# Introduction to Artificial Intelligence

**COMP307/AIML420**

**Neural Networks 2: Back Propagation**

Dr Andrew Lensen

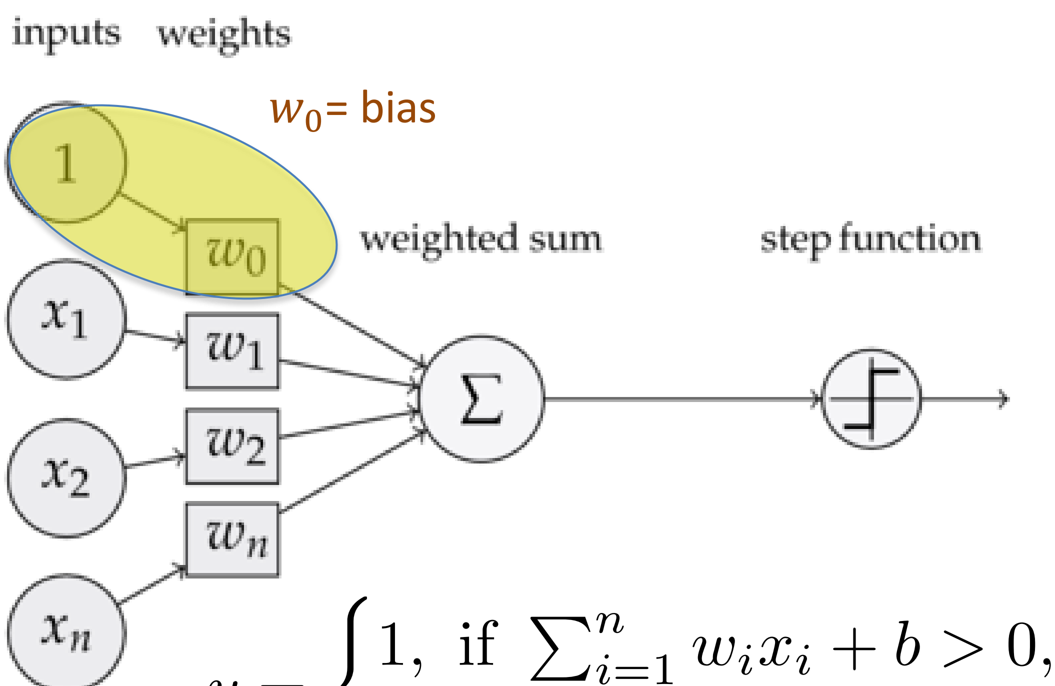
Andrew.Lensen@vuw.ac.nz

# Outline

- Revisiting (Multi-layer) Perceptron
- Feed forward neural network
- Back propagation algorithm to train neural network

# The Perceptron

- A *special* type of artificial neuron
  - Real-valued inputs
  - Binary output
  - Threshold activation function




$$y = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} w_i x_i + b > 0, \\ 0, & \text{otherwise.} \end{cases}$$

# The Perceptron

- Bias or Threshold?
  - They are essentially the same: bias = – threshold

$$y = \begin{cases} 1, & \text{if } \sum_{i=1}^{m} w_i x_i + b > 0 \\ 0, & \text{otherwise} \end{cases} \qquad y = \begin{cases} 1, & \text{if } \sum_{i=1}^{m} w_i x_i - T > 0 \\ 0, & \text{otherwise} \end{cases}$$

- Simplify notation: let $x_0 = 1$, $b = -T = w_0 = w_0 x_0$

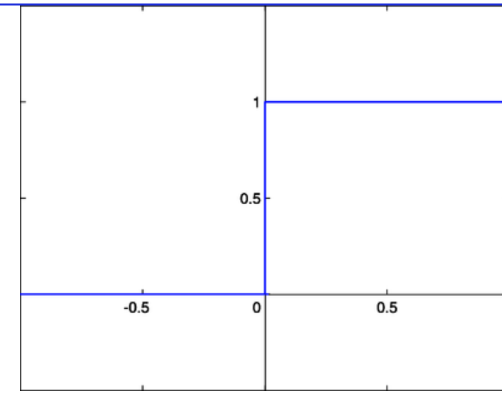
$$y = \begin{cases} 1, & \text{if } \sum_{i=0}^{m} w_i x_i > 0 \\ 0, & \text{otherwise} \end{cases}$$

- So, we have one block of code for changing all the "weights" rather than changing weights and biases separately

# Multi-Layer Perceptron (MLP)

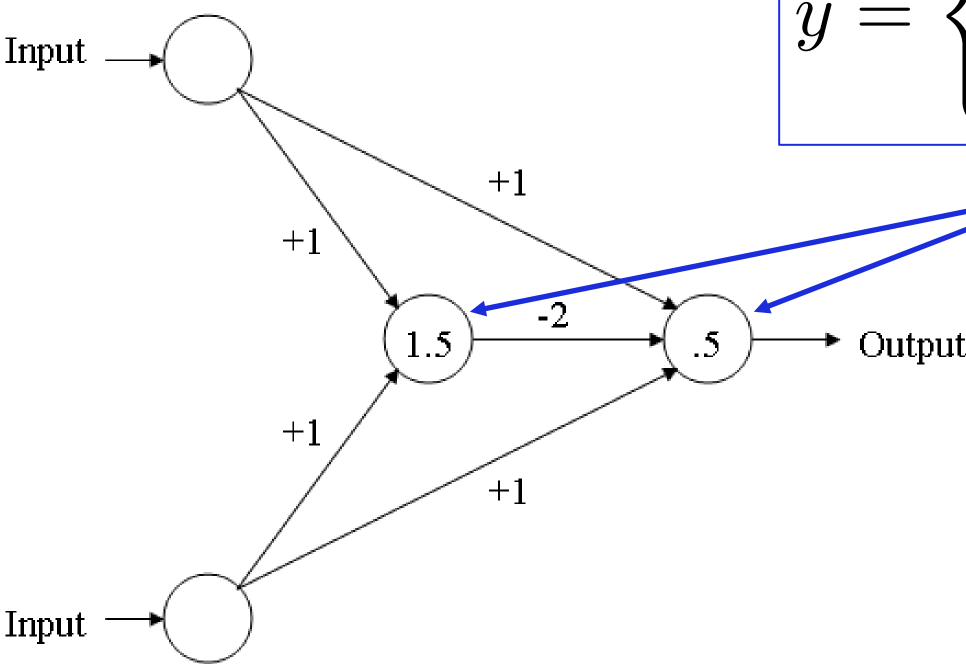
- Add one ***hidden*** node between the inputs and output

| x1 | x2 | y (class) |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |



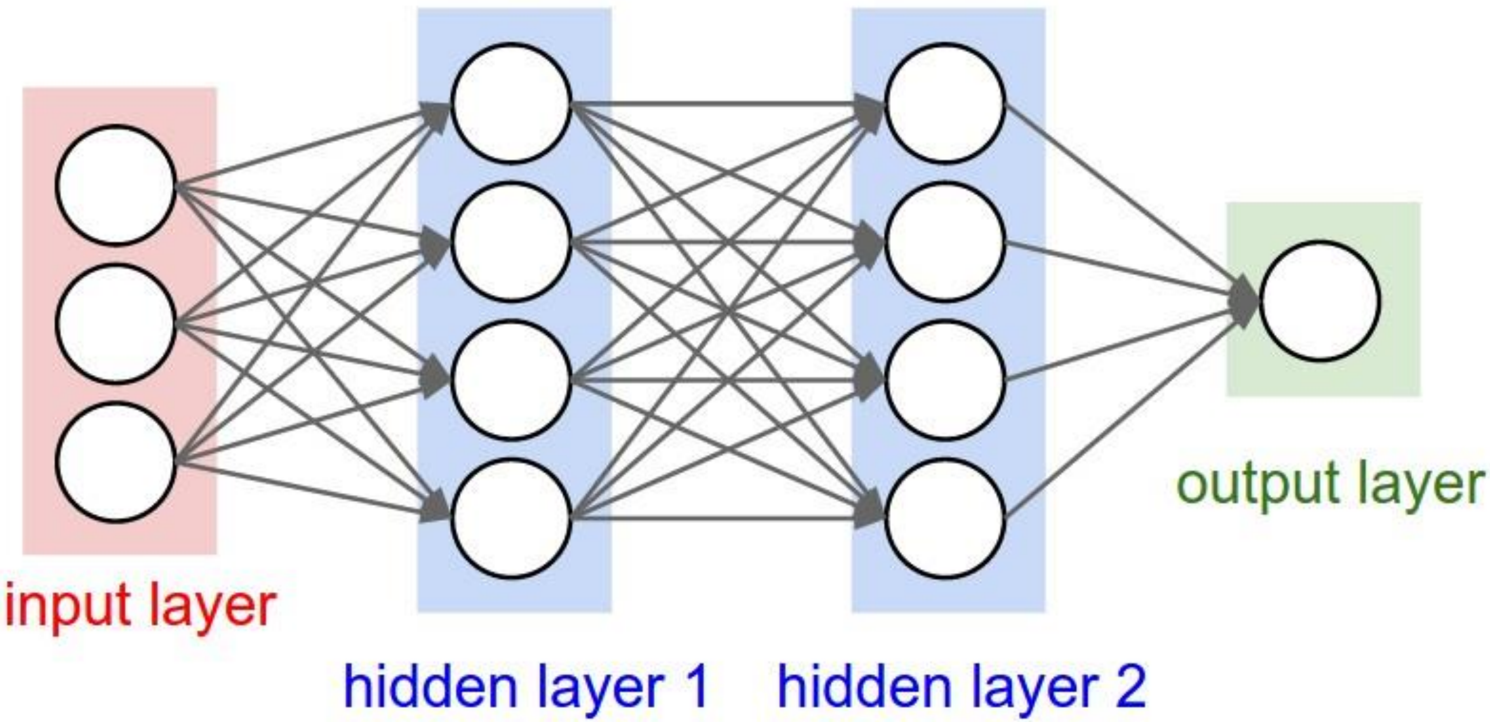

$$y = \begin{cases} 1, & \text{if } \sum_i w_i x_i - T > 0, \\ 0, & \text{otherwise} \end{cases}$$

Threshold

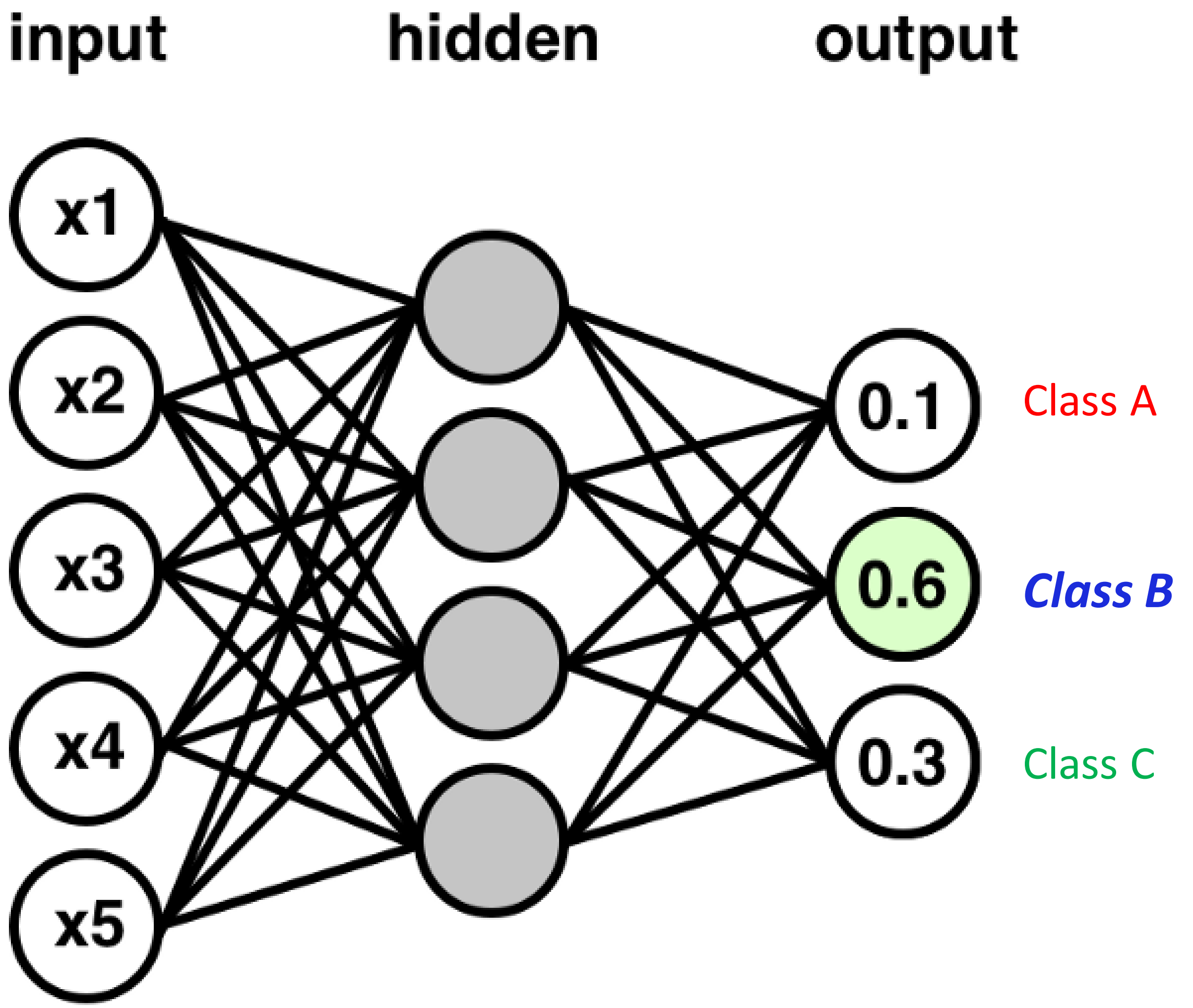


# Feedforward Neural Network

- A more general form of perceptron
  - Most common type of Artificial Neural Network (ANN)
  - Multiple (hidden) layers, multiple nodes in each layer
  - Each node connects to its adjacent layers
  - Fully connected, NO jump connections
  - A **lot** of weights: one per link + one bias per node



# NNs for (Multi-Class) Classification

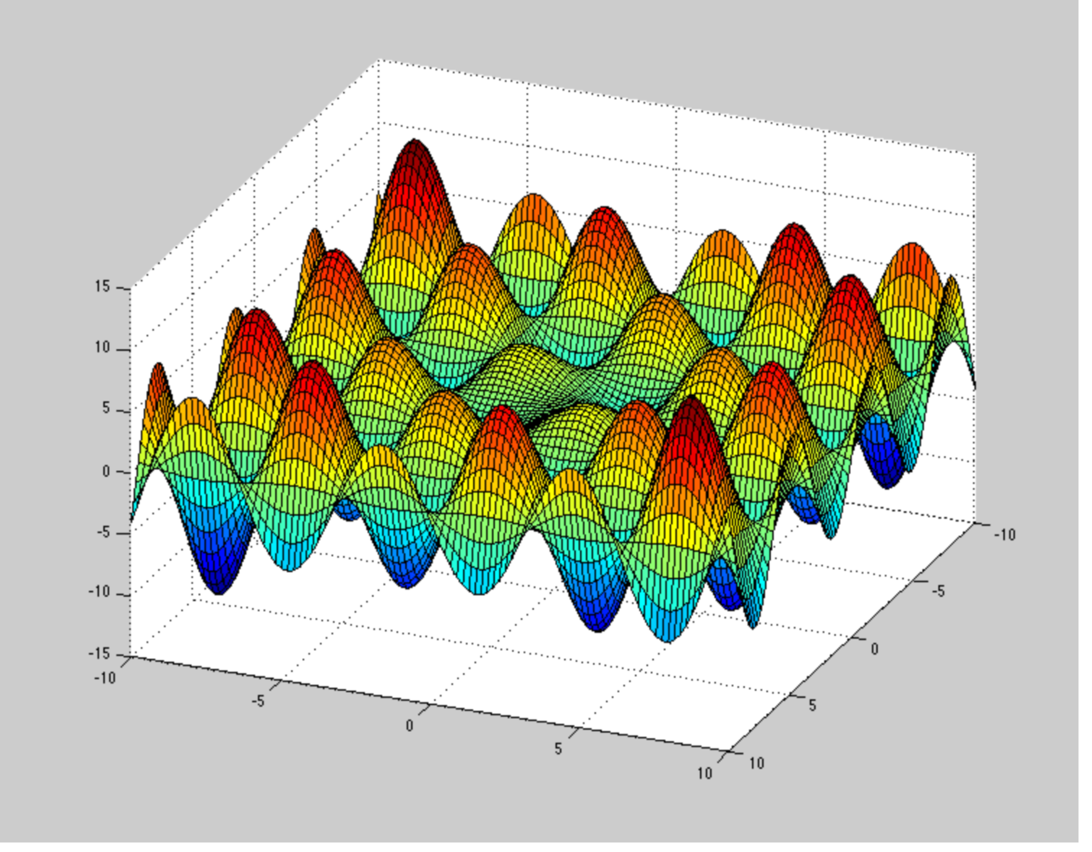


# How Can We Learn ANN Weights?

- A complex optimisation problem!

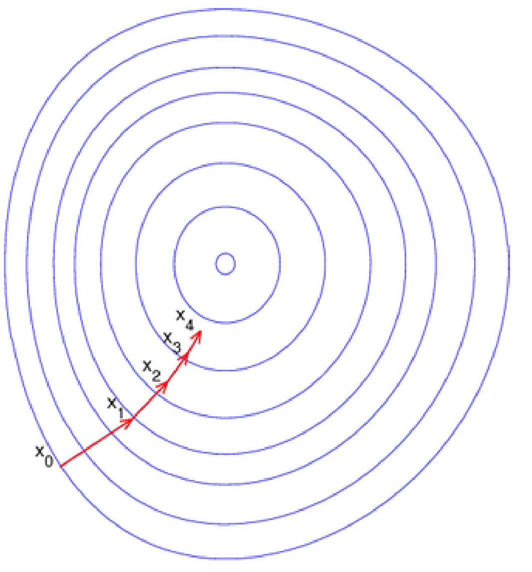
$$\min error = f(w_{ij})$$

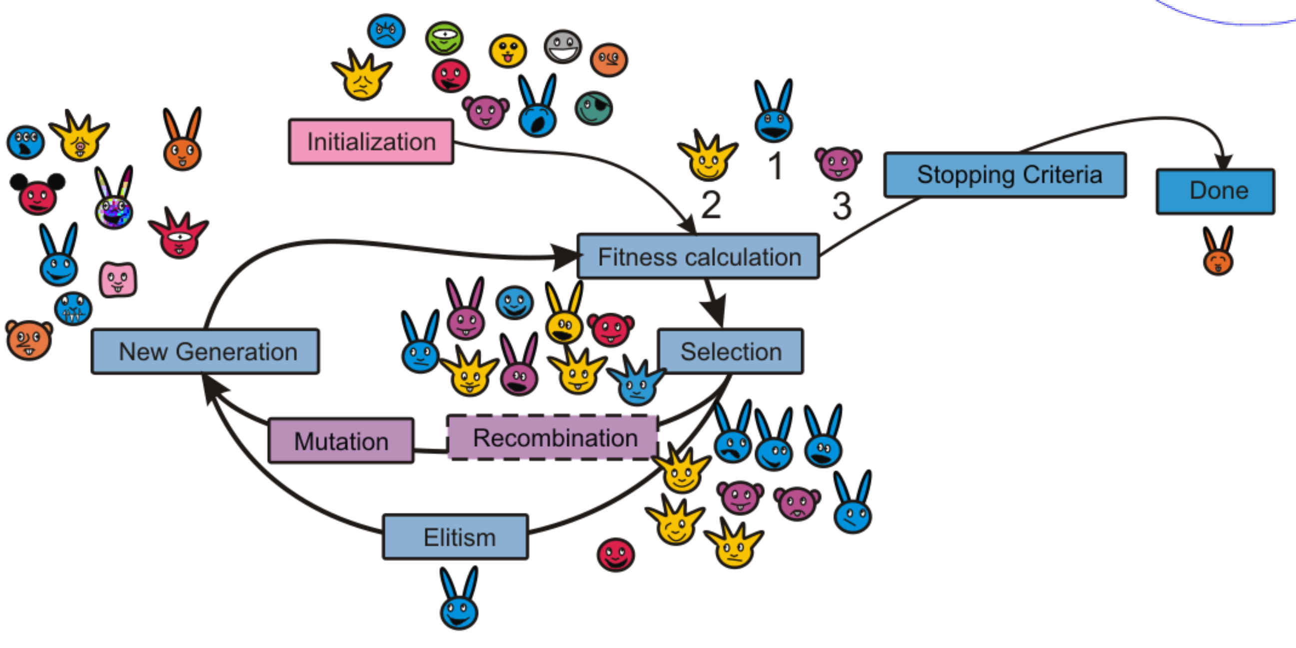
- Usually non-convex (many local optima)
- Extremely high-dimensional
- Not feasible to solve by using exact methods



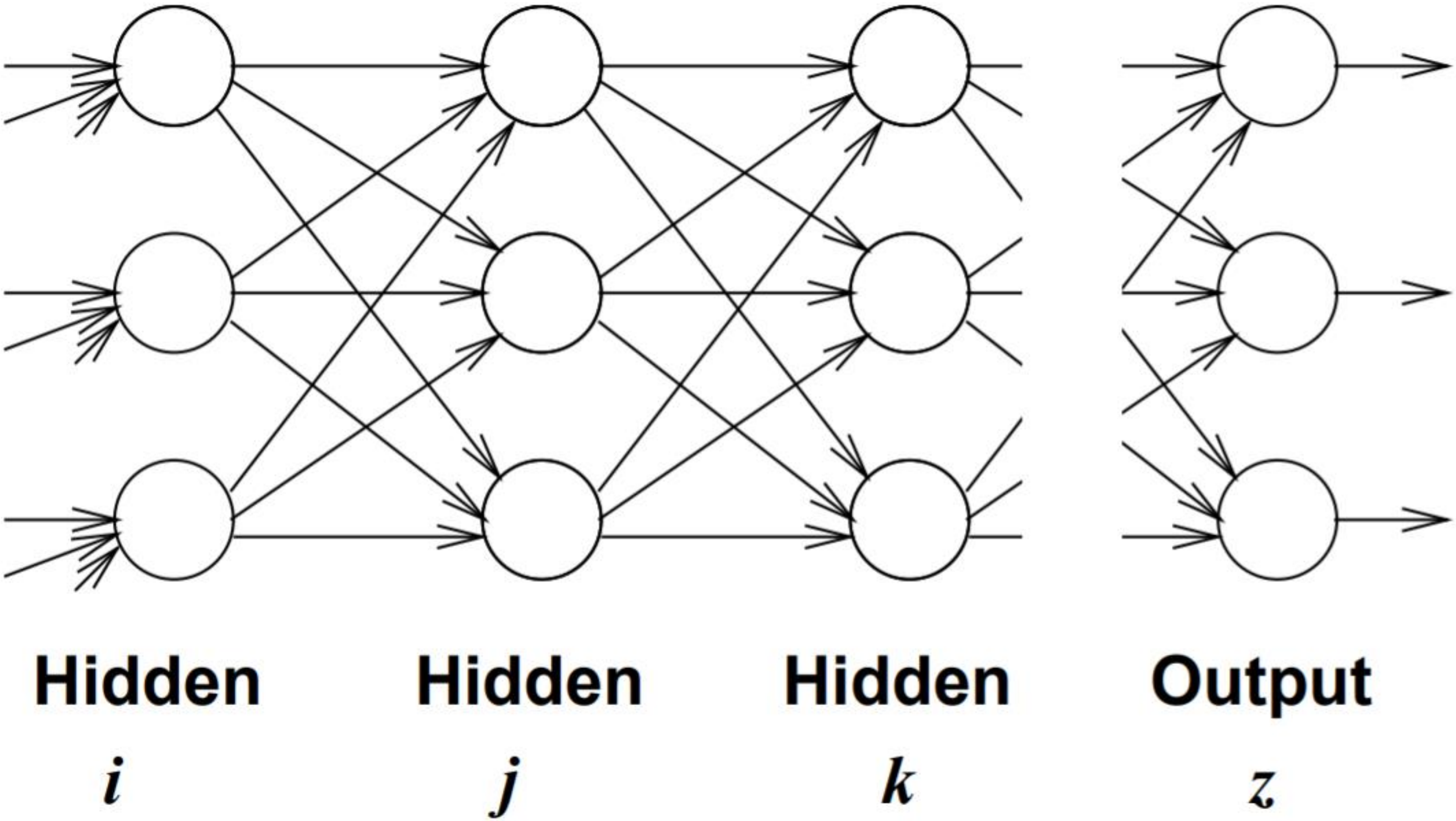
# Learning ANN Weights

- Approximate methods
  - Hill climbing (local search)
  - (Stochastic) gradient descent search
  - Simulated annealing
  - Tabu search
  - Evolutionary computation
  - …



# Training a Neural Network



# Training a Neural Network

- **Initialise** the weights (randomly)

- **Feedforward**
  - For each example, calculate the predicted outputs $o_z$ using the current weights
  - Calculate the total *error* $\sum_z (d_z - o_z)^2$

- If the error is small enough, we can stop.

- Otherwise, we use **back propagation** to adjust the weights to make the error *smaller*.
  - Uses gradient descent (GD)

# Back Propagation (BP) Algorithm

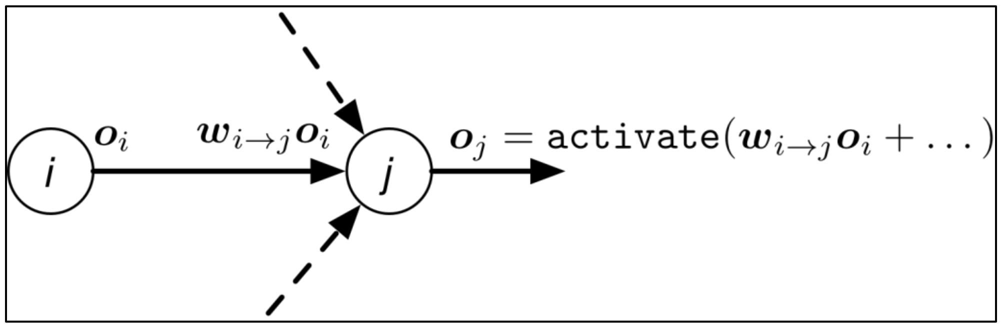
- Estimate the **contribution (gradient)** of each weight to the *error*, i.e. how much the error will be reduced by changing the weight by the gradient.

- Change each weight (simultaneously) proportional to its contribution to reduce the error as much as possible
  - Move in the direction of the **steepest gradient**

- We calculate the contribution/gradient backwards (from the last/output layer to the first hidden layer)
- Error of a single **output** node is $d_z - o_z$
  - $d_z$ means "*desired*"
  - $o_z$ means "output" (i.e. what we *actually* got)

# Back Propagation (BP) Algorithm

- How big a change should we make to weight $w_{i \to j}$?
  - Make a big change if will improve error a lot (big contribution)
  - Make a small change if little effect on error (small contribution)

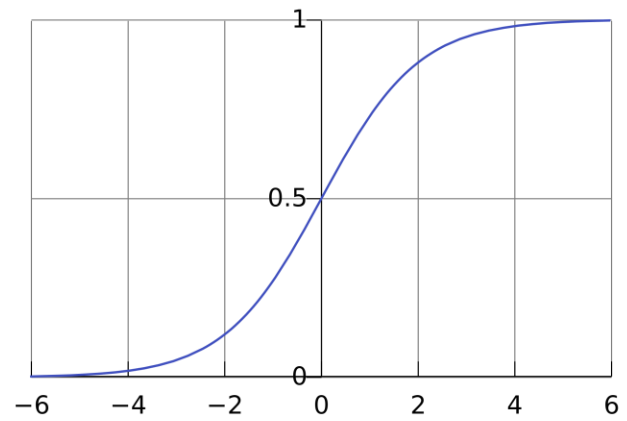



$w_{i \to j}$ $\quad w_{i \to j} o_i$ $\quad o_j = act(w_{i \to j} o_i + \cdots)$ $\quad$ … $\quad error$

- $\beta_j$ is how "*beneficial*" a change is for node $j$ ("error term")
- When changing $w_{i \to j}$, the error change should be:
  - Proportional to the output: $o_i$ (larger output = more effect)
  - Proportional to the *slope* of the activation function at node $j$: $slope_j$
  - Proportional to error term of j ($\beta_j$)

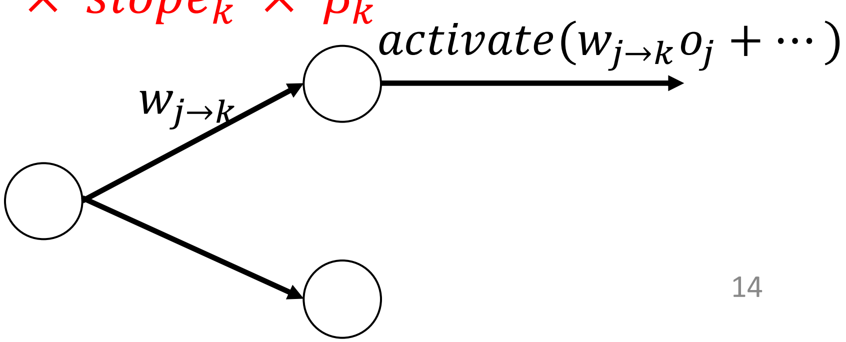
# Back Propagation (BP) Algorithm

- How to calculate $slope_j$?
  - Some calculus knowledge: derivative of the activation function
  - Steeper (larger) the slope, larger the effect of changing the weight
  - We don't expect calculus in this course!




- How to calculate $\beta_j$?
  - *Back-propagated* from later layer

  - The **output layer**: the error $\beta_z = d_z - o_z$
  - **Other layers**: error is $\beta_j = \sum_k w_{j \to k} \times slope_k \times \beta_k$



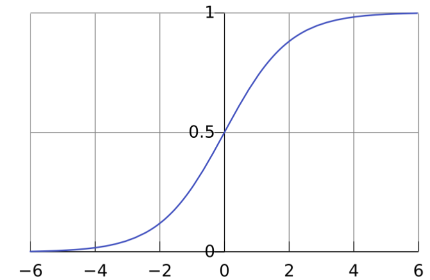
# Back Propagation (BP) Algorithm

- Assume a neural network with:
  - Activation function: sigmoid

$$slope_j = o_j(1 - o_j)$$

  - Target: minimise total sum squared error

$$error = \frac{1}{2} \sum_{s \in examples} \sum_{c \in classes} (d_{sc} - o_{sc})^2$$




$$o_j = \frac{1}{1+e^{-ws_j}}$$

- Output node:

$$\beta_z = d_z - o_z$$

Makes the maths easier! (differentiation)

- Hidden node:

$$\beta_j = \sum_k w_{j \to k} o_k (1 - o_k) \beta_k$$

# BP Algorithm Implementation

- Let $\eta$ be the learning rate ("eta"…)

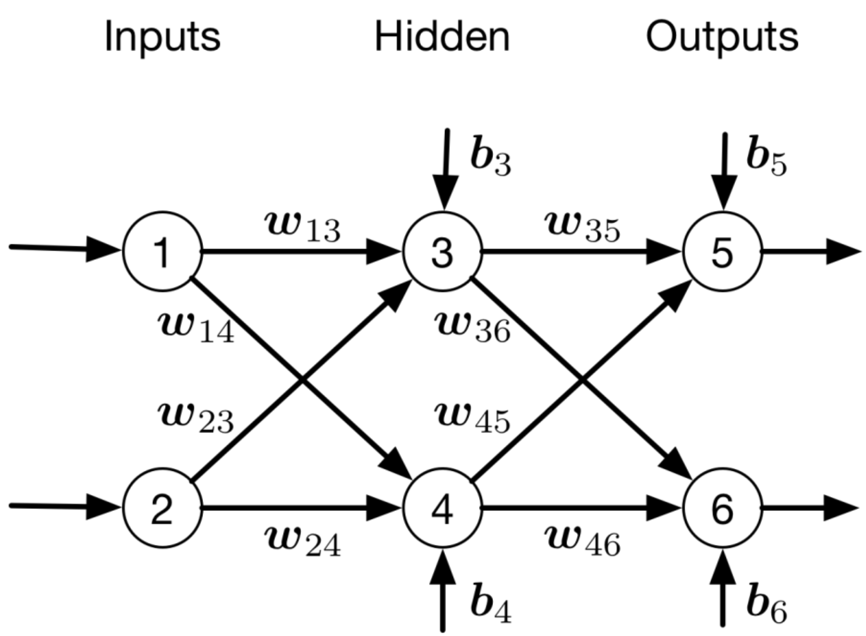
- Initialise all weights (+bias) to small random values

- Until total error is small enough, repeat:
  - For each input example:
    - Feed forward pass to get predicted outputs
    - Compute $\beta_z = d_z - o_z$ for each output node
    - Compute $\beta_j = \sum_k w_{j \to k} o_k (1 - o_k) \beta_k$ for each hidden node (working backwards from last to first layer)
    - Compute (+store) the weight changes for all weights
      $\Delta w_{i \to j} = \eta o_i o_j \left(1 - o_j\right) \beta_j$ (proportional to all 3 factors)
  - Sum up weight changes for all input examples
  - Change weights!

# BP Algorithm Example

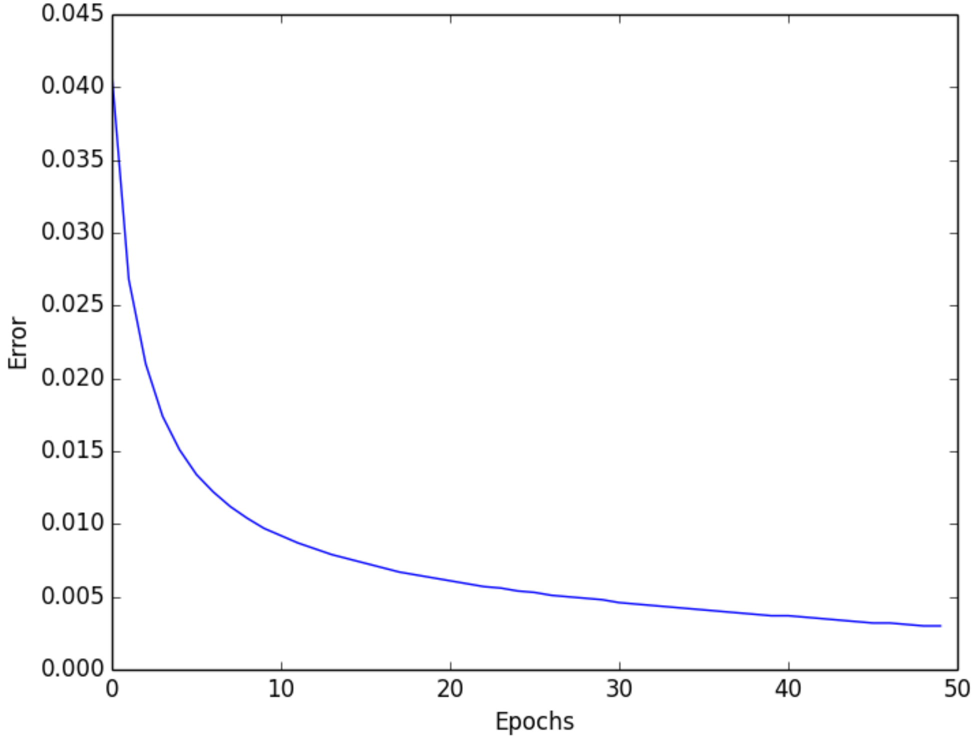
- Calculate one pass of the BP algorithm given the example (feedforward + back propagation)

| Inputs | | Outputs | |
|---|---|---|---|
| $I_1$ | $I_2$ | $d_5$ | $d_6$ |



# Notes on BP Algorithm

- *1 Epoch*: all input examples (entire training set, batch, …)
- A target of 0 or 1 cannot reasonably be reached. Usually interpret an output > 0.9 or > 0.8 as '1'
- Training may require *thousands* of epochs. A convergence curve will help to decide when to stop (over-fitting?)



# Summary

- (Multi-layer) Perceptron
  - Bias and threshold are essential the same
  - Simplify the notation

- Feedforward neural network

- Back propagation
  - Gradient descent
  - Feedforward + error back propagation -> weight change