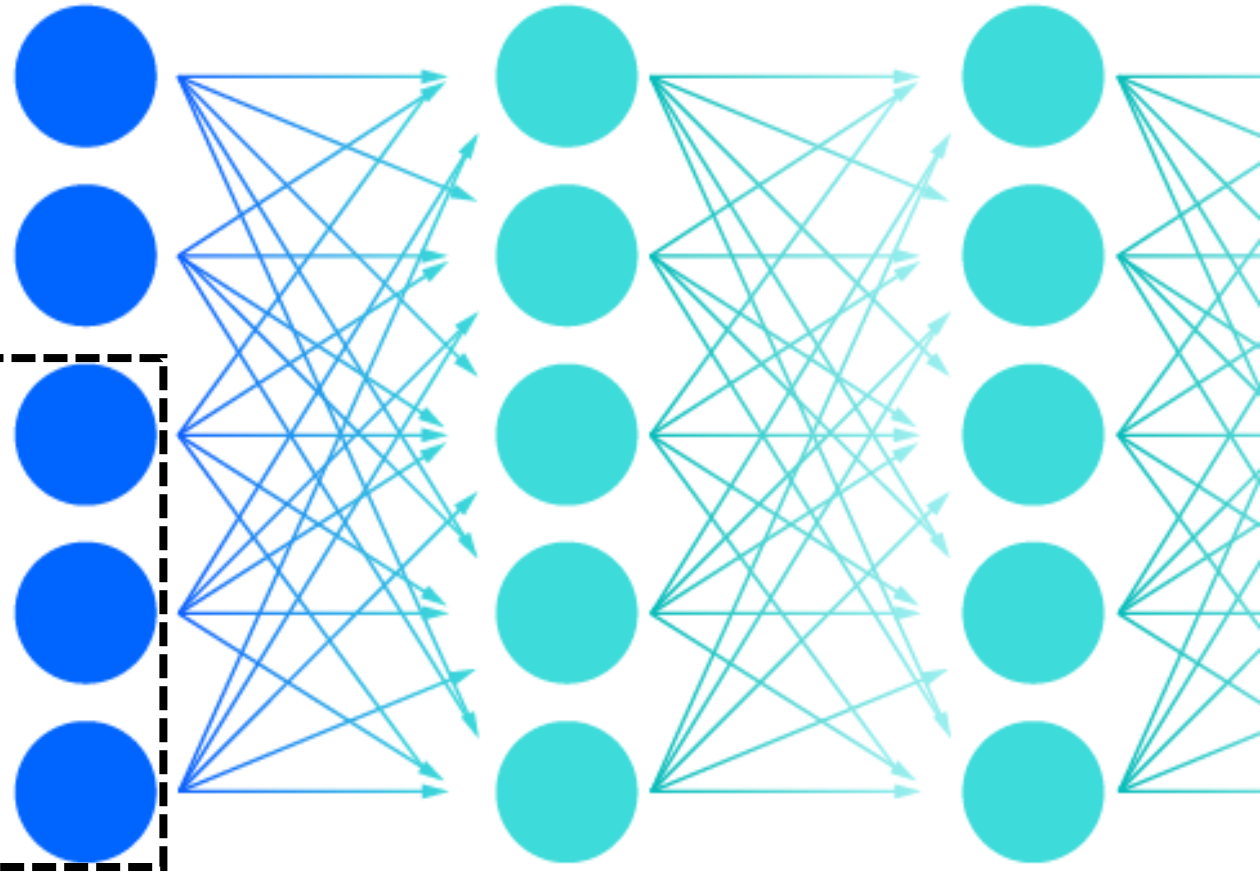
For categorial data with > 2 classes: use one-hot encoding

Sale price ($1000)	Home type	
599	Town.	(1,0,0)
514	Apt.	(0,1,0)
701	Family	(0,0,1)

Is a townhouse? Y (1) or N (0)

Is an apartment? Y (1) or N (0)

Is a family home? Y (1) or N (0)



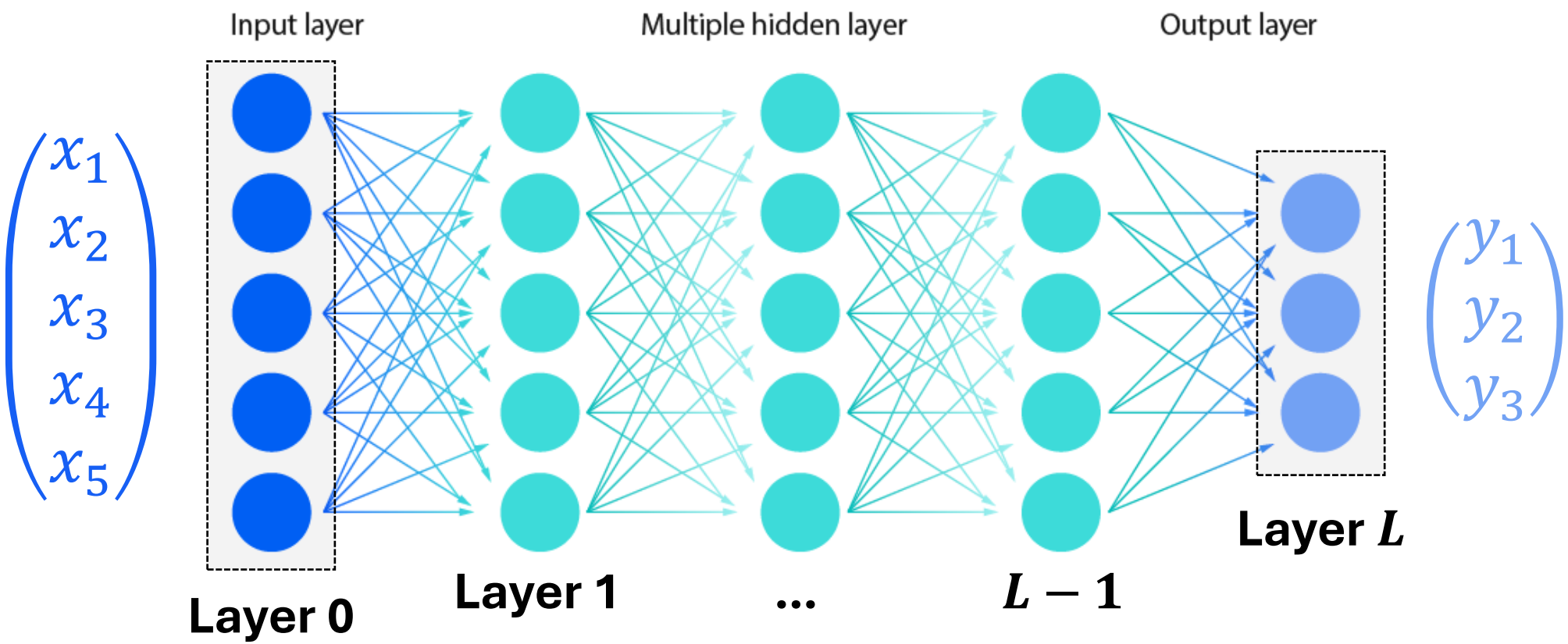
Questions?

A bit of math before next time

How do inputs get “propagated” through the network?

Notation: Input, Output, and Hidden layers

The **input (x)** and **output ($\hat{y}$)** are both **vectors** whose lengths are determined by the numbers of neurons in the input and output layers, respectively.



Notation: The number of neurons in each layer

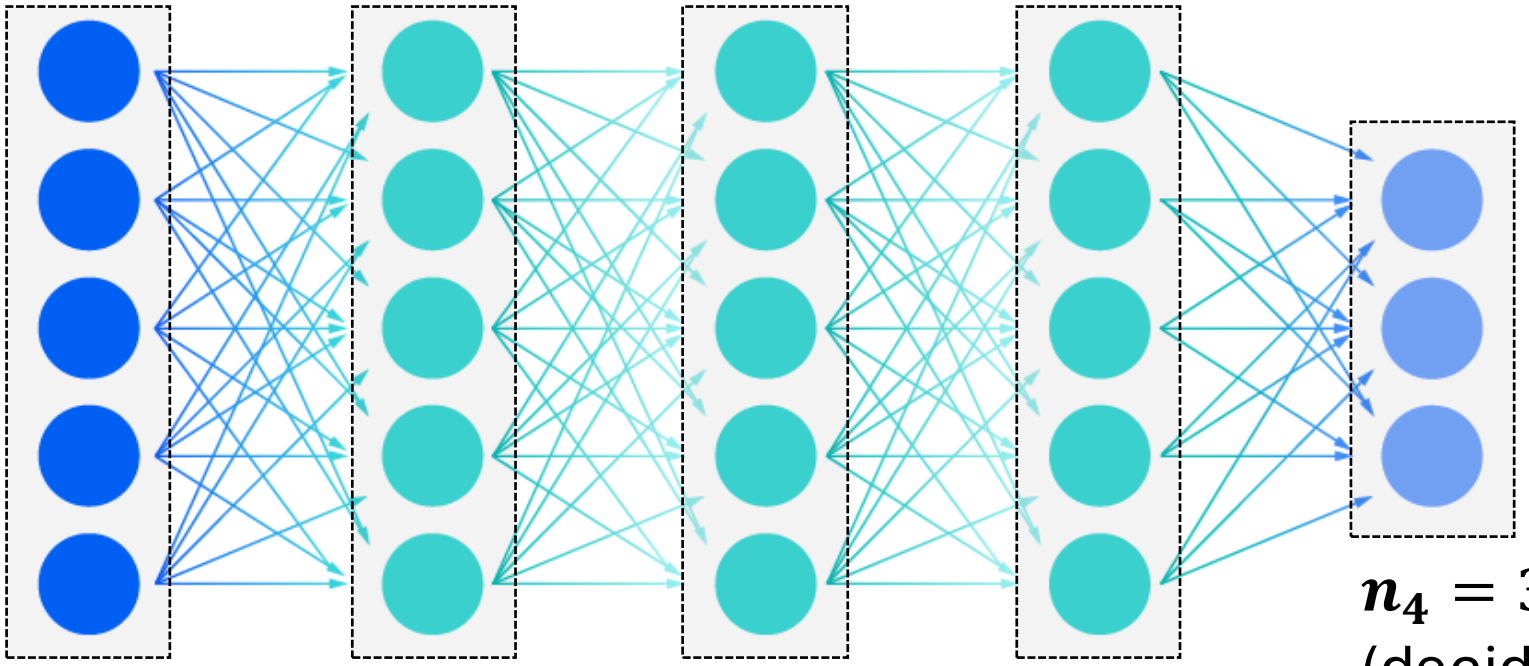
n_i **is a scalar** used to represent the **number of neurons** in each layer

$n_0 = 5$ input features
(decided by the data)

$n_2 = 5$

$n_3 = 5$

Parameters we choose. Controls model complexity.



$n_1 = 5$ neurons in layer 1

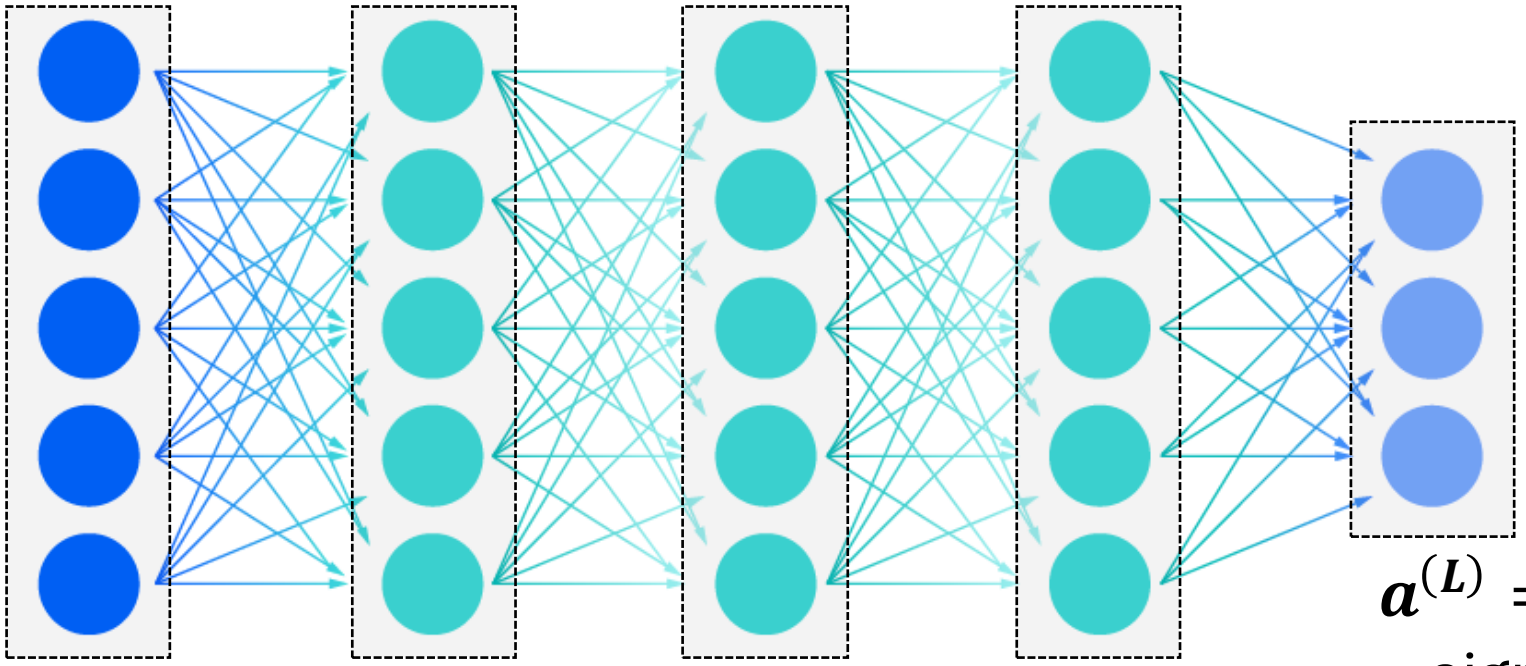
$n_4 = 3$ outputs
(decided by the task)

Notation: Activation functions

$a^{(l)}$ **is a vector** used to represent **activated outputs** from each layer

$a^{(0)}$ = linear

$a^{(l)}$ = Relu (usually)

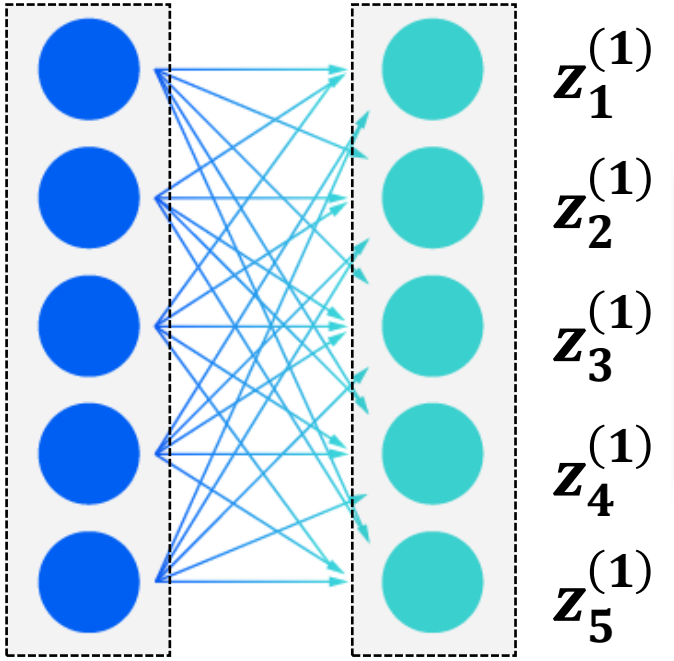


$a^{(L)}$ = linear, sigmoid, or softmax

Notation: Activation functions

$a^{(l)}$ **is a vector** used to represent **activated outputs** from each layer

$z^{(l)}$ **is the vector** representing the values of layer l **before activation**



$$z_1^{(1)}$$
$$z_2^{(1)}$$
$$z_3^{(1)}$$
$$z_4^{(1)}$$
$$z_5^{(1)}$$

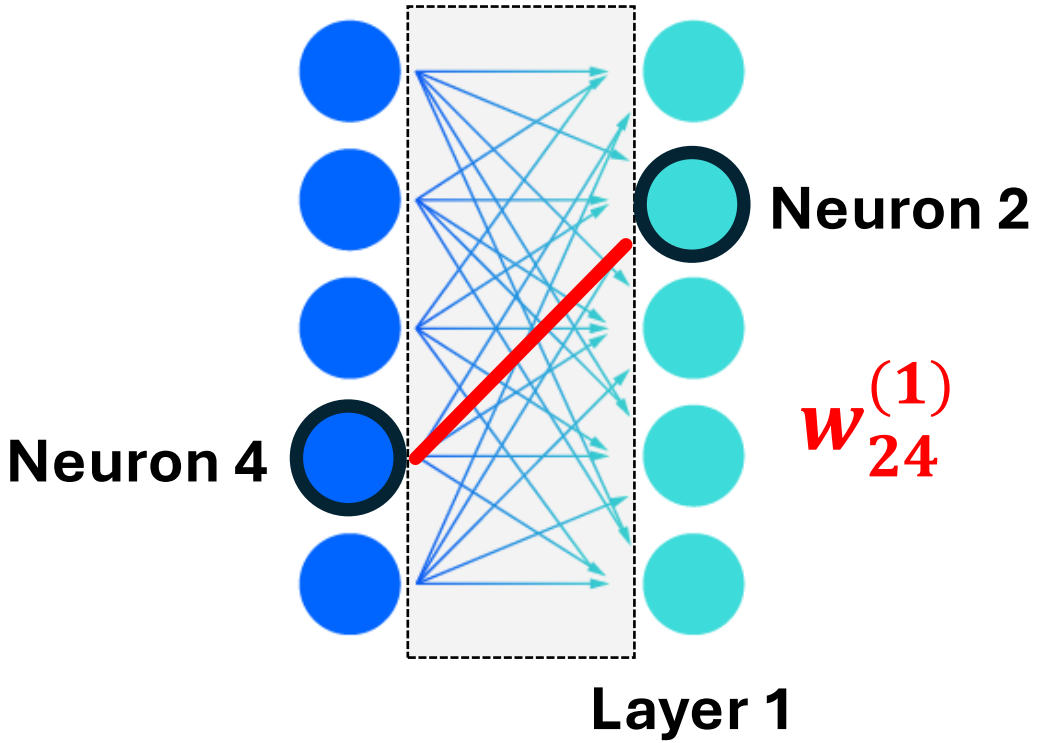
$$a^{(1)} = \begin{pmatrix} \mathbf{Relu}\left(z_1^{(1)}\right) \\ \mathbf{Relu}\left(z_2^{(1)}\right) \\ \dots \\ \mathbf{Relu}\left(z_5^{(1)}\right) \end{pmatrix}$$

This is what then gets passed to the next layer

Notation: Weights (connections)

$w_{ji}^{(l)}$ **is a scalar** representing the **weight** from **neuron i in layer $l-1$ to neuron j in layer l**

Each line we draw in the neural network is one weight



"In this layer"

$$w_{ji}^{(1)}$$

Destination

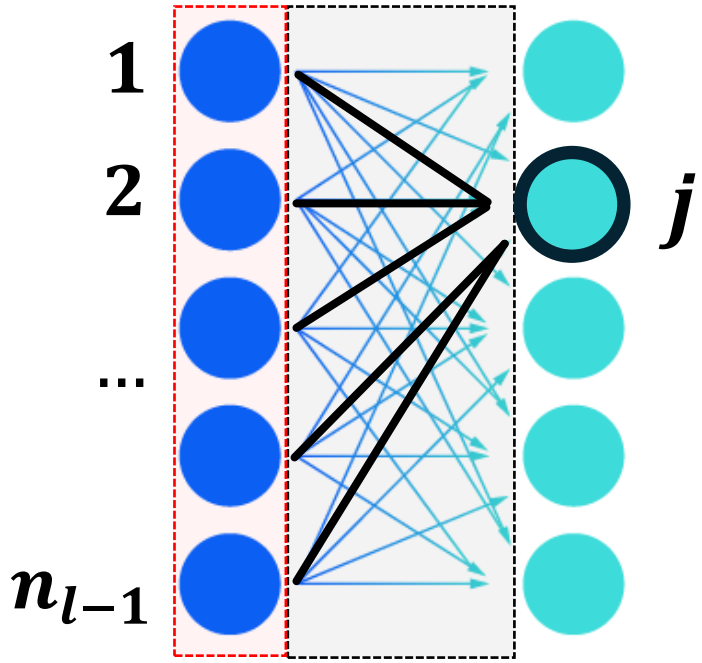
Origin

"To there from here"

Notation: Vector calculation for one neuron

The **weights entering a single neuron *j*** can be collected into a **vector:**

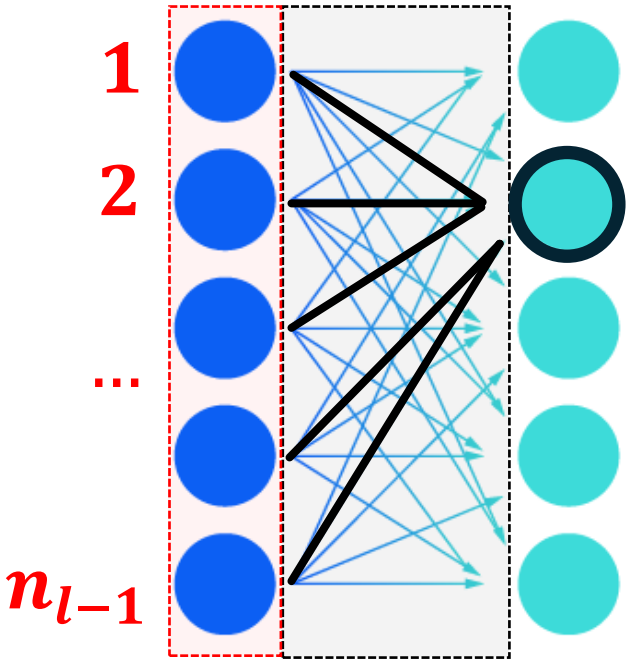
$$w_j^{(l)} = \begin{pmatrix} w_{j1}^{(l)} \\ w_{j2}^{(l)} \\ \dots \\ w_{jn_{l-1}}^{(l)} \end{pmatrix}$$



Notation: Vector calculation for one neuron

The **weights entering a single neuron *j*** can be collected into a **vector:**

$$w_j^{(l)} = \begin{pmatrix} w_{j1}^{(l)} \\ w_{j2}^{(l)} \\ \dots \\ w_{jn_{l-1}}^{(l)} \end{pmatrix}$$



Then, the **pre-activation value of neuron *j*** is:

$$z_j^{(l)} = \left(w_j^{(l)}\right)^T a^{(l-1)} + b_j^{(l)}$$

The bias on *j* (scalar)

$$\begin{pmatrix} w_{j1}^{(l)} & w_{j2}^{(l)} & \dots & w_{jn_{l-1}}^{(l)} \end{pmatrix} \begin{pmatrix} a_1^{(l-1)} \\ a_2^{(l-1)} \\ \dots \\ a_{n_{l-1}}^{(l-1)} \end{pmatrix} = \text{Scalar}$$

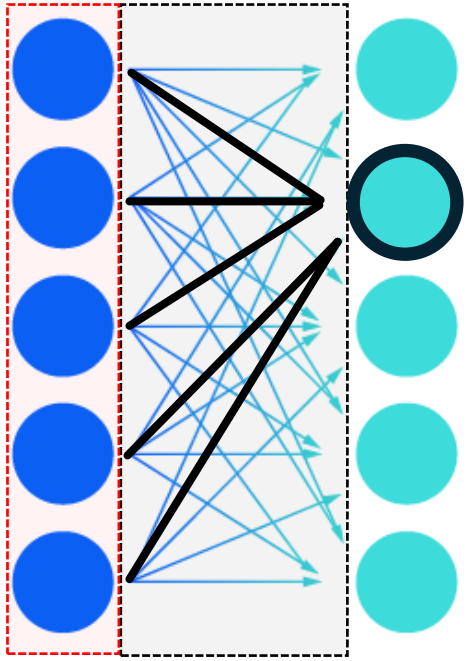
Weights connected to *j*

Activated outputs **from previous layer**

Notation: Vector calculation for one neuron

Replace **Relu** with whatever activation you're using. To keep it general, this is sometimes written as: $a_j^{(l)} = g^{(l)}(z_j^{(l)})$

activation



$$a_j^{(l)} = \mathbf{Relu}(z_j^{(l)})$$

This is what feeds into the next layer

Now **activate** it!

Then, the **pre-activation value of neuron *j*** is:

$$z_j^{(l)} = \left(w_j^{(l)}\right)^T a^{(l-1)} + b_j^{(l)}$$

The bias on *j* (scalar)

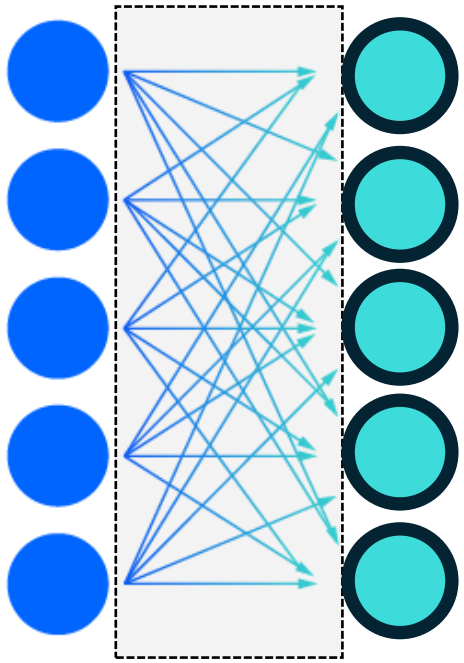
$$\begin{pmatrix} w_{j1}^{(l)} & w_{j2}^{(l)} & \dots & w_{jn_{l-1}}^{(l)} \end{pmatrix} \begin{pmatrix} a_1^{(l-1)} \\ a_2^{(l-1)} \\ \dots \\ a_{n_{l-1}}^{(l-1)} \end{pmatrix} = \text{Scalar}$$

Weights connected to *j*

Activated outputs from previous layer

Notation: *Matrix* calculations for an *entire* *layer*

Instead of computing each neuron separately, we can **compute every neuron in a layer simultaneously.**

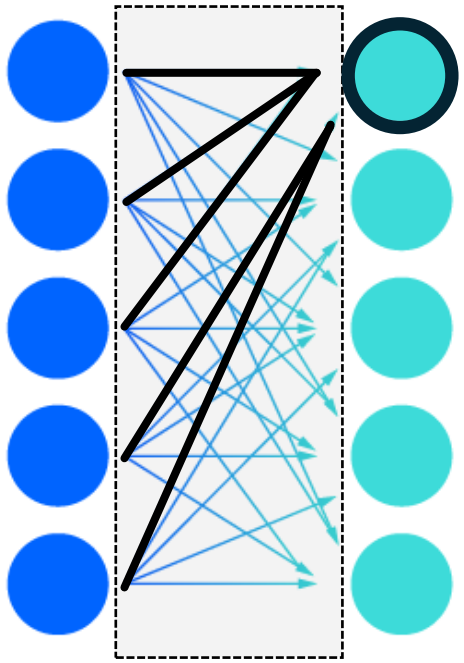


All weights connecting layer $l-1$ to layer l

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \dots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \dots & w_{2n_{l-1}}^{(l)} \\ \dots & \dots & \dots & \dots \\ w_{n_l1}^{(l)} & w_{n_l2}^{(l)} & \dots & w_{n_ln_{l-1}}^{(l)} \end{pmatrix}$$

Notation: *Matrix* calculations for an *entire* *layer*

Instead of computing each neuron separately, we can **compute every neuron in a layer simultaneously.**



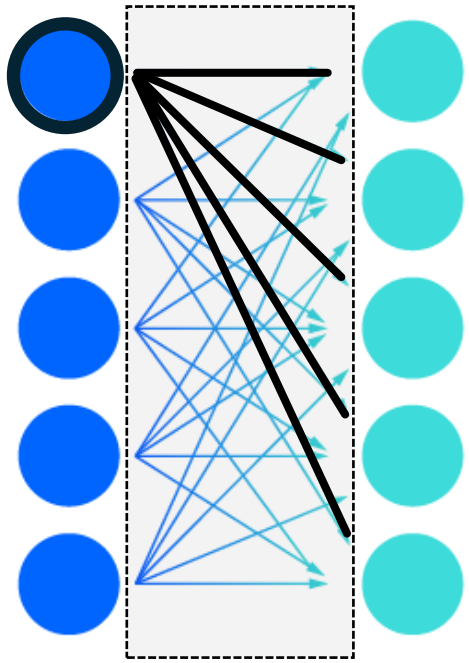
All weights connecting layer $l-1$ to layer l

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \cdots & w_{2n_{l-1}}^{(l)} \\ \cdots & \cdots & \cdots & \cdots \\ w_{n_l 1}^{(l)} & w_{n_l 2}^{(l)} & \cdots & w_{n_l n_{l-1}}^{(l)} \end{pmatrix}$$

Each row describes all the weights feeding into a single neuron in layer l

Notation: *Matrix* calculations for an *entire* *layer*

Instead of computing each neuron separately, we can **compute every neuron in a layer simultaneously.**



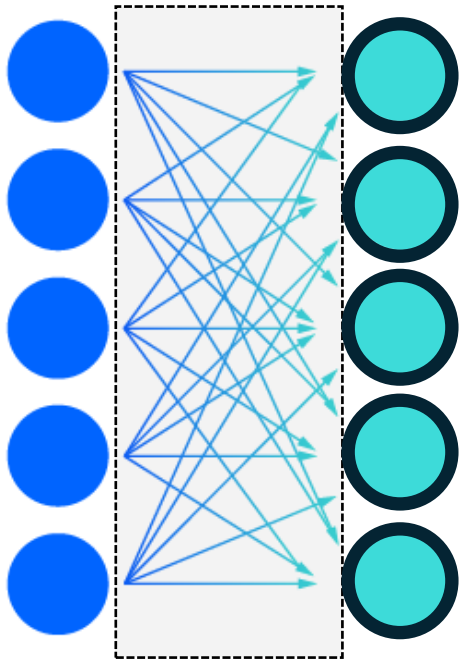
All weights connecting layer $l-1$ to layer l

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \cdots & w_{2n_{l-1}}^{(l)} \\ \cdots & \cdots & \cdots & \cdots \\ w_{n_l 1}^{(l)} & w_{n_l 2}^{(l)} & \cdots & w_{n_l n_{l-1}}^{(l)} \end{pmatrix}$$

Each column describes all the weights originating from a neuron in layer $l-1$

Notation: *Matrix* calculations for an *entire* *layer*

Instead of computing each neuron separately, we can **compute every neuron in a layer simultaneously.**



All weights connecting layer $l-1$ to layer l

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \cdots & w_{2n_{l-1}}^{(l)} \\ \cdots & \cdots & \cdots & \cdots \\ w_{n_l1}^{(l)} & w_{n_l2}^{(l)} & \cdots & w_{n_ln_{l-1}}^{(l)} \end{pmatrix}$$

Similarly, collect the biases into a vector:

Now we have all we need to compute the outputs of layer l from the previous layer: $a^{(l-1)}$

$$b^{(l)} = \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \cdots \\ b_{n_l}^{(l)} \end{pmatrix}$$

Putting it all together into one equation for the layer

$$\boldsymbol{z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}}$$

$$\begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \dots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \dots & w_{2n_{l-1}}^{(l)} \\ \dots & \dots & \dots & \dots \\ w_{n_l 1}^{(l)} & w_{n_l 2}^{(l)} & \dots & w_{n_1 n_{l-1}}^{(l)} \end{pmatrix} \begin{pmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \dots \\ a_{n_{l-1}}^{(l)} \end{pmatrix} + \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \dots \\ b_{n_l}^{(l)} \end{pmatrix}$$

This **yields a vector** with length l, and then we apply our activation to each element of that vector:

$$\boldsymbol{a^{(l)} = g^{(l)}\left(z^{(l)}\right)}$$

All weights connecting layer $l-1$ to layer l

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \dots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \dots & w_{2n_{l-1}}^{(l)} \\ \dots & \dots & \dots & \dots \\ w_{n_l 1}^{(l)} & w_{n_l 2}^{(l)} & \dots & w_{n_l n_{l-1}}^{(l)} \end{pmatrix}$$

Similarly, collect the biases into a vector:

Now we have all we need to compute the outputs of layer l from the previous layer: $a^{(l-1)}$

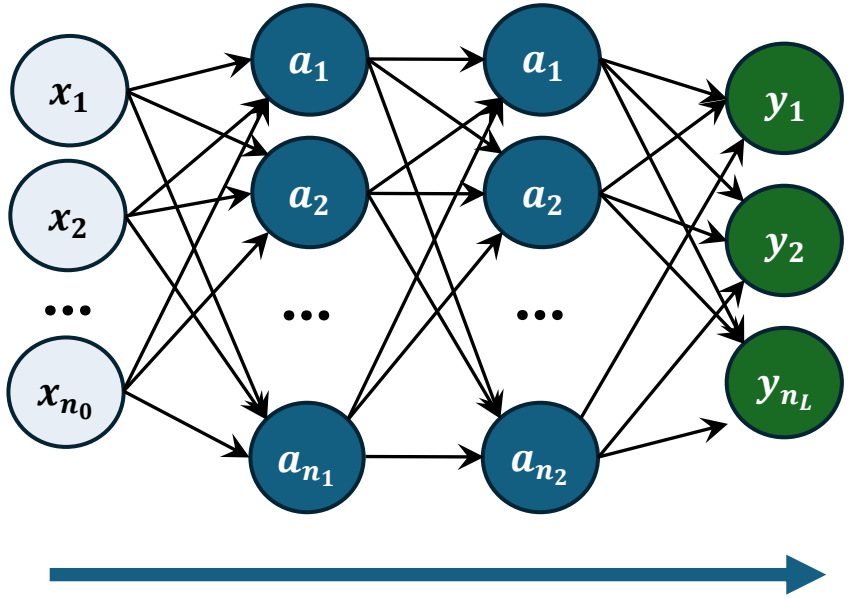
$$b^{(l)} = \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \dots \\ b_{n_l}^{(l)} \end{pmatrix}$$

A complete forward pass

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = g^{(l)}\left(z^{(l)}\right)$$

These equations are all we need to make a **forward pass** through the network: from input to output



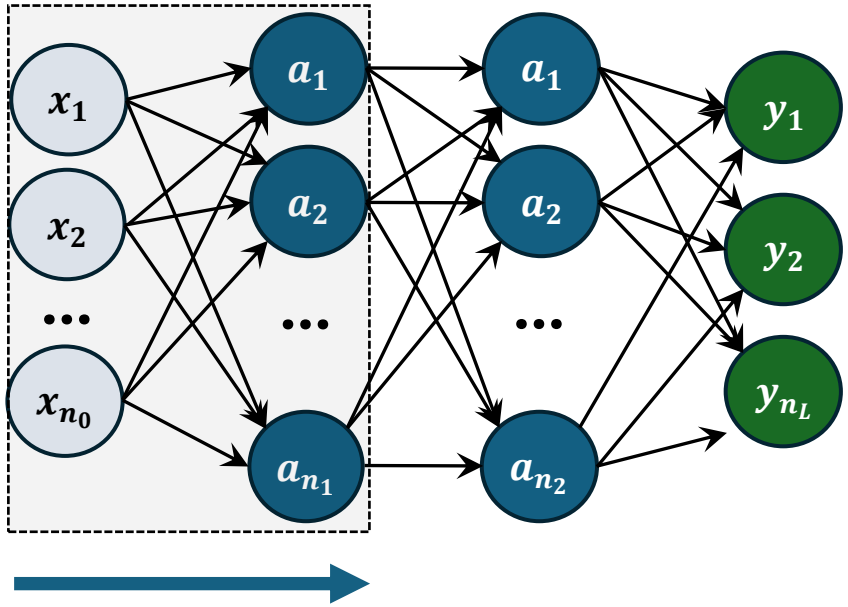
Forward pass

A complete forward pass

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = g^{(l)}\left(z^{(l)}\right)$$

These equations are all we need to make a **forward pass** through the network: from input to output



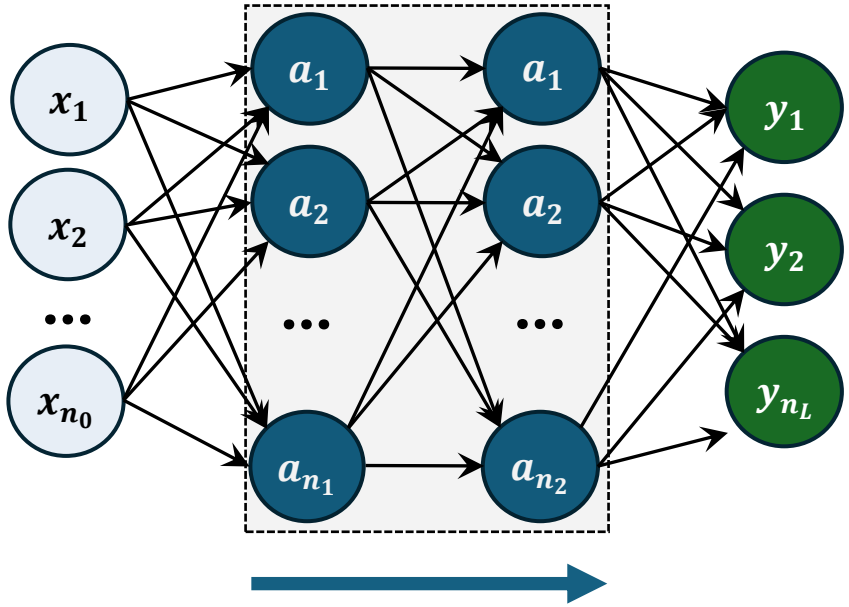
Layer 0→1: $a^{(1)} = g^{(1)}\left(W^{(1)} x + b^{(1)}\right)$

A complete forward pass

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = g^{(l)}\left(z^{(l)}\right)$$

These equations are all we need to make a **forward pass** through the network: from input to output



Layer 0→1: $a^{(1)} = g^{(1)}\left(W^{(1)} x + b^{(1)}\right)$

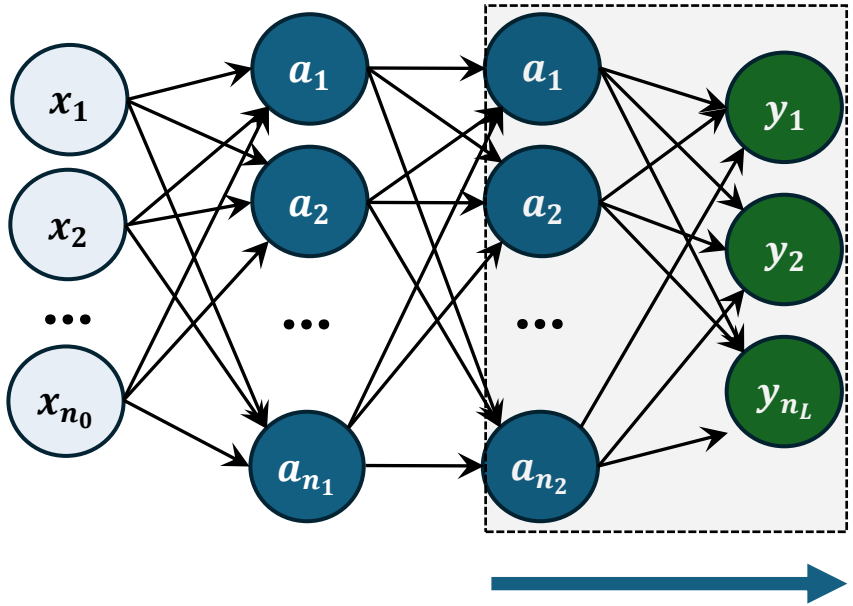
Layer 1→2: $a^{(2)} = g^{(2)}\left(W^{(2)} a^{(1)} + b^{(2)}\right)$

A complete forward pass

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = g^{(l)}\left(z^{(l)}\right)$$

These equations are all we need to make a **forward pass** through the network: from input to output



Layer 0→1: $a^{(1)} = g^{(1)}\left(W^{(1)}x + b^{(1)}\right)$

Layer 1→2: $a^{(2)} = g^{(2)}\left(W^{(2)}a^{(1)} + b^{(2)}\right)$

Layer 2→3: $\hat{y} = g^{(3)}\left(W^{(3)}a^{(2)} + b^{(3)}\right)$

Activation may differ based on task (*e.g.*, softmax)

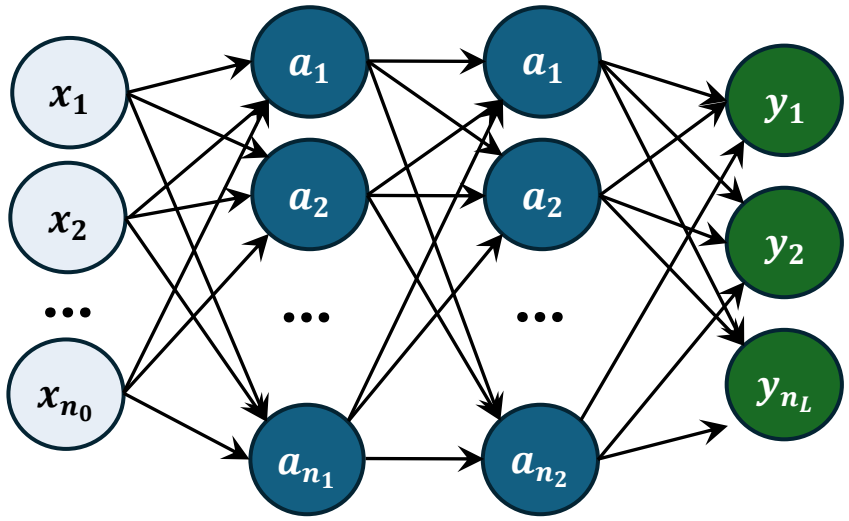
Neural network are just weighted sums (**matrix multiplication**) + **activation functions**. That's it!

$$\hat{y} = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(1)}(x)$$

A *composition* of layer functions

Model size: The number of trainable parameters

How many trainable parameters does our network have?



In total, the size of the NN is:

$$N_{\text{params}} = \sum_{l=1}^{L} n_l(n_{l-1}+1)$$

The weights feeding into layer l are:

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \dots & w_{1n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \dots & w_{2n_{l-1}}^{(l)} \\ \dots & \dots & \dots & \dots \\ w_{n_l1}^{(l)} & w_{n_l2}^{(l)} & \dots & w_{n_1n_{l-1}}^{(l)} \end{pmatrix}$$

This is an $\boldsymbol{n_l \times n_{l-1}}$ matrix

Hence: $(\boldsymbol{n_l})(\boldsymbol{n_{l-1}})$ params

And the bias vector for layer l is:

$$b^{(l)} = \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \dots \\ b_{n_l}^{(l)} \end{pmatrix}$$

We also have $\boldsymbol{n_l}$ params for the bias

Together, layer l has $(\boldsymbol{n_l})(\boldsymbol{n_{l-1}}) + (\boldsymbol{n_l})$ **trainable params**

Modern neural networks are *very* large

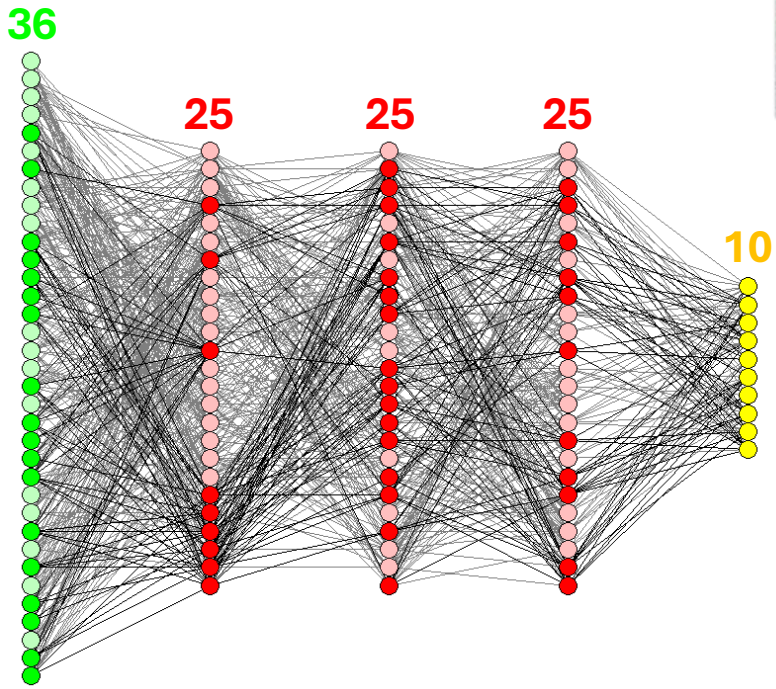
Model	Parameters
GPT-2	1.5 billion
GPT-3	175 billion
GPT-4	1.8 trillion
GPT-5	50 trillion

Training these models… Expensive!



$$N_{\text{params}} = 25(36+1) + 25(25+1) + 25(25+1) + 10(25+1) = \mathbf{2,485}$$

This is a relatively simple NN!





Questions?

What to know for the final exam

- The basics of **perceptrons and neural networks**
- NN architectures, trainable parameters (weights + biases)
  - How architecture affects model **complexity**
- **Activation functions** and their effects on outputs
- Know how to do a **forward pass** calculation

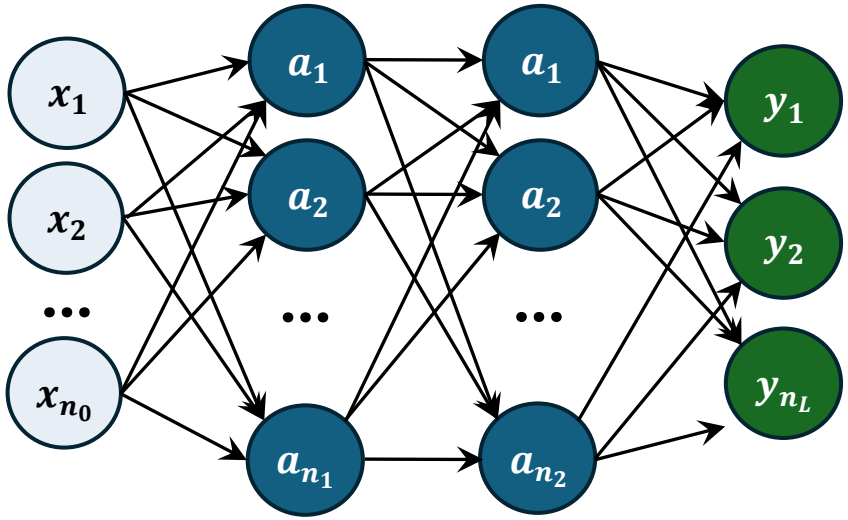
What we did *not* cover today

Model	Parameters
GPT-2	1.5 billion
GPT-3	175 billion
GPT-4	1.8 trillion
GPT-5	50 trillion

Training these models… Expensive!

How do we go about finding the best weights for a given task?

Next time: **Backpropagation** and **gradient descent** for training



Backward pass