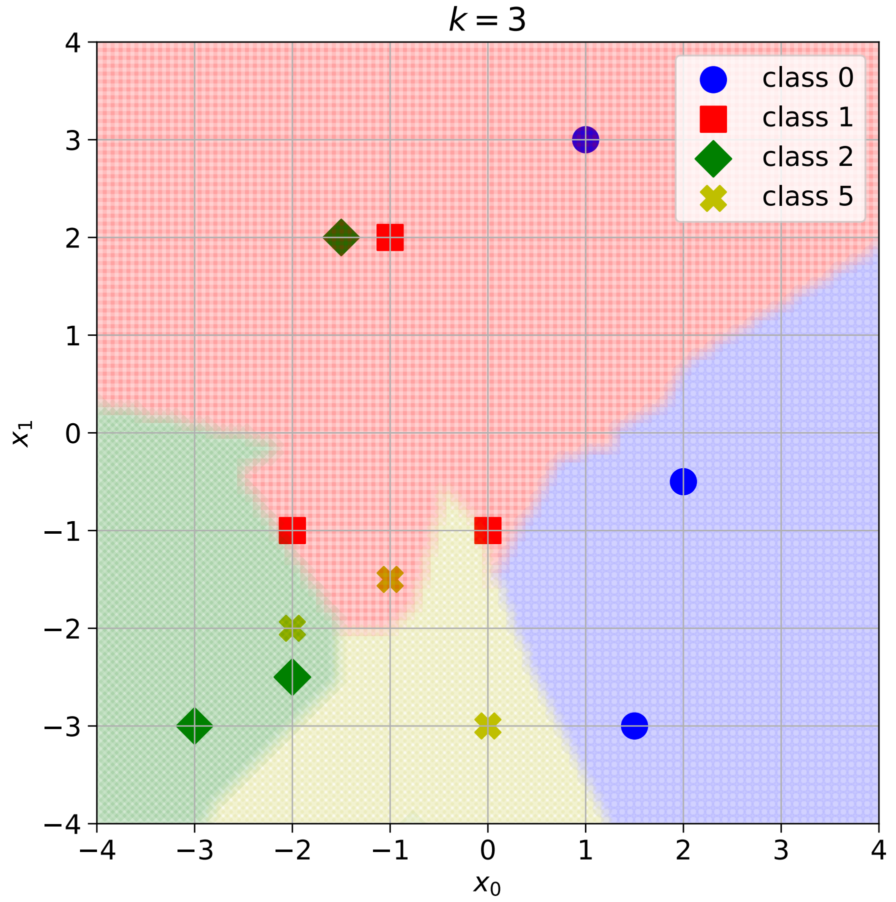
Data Analysis and Machine Learning 4

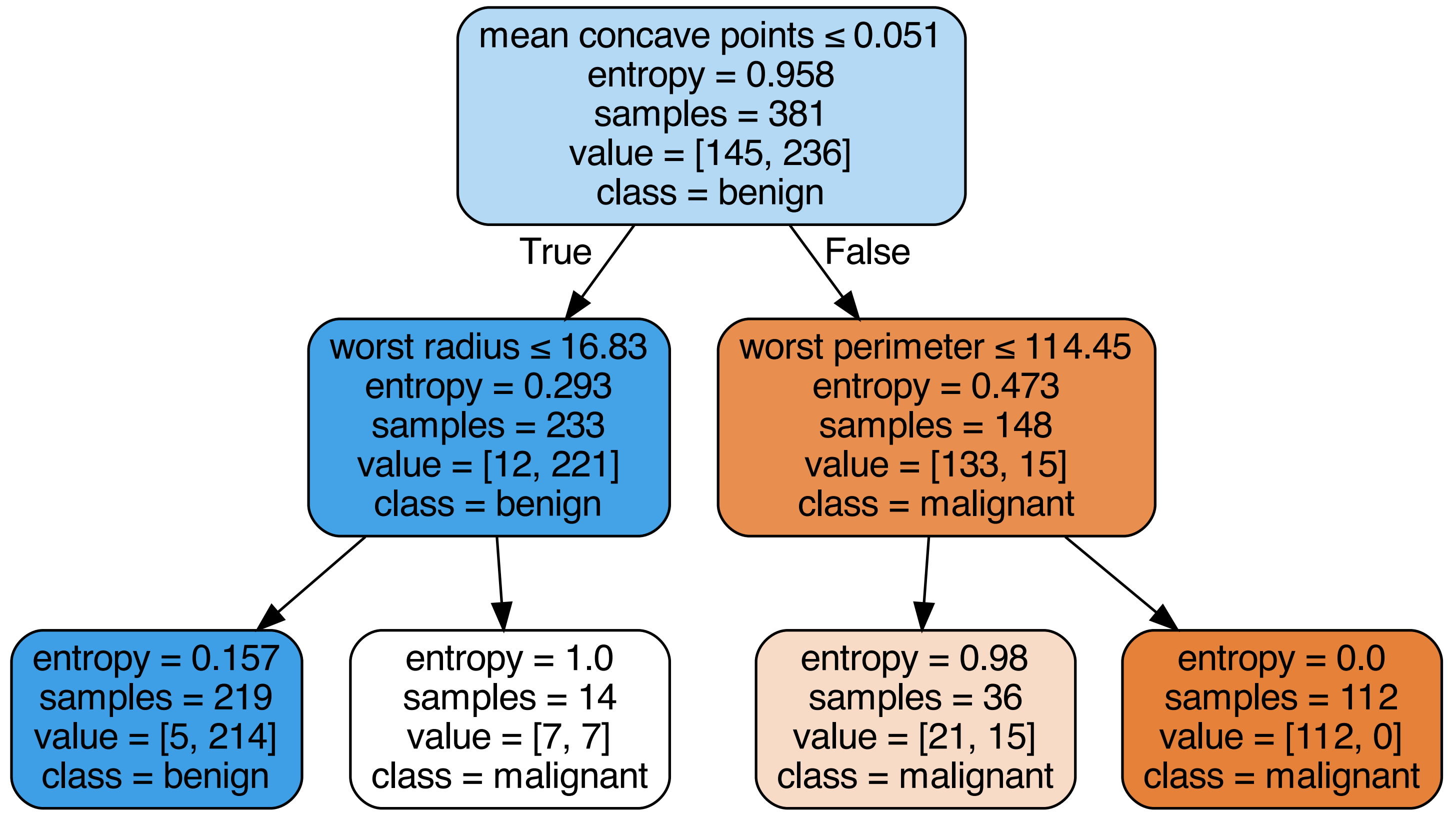
Week 9: Deep neural networks

Elliot J. Crowley, 20th March 2023

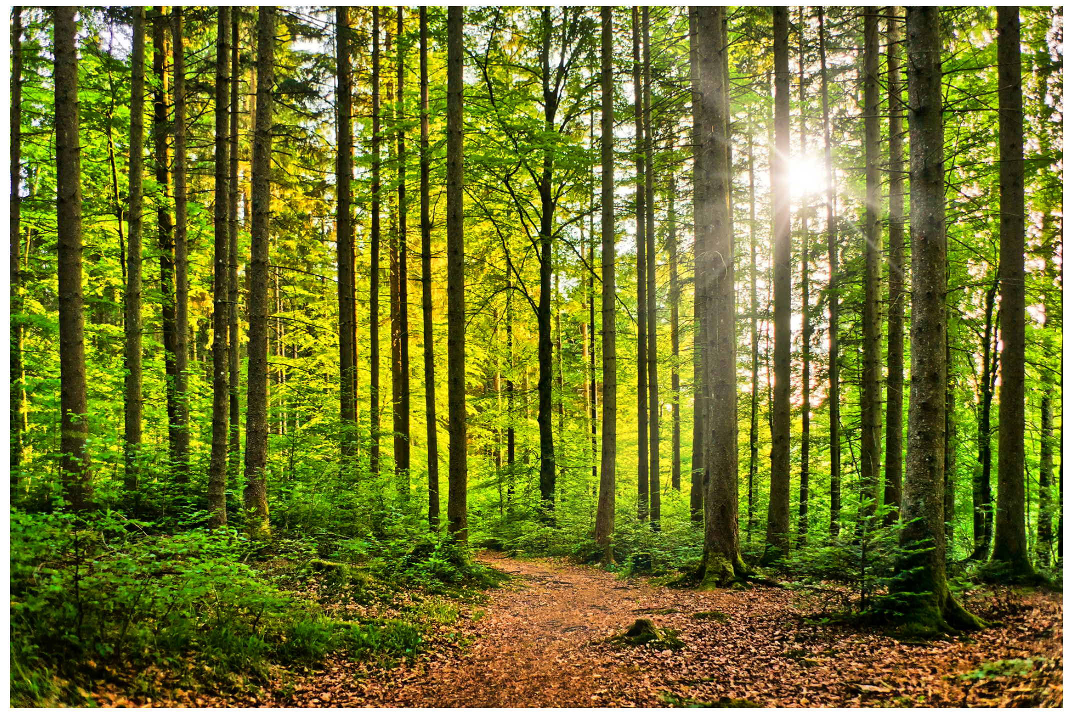
Recap

- We learnt about about k-NN and decision trees





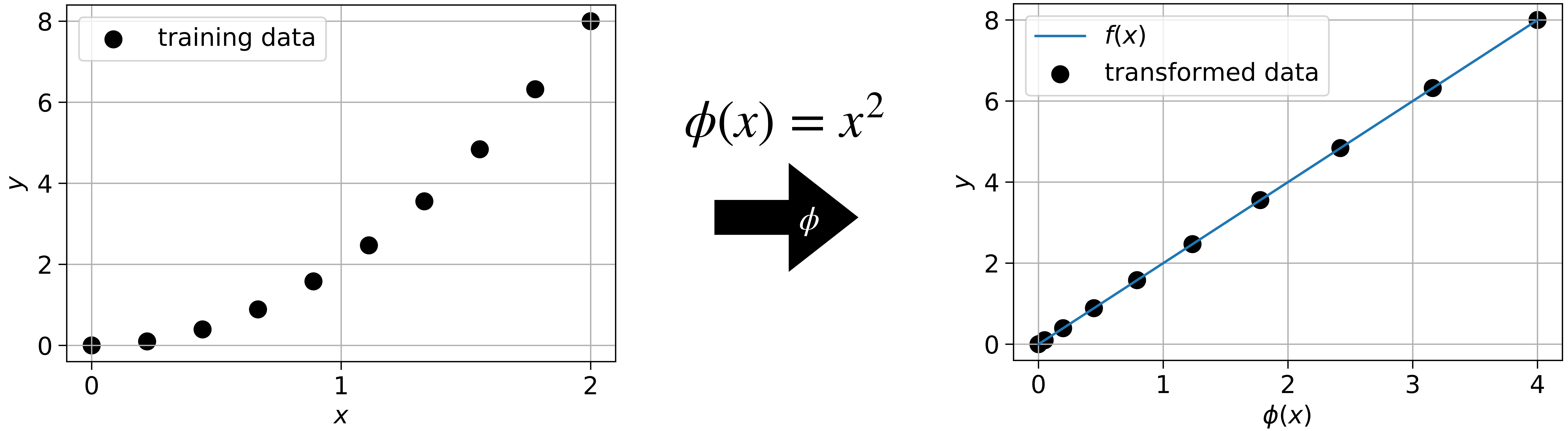
- We found out how an ensemble of decision trees called a random forest can be created using bagging and feature subsampling



Deep Learning

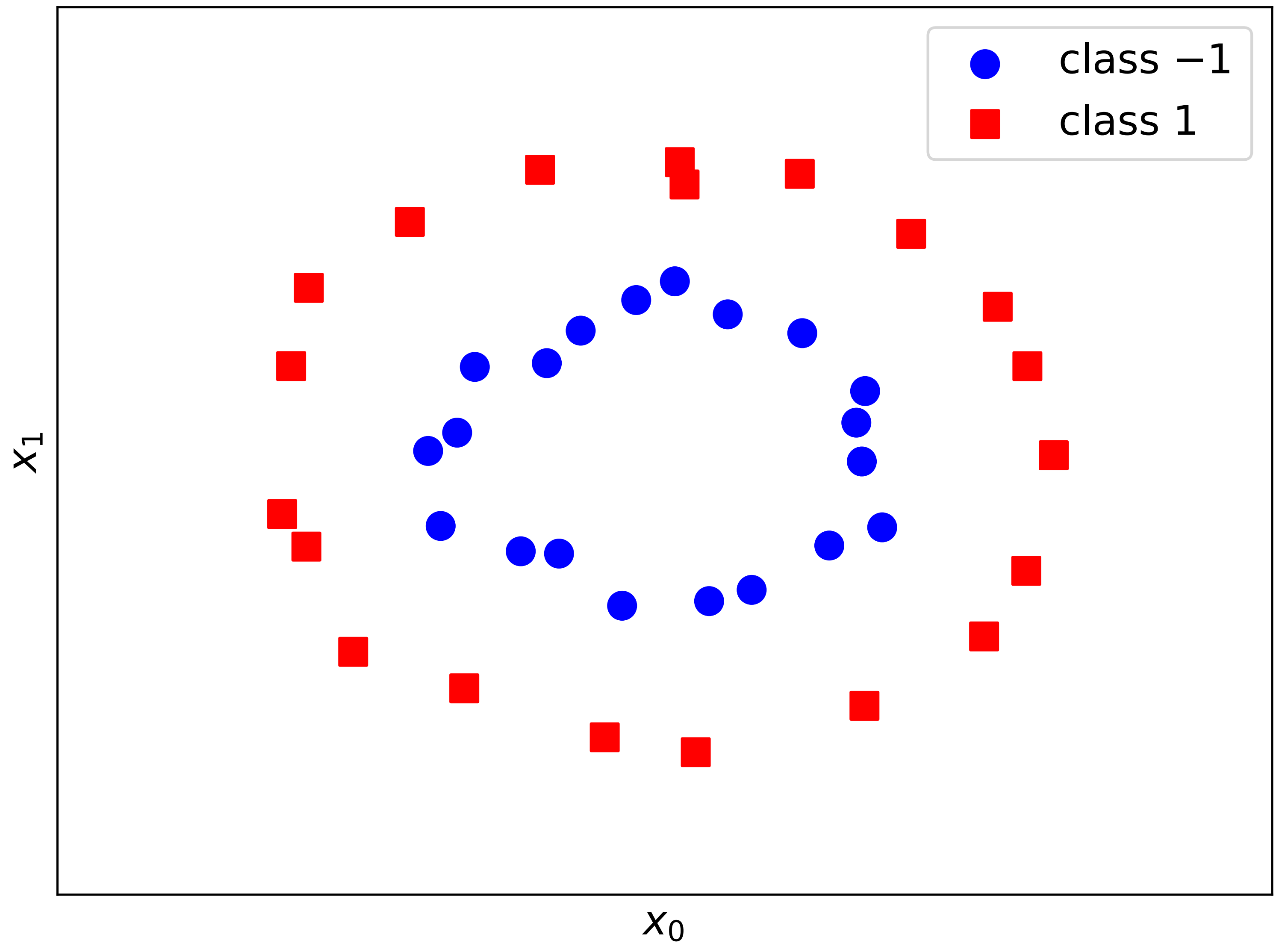
Linear regression

- Given training data $\{\mathbf{x}^{(n)}, y^{(n)}\}_{n=0}^{N-1}$ $(\mathbf{x} \in \mathbb{R}^D, y \in \mathbb{R}^1)$ we can learn a model:
    - $f(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x}) + b$ s.t. $y^{(n)} \approx f(\mathbf{x}^{(n)}) \, \forall n$
- We want ϕ to map the data to a space where we can fit a hyperplane to it

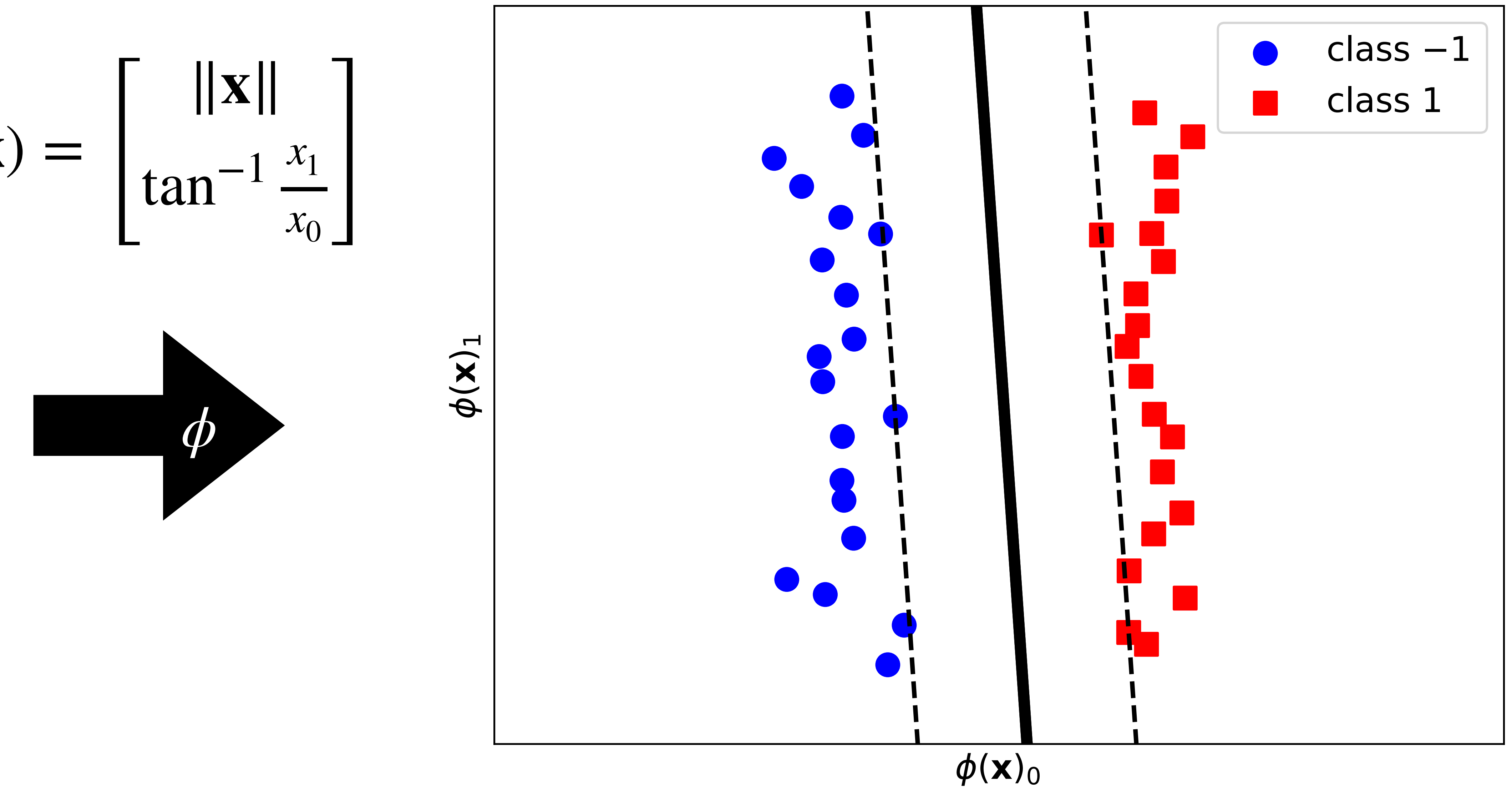


(Binary) linear classifiers

- Given training data $\{\mathbf{x}^{(n)}, y^{(n)}\}_{n=0}^{N-1}$ ($\mathbf{x} \in \mathbb{R}^D$, $y \in \{0,1\}$) we can learn a model:
    - $f(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x}) + b$ s.t. the hyperplane $f(\mathbf{x}) = 0$ separates the classes
- We want ϕ to map the data to a space where classes can be separated by a hyperplane



$$\phi(\mathbf{x}) = \begin{bmatrix} \|\mathbf{x}\| \\ \tan^{-1} \frac{x_1}{x_0} \end{bmatrix}$$



Multi-dimensional output

- What if we want to perform multi-class classification or regress to a multi-dimensional output $f(\mathbf{x}) \in \mathbb{R}^K$?

$f(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x}) + b$ with $\mathbf{w} \in \mathbb{R}^Z$ and $b \in \mathbb{R}^1$

becomes

$f(\mathbf{x}) = \mathbf{W}\phi(\mathbf{x}) + \mathbf{b}$ with with $\mathbf{W} \in \mathbb{R}^{Z \times K}$ and $\mathbf{b} \in \mathbb{R}^K$

- We will assume this is the default output from now on as it is more general

Feature learning

- There are plenty of off-the-shelf feature maps ϕ
- But how do we know if we've got the best one for a particular problem?
- Trying to design ϕ for a new problem can be tedious or impossible!
- What if we could learn ϕ directly from our training data?
- This is what **deep learning** entails. It's **feature learning!**

Deep (feedforward) neural networks (DNNs)

- These are non-linear models consisting of $\mathscr{L}$ functional layers

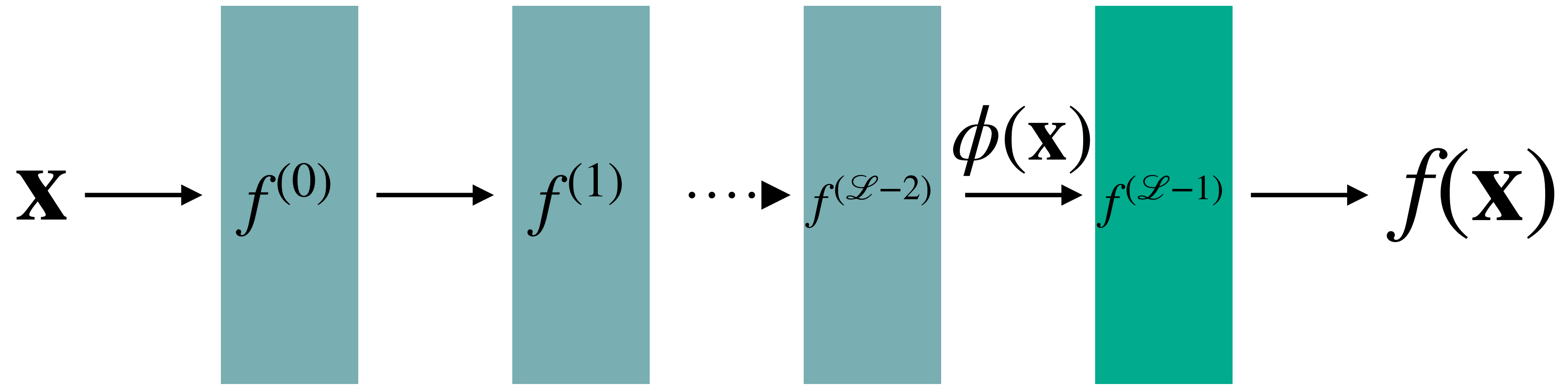
$$f(\mathbf{x}) = f^{(\mathscr{L}-1)} \circ f^{(\mathscr{L}-2)} \circ \ldots \circ f^{(1)} \circ f^{(0)}(\mathbf{x})$$

- The first $\mathscr{L} - 1$ layers form a **learnable** feature map $\phi(\mathbf{x}) \in \mathbb{R}^Z$. These are known as **hidden layers**

$$\phi(\mathbf{x}) = f^{(\mathscr{L}-2)} \ldots f^{(1)} f^{(0)}(\mathbf{x})$$

- The last layer is a linear transformation of the features (this can perform e.g. linear classification or linear regression)

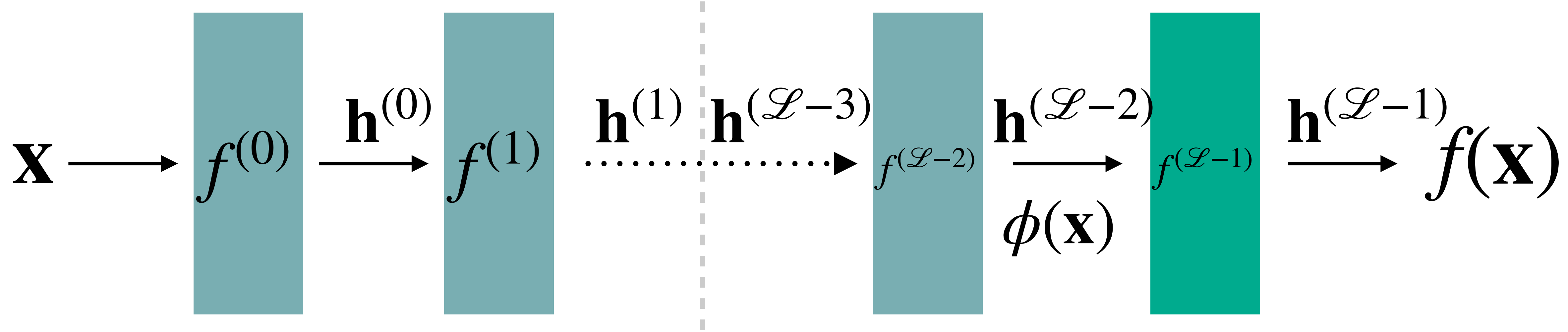
$$f(\mathbf{x}) = f^{(\mathscr{L}-1)}(\phi(\mathbf{x})) = \mathbf{W}^{(\mathscr{L}-1)}\phi(\mathbf{x}) + \mathbf{b}^{(\mathscr{L}-1)}$$



The multilayer perceptron (MLP)

- A DNN takes the form

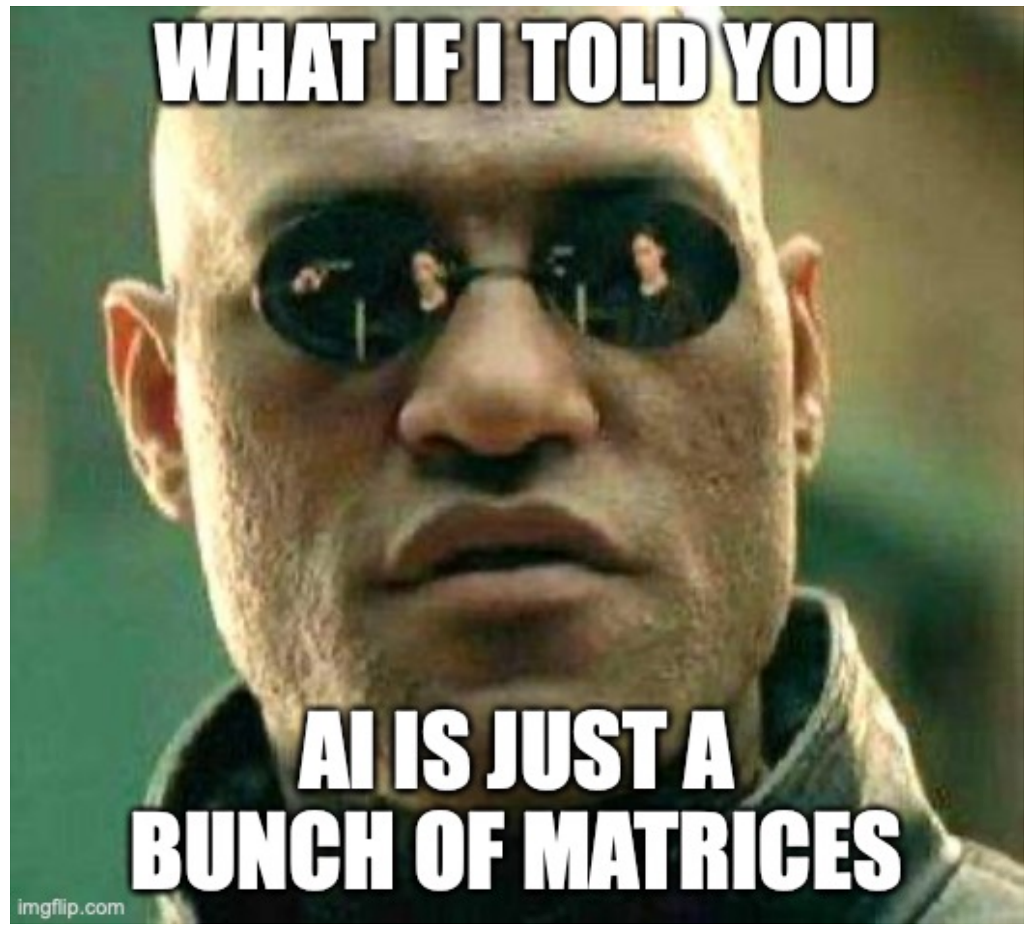
$$f(\mathbf{x}) = f^{(\mathscr{L}-1)} \circ f^{(\mathscr{L}-2)} \circ \ldots \circ f^{(1)} \circ f^{(0)}(\mathbf{x})$$



- An MLP is a network where each hidden layer output $\mathbf{h}^{(l)} \in \mathbb{R}^{H_l}$ is

$$\mathbf{h}^{(l)} = f^{(l)}(\mathbf{h}^{(l-1)}) = g(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}) \text{ for } l = 0,1,\ldots,\mathscr{L}-2$$

  - The layer input is the output of the previous layer $\mathbf{h}^{(l-1)} \in \mathbb{R}^{H_{l-1}}$
  - This undergoes a linear transformation
  - It then passes through a **non-linear** element-wise function g



g is called an **activation function** and layer outputs are called **activations**

Layers in an MLP are known as **fully-connected** or **dense** layers

Two layer MLP

- For a 2 layer MLP with $\mathbf{x} \in \mathbb{R}^D$ and $f(\mathbf{x}) \in \mathbb{R}^K$ we have:

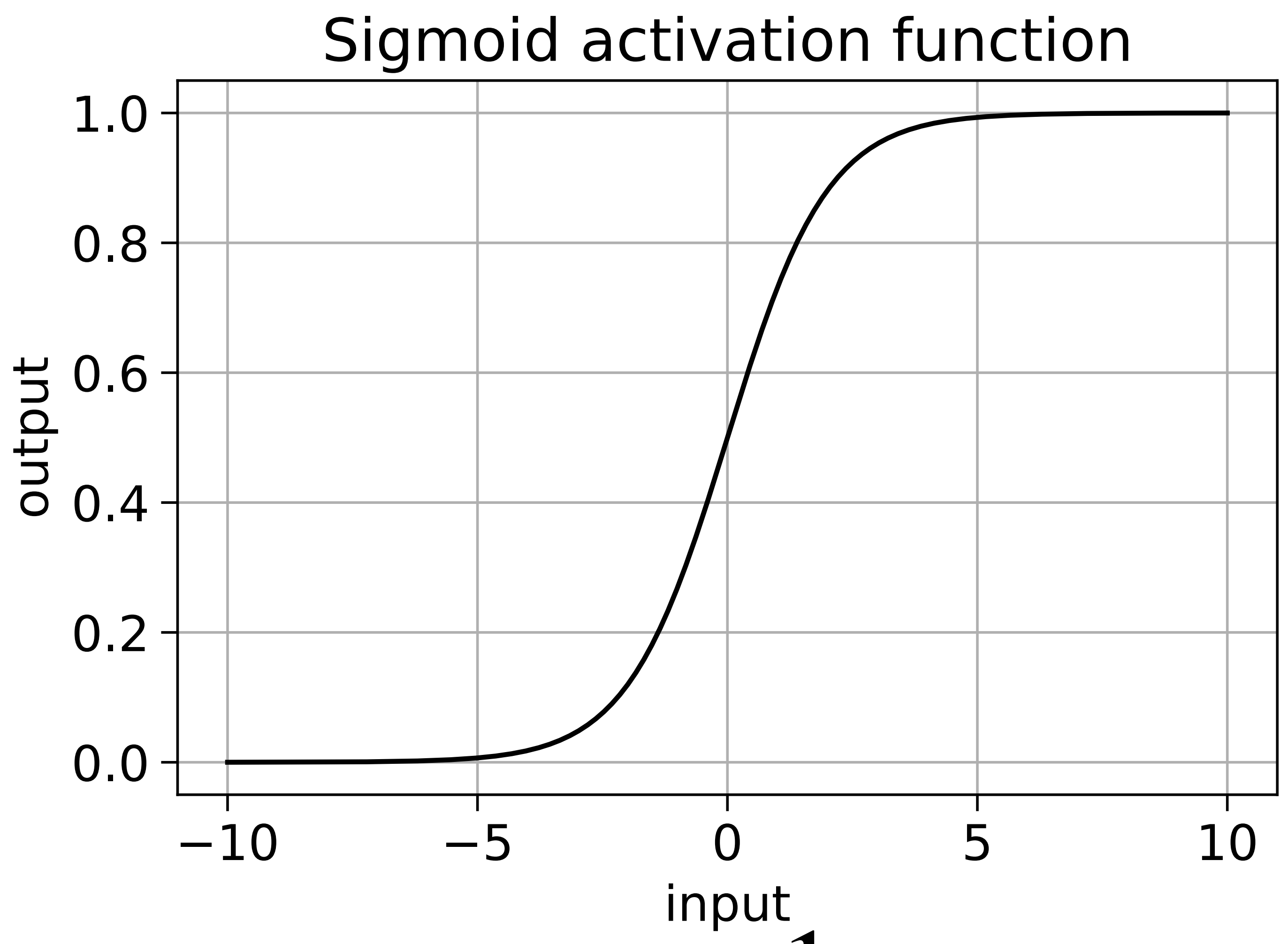
$$\phi(\mathbf{x}) = \mathbf{h}^{(0)} = g(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)})$$

$$f(\mathbf{x}) = \mathbf{h}^{(1)} = \mathbf{W}^{(1)}\mathbf{h}^{(0)} + \mathbf{b}^{(1)}$$

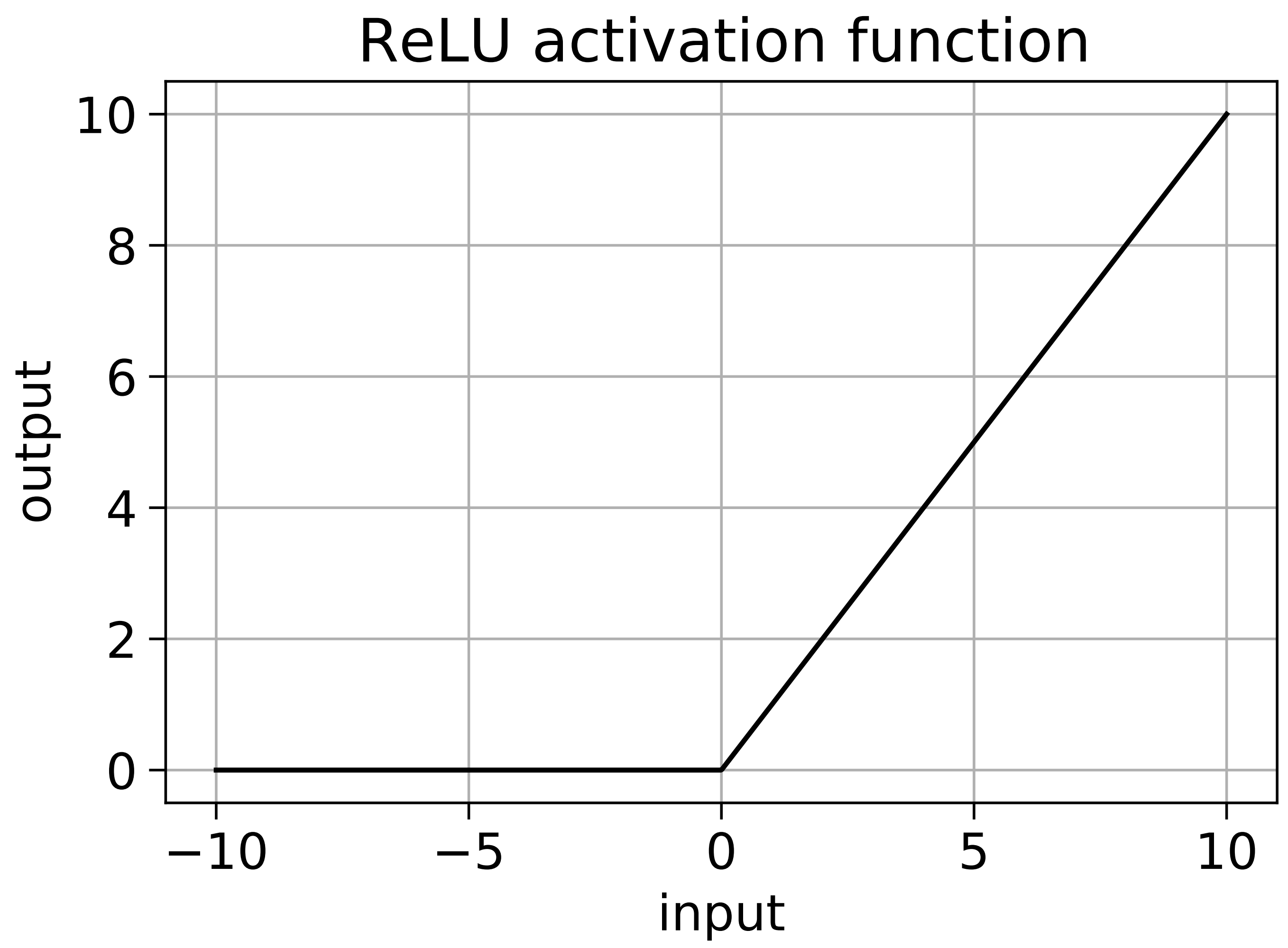
- We can write the whole MLP as $f(\mathbf{x}) = \mathbf{W}^{(1)}g(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}) + \mathbf{b}^{(1)}$
- We have to decide on the dimensionality of $\mathbf{h}^{(0)}$ (the **width** of the hidden layer)
- We also have to pick a non-linearity g

Activation functions

- These make our function non-linear. Without them an MLP collapses into a single linear transformation
- They are element-wise functions which means each element of the input vector is individually transformed



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



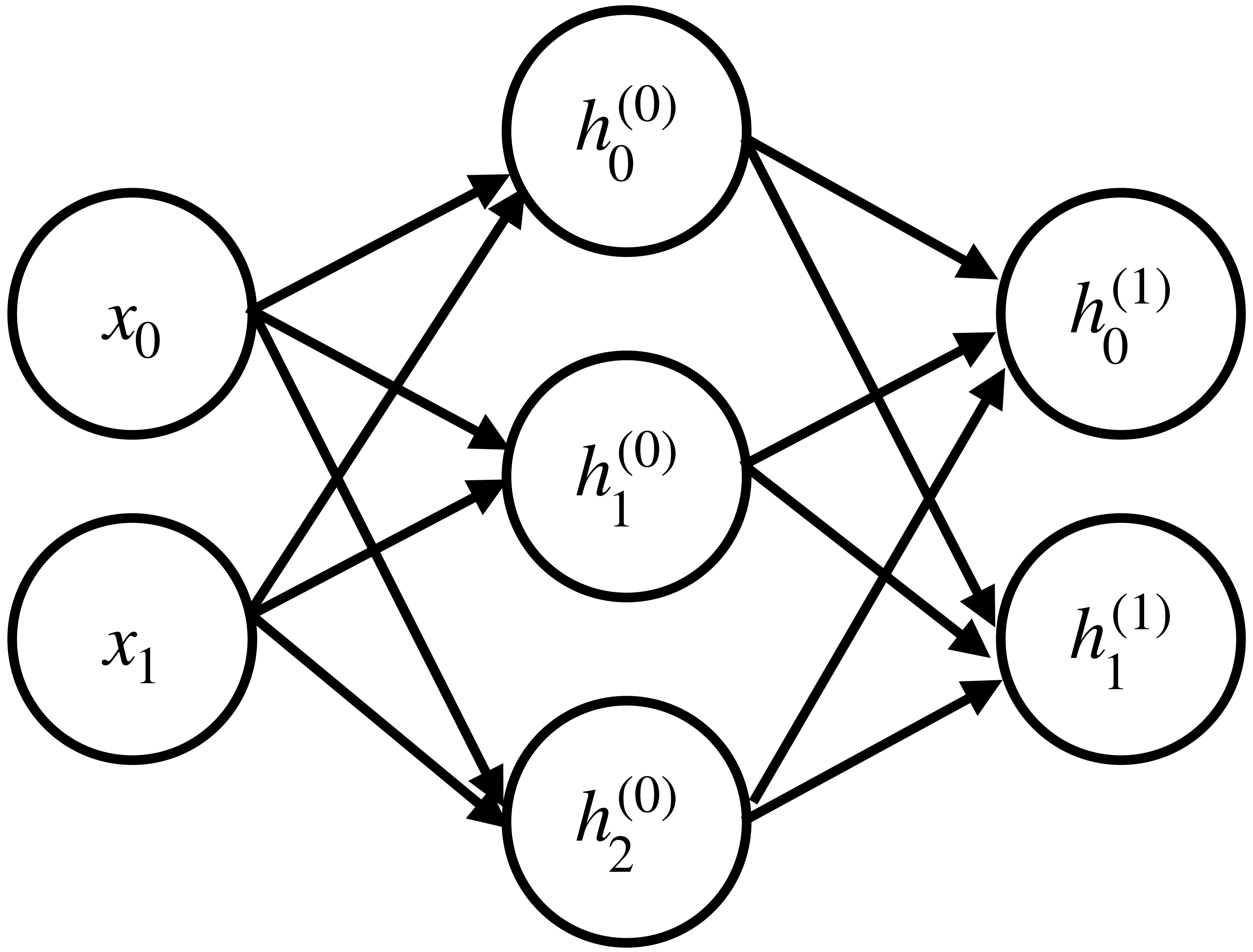
$$g(z) = \max(0, z)$$

ReLU or "rectified linear unit" is the most prevalent activation function and is what will we consider for the rest of this course

Alternate view of our MLP

$$\mathbf{h}^{(0)} = g(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)})$$

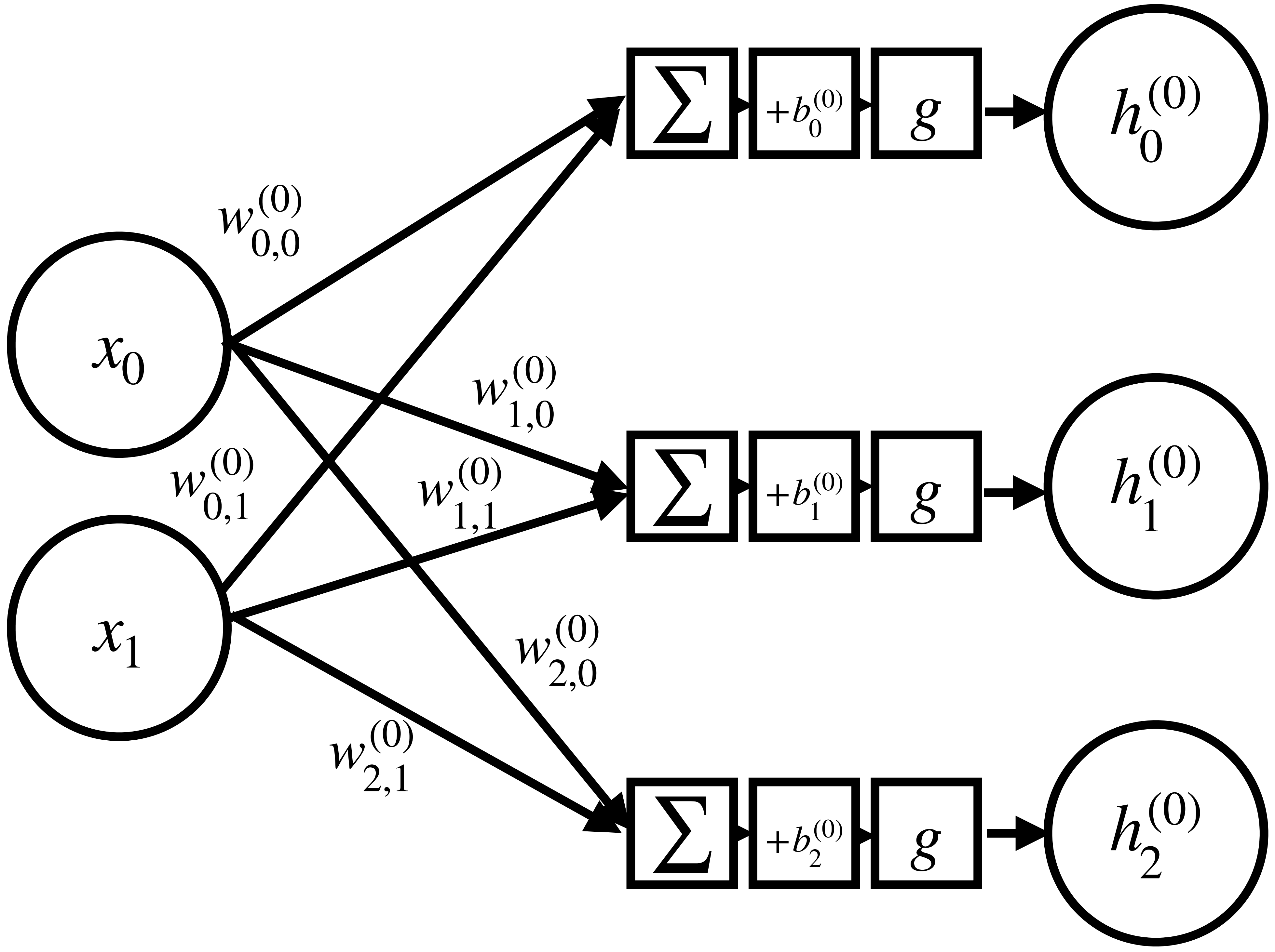
$$\mathbf{h}^{(1)} = \mathbf{W}^{(1)}\mathbf{h}^{(0)} + \mathbf{b}^{(1)}$$



- Sometimes you see MLPs drawn as graphs
- Here, the elements of $\mathbf{x} \in \mathbb{R}^2, \mathbf{h}^{(0)} \in \mathbb{R}^3, \mathbf{h}^{(1)} \in \mathbb{R}^2$ are represented by nodes
- Stuff is happening at the node inputs!
- It follows that $\mathbf{W}^{(0)} \in \mathbb{R}^{3\times2}$, $\mathbf{b}^{(0)} \in \mathbb{R}^3$
- And also that $\mathbf{W}^{(1)} \in \mathbb{R}^{2\times3}$, $\mathbf{b}^{(1)} \in \mathbb{R}^2$
- Sometimes these nodes are referred to as *neurons*

MLP: Layer 0

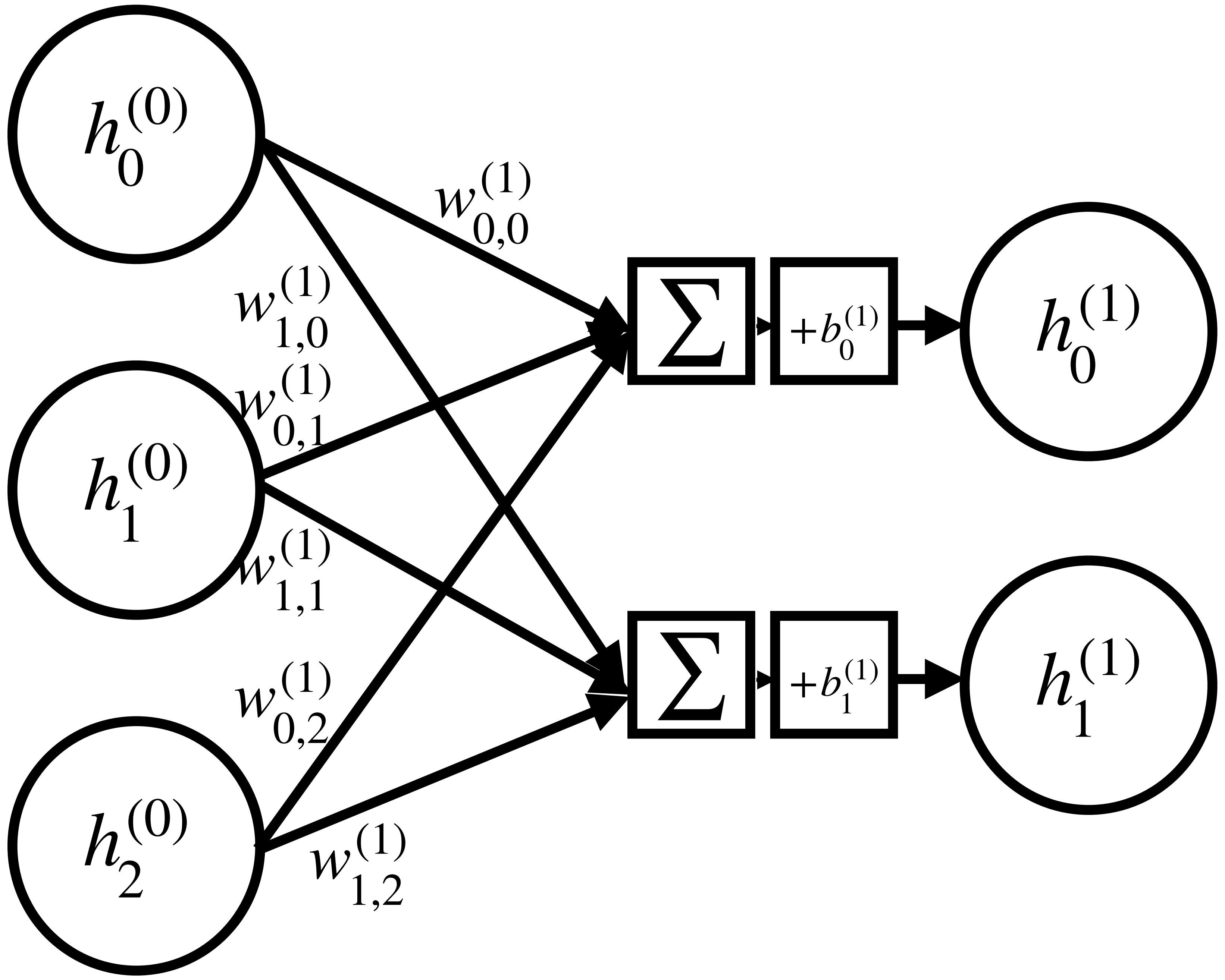
$$\mathbf{h}^{(0)} = \begin{bmatrix} h_0^{(0)} \\ h_1^{(0)} \\ h_2^{(0)} \end{bmatrix} = g(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}) = g(\begin{bmatrix} w_{0,0}^{(0)} & w_{0,1}^{(0)} \\ w_{1,0}^{(0)} & w_{1,1}^{(0)} \\ w_{2,0}^{(0)} & w_{2,1}^{(0)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} + \begin{bmatrix} b_0^{(0)} \\ b_1^{(0)} \\ b_2^{(0)} \end{bmatrix})$$



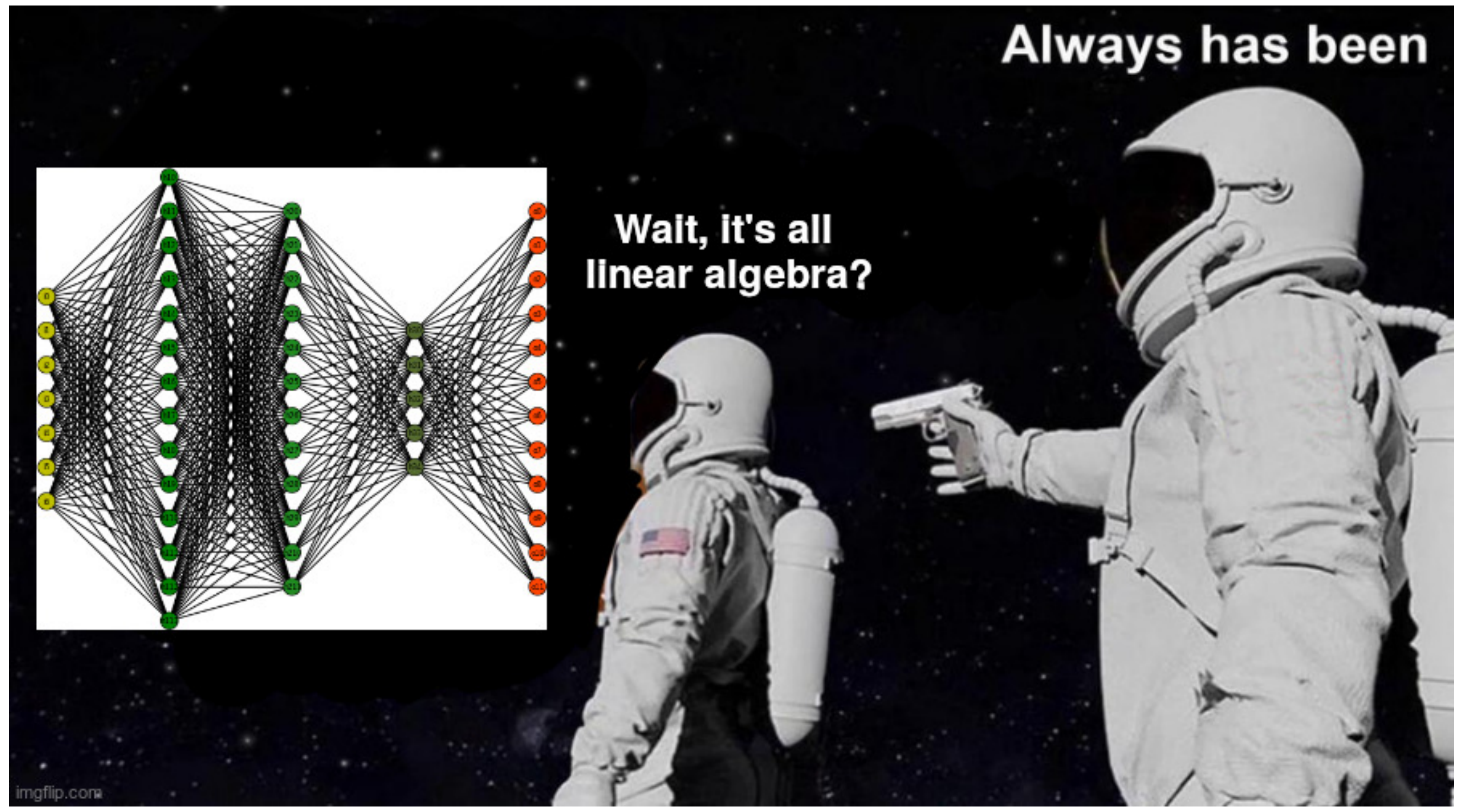
- Consider one of the *neurons of* $\mathbf{h}^{(0)}$
- It receives a weighted sum of the input neurons, to which a bias is added
- This quantity is known as a *pre-activation* and it goes into an activation function g
- If we are using ReLU activations $g(z) = \max(0,z)$ then the pre-activation must be positive to pass through
- If this happens we say that the neuron has **activated**

MLP: Layer 1

$$\mathbf{h}^{(1)} = \begin{bmatrix} h_0^{(1)} \\ h_1^{(1)} \end{bmatrix} = \mathbf{W}^{(1)}\mathbf{h}^{(\mathbf{0})} + \mathbf{b}^{(1)} = \begin{bmatrix} w_{0,0}^{(1)} & w_{0,1}^{(1)} & w_{0,2}^{(1)} \\ w_{1,0}^{(1)} & w_{1,1}^{(1)} & w_{1,2}^{(1)} \end{bmatrix} \begin{bmatrix} h_0^{(0)} \\ h_1^{(0)} \\ h_2^{(0)} \end{bmatrix} + \begin{bmatrix} b_0^{(1)} \\ b_1^{(1)} \end{bmatrix}$$



- There is no activation function for the last layer
- It’s just a matrix multiplied by a vector plus another vector
- The previous layer was the same + a non-linearity



https://www.reddit.com/r/machinelearningmemes/comments/hst89w/always_has_been/

Batch processing

- Consider a $N \times D$ dataset matrix $\mathbf{X} = \begin{bmatrix} \mathbf{x}^{(0)\top} & \mathbf{x}^{(1)\top} & \ldots \mathbf{x}^{(N-1)\top} \end{bmatrix}^\top$
- If we want to collect all the layer 0 outputs in a $N \times H_0$ matrix $\mathbf{H}^{(0)}$ then we can compute $\mathbf{H}^{(0)} = g(\mathbf{X}\mathbf{W}^{(0)\top} + \mathbf{b}^{(0)}\mathbf{1}^\top)$
- We can similarly collect all the layer 1 outputs in a $N \times Z$ matrix using $\mathbf{H}^{(1)} = \mathbf{H}^{(0)}\mathbf{W}^{(1)\top} + \mathbf{b}^{(1)}\mathbf{1}^\top$
- This is how it's done in PyTorch which is the deep learning framework we'll use

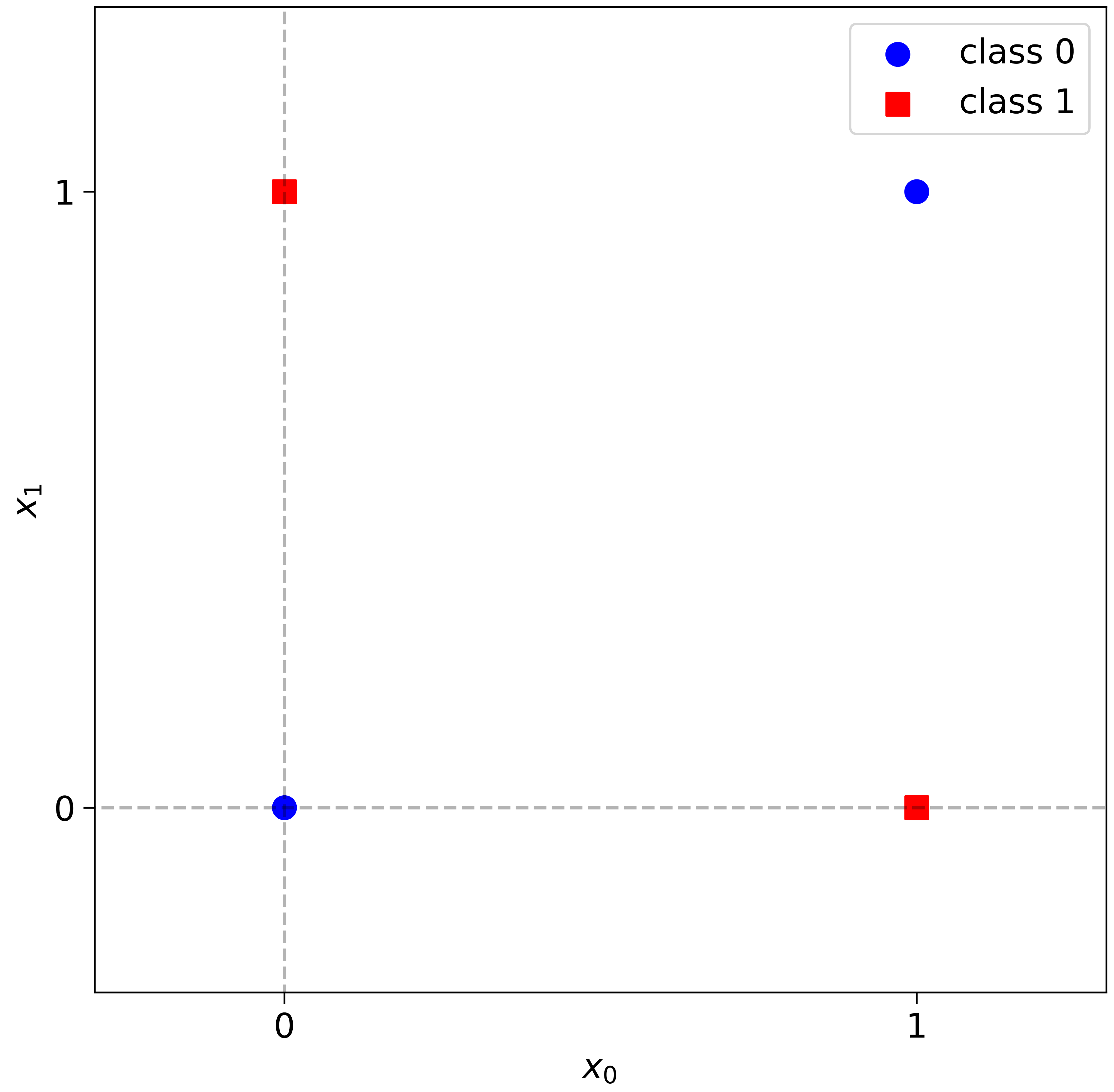
$\mathbf{1}$ is a vector of ones the same size as whatever it is being multiplied by :)

Binary classification with a 2 layer MLP

- This data is not linearly separable
- We will run through how a 2 layer MLP can deal with this in batch
- $\mathbf{H}^{(0)} = g(\mathbf{X}\mathbf{W}^{(0)\top} + \mathbf{b}^{(0)}\mathbf{1}^\top)$
- $\mathbf{H}^{(1)} = \mathbf{H}^{(0)}\mathbf{W}^{(1)\top} + \mathbf{b}^{(1)}\mathbf{1}^\top$
- Rows of $\mathbf{H}^{(0)}$ are feature vectors
- Rows of $\mathbf{H}^{(1)}$ are the corresponding $f(\mathbf{x})$ for each feature vector

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\mathbf{y} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$



For binary classification $\mathbf{H}^{(1)}$ is a vector, $\mathbf{W}^{(1)}$ is a vector, and $\mathbf{b}^{(1)}$ is a scalar but I'm keeping the more general notation for multi-class

This is equivalent to learning XOR. This example was based on https://www.deeplearningbook.org/contents/mlp.html 6.1

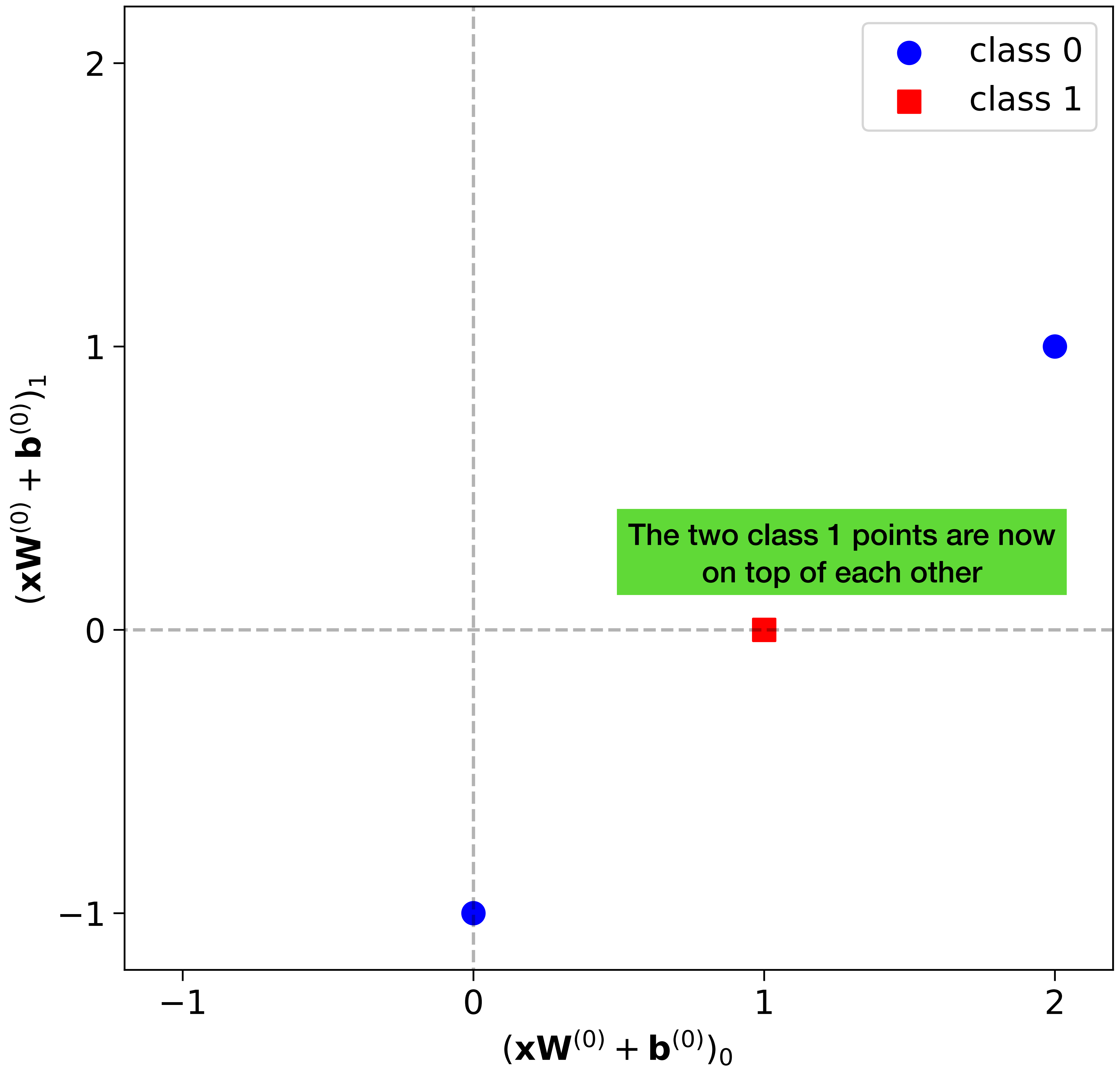
Layer 0: pre-activations

$$\mathbf{X} \longrightarrow \boxed{@\,\mathbf{W}^{(0)\top} + \mathbf{b}^{(0)}\mathbf{1}^\top} \longrightarrow \mathbf{X}\mathbf{W}^{(0)\top} + \mathbf{b}^{(0)}\mathbf{1}^\top$$

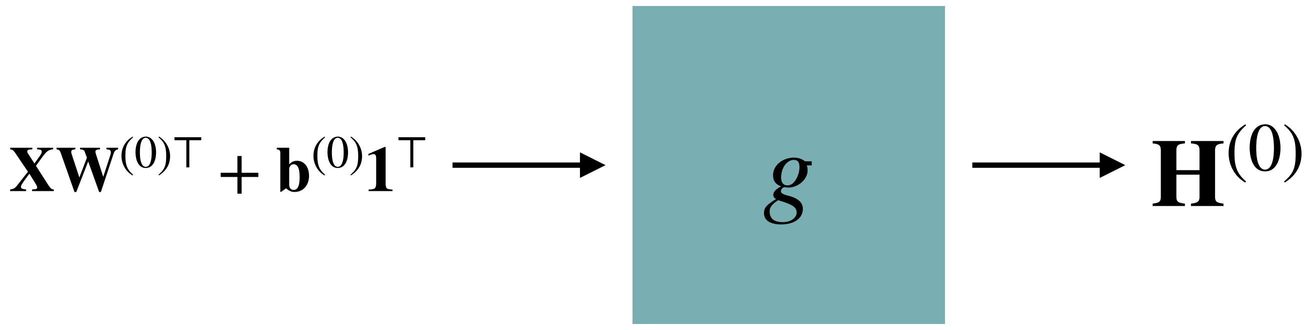
- We will use ReLU for g and $H_0 = 2$
- Layer 0 is $\mathbf{H}^{(0)} = g(\mathbf{X}\mathbf{W}^{(0)\top} + \mathbf{b}^{(0)}\mathbf{1}^\top)$
- Consider the *pre-activations* $\mathbf{X}\mathbf{W}^{(0)\top} + \mathbf{b}^{(0)}\mathbf{1}^\top$ given the following:

$$\mathbf{W}^{(0)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \mathbf{b}^{(0)} = [0 \quad -1]^\top$$

$$\mathbf{X}\mathbf{W}^{(0)\top} + \mathbf{b}^{(0)}\mathbf{1}^\top = \begin{bmatrix} 0 & -1 \\ 2 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$$

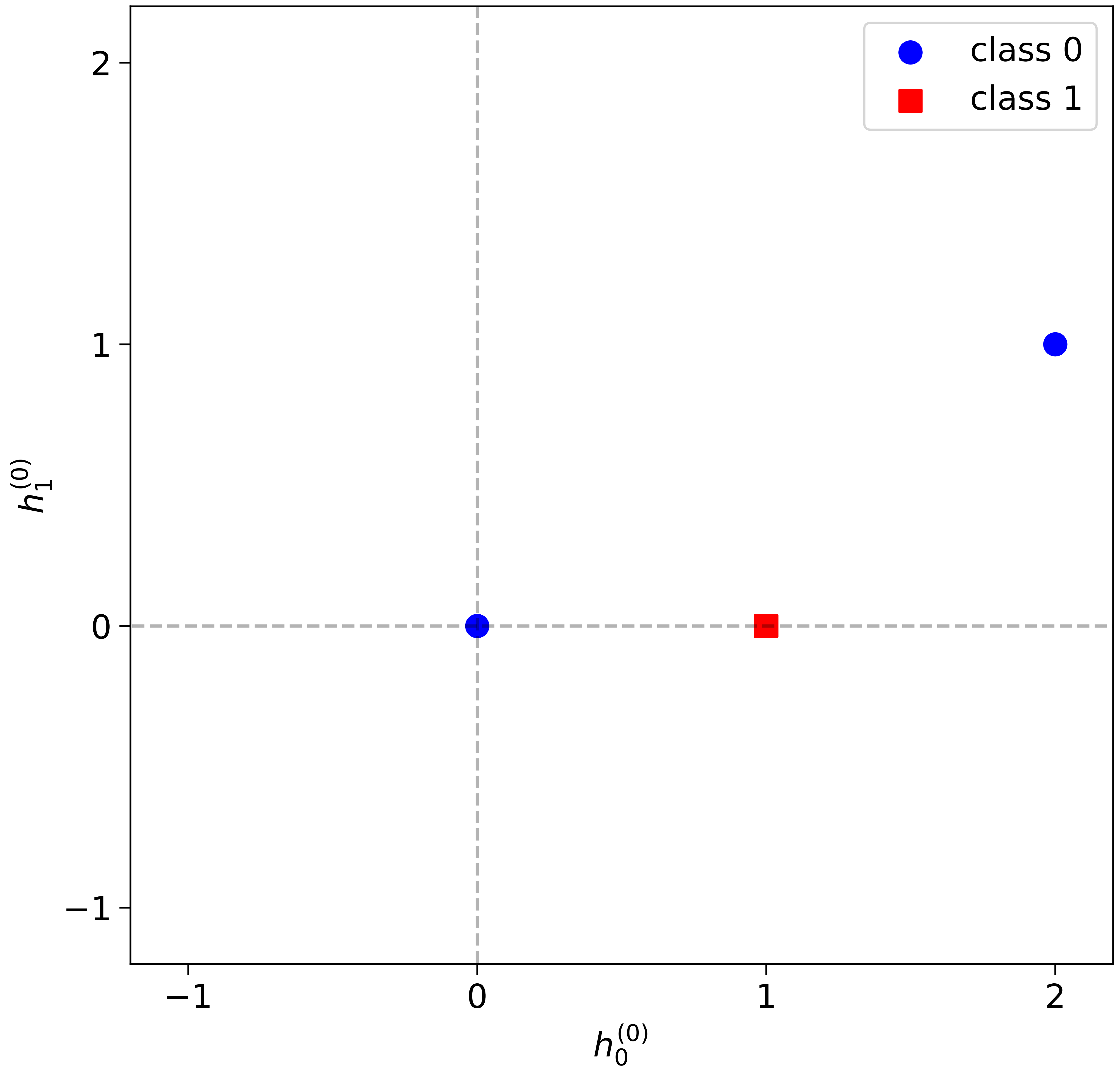


Layer 0: activations

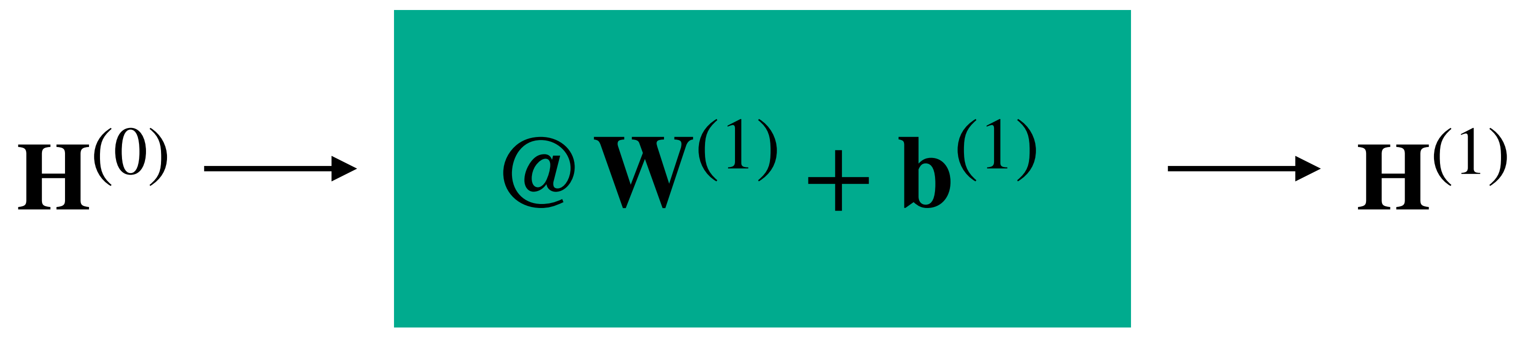


- Layer 0 is $\mathbf{H}^{(0)} = g(\mathbf{X}\mathbf{W}^{(0)\top} + \mathbf{b}^{(0)}\mathbf{1}^\top)$
- ReLU moves all negative values in each dimension to zero
- This makes things linearly separable in our example :)

$$\mathbf{X}\mathbf{W}^{(0)\top} + \mathbf{b}^{(0)}\mathbf{1}^\top = \begin{bmatrix} 0 & -1 \\ 2 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \qquad \mathbf{H}^{(0)} = \begin{bmatrix} 0 & 0 \\ 2 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$$



Layer 1



- Layer 1 $\mathbf{H}^{(1)} = \mathbf{H}^{(0)}\mathbf{W}^{(1)\top} + \mathbf{b}^{(1)}\mathbf{1}^\top$ is just a linear classifier

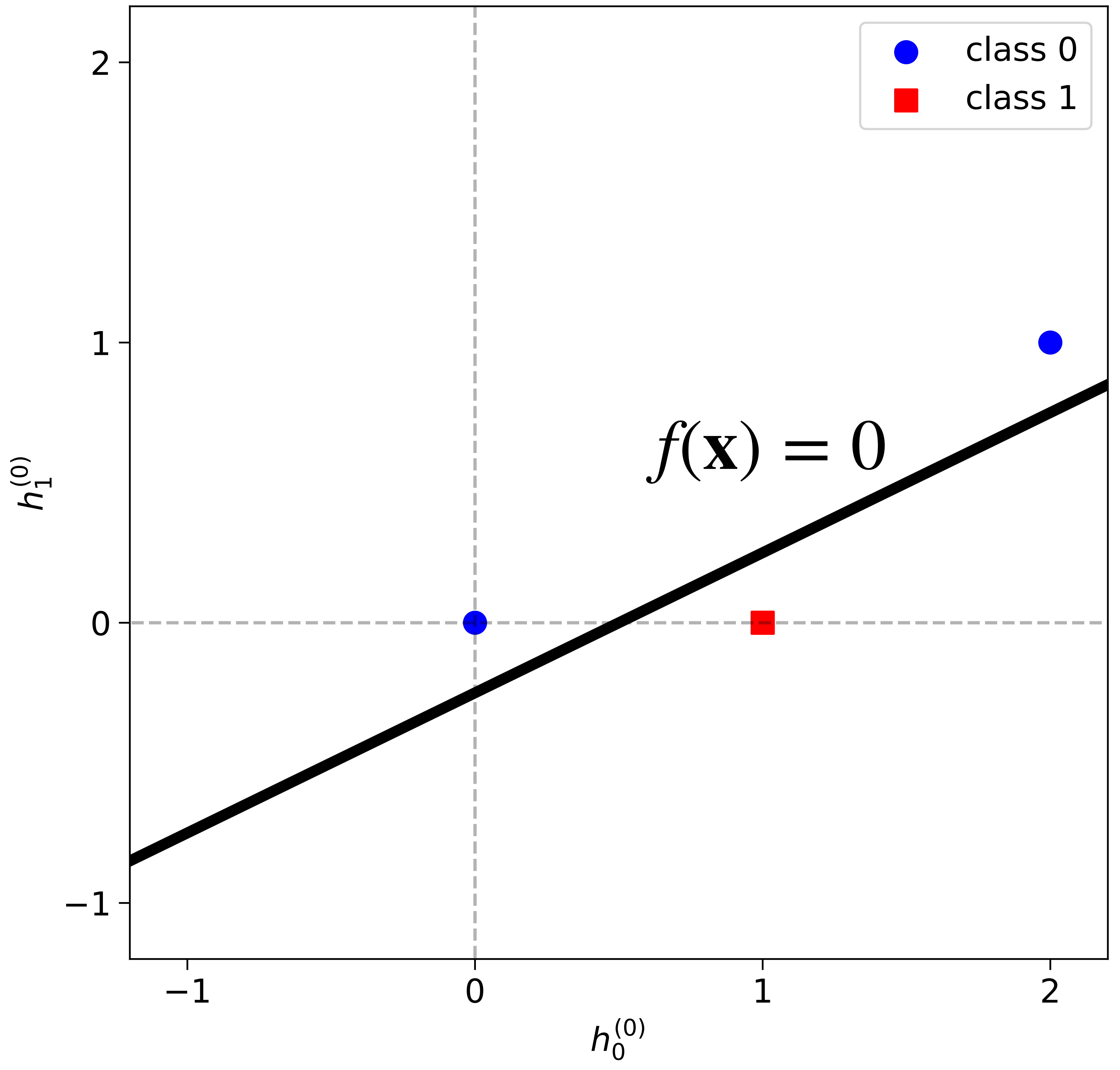
- The n^{th} row of $\mathbf{H}^{(1)}$ is $f(\mathbf{x}^{(n)})$ and points are classified according to the sign of $f(\mathbf{x})$

$$\mathbf{W}^{(1)} = [1 \quad -2]^\top \qquad \mathbf{b}^{(1)} = 0.5$$

$$\mathbf{H}^{(1)} = \begin{bmatrix} -0.5 \\ -0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$$

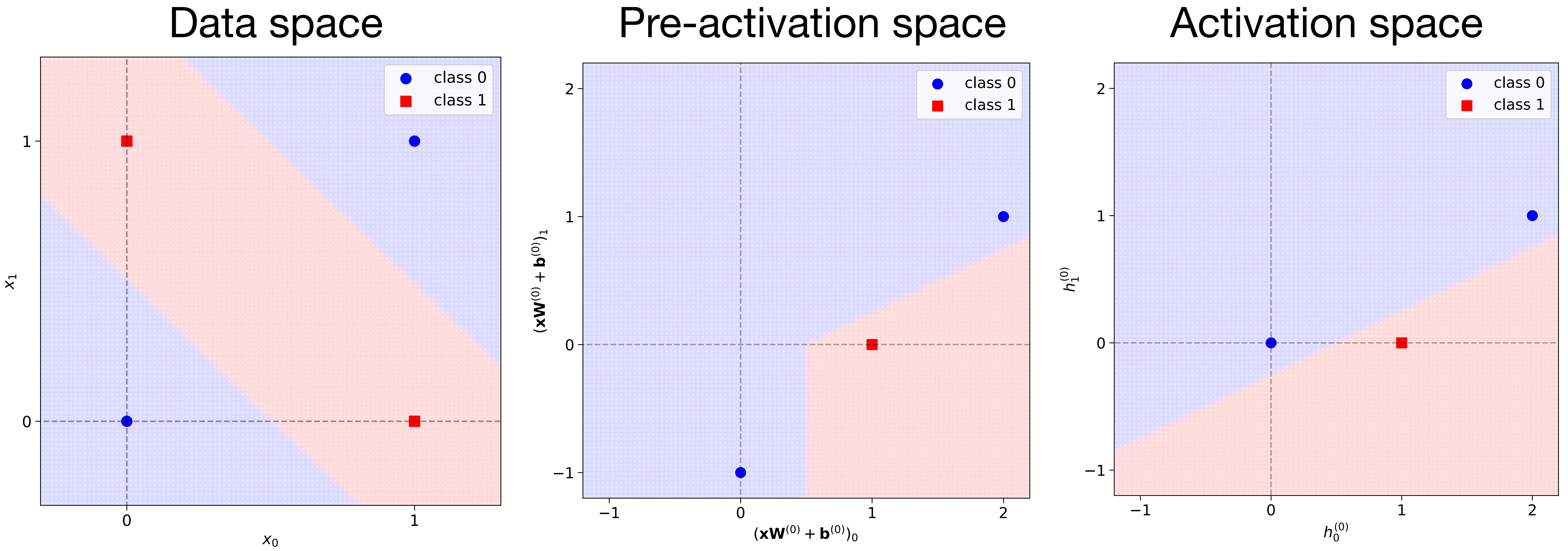
Correct classifications!

- The decision boundary is $f(\mathbf{x}) = 0$



Decision boundaries

The decision boundary is non-linear in the original and pre-activation space



Learning the parameters of a 2 layer MLP

- For $\mathbf{x} \in \mathbb{R}^D$ we can push a dataset $\mathbf{X} \in \mathbb{R}^{N \times D}$ through a 2 layer MLP using

$$\mathbf{H}^{(0)} = g(\mathbf{X}\mathbf{W}^{(0)\top} + \mathbf{b}^{(0)}\mathbf{1}^\top)$$

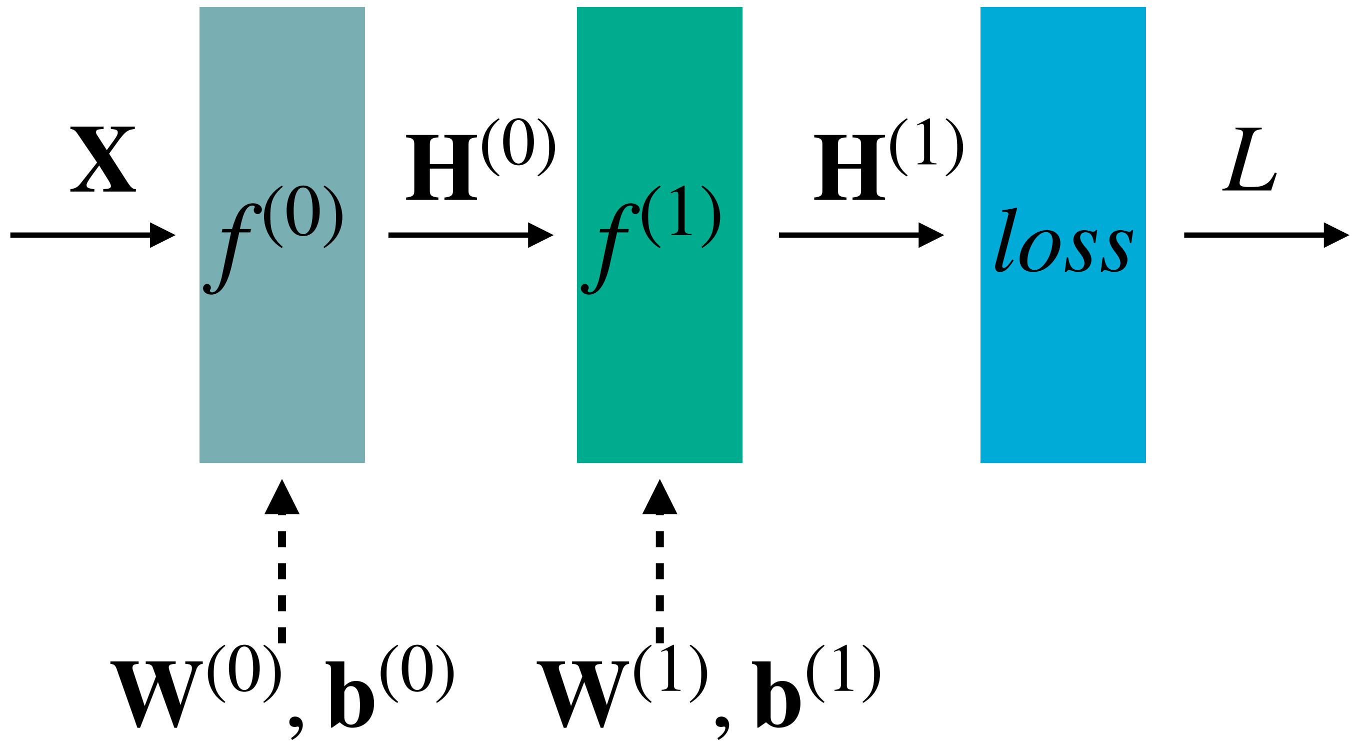
$$\mathbf{H}^{(1)} = \mathbf{H}^{(0)}\mathbf{W}^{(1)\top} + \mathbf{b}^{(1)}\mathbf{1}^\top$$

- The learning process is very similar to that of linear models
- We pick an appropriate loss function L e.g. cross-entropy for classification
- We then find the parameters that minimise the loss
- i.e. we solve $\underset{\boldsymbol{\theta}}{\text{minimise}}\, L$ where $\boldsymbol{\theta} = \{\mathbf{W}^{(0)}, \mathbf{b}^{(0)}, \mathbf{W}^{(1)}, \mathbf{b}^{(1)}\}$

The chain rule

- We can solve $\min_{\boldsymbol{\theta}} L$ for $\boldsymbol{\theta} = \{\mathbf{W}^{(0)}, \mathbf{b}^{(0)}, \mathbf{W}^{(1)}, \mathbf{b}^{(1)}\}$ using GD
- This involves computing gradients $\nabla_{\boldsymbol{\theta}} L = \{\frac{\partial L}{\partial \mathbf{W}^{(0)}}, \frac{\partial L}{\partial \mathbf{b}^{(0)}}, \frac{\partial L}{\partial \mathbf{W}^{(1)}}, \frac{\partial L}{\partial \mathbf{b}^{(1)}}\}$
- We can obtain expressions for these using the chain rule

$$\mathbf{H}^{(0)} = g(\mathbf{X}\mathbf{W}^{(0)\top} + \mathbf{b}^{(0)}\mathbf{1}^\top)$$
$$\mathbf{H}^{(1)} = \mathbf{H}^{(0)}\mathbf{W}^{(1)\top} + \mathbf{b}^{(1)}\mathbf{1}^\top$$



$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = \frac{\partial L}{\partial \mathbf{H}^{(1)}} \frac{\partial \mathbf{H}^{(1)}}{\partial \mathbf{W}^{(1)}} \qquad \frac{\partial L}{\partial \mathbf{b}^{(1)}} = \frac{\partial L}{\partial \mathbf{H}^{(1)}} \frac{\partial \mathbf{H}^{(1)}}{\partial \mathbf{b}^{(1)}}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(0)}} = \frac{\partial L}{\partial \mathbf{H}^{(1)}} \frac{\partial \mathbf{H}^{(1)}}{\partial \mathbf{H}^{(0)}} \frac{\partial \mathbf{H}^{(0)}}{\partial \mathbf{W}^{(0)}} \qquad \frac{\partial L}{\partial \mathbf{b}^{(0)}} = \frac{\partial L}{\partial \mathbf{H}^{(1)}} \frac{\partial \mathbf{H}^{(1)}}{\partial \mathbf{H}^{(0)}} \frac{\partial \mathbf{H}^{(0)}}{\partial \mathbf{b}^{(0)}}$$

Automatic differentiation

- Computers can perform automatic differentiation (/auto-diff/autograd/magic)
- We don't need to work out closed form expressions for any derivatives!

$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = \frac{\partial L}{\partial \mathbf{H}^{(1)}} \frac{\partial \mathbf{H}^{(1)}}{\partial \mathbf{W}^{(1)}}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(0)}} = \frac{\partial L}{\partial \mathbf{H}^{(1)}} \frac{\partial \mathbf{H}^{(1)}}{\partial \mathbf{H}^{(0)}} \frac{\partial \mathbf{H}^{(0)}}{\partial \mathbf{W}^{(0)}}$$

$$\frac{\partial L}{\partial \mathbf{b}^{(1)}} = \frac{\partial L}{\partial \mathbf{H}^{(1)}} \frac{\partial \mathbf{H}^{(1)}}{\partial \mathbf{b}^{(1)}}$$

$$\frac{\partial L}{\partial \mathbf{b}^{(0)}} = \frac{\partial L}{\partial \mathbf{H}^{(1)}} \frac{\partial \mathbf{H}^{(1)}}{\partial \mathbf{H}^{(0)}} \frac{\partial \mathbf{H}^{(0)}}{\partial \mathbf{b}^{(0)}}$$

Why an MLP?

- We've gone from **learning your own feature** to this two layer MLP

$$\phi(\mathbf{x}) = \mathbf{h}^{(0)} = g(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)})$$

$$f(\mathbf{x}) = \mathbf{h}^{(1)} = \mathbf{W}^{(1)}\mathbf{h}^{(0)} + \mathbf{b}^{(1)}$$

- There is a practical reason: apart from the activation function it's all just matrix multiplies which computers are very good at
- There is also theory in the form of a universal approximation theorem
- This basically tells us an MLP with at least 2 layers (and appropriate g) can represent a wide range of functions when they have the right weights

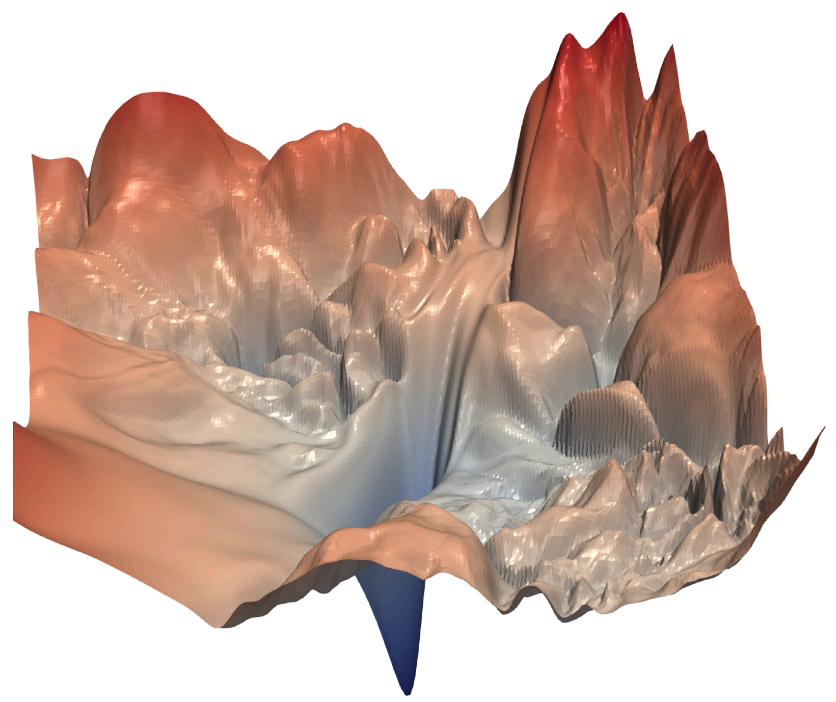
See https://www.deeplearningbook.org/contents/mlp.html 6.4.1 and https://cognitivemedium.com/magic_paper/assets/Hornik.pdf

Too good to be true?

~~Step 1: Use a 2 layer MLP to solve intelligence~~

~~Step 2: Use that to solve everything else~~

- The universal approximation theorem tells us an appropriate 2 layer MLP exists for lots of functions
- It doesn't tell us how wide the hidden layer should be or what weights to use!
- To make things worse, losses involving DNNs are generally **non-convex** :(



https://arxiv.org/pdf/1712.09913.pdf

Going deeper

- Empirically, deeper networks (those with more layers) tend to work better up to a certain point

- Now is good time to mention that deep learning is **very empirical**
- There are rules of thumb for e.g. the number of layers, layer widths
- However, often you need to try stuff out (or use existing models)

https://knowyourmeme.com/photos/531557-we-need-to-go-deeper

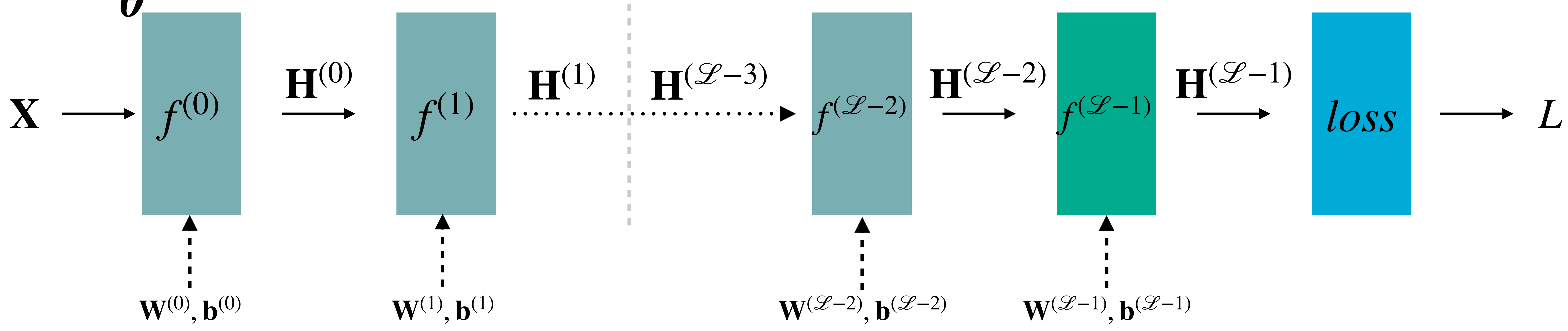
Learning the parameters of an $\mathscr{L}$ layer MLP

- For a dataset matrix $\mathbf{X}$ our $\mathscr{L}$ layer MLP is given by:

$$\mathbf{H}^{(l)} = g^{(l)}(\mathbf{H}^{(l-1)}\mathbf{W}^{(l)\top} + \mathbf{b}^{(l)}\mathbf{1}^\top) \text{ for } l = 0,1,\ldots,\mathscr{L}-1$$

- $\mathbf{H}^{(0)} = \mathbf{X}$ and $g^{(l)}$ is a non-linear activation function e.g. ReLU for all layers but the last, which is just the identity

- The loss function takes in $\mathbf{H}^{(\mathscr{L}-1)}$ (and some labels/targets) and we want to solve $\text{minimise}_{\boldsymbol{\theta}} L$ where $\boldsymbol{\theta} = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=0}^{\mathscr{L}-1}$

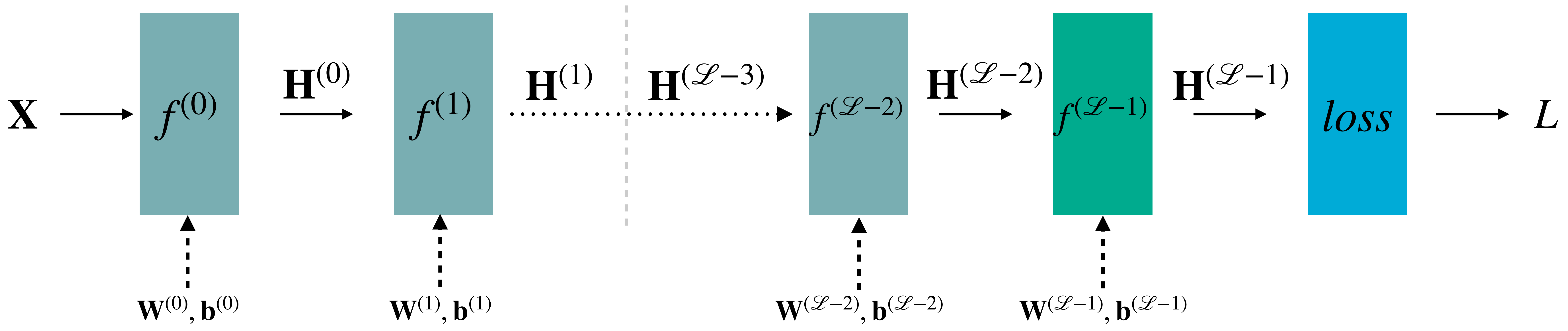


More chain rule!

- To use GD we need to compute $\nabla_{\boldsymbol{\theta}} L = \left\{ \frac{\partial L}{\partial \mathbf{W}^{(l)}}, \frac{\partial L}{\partial \mathbf{b}^{(l)}} \right\}_{l=0}^{\mathscr{L}-1}$
- We start with the last layer and can use the chain rule to write

$$\frac{\partial L}{\partial \mathbf{W}^{(\mathscr{L}-1)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{W}^{(\mathscr{L}-1)}} \qquad \frac{\partial L}{\partial \mathbf{b}^{(\mathscr{L}-1)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{b}^{(\mathscr{L}-1)}}$$

- These expression are very similar so I'll just consider the $\mathbf{W}$ gradients for now, knowing we can obtain the $\mathbf{b}$ gradients in the same way



What do you notice?

$$\frac{\partial L}{\partial \mathbf{W}^{(\mathscr{L}-1)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{W}^{(\mathscr{L}-1)}}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(\mathscr{L}-2)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{H}^{(\mathscr{L}-2)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-2)}}{\partial \mathbf{W}^{(\mathscr{L}-2)}}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(\mathscr{L}-3)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{H}^{(\mathscr{L}-2)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-2)}}{\partial \mathbf{H}^{(\mathscr{L}-3)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-3)}}{\partial \mathbf{W}^{(\mathscr{L}-3)}}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(\mathscr{L}-4)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{H}^{(\mathscr{L}-2)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-2)}}{\partial \mathbf{H}^{(\mathscr{L}-3)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-3)}}{\partial \mathbf{H}^{(\mathscr{L}-4)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-4)}}{\partial \mathbf{W}^{(\mathscr{L}-4)}}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(\mathscr{L}-5)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{H}^{(\mathscr{L}-2)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-2)}}{\partial \mathbf{H}^{(\mathscr{L}-3)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-3)}}{\partial \mathbf{H}^{(\mathscr{L}-4)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-4)}}{\partial \mathbf{H}^{(\mathscr{L}-5)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-5)}}{\partial \mathbf{W}^{(\mathscr{L}-5)}}$$

The same terms keep cropping up

$$\frac{\partial L}{\partial \mathbf{W}^{(\mathscr{L}-1)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{W}^{(\mathscr{L}-1)}}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(\mathscr{L}-2)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{H}^{(\mathscr{L}-2)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-2)}}{\partial \mathbf{W}^{(\mathscr{L}-2)}}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(\mathscr{L}-3)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{H}^{(\mathscr{L}-2)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-2)}}{\partial \mathbf{H}^{(\mathscr{L}-3)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-3)}}{\partial \mathbf{W}^{(\mathscr{L}-3)}}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(\mathscr{L}-4)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{H}^{(\mathscr{L}-2)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-2)}}{\partial \mathbf{H}^{(\mathscr{L}-3)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-3)}}{\partial \mathbf{H}^{(\mathscr{L}-4)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-4)}}{\partial \mathbf{W}^{(\mathscr{L}-4)}}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(\mathscr{L}-5)}} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-1)}}{\partial \mathbf{H}^{(\mathscr{L}-2)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-2)}}{\partial \mathbf{H}^{(\mathscr{L}-3)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-3)}}{\partial \mathbf{H}^{(\mathscr{L}-4)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-4)}}{\partial \mathbf{H}^{(\mathscr{L}-5)}} \frac{\partial \mathbf{H}^{(\mathscr{L}-5)}}{\partial \mathbf{W}^{(\mathscr{L}-5)}}$$

- We can write $\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \mathbf{G}^{(l)} \frac{\partial \mathbf{H}^{(l)}}{\partial \mathbf{W}^{(l)}}$ where $\mathbf{G}^{(l)} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}} \prod_{m=1}^{\mathscr{L}-l-1} \frac{\partial \mathbf{H}^{(\mathscr{L}-m)}}{\partial \mathbf{H}^{(\mathscr{L}-m-1)}}$
- We can iteratively compute $\mathbf{G}^{(l-1)} = \mathbf{G}^{(l)} \frac{\partial \mathbf{H}^{(l)}}{\partial \mathbf{H}^{(l-1)}}$ so we don't have to repeatedly calculate the same terms

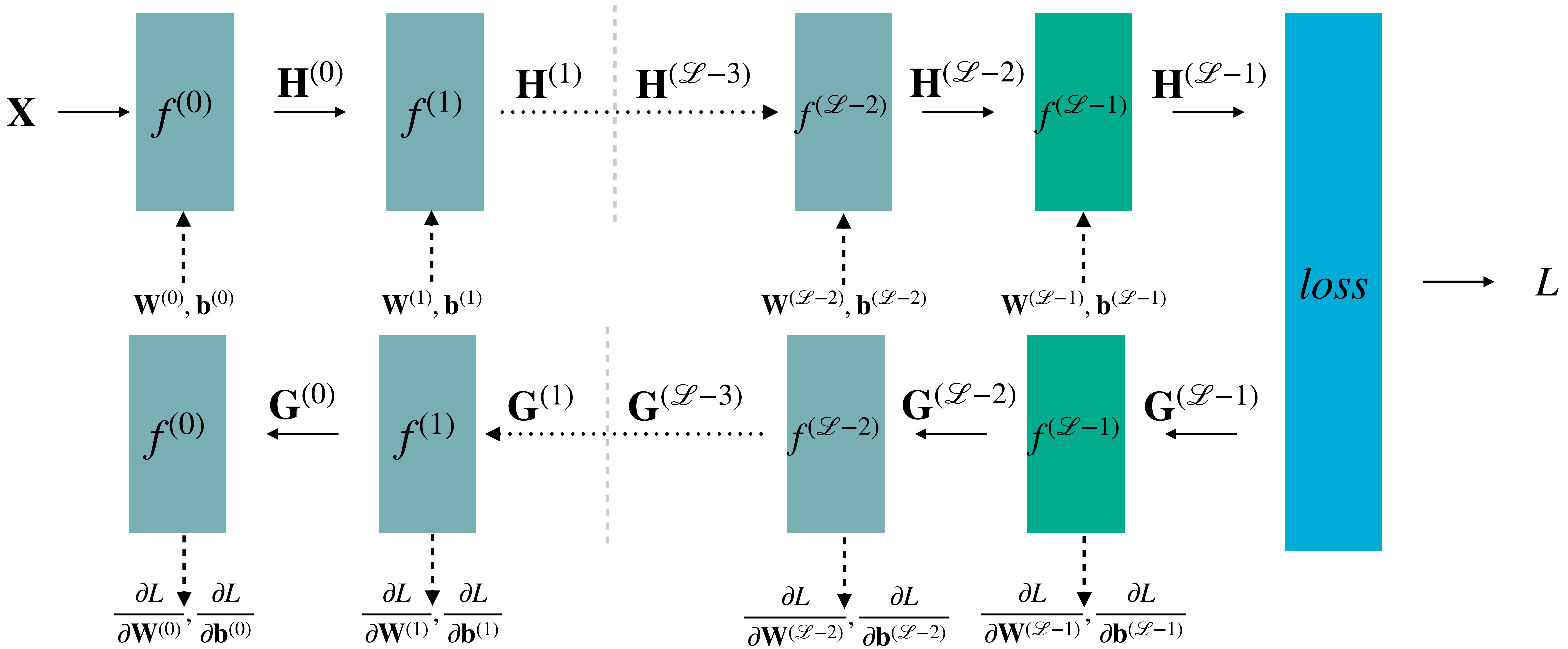
The backpropagation algorithm

- Goal: Obtain gradients $\nabla_{\boldsymbol{\theta}} L = \{\frac{\partial L}{\partial \mathbf{W}^{(l)}}, \frac{\partial L}{\partial \mathbf{b}^{(l)}}\}_{l=0}^{\mathscr{L}-1}$
- Compute $\mathbf{G}^{(\mathscr{L}-1)} = \frac{\partial L}{\partial \mathbf{H}^{(\mathscr{L}-1)}}$
- For l in $\mathscr{L}-1, \mathscr{L}-2, \ldots, 1, 0$:
  1. Compute $\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \mathbf{G}^{(l)} \frac{\partial \mathbf{H}^{(l)}}{\partial \mathbf{W}^{(l)}}$ and $\frac{\partial L}{\partial \mathbf{b}^{(l)}} = \mathbf{G}^{(l)} \frac{\partial \mathbf{H}^{(l)}}{\partial \mathbf{b}^{(l)}}$
  2. Compute $\mathbf{G}^{(l-1)} = \mathbf{G}^{(l)} \frac{\partial \mathbf{H}^{(l)}}{\partial \mathbf{H}^{(l-1)}}$

See Murphy Section 13.3 for a more detailed and rigorous description!

Backpropagation is efficient

- Going forward, you have to keep all the activations in memory
- Going backward, you can throw stuff away after it's used to update $\mathbf{G}^{(l)}$



SGD for neural network training

Storing lots of activations for a whole dataset $\mathbf{X} \in \mathbb{R}^{N \times D}$ can be expensive. Because of this SGD is typically used for DNN training. The procedure is:

- Initialise DNN weights at random e.g. from a normal distribution
- For e in range(E):
  - Split dataset into equal sized **mini-batches** $\{\mathbf{X}^{(b)}, \mathbf{y}^{(b)}\}_{b=0}^{B-1}$ at random
  - For b in range(B):
    1. Compute $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathbf{X}^{(b)}, \mathbf{y}^{(b)})$ using backpropagation
    2. Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathbf{X}^{(b)}, \mathbf{y}^{(b)})$

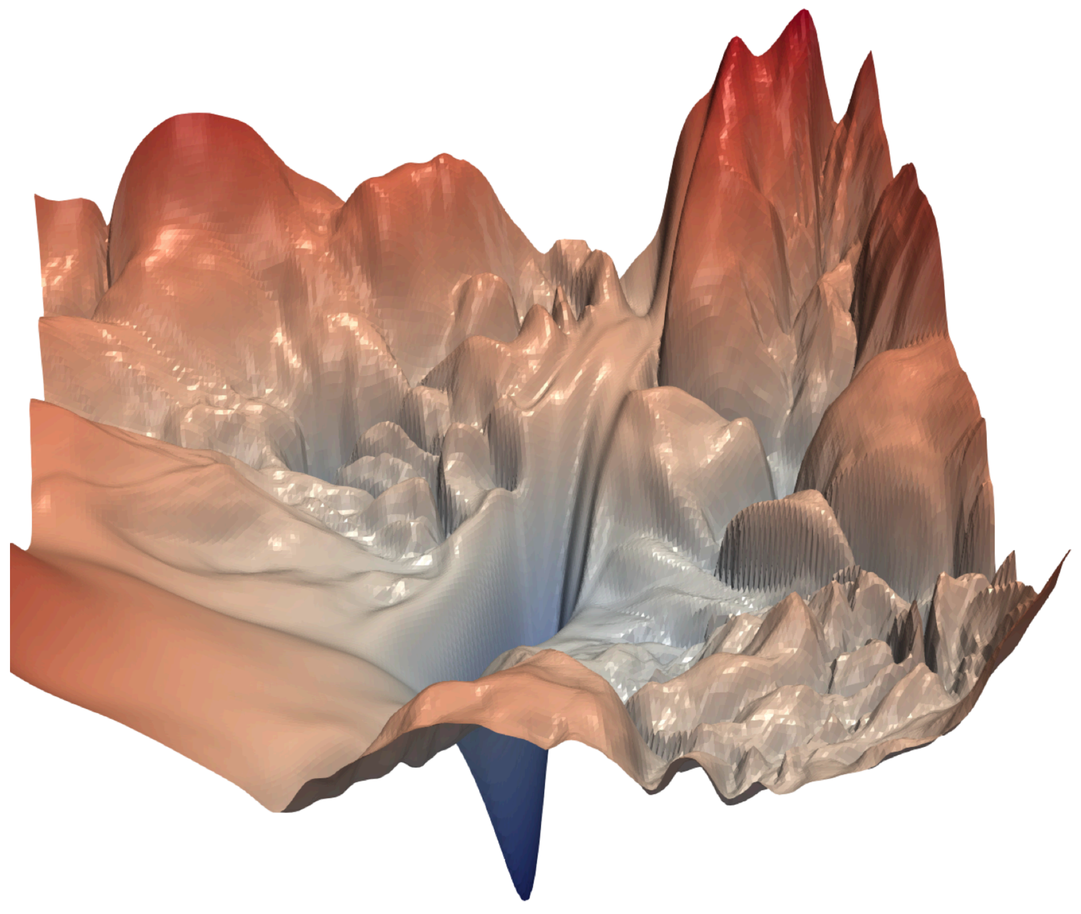
Each outer loop across the whole dataset is known as an *epoch*

SGD + momentum

- As the loss functions for DNNs are non-convex it is possible to get stuck in an undesirable local minimum as the gradient is zero
- In SGD + momentum we update parameters using the current gradient and an exponential moving average of previous gradients
- This makes it harder to get stuck, and tends to accelerate training
- At time step t:

    1. Compute $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_{t=i}, \mathbf{X}^{(b)}, \mathbf{y}^{(b)})$ using backpropagation
    2. Update velocity $v_{t=i+1} = \mu v_{t=i} + \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_{t=i}, \mathbf{X}^{(b)}, \mathbf{y}^{(b)})$
    3. Update $\boldsymbol{\theta}_{t=i+1} = \boldsymbol{\theta}_{t=i} - \alpha v_{t=i+1}$

μ is the momentum

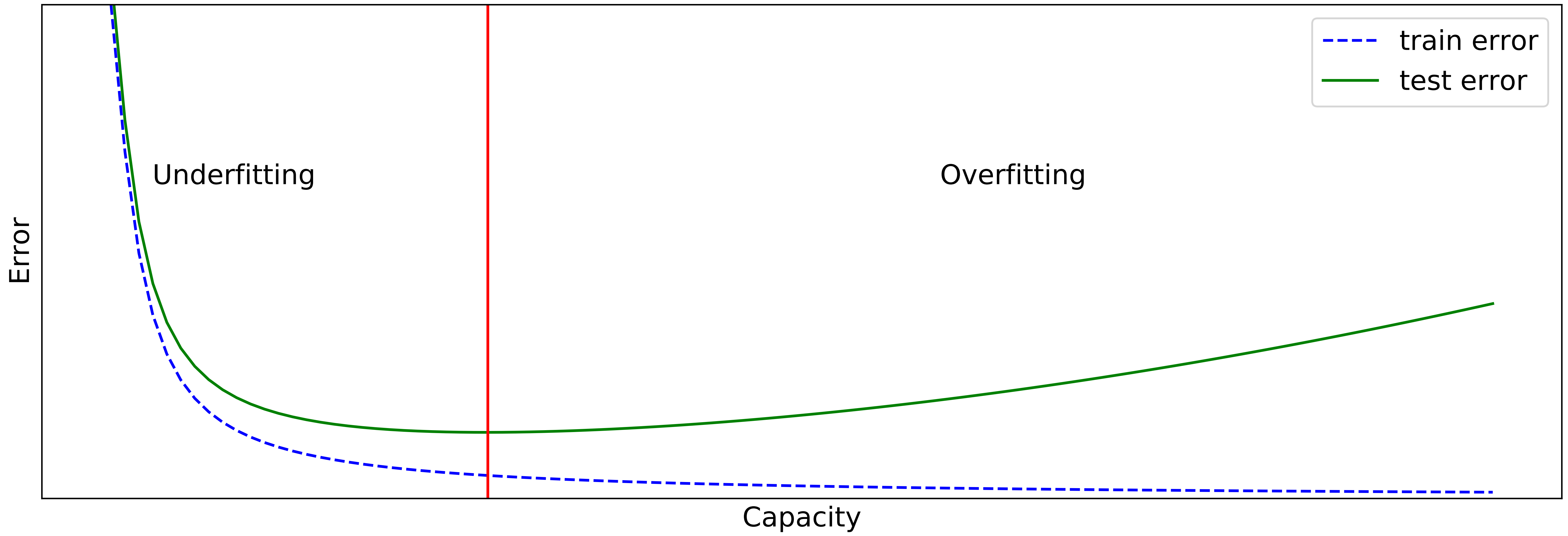


Other optimisers are available

- e.g. the Adam optimiser (pictured right)
- Almost all take the gradients from backprop and do something with them
- You don't need to know about any optimisers other than GD and SGD (+ momentum) for this course
- See https://pytorch.org/docs/stable/optim.html#algorithms if you're curious how others function

DNNs can overfit

- DNNs can represent lots of functions. They are high capacity models
- They are very susceptible to overfitting!
- Remember, we care about a model’s ability to **generalise** to unseen data
- Regularisation is very important in DNNs!

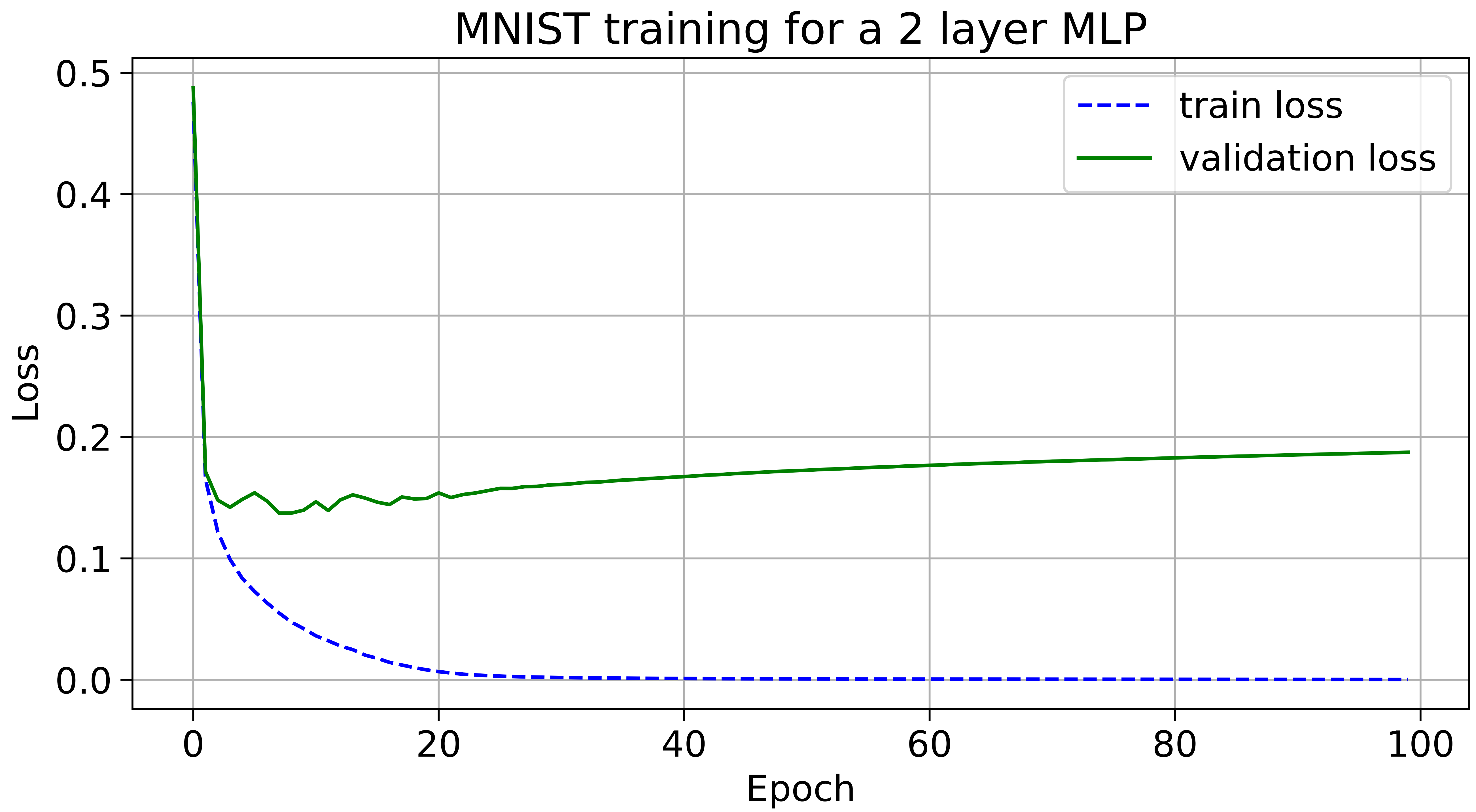


Early stopping

- Fitting to the test set is not allowed
- We can however look at the validation set throughout training as a proxy
- The model starts to overfit once validation loss stops decreasing with train loss
- We can stop training at this point

This looks very similar to the last figure!

Over training models tend to underfit and then overfit to the training data

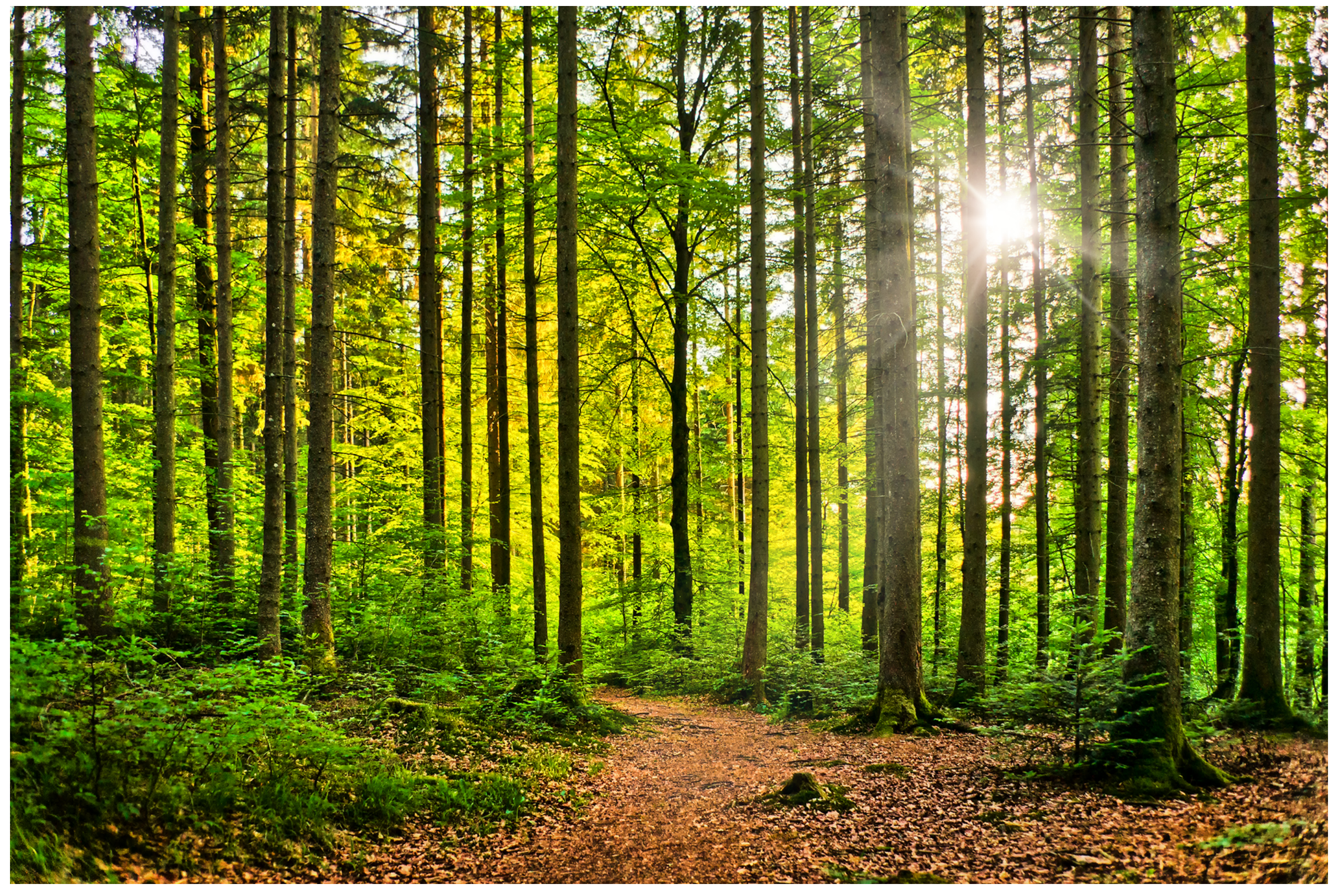


Weight decay

- Models that overfit tend to have large weights
- To mitigate this, we multiply all the weights by $1 - \lambda$ whenever we perform an update step in e.g. SGD
- λ is the amount of weight decay as is usually very small e.g. 10^{-4}
- This is basically equivalent to having L2 regularisation in the loss function

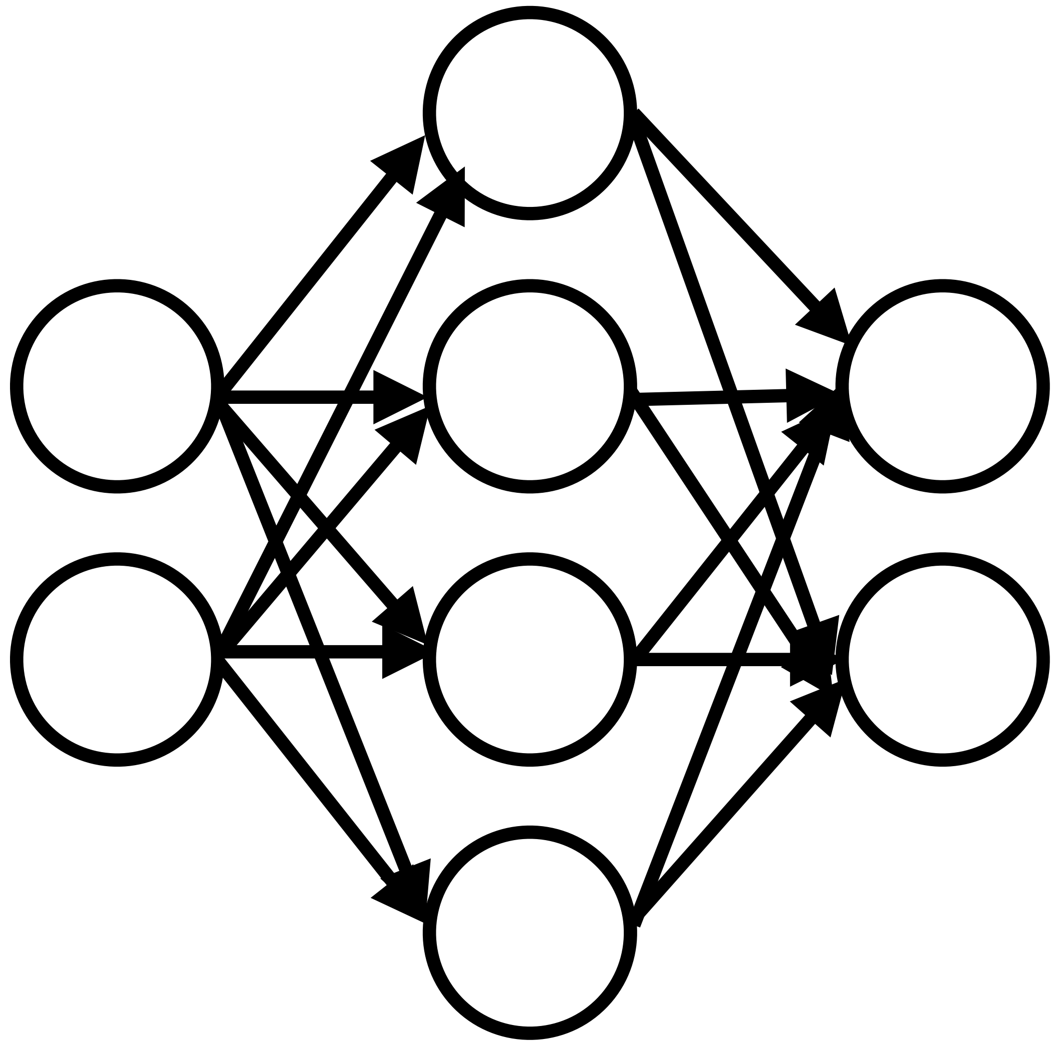
Ensembles as regularisers

- Recall that decision trees tended to overfit
- We mitigated this by forming an ensemble in the form of a random forest
- Ensemble learning is a form of regularisation
- But DNN training is costly so we don't want to train lots of them



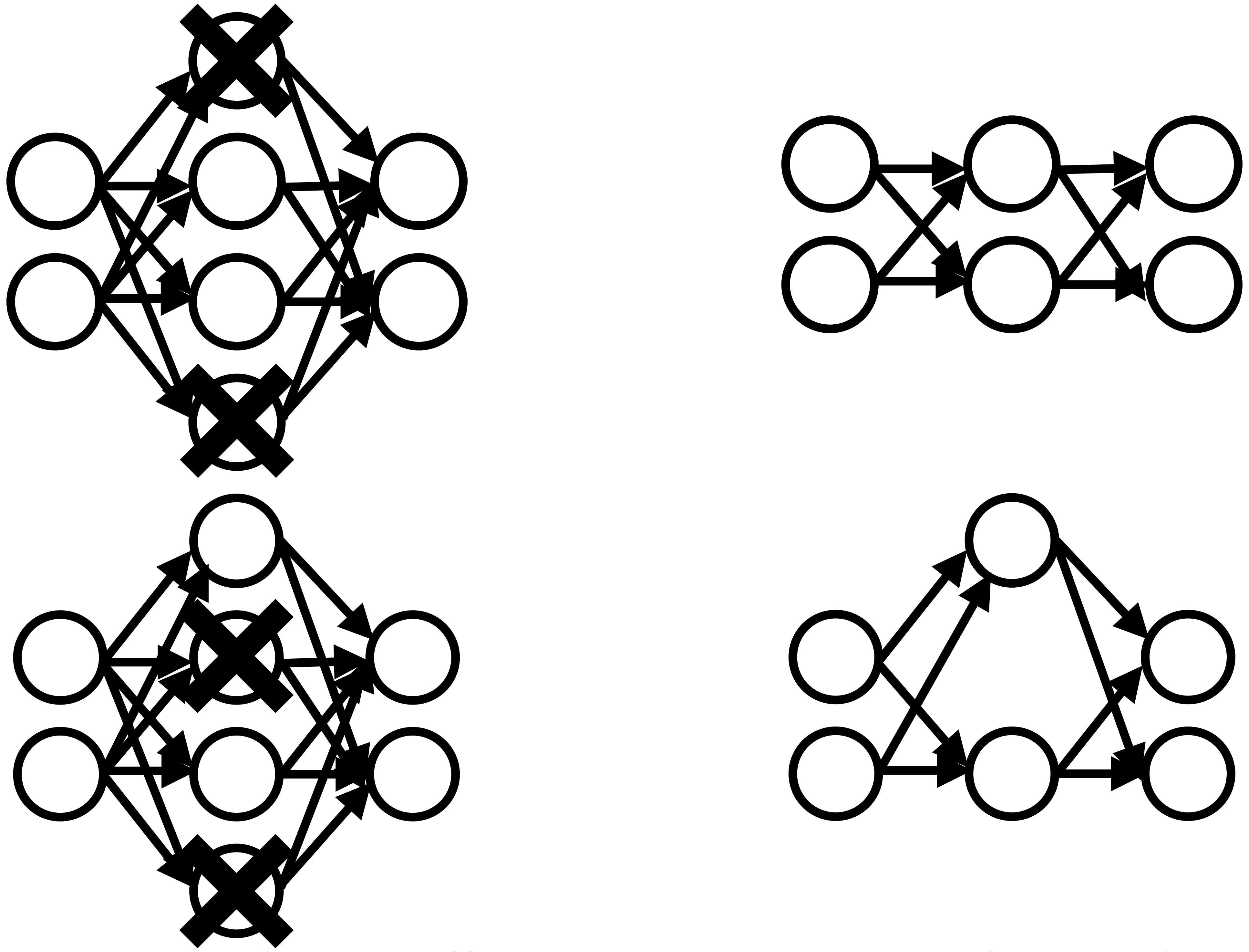
Dropout

- At each iteration of training, each hidden neuron has a chance (usually 50%) of being switched off for that forward and backward pass
- We can view this as cheaply training an ensemble of subnetworks



Iteration 0

Iteration 1



See https://www.deeplearningbook.org/contents/regularization.html 7.12 for more details

Summary

- We have considered learning our features instead of using a pre-existing map
- We have seen how the structure of a DNN facilitates feature learning
- We have looked at the MLP architecture and worked through some examples
- We have found out how to train an MLP using backpropagation + SGD
- We looked at different ways to regularise DNNs