Deep learning, neural networks and frameworks

The Academy of AI 2018/19

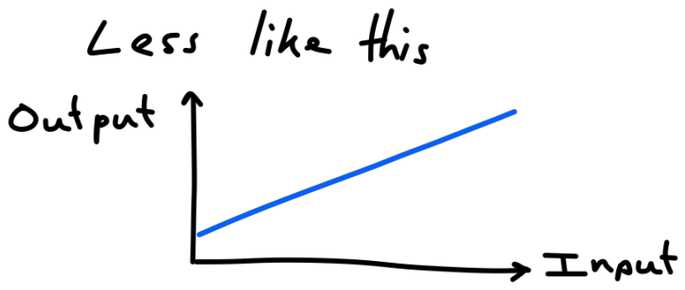
Session 7

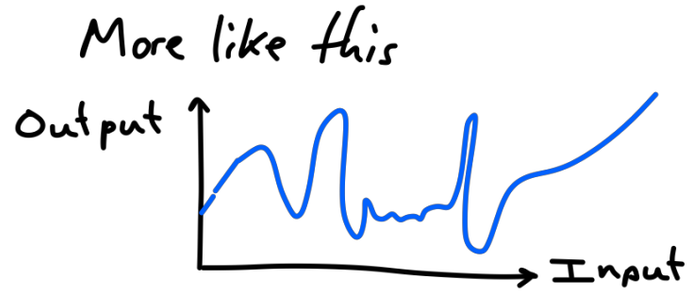
Students for AI

Problems motivating deep learning

Modelling complex functions

Useful input-output relationships are too complex to guess what model might fit them





sin wave?
high order polynomial?
gaussian?

Assuming any particular function as the model makes it biased

How can we hope to guess a function that can learn a sufficient model?

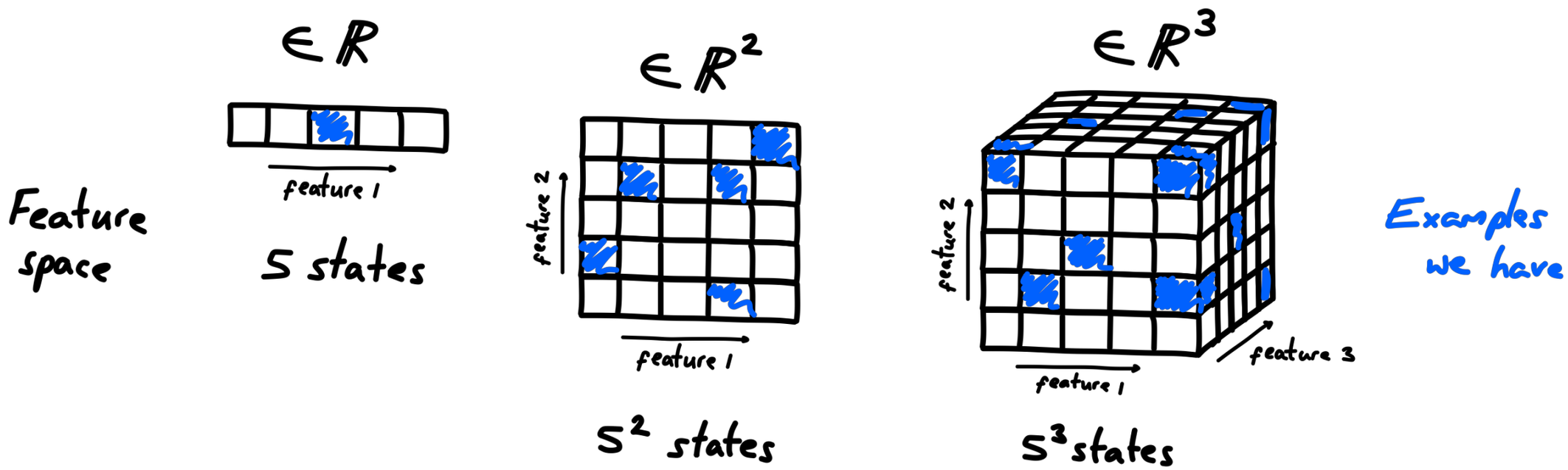
We need our model to be able to fit any shape

Problems motivating deep learning

The curse of dimensionality

As the number of dimensions (features) of a problem increase, there become exponentially more possible states that each example can take.

With more possible states, we need to see more examples to cover the space so that we can form a good representation of how an example at that point is mapped to an output.



Problems motivating deep learning

Abstraction

Us humans we don't pay attention to every individual raw input.

Low level feature		High level feature
Each photon that hits your retina		"I see a car"
Frequency and amplitude of sound over time	abstraction →	"I hear a voice"
Locations and pressures at each point on skin		"I feel the apple in my hand"

Forming abstract, hierarchical representations of low level data alleviates the problem of the curse of dimensionality.

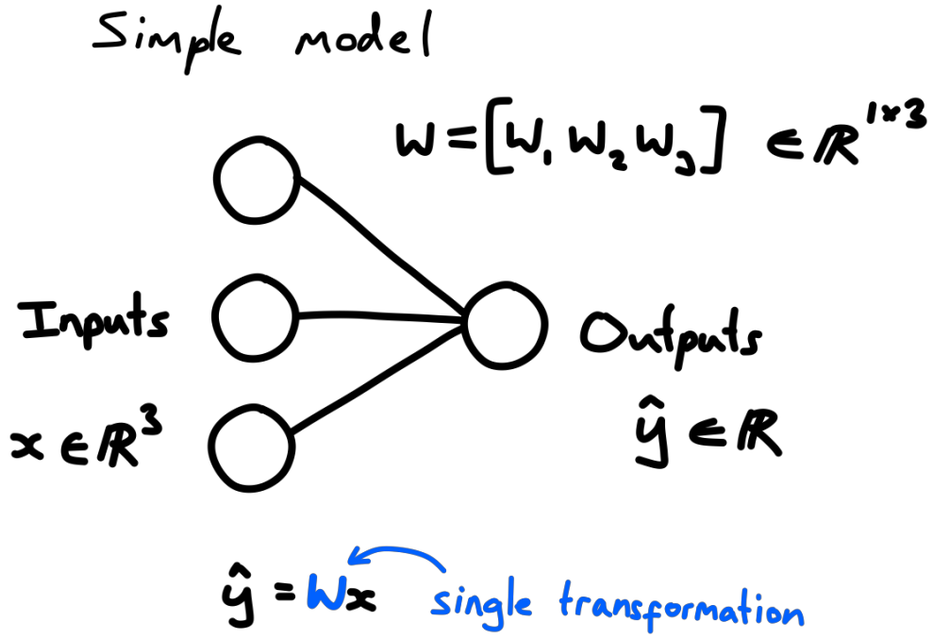
As such, we want our models to be able to make abstractions

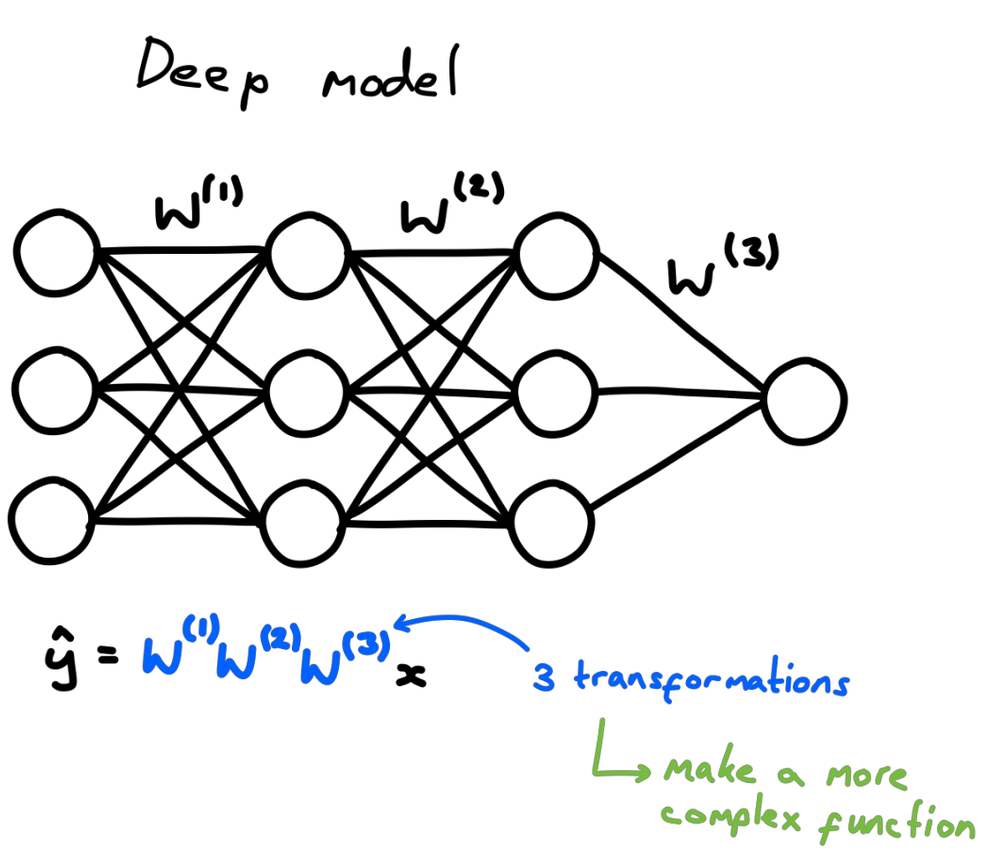
Modelling complex transformations

The models that we've seen so far only apply one function to transform the data

But we need to be able to model non-linear relationships

We can represent complex functions by repeatedly applying simple functions.

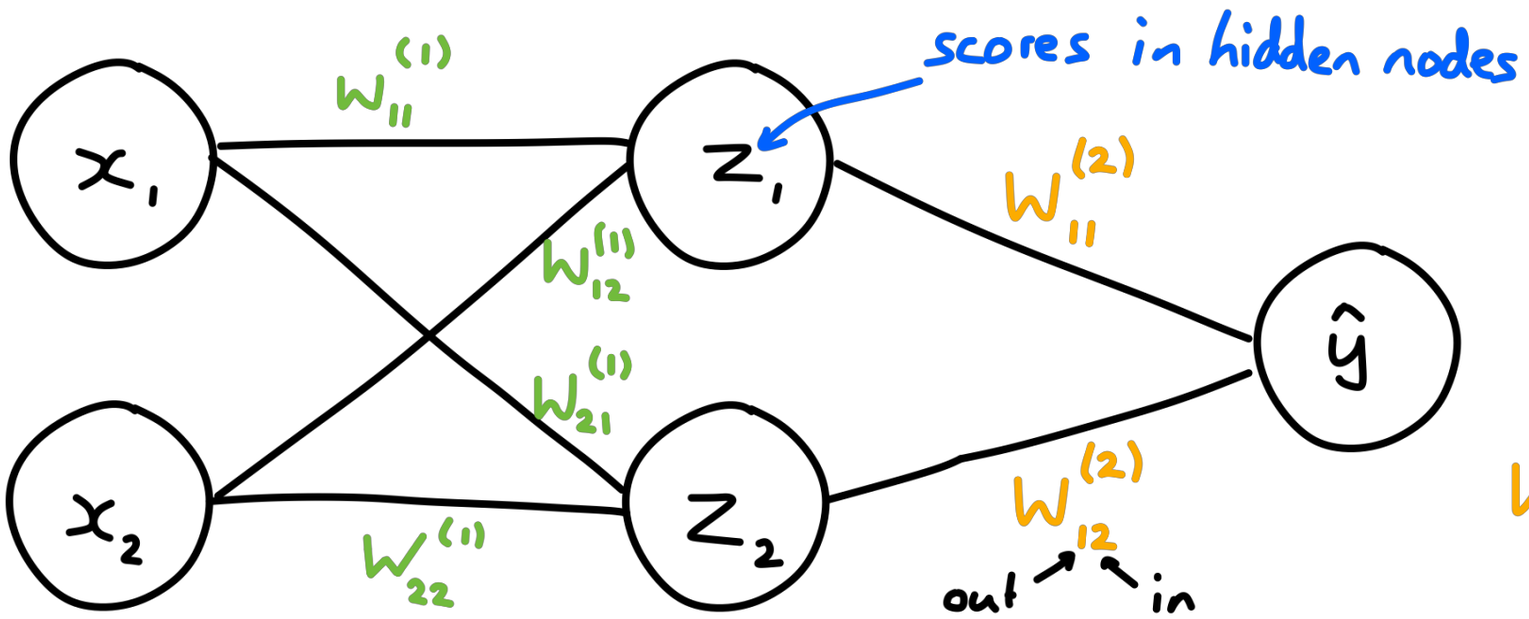




Linear factorisation

However...

...repeatedly applied linear transformations are equivalent to a single linear transformation



$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \end{bmatrix} \in \mathbb{R}^{2\times 2}$$

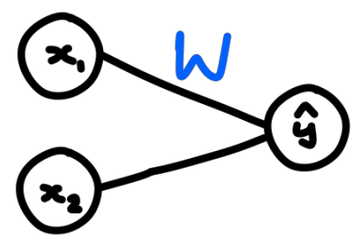
$$W^{(2)} = \begin{bmatrix} W_{11}^{(2)} & W_{12}^{(2)} \end{bmatrix} \in \mathbb{R}^{1\times 2}$$

$\uparrow$ out x in

$\in \mathbb{R}$

reduced to single transformation

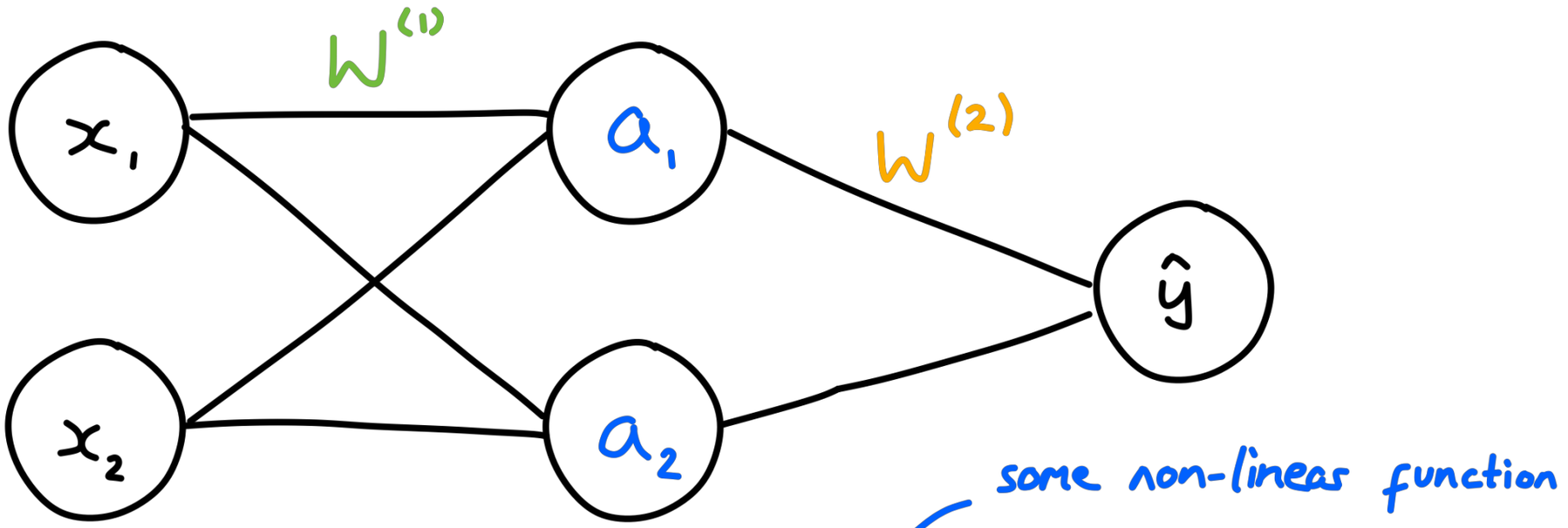
$$\hat{y} = W^{(2)}W^{(1)}x = \begin{bmatrix} W_{11}^{(2)} & W_{12}^{(2)} \end{bmatrix} \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} W_{11}^{(2)}W_{11}^{(1)} & W_{11}^{(2)}W_{12}^{(1)} \\ W_{12}^{(2)}W_{21}^{(1)} & W_{12}^{(2)}W_{22}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = Wx$$



In order to prevent this happening, we use **activation functions**...

Activation functions

Activation functions are **non-linear** transformations
that we apply to the input of each hidden node



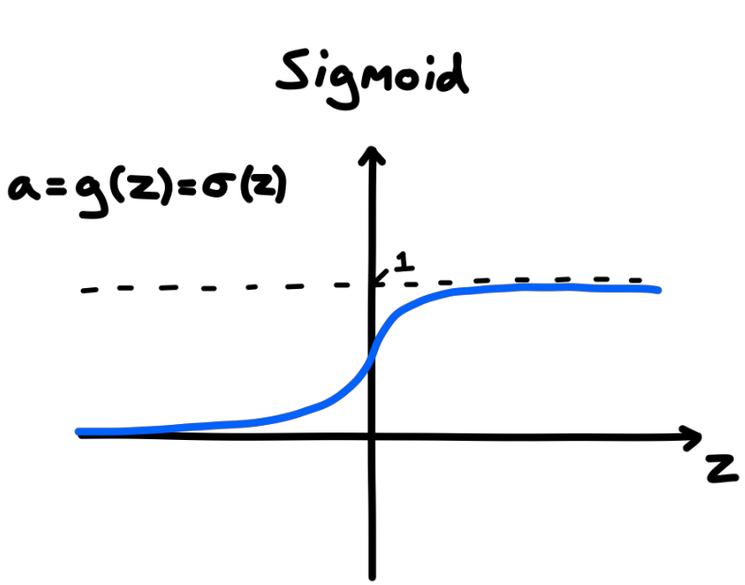
$$a = g(z) \longrightarrow \hat{y} = W^{(2)} g(W^{(1)} x)$$

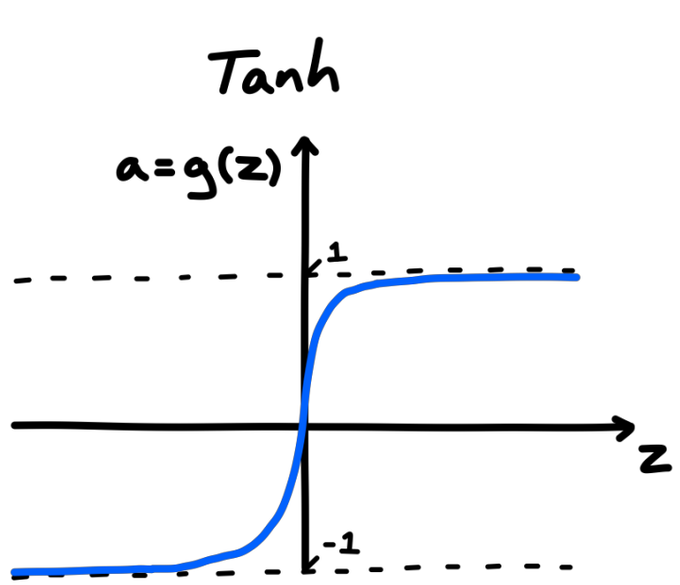
The function can now no longer be reduced to a single linear transformation

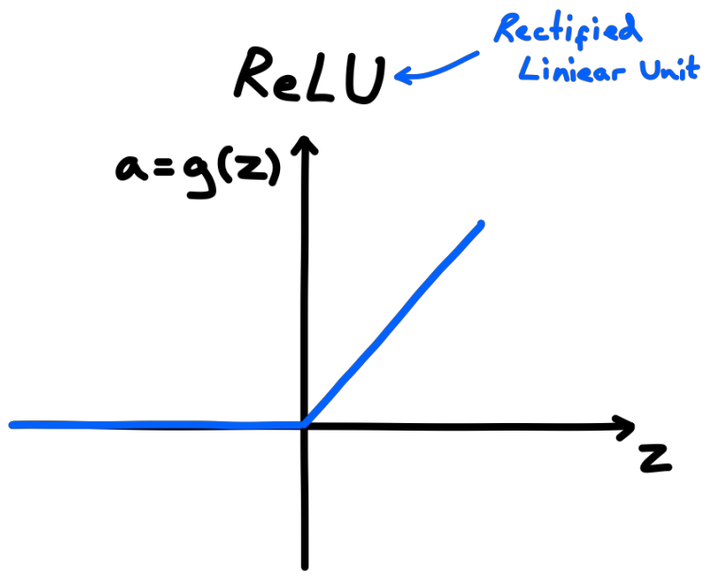
We have modelled a non-linear function

Activation functions

What functions do we use as activation functions?







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

$$\tanh(z) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

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

All an activation function needs to do, is to add nonlinearity so that we can learn nonlinear mappings

Activation functions that perform comparably to the best ones we know are so common as to be uninteresting

The neural network

$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \\ W_{41}^{(1)} & W_{42}^{(1)} \end{bmatrix}$$

weighting of input unit 1 to output unit 4

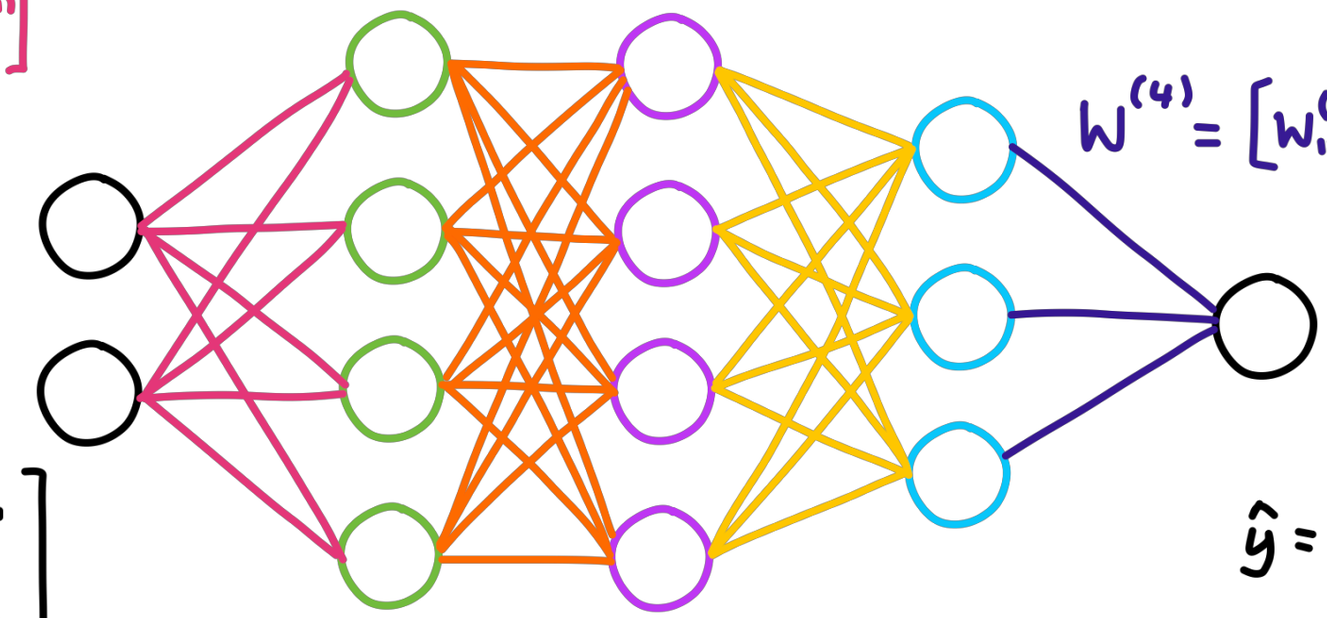
$$W^{(2)} = \begin{bmatrix} W_{11}^{(2)} & W_{12}^{(2)} & W_{13}^{(2)} & W_{14}^{(2)} \\ W_{21}^{(2)} & W_{22}^{(2)} & W_{23}^{(2)} & W_{24}^{(2)} \\ W_{31}^{(2)} & W_{32}^{(2)} & W_{33}^{(2)} & W_{34}^{(2)} \\ W_{41}^{(2)} & W_{42}^{(2)} & W_{43}^{(2)} & W_{44}^{(2)} \end{bmatrix}$$

input dims →

$$W^{(3)} = \begin{bmatrix} W_{11}^{(3)} & W_{12}^{(3)} & W_{13}^{(3)} & W_{14}^{(3)} \\ W_{21}^{(3)} & W_{22}^{(3)} & W_{23}^{(3)} & W_{24}^{(3)} \\ W_{31}^{(3)} & W_{32}^{(3)} & W_{33}^{(3)} & W_{34}^{(3)} \end{bmatrix}$$

↓ output dims

$W \in \mathbb{R}^{\text{outxin}}$



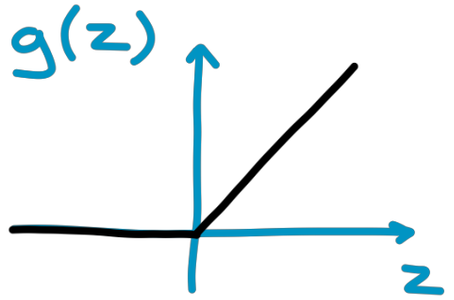
$$W^{(4)} = \begin{bmatrix} W_{11}^{(4)} & W_{12}^{(4)} & W_{13}^{(4)} \end{bmatrix}$$

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$\hat{y} = W^{(4)} a^{(3)}$$

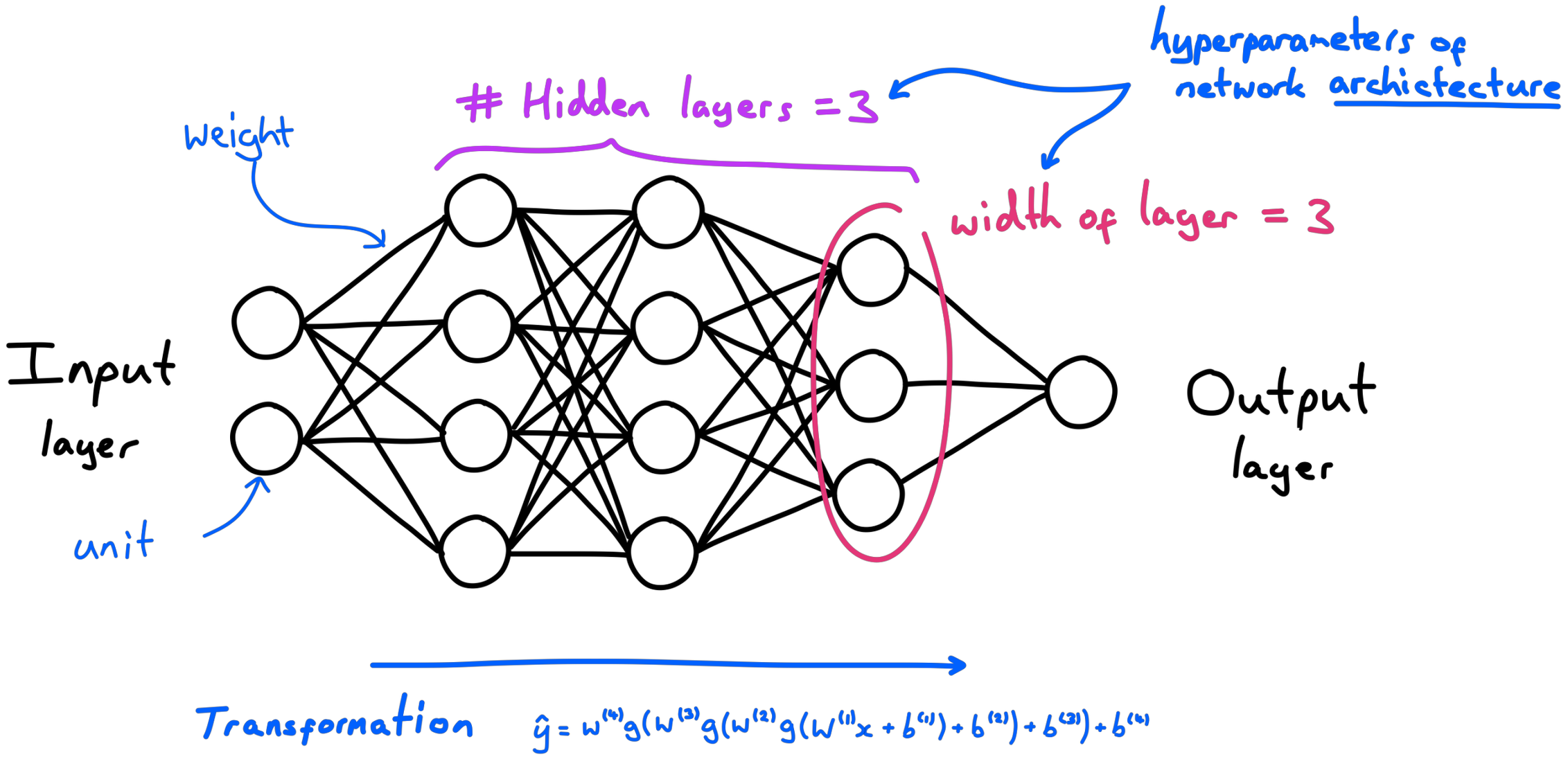
$$a^{(1)} = \begin{bmatrix} g(z_1^{(1)}) \\ \vdots \\ g(z_4^{(1)}) \end{bmatrix} \quad a^{(2)} = \begin{bmatrix} g(z_1^{(2)}) \\ \vdots \\ g(z_4^{(2)}) \end{bmatrix} \quad a^{(3)} = \begin{bmatrix} g(z_1^{(3)}) \\ \vdots \\ g(z_4^{(3)}) \end{bmatrix}$$

Activation function



$$\hat{y} = W^{(4)} g(W^{(3)} g(W^{(2)} g(W^{(1)} x + b^{(1)}) + b^{(2)}) + b^{(3)}) + b^{(4)}$$

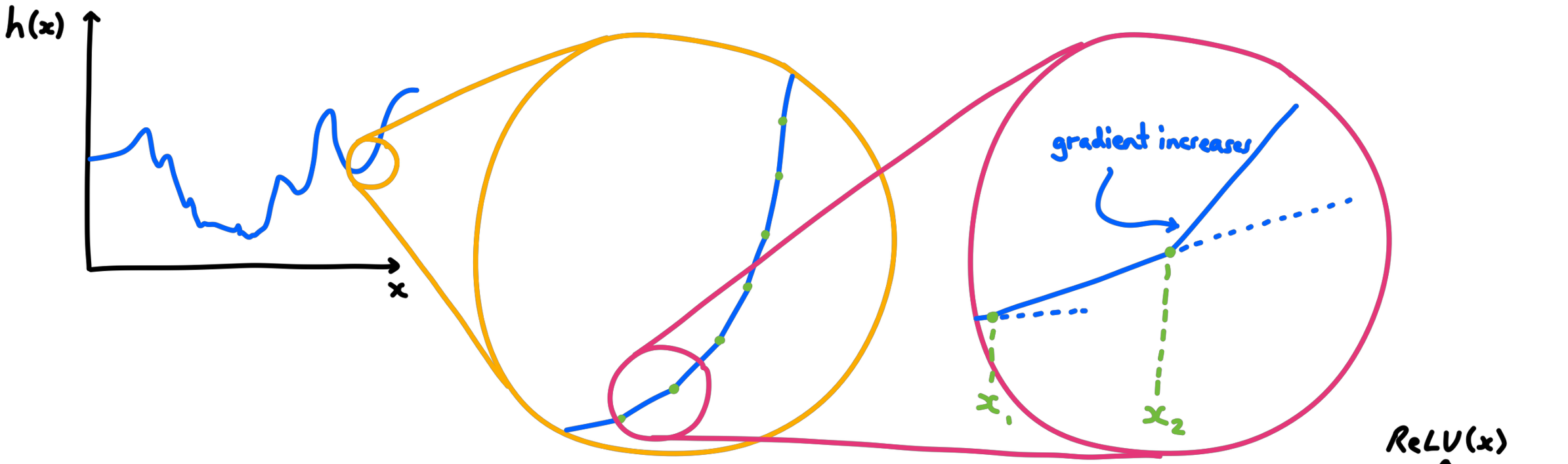
The neural network



Neural networks can be trained to model almost ANY function, no matter how ridiculous...

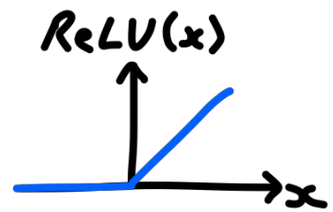
...but they need to be deep and wide enough to do so.

Simple demonstration of universal approximation



$$h(x) = \ldots + W_1 ReLU(x - x_1) + W_2 ReLU(x - x_2) + \ldots$$ ← shows how changing gradient can be implemented

Remember:



Each ReLU adds a linear contribution to the model

This contribution is not added over the whole domain, and hence the model is piece-wise linear

With an increasing number of hidden units, the model becomes increasingly continuous

Hierarchical feature learning

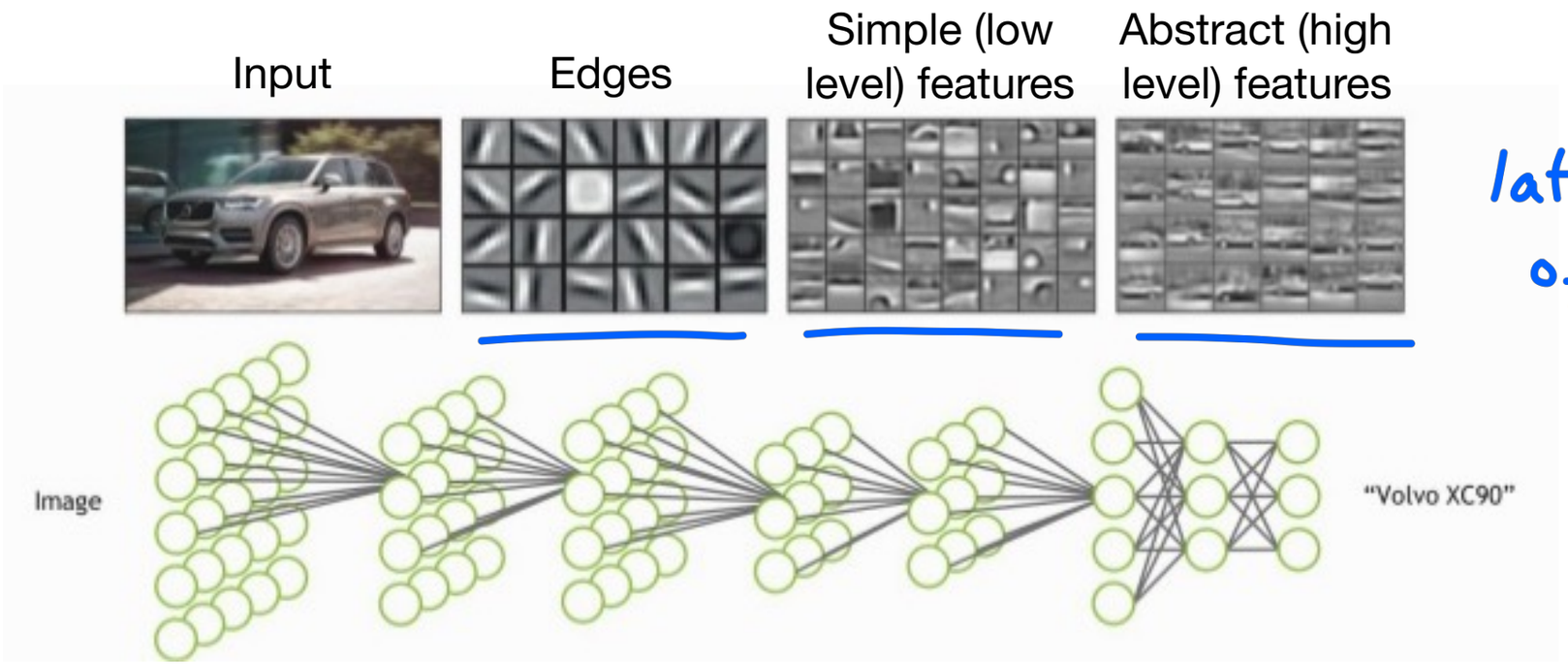
The output of each layer is latent features of the input.

Latent features means features that we don't observe directly

E.g. the edges in an image rather than the raw pixels

Because each layer builds on the features output by the previous layer, each layer can learn to output more abstract features than those input.

This allows neural networks to learn **hierarchical representations**



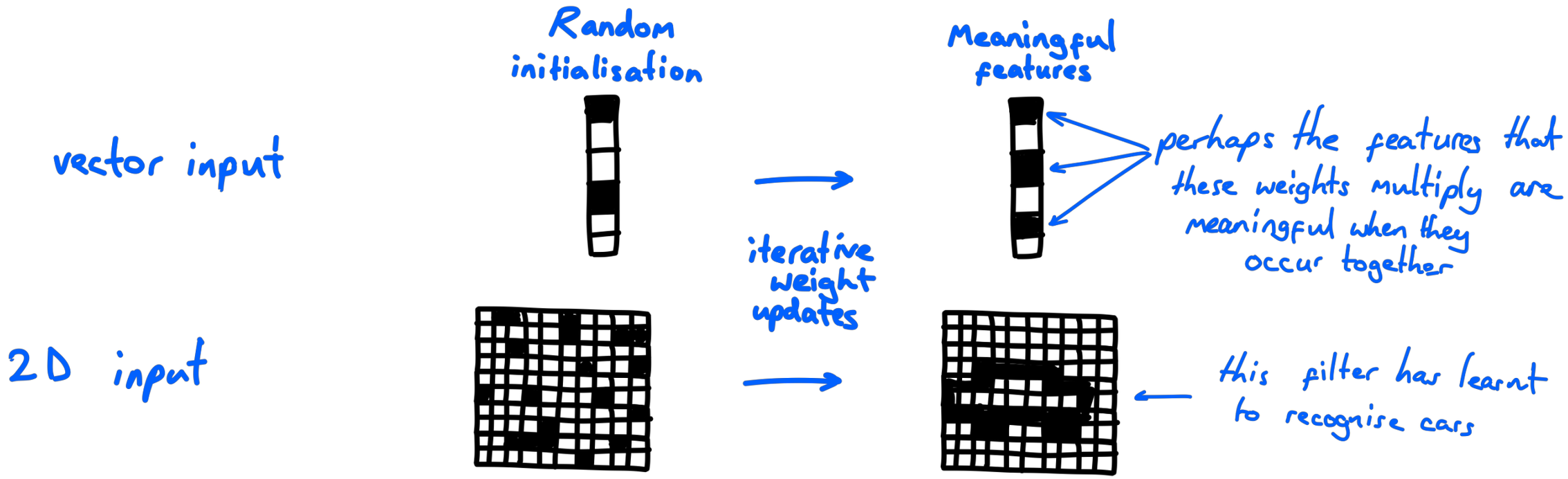
Automatic feature extraction

Gradient based optimisers (like SGD) update model parameters in a way that reduces the loss (improves performance) for the input examples

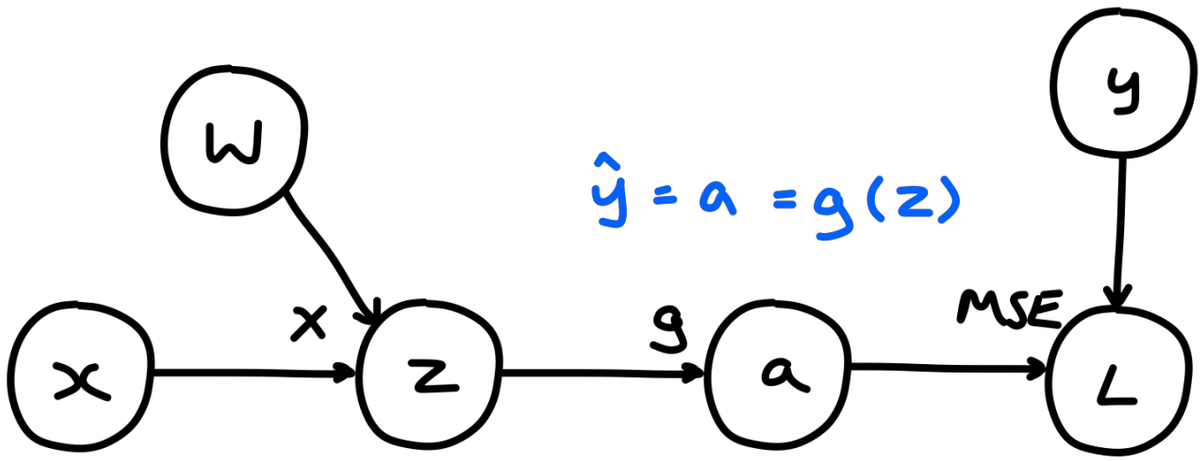
Intuitively, a model will perform better if it can identify meaningful features

So as the model improves, it does so by learning weights that activate units in the network when **meaningful features** are present in the input.

This is called **automatic feature extraction**



Computational graphs



This is a simple computational graph of a logistic regression model.

It shows the relationships between variables that appear in the model and makes it easy to compute derivatives, which we need for gradient based leaning.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial g}\frac{\partial g}{\partial z}\frac{\partial z}{\partial w}$$

We can obtain the derivatives of any variable with respect to another using the chain rule

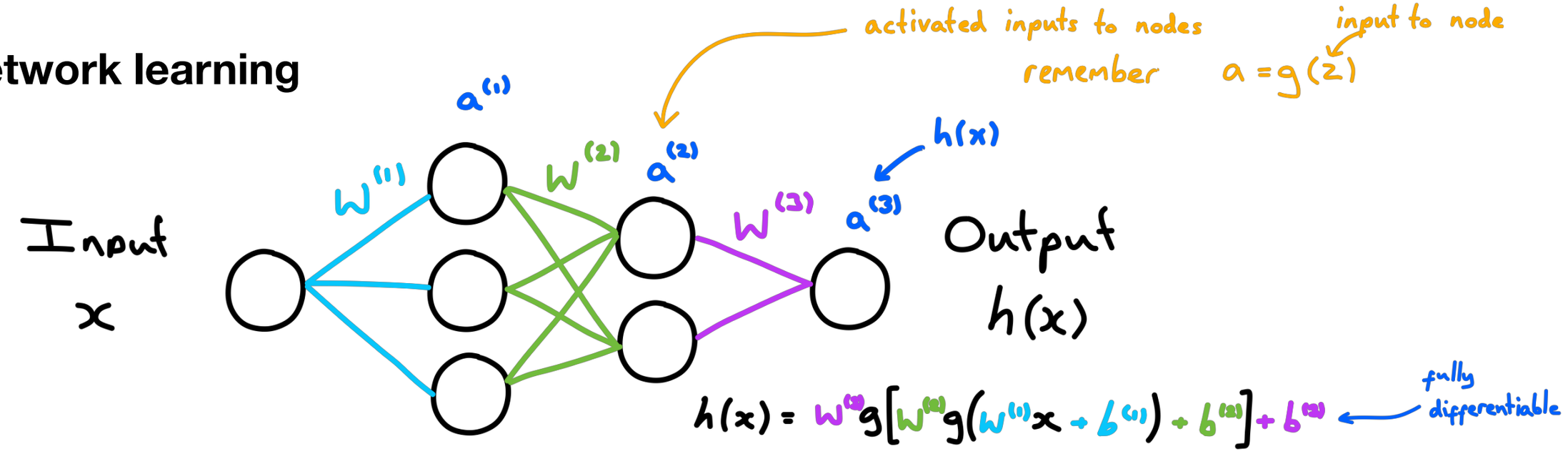
The derivative of each operation is simply a function of its inputs

e.g. $a = g(z) = \frac{1}{1+e^{-x}}$ $\frac{\partial a}{\partial z} = \frac{e^{-x}}{(1+e^{-x})^2} = g(x)(1-g(x))$

So we can just program functions that output the derivative of each particular operation in the graph.

Then apply them to any variable that appears in our graph, as long as we store how they were computed and from what variables.

Neural network learning



To learn, we need to find how each weight affects the error (find the derivatives)

$$\frac{\partial L}{\partial W^{(3)}} = \frac{\partial L}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial W^{(3)}}$$

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}}$$

$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(1)}}$$

Parts of the equation reappear

so we can reuse them to save unnecessary computation

We can propagate the error backwards through the model to find the gradients required for gradient based optimisers to make it learn

This is called **backpropagation**