Neural Networks

Sarah Jane Delany

Based on ML for PDA textbook

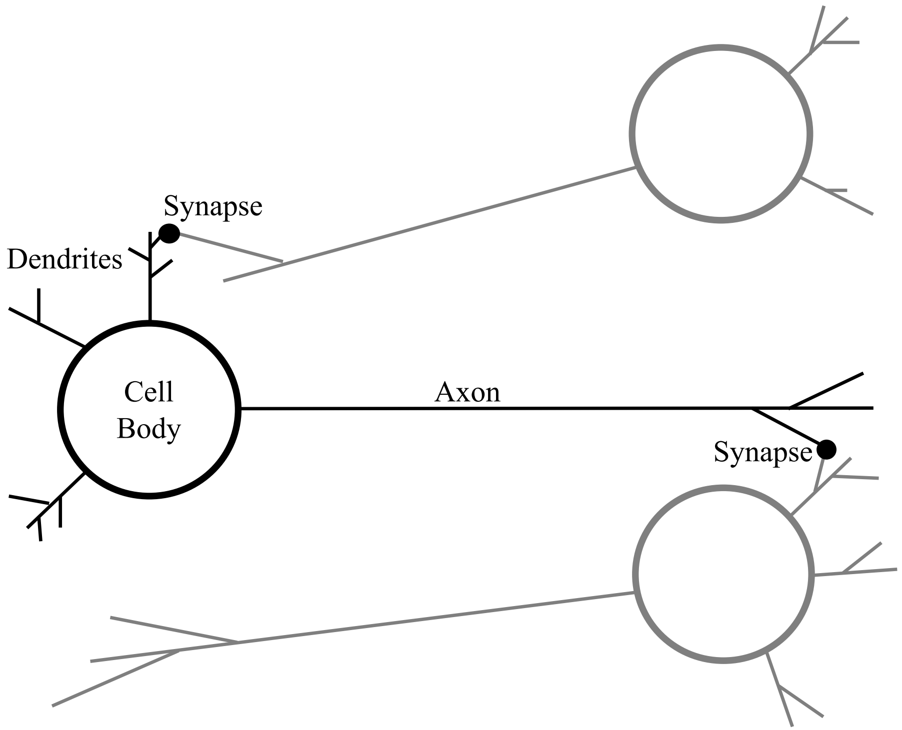
Big Idea

- Artificial Neural Networks (ANNs) are inspired by the structure and operations of the human brain
  - Propagates electrical signals through a massive network of interconnected cells (neurons)

Neuron is a simple signal processing unit

(i) Cell body
(ii) Dendrites (inputs)
(iii) Axon (output)

Connector is the synapse



All-or-none switch: If the electrical signals gathered by its dentrites are strong enough, the neuron transmits an electrical pulse (action potential) along its axon, otherwise it has no output

Motivation: Decision Making

Q. “Will a customer wait for a restaurant table?” - "Yes" or "No"

- This decision might depend on a number of factors:

  Is the restaurant full? (0 or 1)

  Is the customer hungry? (0 or 1)

  Is a suitable alternative restaurant nearby? (0 or 1)

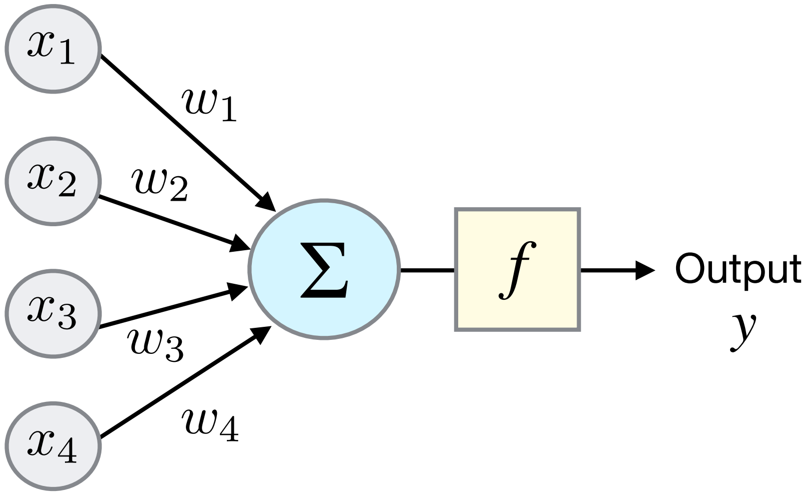
- We could determine weights w_i indicating how important each factor is in making the decision.
- For example, if x_2 is the most important factor, we might choose weights $w_1 = 0.2,\ w_2 = 0.6,\ w_3 = 0.2$
- If the weighted sum is greater than some predefined threshold, the customer might decide to move on to another restaurant ("No")

$$w_1 x_1 + w_2 x_2 + w_3 x_3 \geq \text{threshold}$$

e.g. $(0.2 \times x_1) + (0.6 \times x_2) + (0.2 \times x_3) \geq 0.8$

Modelling Neurons

- An artificial neuron makes decisions by weighing up evidence. It takes many input signals $\{x_1, x_2, \ldots\}$ and produces a single output.
- The inputs each have weights $\{w_1, w_2, \ldots\}$. These are real numbers which indicate the importance of the inputs to the output.
- The output y is computed by applying some function f to the weighted sum of the input signals $\{x_1, x_2, \ldots\}$. This is often called the activation function.
- **Example:** Neuron with 4 inputs and 4 corresponding weights.

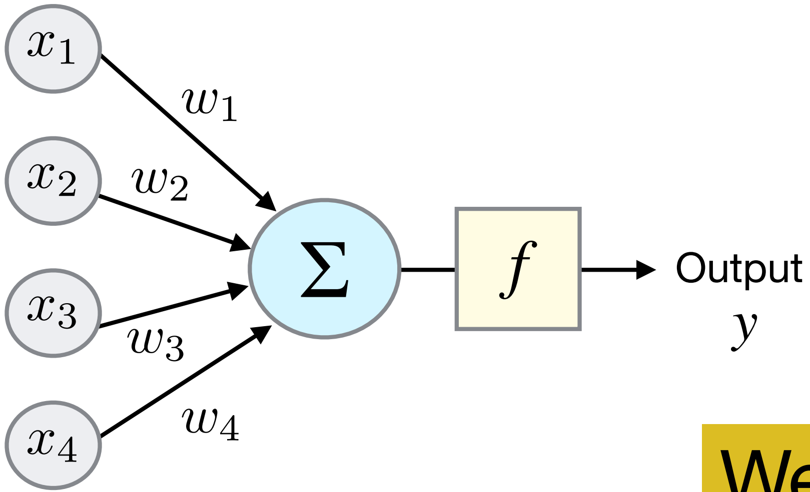


$$y = f(w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4)$$

f: activation function

Modelling Neurons

- An artificial neuron makes decisions by weighing up evidence. It takes many input signals $\{x_1, x_2, \ldots\}$ and produces a single output.
- The inputs each have weights $\{w_1, w_2, \ldots\}$. These are real numbers which indicate the importance of the inputs to the output.
- The output y is computed by applying some function f to the weighted sum of the input signals $\{x_1, x_2, \ldots\}$. This is often called the activation function.
- **Example:** Neuron with 4 inputs and 4 corresponding weights.



$$y = f(w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4)$$

f: activation function

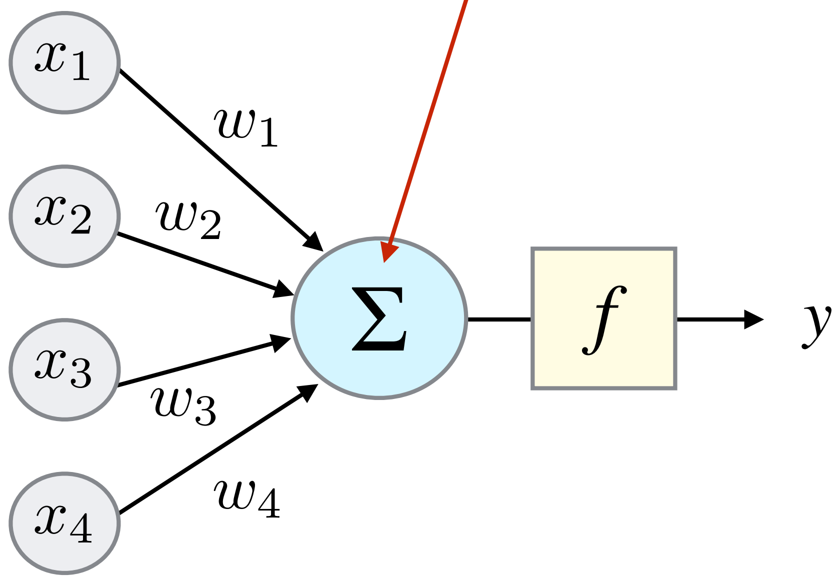
We have seen this before!

Modelling Neurons

Multivariate Linear Regression

$$\mathbb{M}_{\mathbf{w}}(\mathbf{d}) = \mathbf{w} \cdot \mathbf{d}$$

$$\begin{aligned}\mathbb{M}_{\mathbf{w}}(\mathbf{d}) &= \mathbf{w}[0] \times \mathbf{d}[0] + \mathbf{w}[1] \times \mathbf{d}[1] + \ldots + \mathbf{w}[m] \times \mathbf{d}[m] \\ &= \sum_{j=0}^{m} \mathbf{w}[j] \times \mathbf{d}[j] \\ &= \mathbf{w} \cdot \mathbf{d}\end{aligned}$$

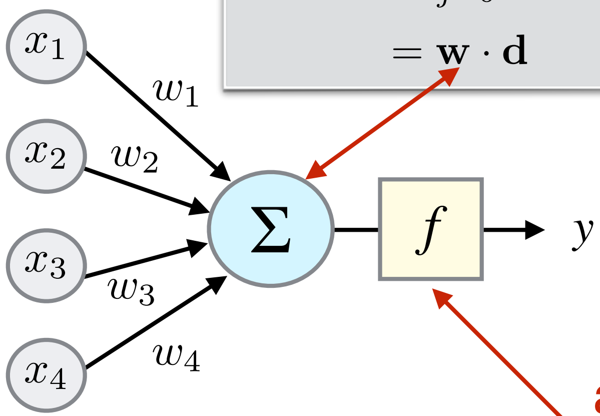


Modelling Neurons

Multivariate Linear Regression

$$\mathbb{M}_{\mathbf{w}}(\mathbf{d}) = \mathbf{w} \cdot \mathbf{d}$$

$$\begin{aligned}\mathbb{M}_{\mathbf{w}}(\mathbf{d}) &= \mathbf{w}[0] \times \mathbf{d}[0] + \mathbf{w}[1] \times \mathbf{d}[1] + \ldots + \mathbf{w}[m] \times \mathbf{d}[m] \\ &= \sum_{j=0}^{m} \mathbf{w}[j] \times \mathbf{d}[j] \\ &= \mathbf{w} \cdot \mathbf{d}\end{aligned}$$



activation function

Logistic Regression

$$\begin{aligned}\mathbb{M}_{\mathbf{w}}(\mathbf{d}) &= Logistic(\mathbf{w} \cdot \mathbf{d}) \\ &= \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{d}}}\end{aligned}$$

$$\begin{aligned}\mathbb{M}_{\mathbf{w}}(\mathbf{d}) &= Logistic(\mathbf{w} \cdot \mathbf{d}) \\ &= \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{d}}}\end{aligned}$$

Modelling Neurons

- An artificial neuron can be defined as:

$$\mathbb{M}_{\mathbf{w}}(\mathbf{d}) = \varphi\left(\mathbf{w}[0] \times \mathbf{d}[0] + \mathbf{w}[1] \times \mathbf{d}[1] + \cdots + \mathbf{w}[m] \times \mathbf{d}[m]\right)$$
$$= \varphi\left(\sum_{i=0}^{m} w_i \times d_i\right) = \varphi\left(\underbrace{\mathbf{w} \cdot \mathbf{d}}_{dot\ product}\right)$$

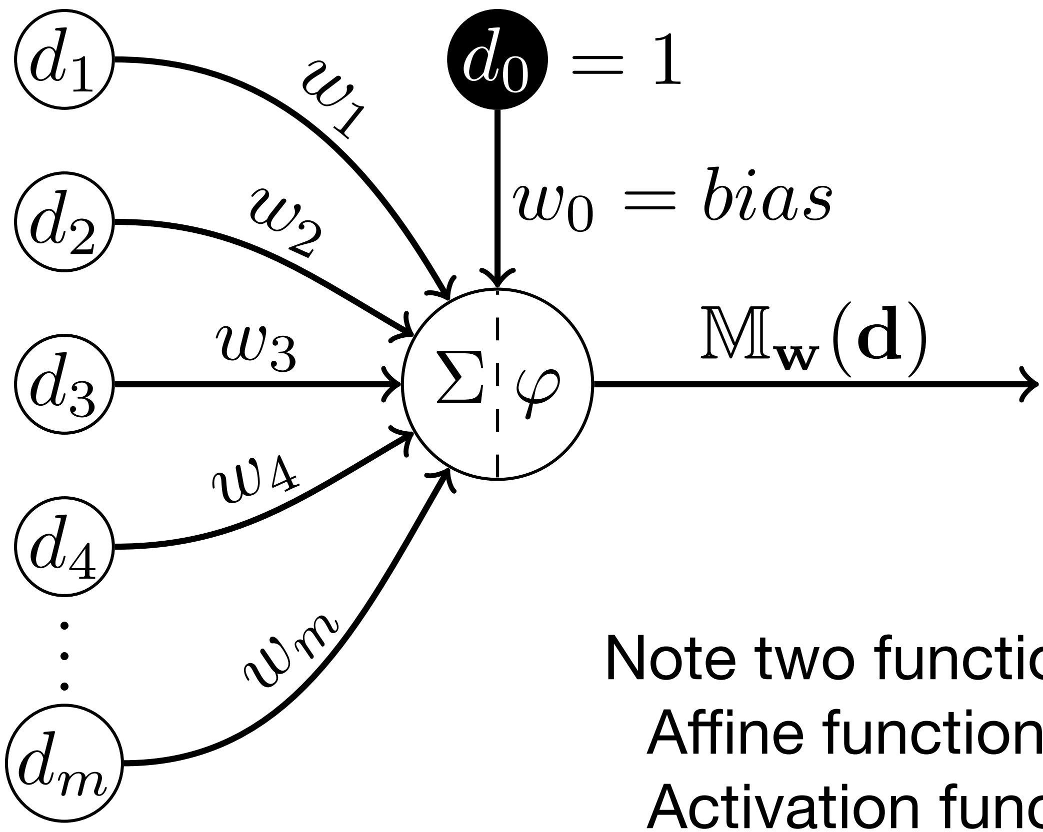
where φ is the activation function

- $w[0]$ is known as the bias (remember the dummy variable in linear regression...)
  - in the absence of other inputs the output is set to value of $w[0]$
  - changes from a linear function on the inputs to an affine function on the inputs

Affine function = Linear Function + Transformation

  - closely related so term linear is often used for both

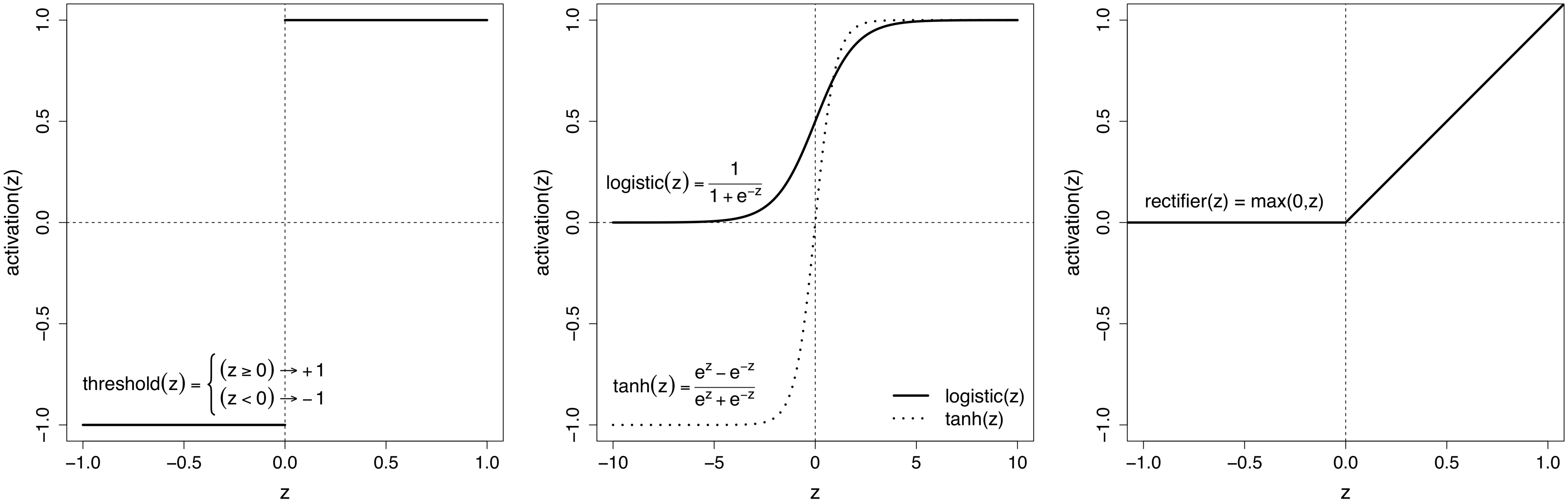
Modelling Neurons



Note two functions (or stages):
- Affine function on inputs
- Activation function to determine whether neuron fires or not

Activation functions

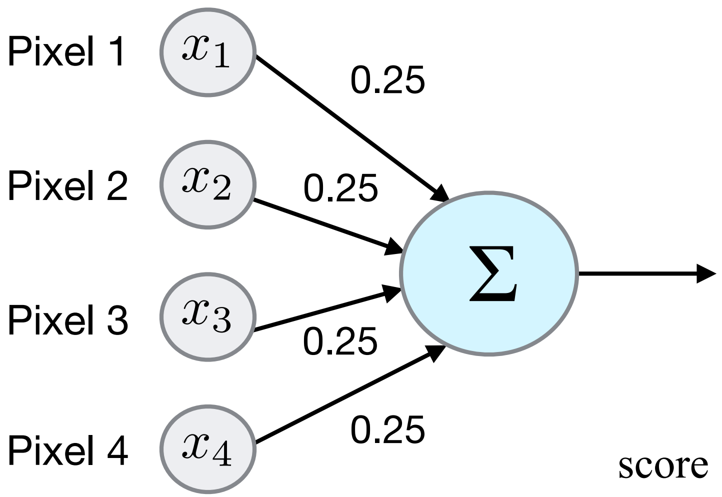
- Popular activation functions over the years



- A neuron is also known as unit or node
  - neuron that uses a logistic activation function is a logistic unit
  - neuron that uses a rectified linear activation function (rectifier) is a rectified linear unit or ReLU

Example: Perceptron (Rosenblatt, 1958)

- Input is a 2x2 B&W image (i.e 4 pixels). Task is to classify the brightness of the image. All weights are equal (0.25) and using threshold value of 0.6. Output is "Dark" (0) or "Bright" (1).



Apply threshold activation function

$$\text{Output} = \begin{cases} 0 \text{ (Dark)} & \text{if score} < 0.6 \\ 1 \text{ (Bright)} & \text{if score} \geq 0.6 \end{cases}$$

$$\text{score} = (0.25 \times x_1) + (0.25 \times x_2) + (0.25 \times x_3) + (0.25 \times x_4)$$

1.0	1.0
1.0	1.0

score = 1.0

$\geq 0.6 \implies \text{Bright}$

1.0	0.5
0.5	1.0

score = 0.75

$\geq 0.6 \implies \text{Bright}$

1.0	0.5
0.5	0.0

score = 0.5

$< 0.6 \implies \text{Dark}$

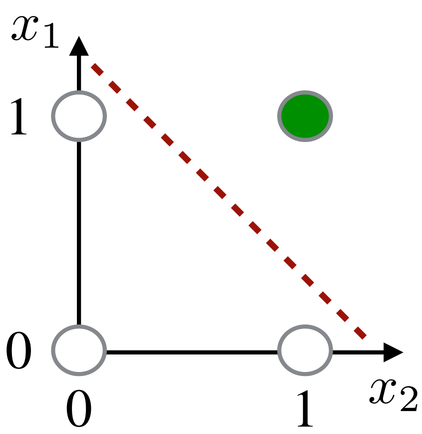
0.0	0.5
0.5	0.0

score = 0.25

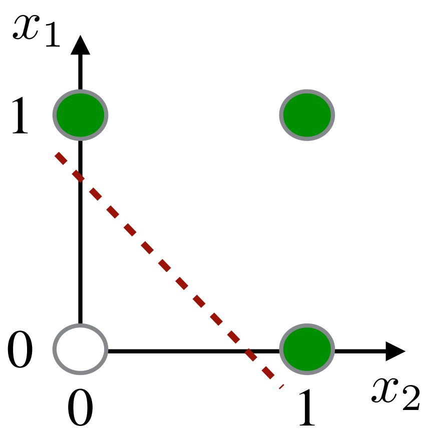
$< 0.6 \implies \text{Dark}$

Perceptron Limitations

- Single layer perceptron is a linear classifier can only handle linearly separable problems
- **Example:** Boolean AND and OR functions are linearly separable.



x_1 AND x_2



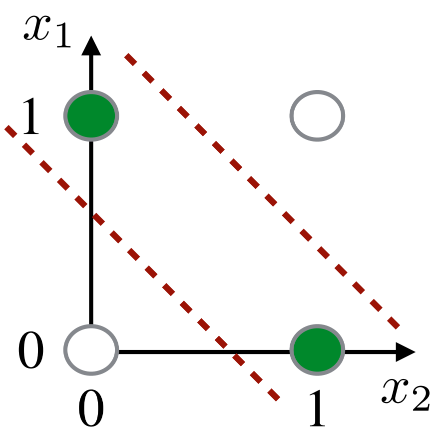
x_1 OR x_2

x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

Perceptron Limitations

- Single layer perceptron is a linear classifier can only handle linearly separable problems
- **Example:** Boolean AND and OR functions are linearly separable. But the XOR ("Exclusive OR") function is not linearly separable.



x_1 XOR x_2

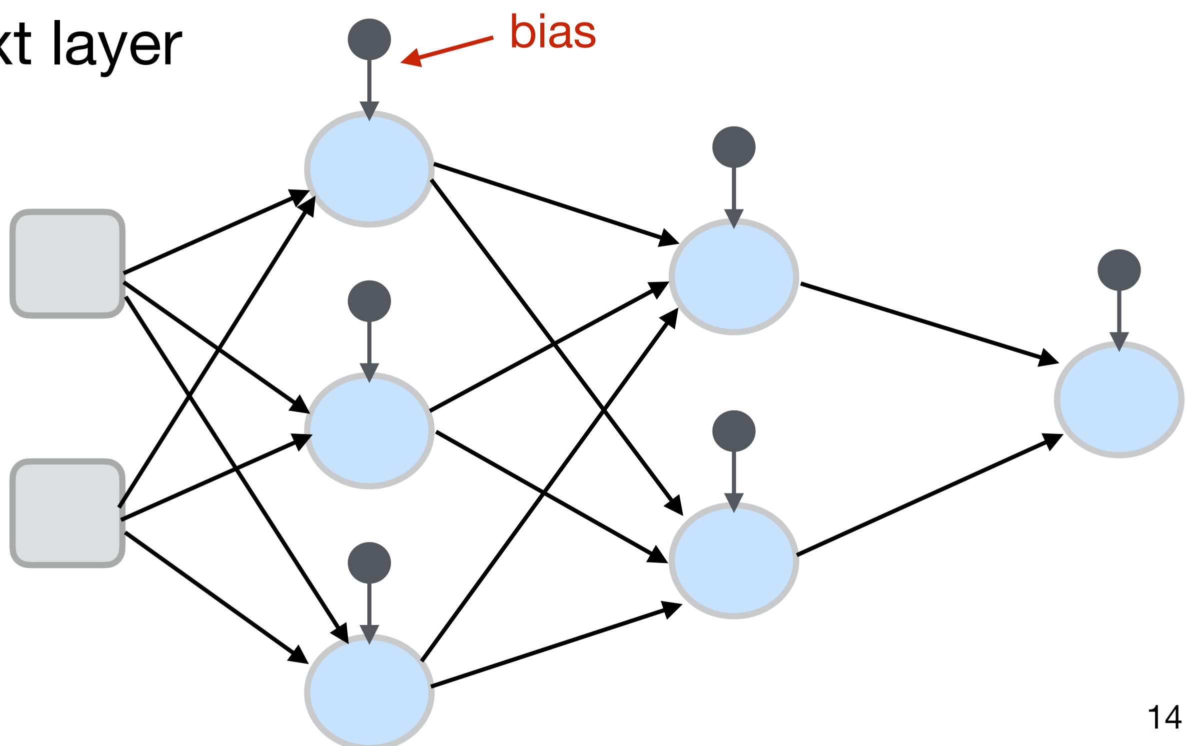
x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

(Minsky & Papert, 1969)

Q. How can we solve non-linearly separable problems like this?

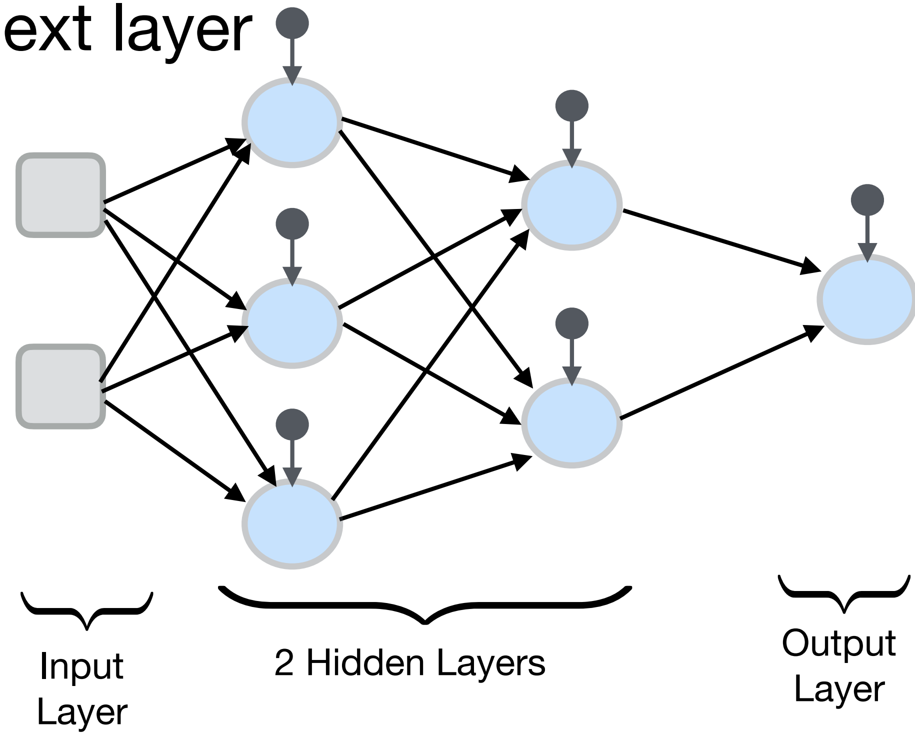
Multilayer Networks

- By building a slightly more complicated neural network with an intermediary layer, we can solve non-linear problems that cannot be solved using only a single layer of inputs and outputs.
- Input layer are *sensory* nodes, they sense the input, but do not transform or process it
- All other nodes are *processing* nodes, applying the two stages of processing, affine/linear function + activation function
- Output is passed onto next layer in the network
- Arrows indicate the flow of processing
- Networks can have multiple outputs



Multilayer network

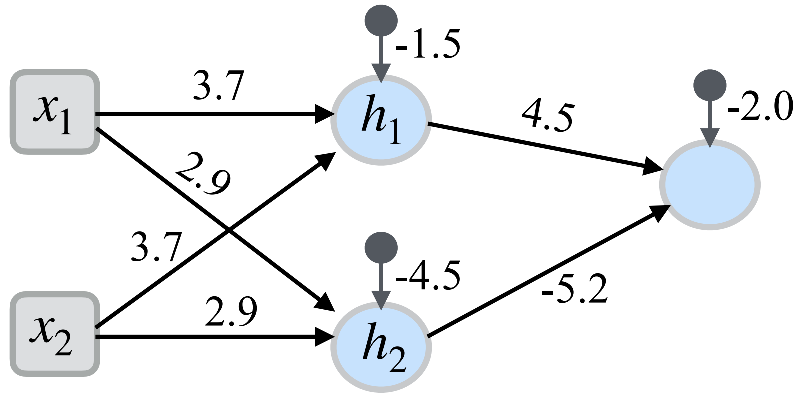
- Layers of nodes between the input and output layers are known as hidden layers
- Feedforward network is one where there are no loops or cycles. Activations always flow forward
- A fully connected network is one where each node receives input from all nodes in preceding layer and passes its output activation to all nodes in the next layer
- The depth of a network is the number of hidden layers + output layer



Example:

- **Configuration:** One input layer with 2 inputs. One hidden layer, one output. Using a sigmoid activation function.

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



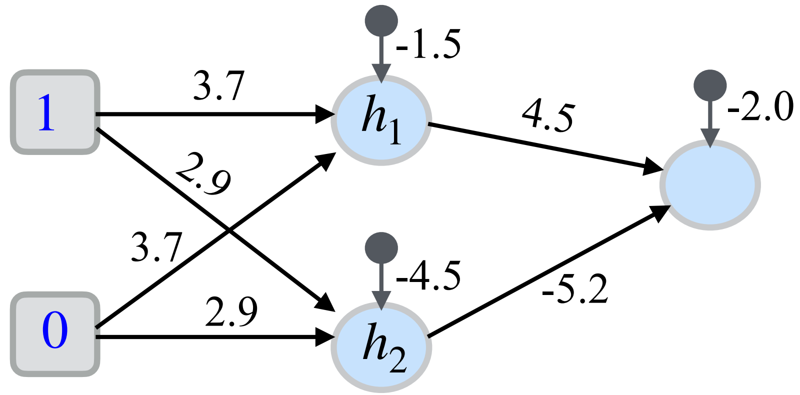
Example: Multilayer Network

- **Configuration:** One input layer with 2 inputs. One hidden layer, one output. Using a sigmoid activation function.

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

- Consider the case of the inputs:

$$x_1 = 1, x_2 = 0$$



- Compute values based on the inputs:

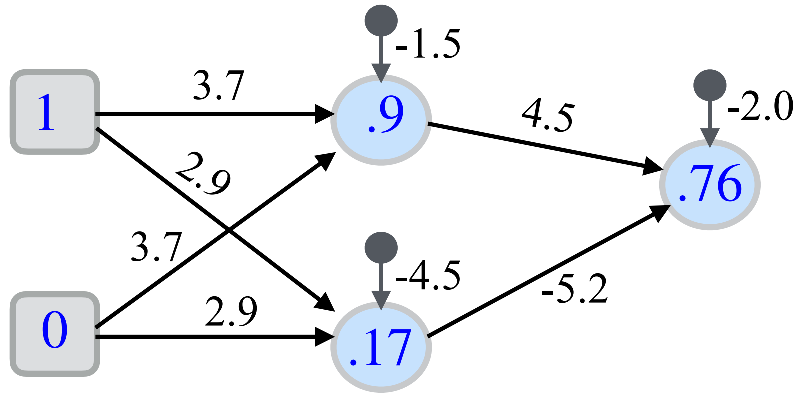
$$h_1 = \sigma((1 \times 3.7) + (0 \times 3.7) + (1 \times -1.5)) = \sigma(2.2) = \frac{1}{1+e^{-2.2}} = 0.90$$

$$h_2 = \sigma((1 \times 2.9) + (0 \times 2.9) + (1 \times -4.5)) = \sigma(-1.6) = \frac{1}{1+e^{1.6}} = 0.17$$

Example: Multilayer Network:

- We now use these values as inputs to the output layer, in order to compute the output value of the network.

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



- Original Inputs

  $x_1 = 1, x_2 = 0$

- Inputs to output layer:

  $h_1 = 0.9, h_2 = 0.17$

- Compute output value:

$$y = \sigma((0.9 \times 4.5) + (0.17 \times -5.2) + (1 \times -2.0)) = \sigma(1.17)$$

$$= \frac{1}{1 + e^{-1.17}} = 0.76$$

Example: Multilayer Network

- We can compute other network outputs for different pairs of binary inputs in the same way:

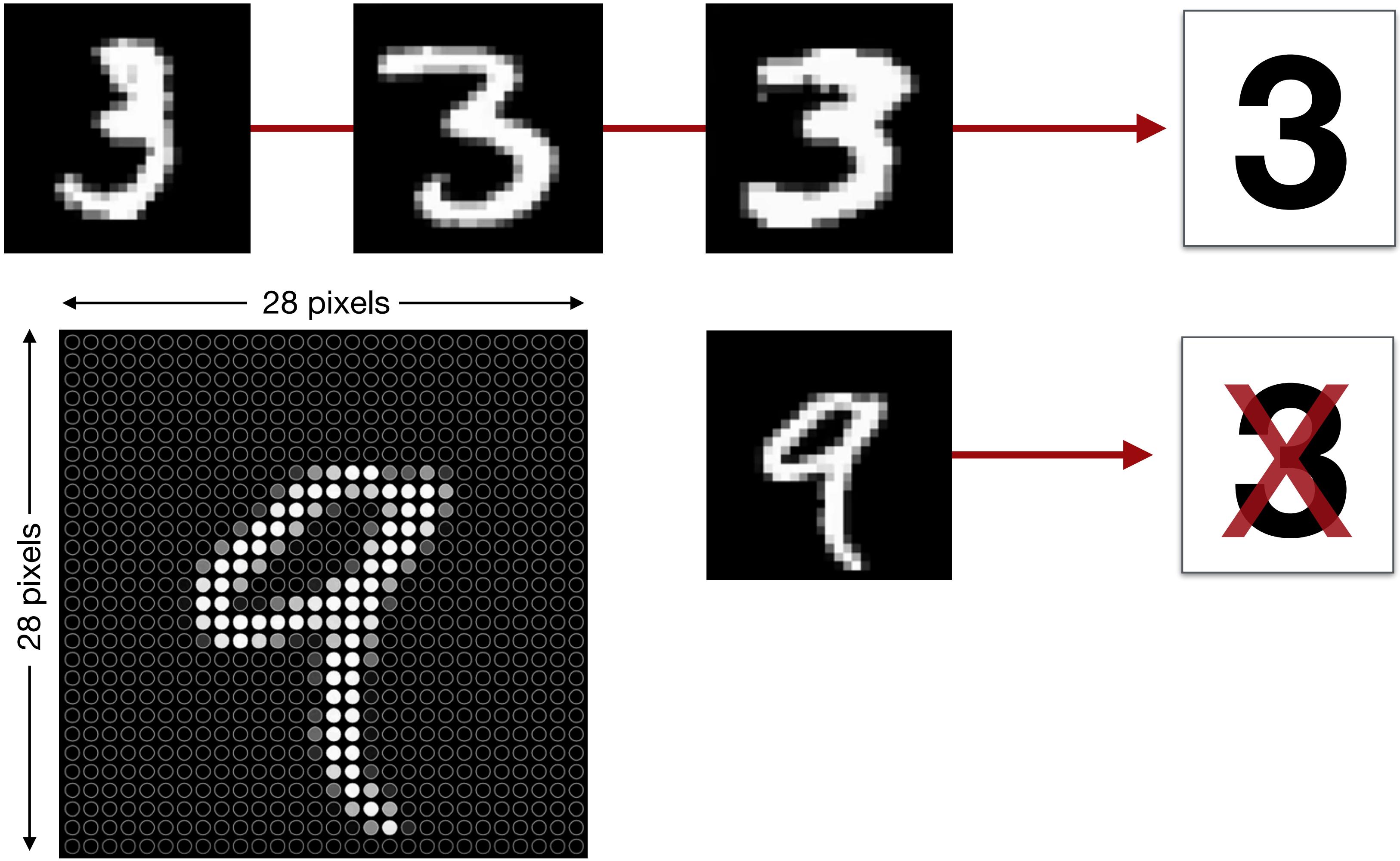
Input x_1	Input x_2	Hidden h_1	Hidden h_2	Output	Approx.
0	0	0.18	0.01	0.23	$\Longrightarrow 0$
0	1	0.90	0.17	0.76	$\Longrightarrow 1$
1	0	0.90	0.17	0.76	$\Longrightarrow 1$
1	1	1.00	0.79	0.17	$\Longrightarrow 0$

- Notice that this network roughly implements the XOR function.
- The hidden node h_1 implements the OR function.
- The hidden node h_2 implements the AND function.

➡ By chaining these, we can solve a more complex problem.

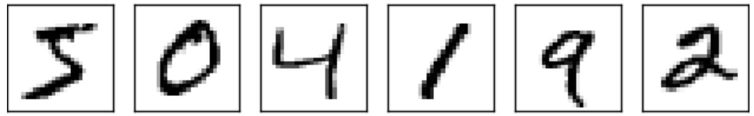
Example: Handwritten Digits

- **Task:** How can we learn to automatically recognise hand written digits, based on their pixel representations?



Example: Handwritten Digits

- **Goal:** Classify handwritten digit images into classes (0,1,…9)
- Input: Training set of many 28x28 pixel images labelled with correct digit.



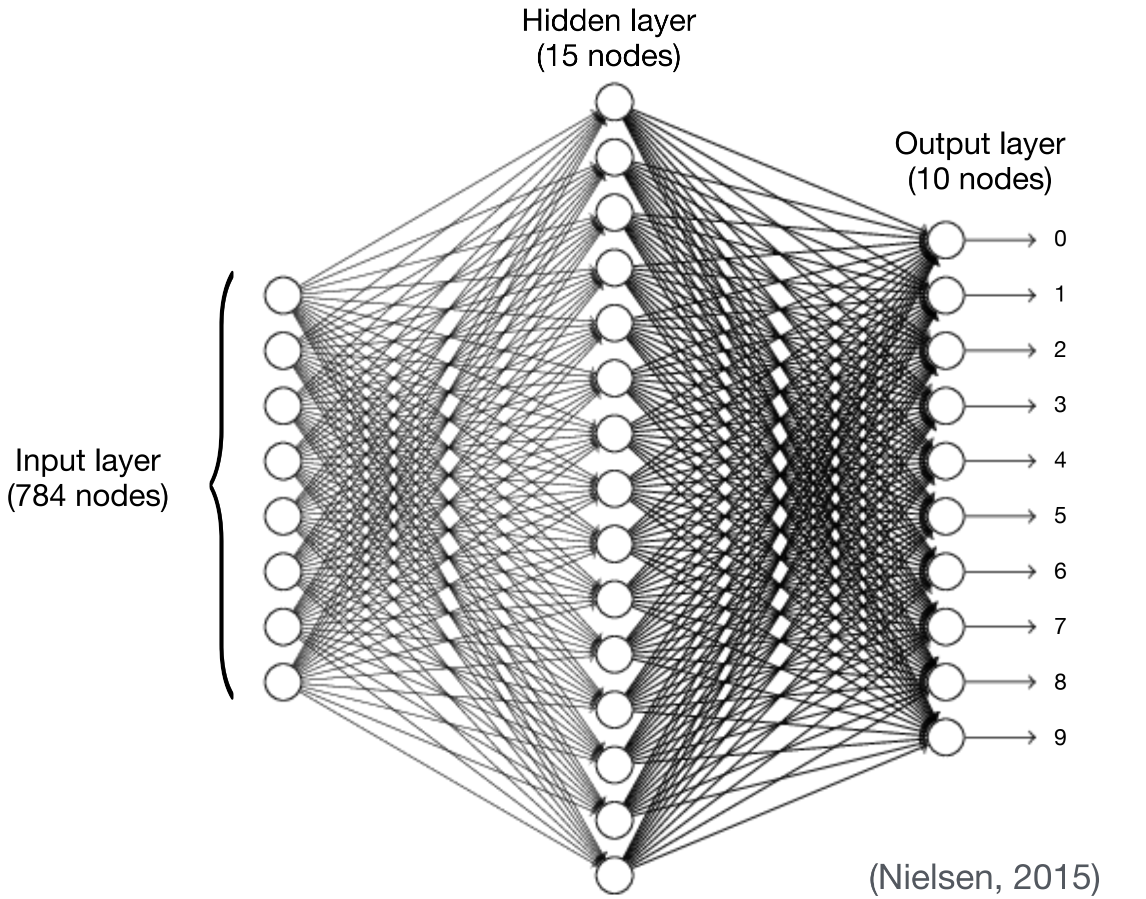
5 0 4 1 9 2

- Neural Network:

  Input layer:
  28x28 pixels=784 neurons

  Hidden layer:
  15 nodes

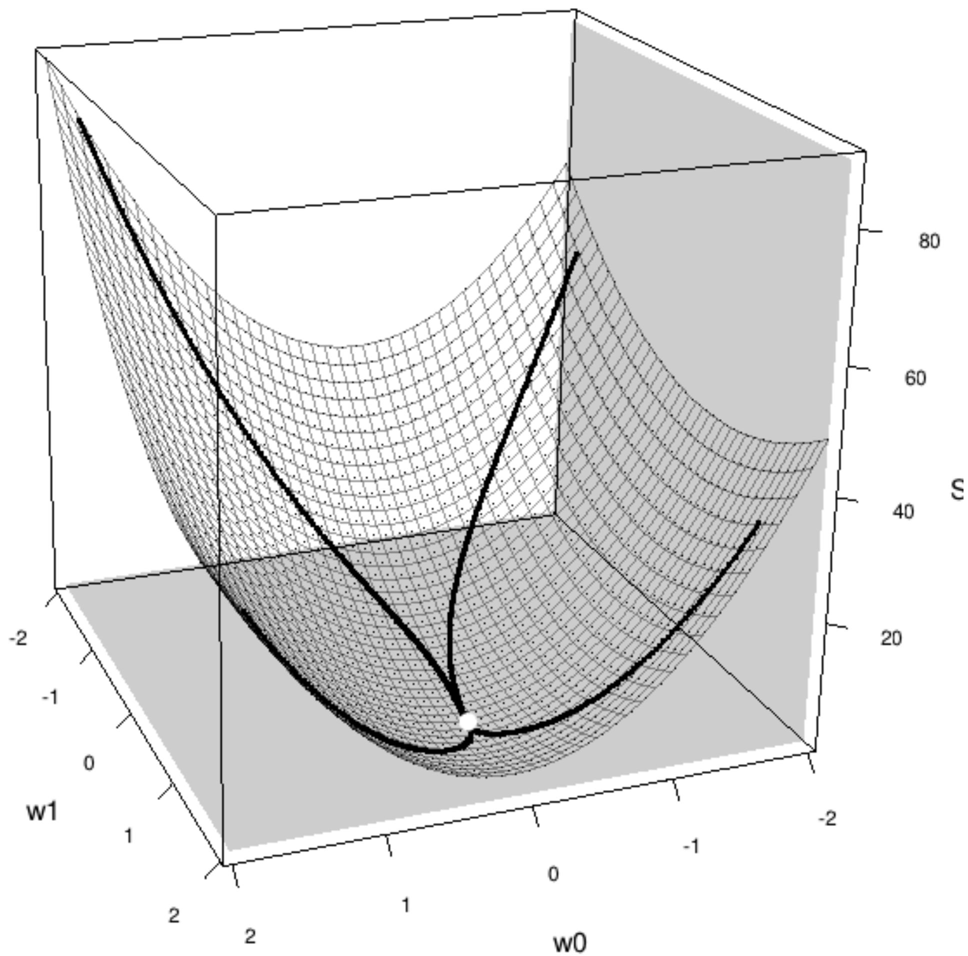
  Output layer:
  One output node per digit.
  To determine which class to to assign for an input, we look at which of the output nodes has the largest value.



(Nielsen, 2015)

Recap: Gradient Descent Algorithm

- Select a random point in the weight space (i.e. each weight is assigned a random value within some sensible range)
- Calculate the loss on the training data, which defines a point on the error surface
- Determine the slope of the error function, by evaluating the derivative at this point on the error surface
- Adjust the weights using the direction of the error surface gradient and move to a new position on the error surface
- Repeat until the global minimum is reached



Adjusting the weights

- Each weight is considered independently and a small value, called a delta value, is added
- Delta value must ensure that the change leads to a movement *downwards* on the error surface
- The *direction* and *magnitude* of the weight adjustment is determined by the gradient of the error surface at the current position
- The error surface is defined by the error or loss function
- The gradient of the error surface is given by the value of the partial derivative of the loss function wrt a particular weight at that point

Gradient Descent Algorithm

Require: set of training instances $\mathcal{D}$

Require: a learning rate α that controls how quickly the algorithm converges

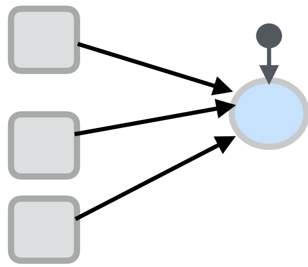
Require: a function, **errorDelta**, that determines the direction in which to adjust a given weight, **w**[*j*], so as to move down the slope of an error surface determined by the dataset, $\mathcal{D}$

Require: a convergence criterion that indicates that the algorithm has completed

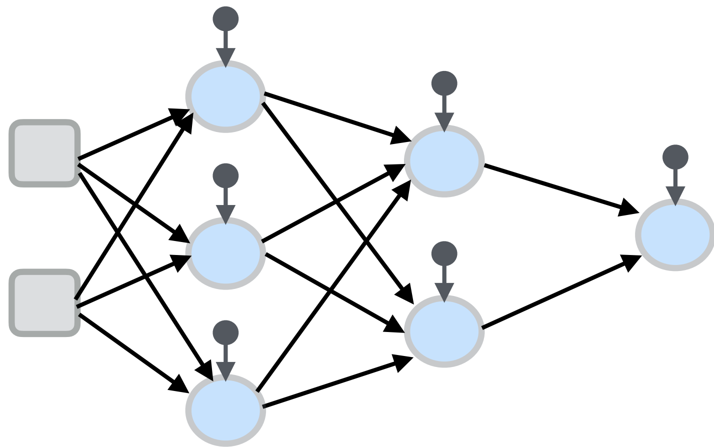
w $\leftarrow$ random starting point in the weight space
repeat
    for each **w**[*j*] in **w do**
        w[*j*] $\leftarrow$ **w**[*j*] $+ \alpha \times$ **errorDelta**($\mathcal{D}$, **w**[*j*])
    end for
until convergence occurs

Training a Neural Network

- A node is structurally equivalent to a logistic regression model so a logistic unit can be trained by gradient descent
- Nodes that use other activation functions can also use gradient descent as long as the activation function is differentiable
- Easy to train a perceptron using gradient descent



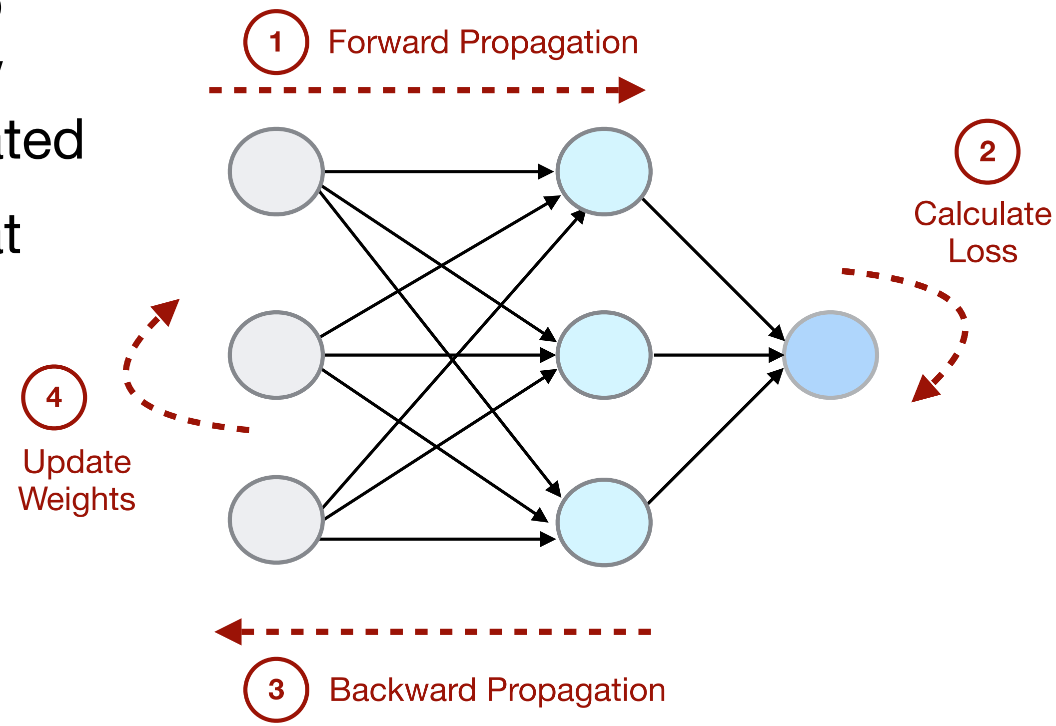
- How do we train a network with hidden layers?



How do you calculate the error gradient at nodes in the hidden layers?

Back Propagation

- Need a measure of how much each node contributed to the overall error of the network at the output layer, known as blame assignment
- The Back Propagation algorithm provides an estimate of gradient error at each node and then the weights can be updated before the next iteration
- Steps:
  1. Training instance is presented to the network and activations flow forward until an output is generated
  2. Error or loss is calculated for that training instance
  3. This error is shared back (propagated) through the network on a layer-by-layer basis to input layer
  4. Weights are updated



Back propagation of the error

- During back propagation a share of the error δ is calculated for each node from the output node backwards
- The δ term describes the rate of change of the error of the network wrt changes in the weighted sum at that node (z)
  - i.e. the partial derivative of the loss function wrt weighted sum of the node
- Calculated as the product of two terms:
  - rate of change (partial derivative) of the error (loss function) wrt changes in the activation
  - rate of change (partial derivative) of the activation of the node wrt changes in the weighted sum
- Requires: + activation function to be differentiable
  + storing the weighted sum of each node during the forward pass

Updating the weights

- Weights are initialised randomly
- Updated using:

$$w_{i,k} \leftarrow w_{i,k} - \alpha \times \delta_i \times a_k$$

where α is the learning rate
δ_i is the gradient error at node i
a_k is the activation at node k

$\delta_i \times a_k$ represents the sensitivity of the error of the network wrt changes in the weight

Batch vs Stochastic Gradient Descent

- Stochastic gradient descent updates weights after each training example
  - Slower as liable to move orthogonal to gradient
- Batch gradient descent involves calculating error gradients for each weight for all example in the dataset, summing the gradients for each weight, and only then updating weights
  + Smoother descent so can use a larger learning rate
  + Can process multiple examples in parallel —> Faster
  - Training datasets can be large —
- Use mini-batch gradient descent

- An epoch is a single pass through all training examples
- An iteration is a single forward, backward and weights update

How many iterations in an epoch in stochastic and batch GD?

Handling Categorical Target Features

- To perform multi-class prediction with a ANN:
- Represent the target feature using one-hot-encoding
- Change the output layer to a softmax layer
  - One node for each class
  - Softmax activation function in the output/softmax layer

$$\varphi_{sm}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_m}}$$

  - Activation of the node can be interpreted as a probability of the class that the node represents
- Change the error (loss) function to be the cross-entropy function which measures the dissimilarity between the true probability distribution $\mathbf{t}$ and the predicted distribution $\hat{\mathbf{P}}$

$$L_{CE}\left(\mathbf{t}, \hat{\mathbf{P}}\right) = -\sum_{j} \mathbf{t}_j \ln\left(\hat{\mathbf{P}}_j\right)$$

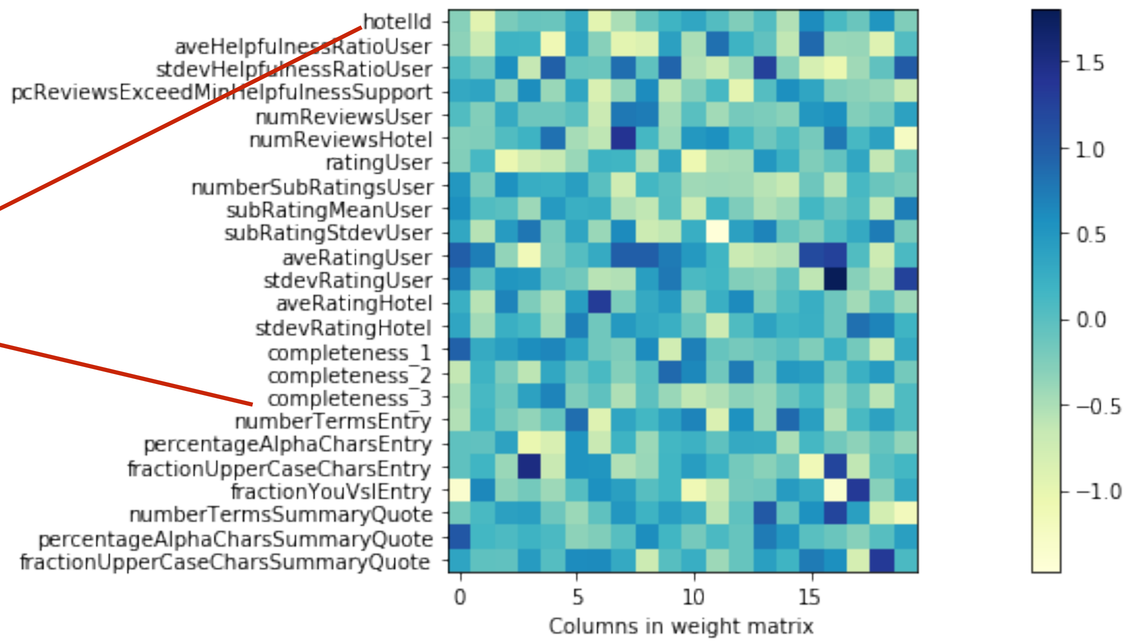
Neural Networks in sklearn

```
y = HR.pop('reviewHelpfulness').values
scaler = preprocessing.StandardScaler().fit(HR)
X_scaled = scaler.transform(HR)
In [55]:
mlp = MLPClassifier(max_iter=2000, random_state=2,
                    hidden_layer_sizes=[20])
mlp.fit(X_scaled, y)
acc = mlp.score(X_scaled, y)
print("Accuracy on training set: {:.2f}".format(acc))

Accuracy on training set: 1.00
```

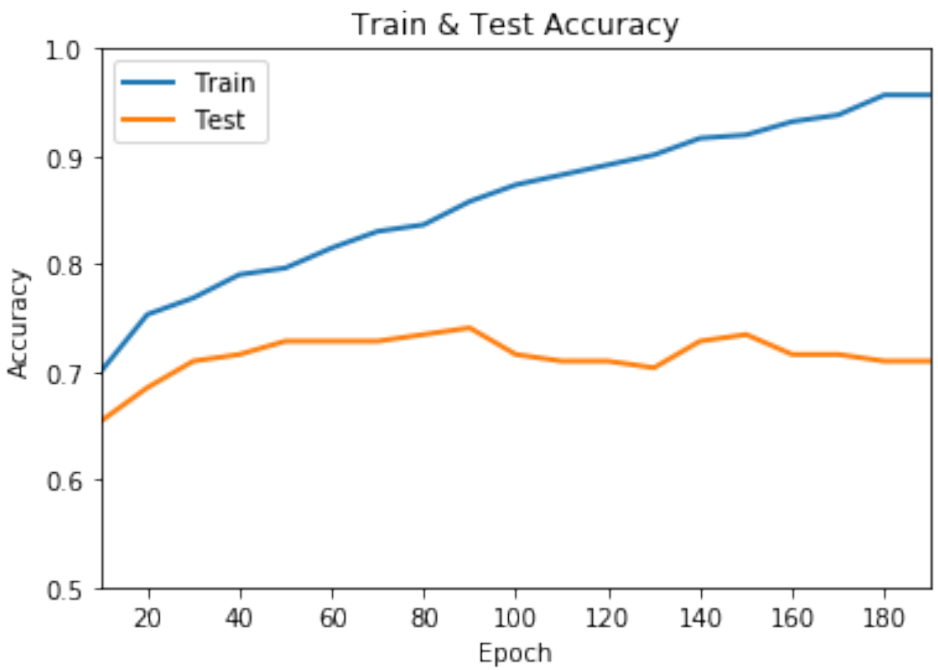
Weights

Some features not Influential

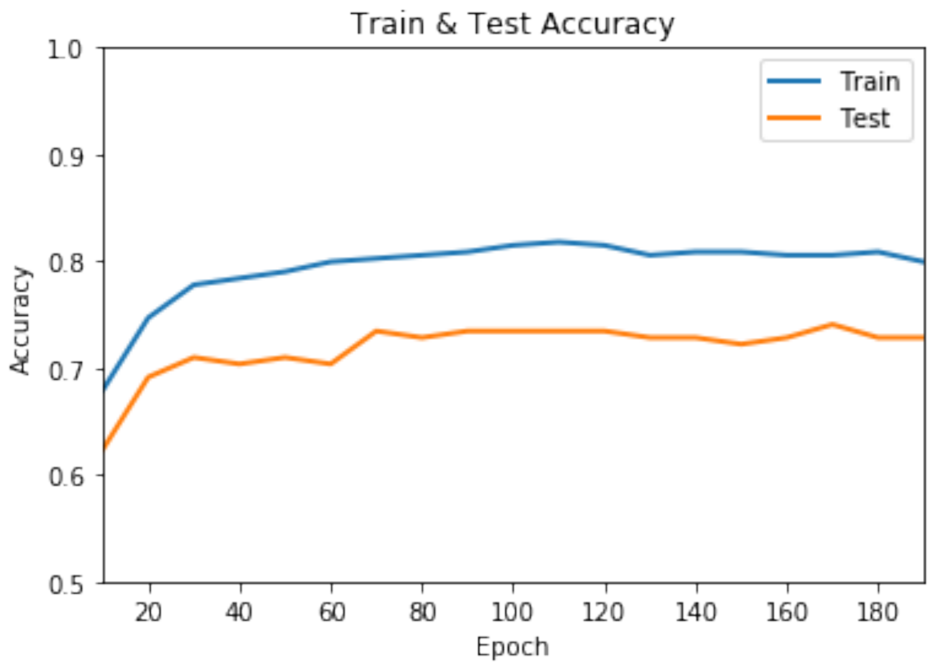


Overfitting

HoldOut Test



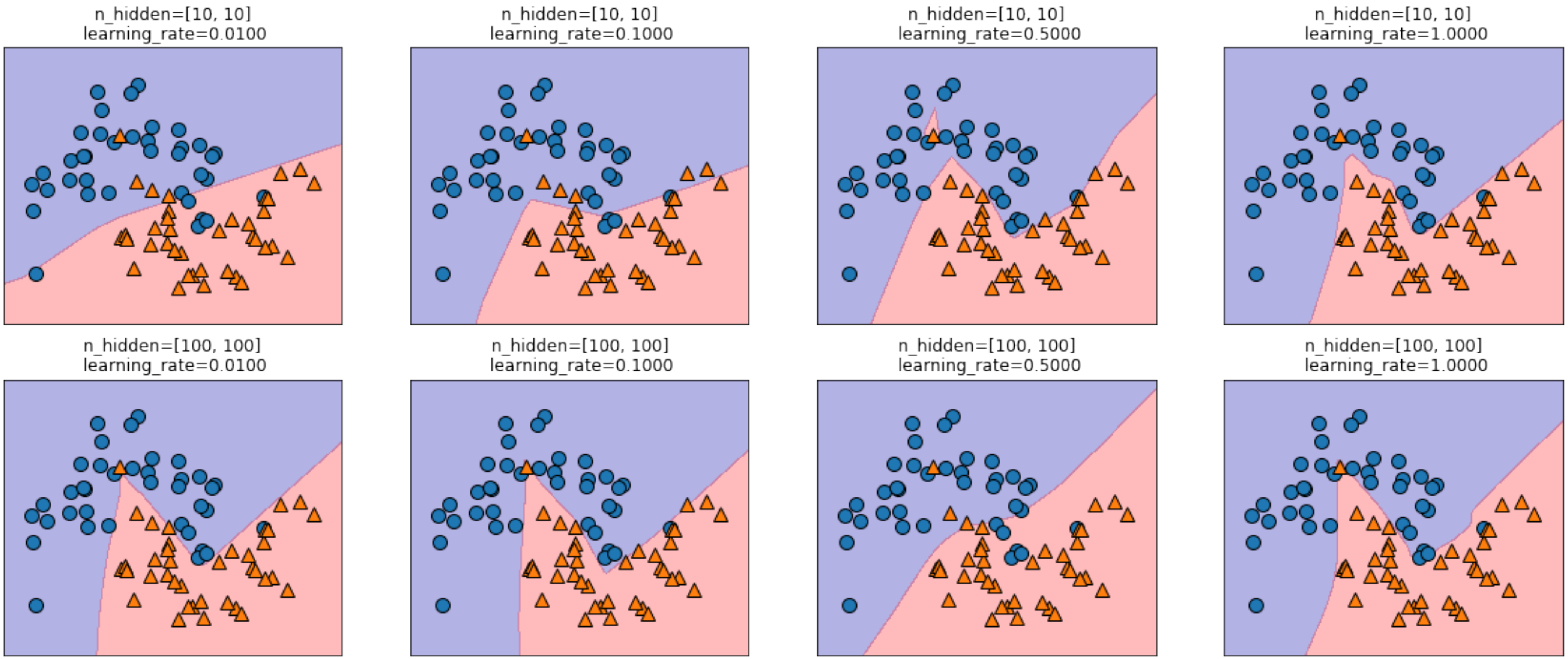
Use regularisation to control overfitting (alpha = 5)



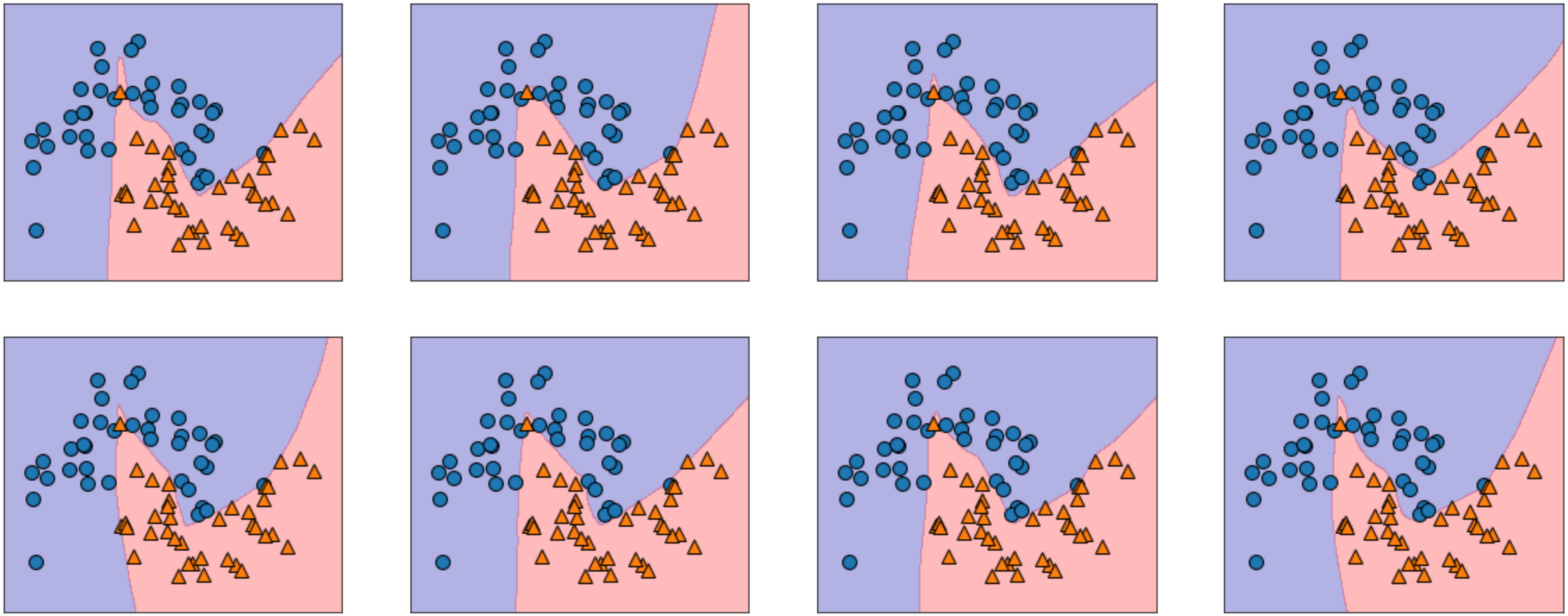
- *alpha* is a parameter that restricts the size of the weights
- Overfitting can also be controlled by restricting network complexity,
  - e.g. reduce units in hidden layer

NNs sensitive to model parameters

- 2 hidden layers [10,10] or [100,100]
- Learning rate 0.01, 0.1, 0.5, 1



Also sensitive to weight initialisation



What are Neural Networks good for?

- **Advantages**
  - Can learn and model non-linear and complex relationships.
  - Work well when training data is noisy or inaccurate.
  - Fast performance once a network is trained.
- **Disadvantages**
  - Often require a large number of training examples.
  - Training time can be very long.
  - Network is like a “black box”. A human cannot look inside and easily understand the model or interpret the outputs.