

# ML Study Group

MEET UP #1

**An Introduction to Deep Learning  
and  
The Multilayer Perceptron**

Shaun Irwin

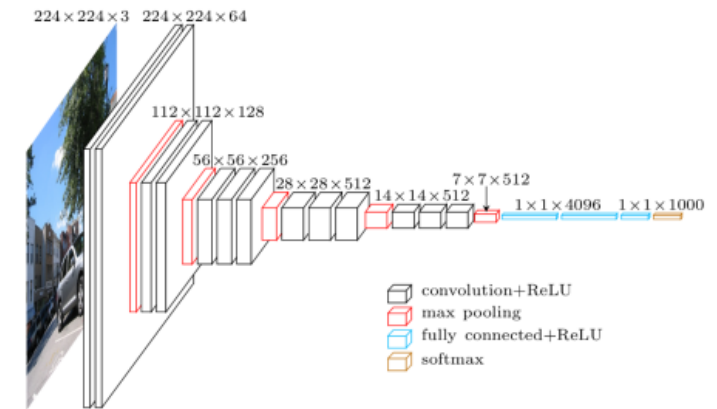
# Overview

The Concept of Deep Learning

Single Layer Perceptron

Multilayer Perceptron

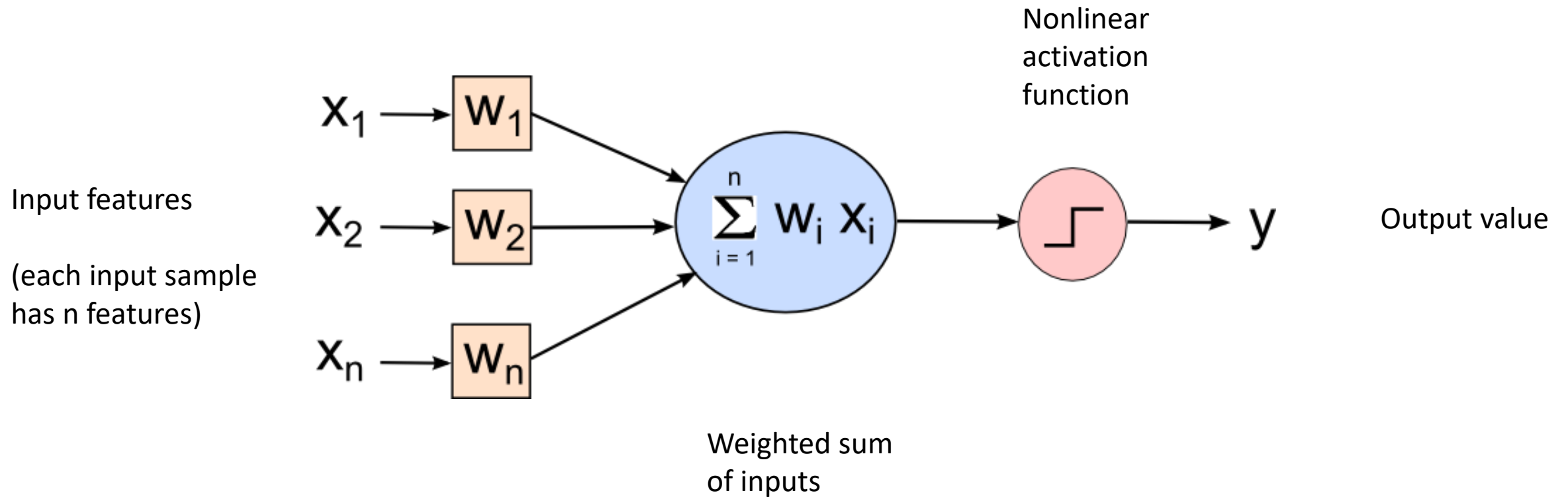
# Deep Learning



- **Neural nets with many layers of neurons**
  - Often millions or billions of neurons (some are 100s of layers deep)
  - The layers of neurons transform the input data into a linearly separable representation (for a classification problem)
  - Able to fit extremely complex functions
    - State-of-the-art results on computer vision, language translation and other domains
- **(Relatively) efficient methods to optimise parameters**
  - Stochastic gradient descent is able to efficiently calculate a (local) optimum for the parameters despite having so many parameters to solve for
- **Well suited to GPUs**
  - Mostly matrix operations => Highly parallelisable
- **Difficult to understand what the networks are learning!**
  - Due to the vast size of the networks, understanding what they are learning is not always apparent
  - Lots of research into incorporating uncertainty estimates and explainability into the networks (important for use in safety critical applications)

# The Perceptron

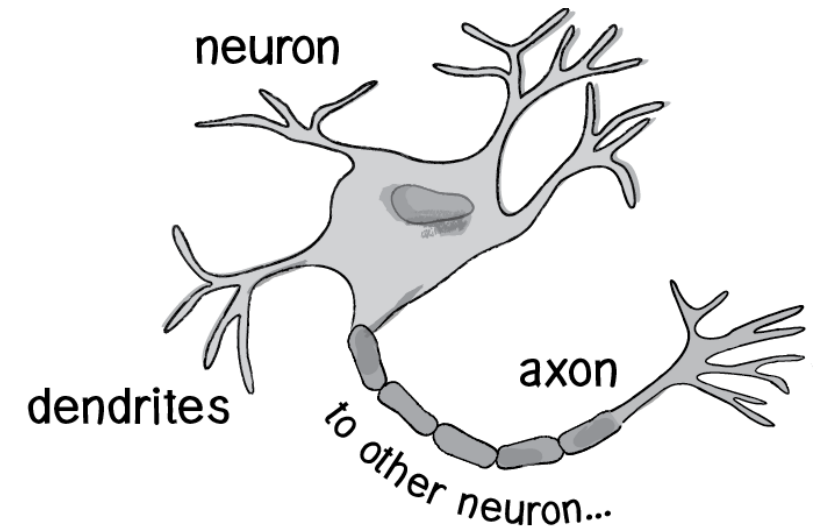
The basic building block of many deep neural networks



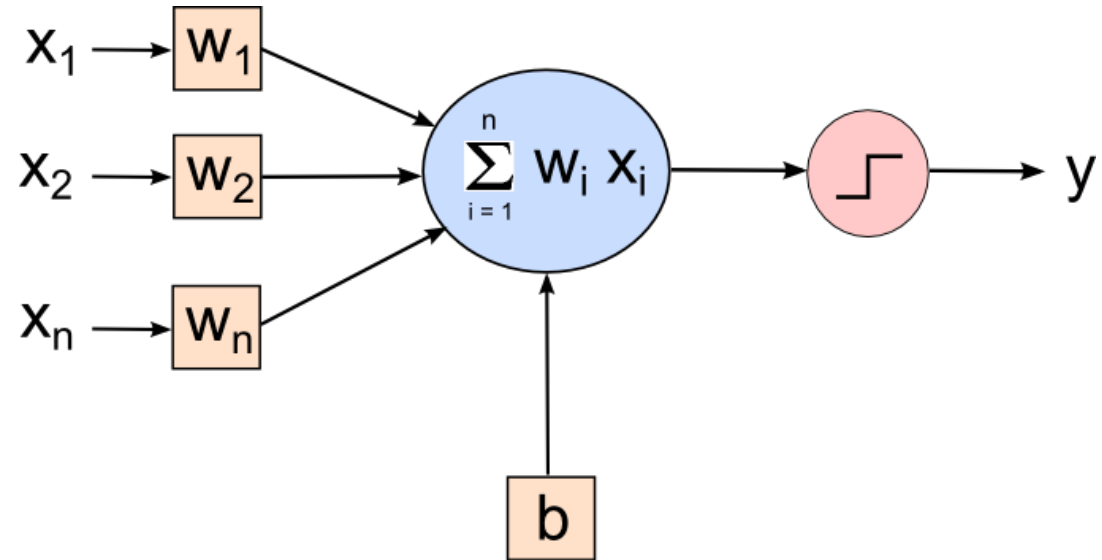
# Biological neurons

The perceptron was inspired by and *loosely* based on biological neurons:

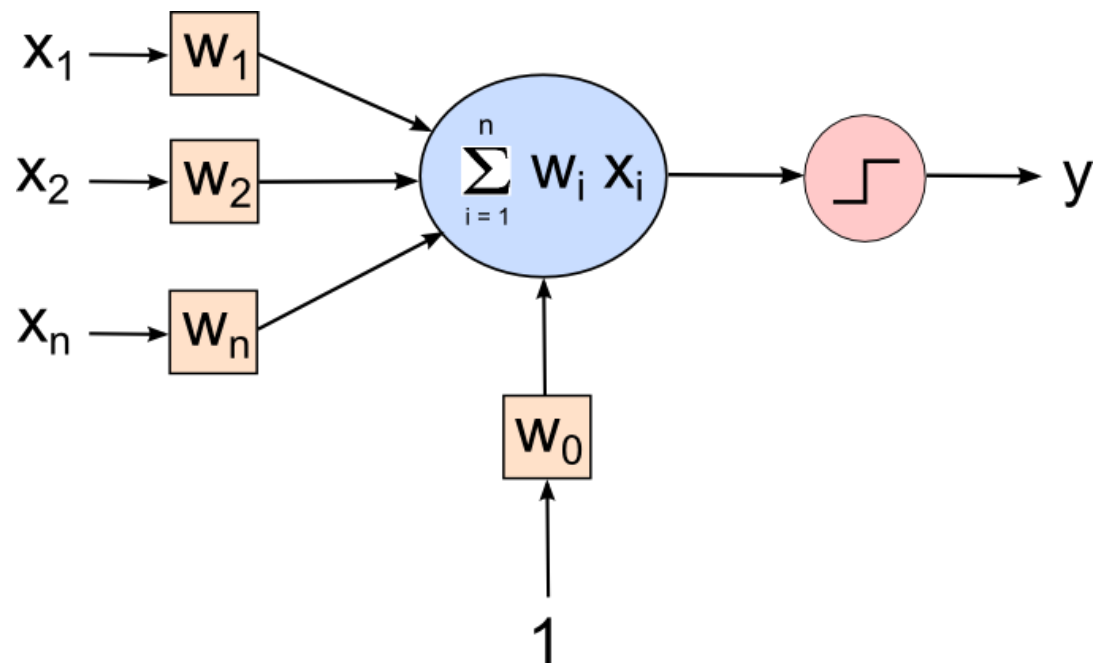
- Each neuron receives input signals from potentially 1000s of other neurons
- “All-or-None” principle:
  - A neuron is either activated or not
  - If a neuron reacts, it reacts completely
  - Increasing its stimulus further does not increase its output intensity
    - Although the frequency of its firing can increase
- A human brain has an estimated ~80 billion neurons



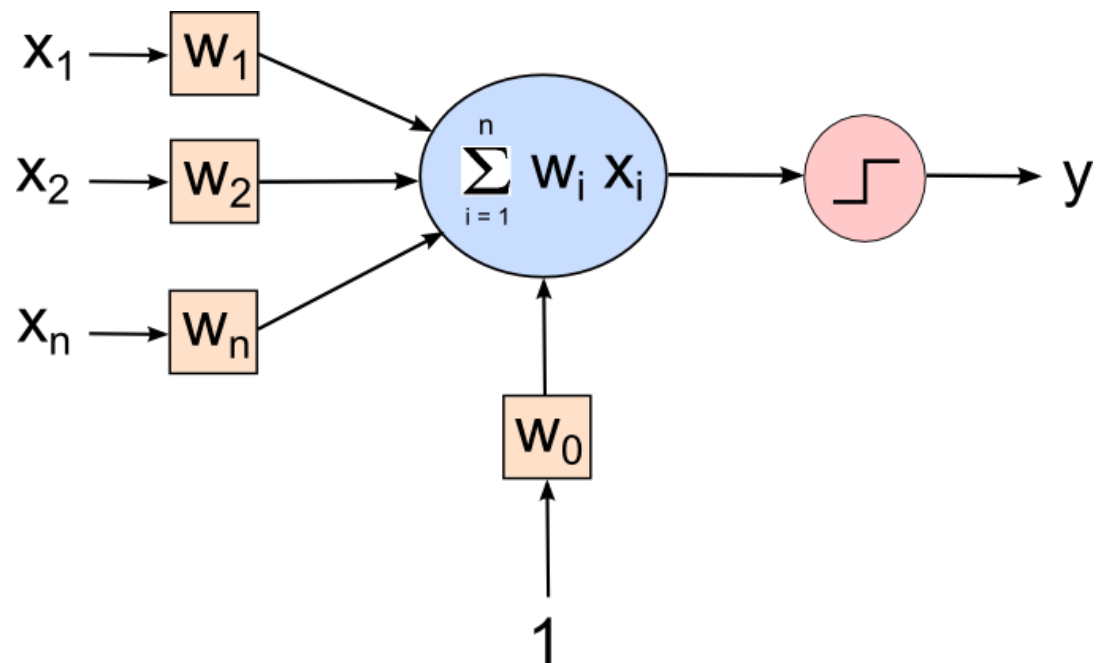
# Adding a bias term



# Adding a bias term




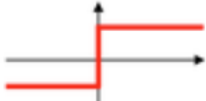
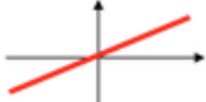
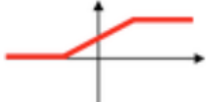
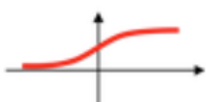

# Adding a bias term





# Activation functions

Activation functions apply a nonlinear mapping to the weighted sum of input variables

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer NN	

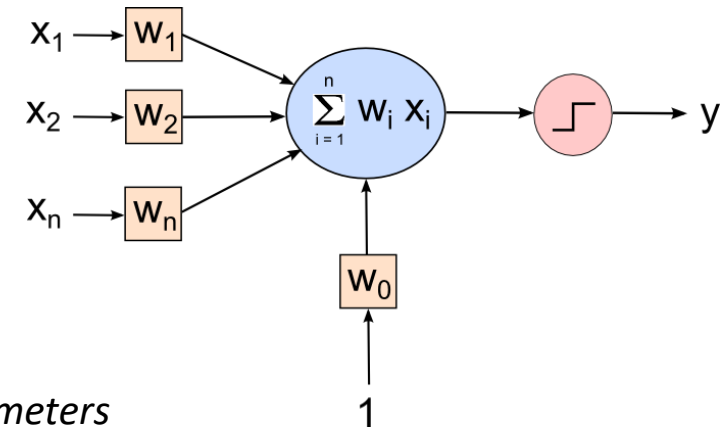
# Finding the weights

To find the optimal weight values for the perceptron on a dataset we do the following:

1. Choose a learning rate:  $0 < \text{learning\_rate} < 1$
2. Initialise the weights  
*e.g. randomly or to zero*
3. Forward propagate the input values  
*i.e. generate a prediction  $y$  for an input sample and current weight parameters*
4. Calculate the error between the prediction and true (labelled) output and adapt the weights:

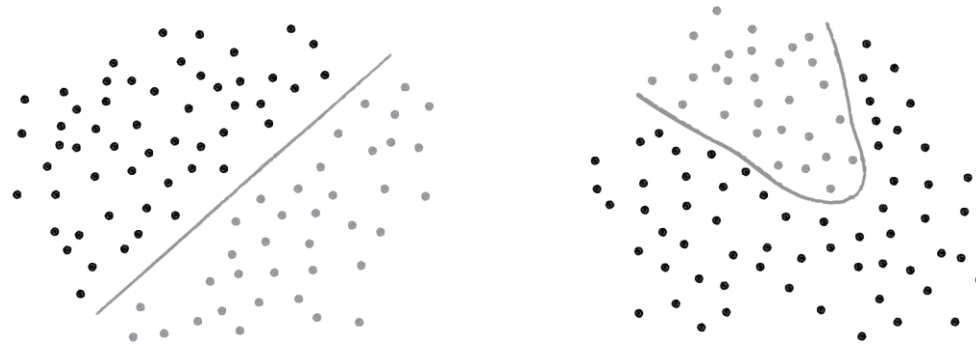
$$w_i(t+1) = w_i(t) + (d_j - y_j(t))x_{j,i}$$

5. Repeat 3. and 4. until convergence



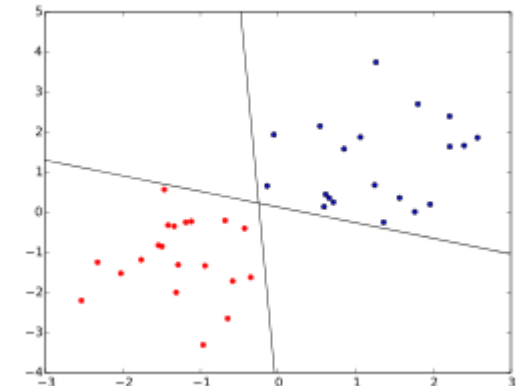
# Limitations of the perceptron

Most problems that we encounter are not linearly separable



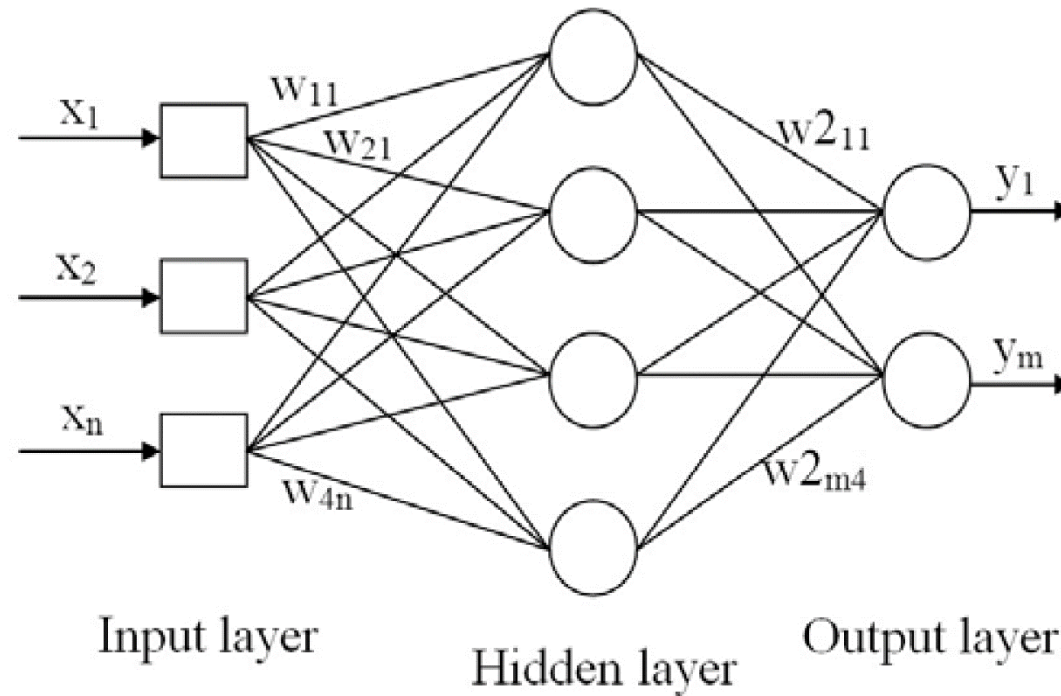
Although guaranteed to converge (if dataset is linearly separable), no guarantees as to *which* solution may be found

- *The linear support vector machine addresses this shortcoming*



# Multilayer Perceptron

To allow for nonlinear decision boundaries we can arrange multiple perceptrons in layers



# Error backpropagation

Only the output neurons in a multilayer perceptron have labelled data that can be used to compare the outputs to.

However, we can propagate the errors in the output neurons back through the network in order to decide the magnitude and direction of the changes to each of the weights.

Provided all operations are *differentiable*, we can apply the *chain rule* to determine each weight's influence on the error at the outputs.

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i} \frac{\partial E}{\partial z_j} = \sum_j w_{ij} \frac{\partial E}{\partial z_j}$$

We can then update each weight value in a similar manner to the single layer perceptron!

# Why are deep networks so powerful?

Think of the multilayer perceptron (and other deep neural networks) as a **composition of functions**.

Provided all of the operations are differentiable, we can append additional neurons and even whole networks on top of each other, similar to Lego bricks.

We can then apply the same forward and back propagation ideas to these models.



# More info

Here are a few sources of info that I found useful:

- Perceptron
  - <https://www.youtube.com/watch?v=1XkjVI-j8MM>
  - <https://www.youtube.com/watch?v=S3iQgcoQVbc>
- Multilayer perceptron and back propagation
  - <https://youtu.be/UJwK6jAStmg>
  - <https://jeremykun.com/2012/12/09/neural-networks-and-backpropagation/>

Time to code!