

# Introduction to Deep Learning

Esther Puyol

Medical Imaging-Deep Learning (MIDL) satellite meeting

11 July 2019

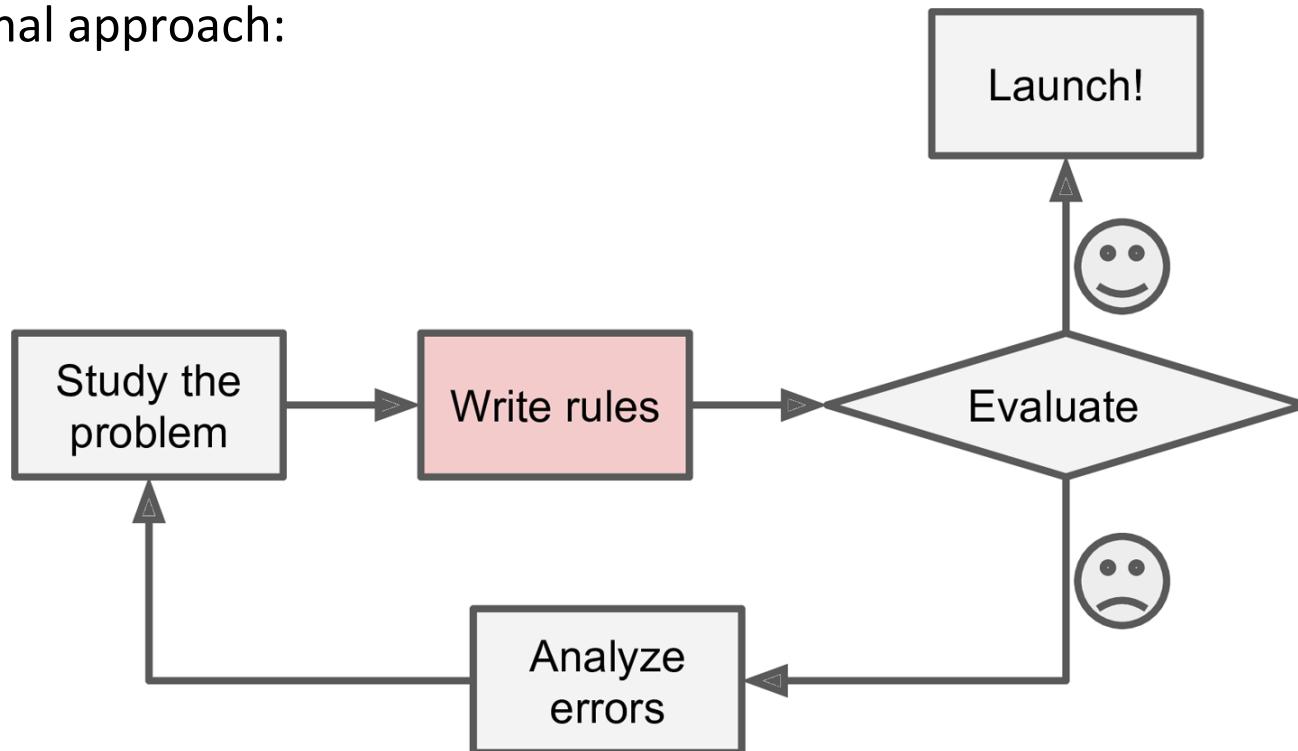
[https://github.com/estherpuyol/MRAI\\_workshop](https://github.com/estherpuyol/MRAI_workshop)



# What is AI?

The science (and art) of programming computers so they can “*learn from and make predictions on data*”

Traditional approach:

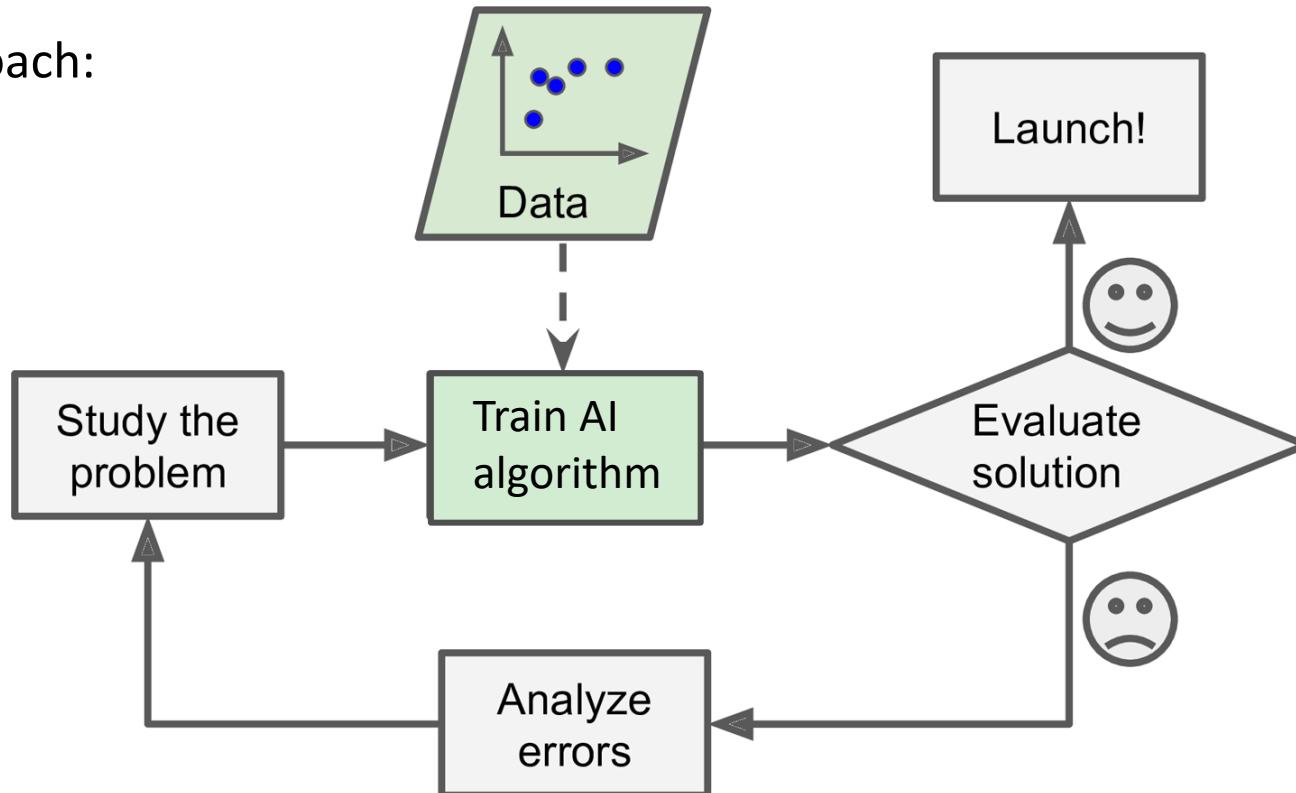




# What is AI?

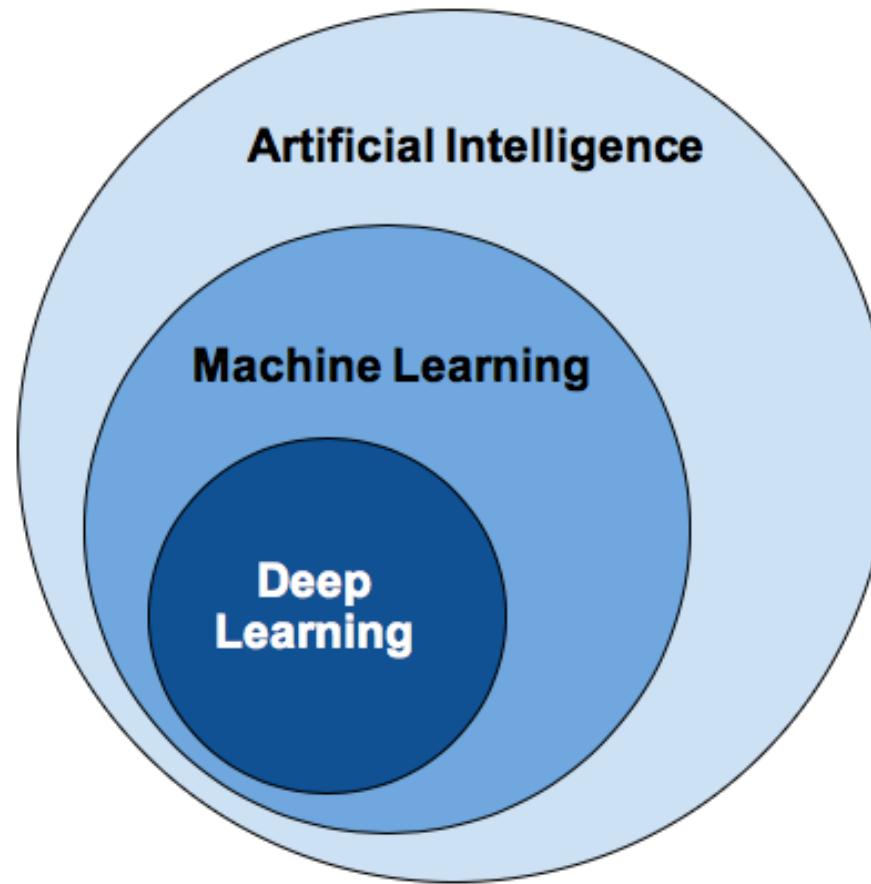
The science (and art) of programming computers so they can “*learn from and make predictions on data*”

AI approach:





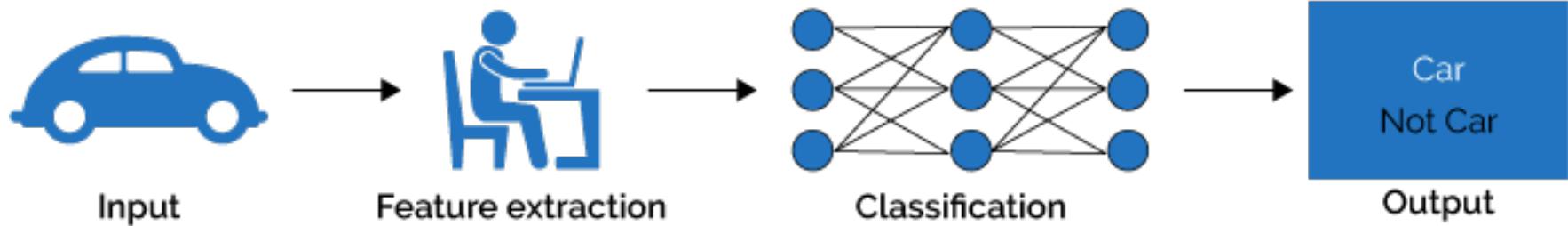
# AI, Machine Learning and Deep Learning:



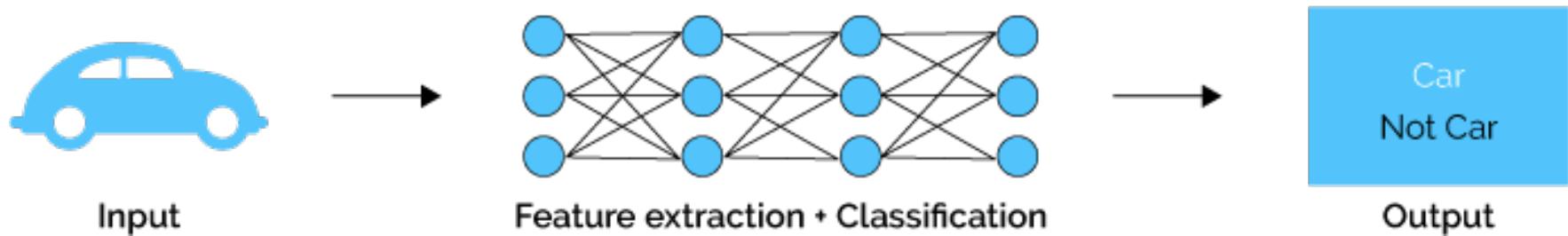


# AI, Machine Learning and Deep Learning:

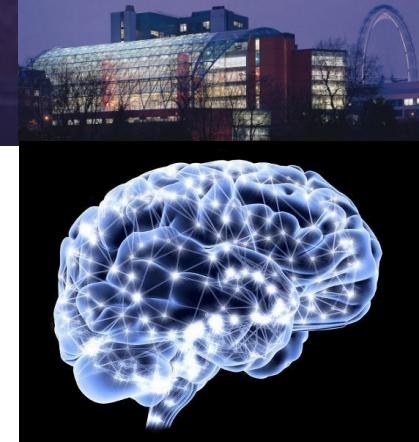
## Machine Learning



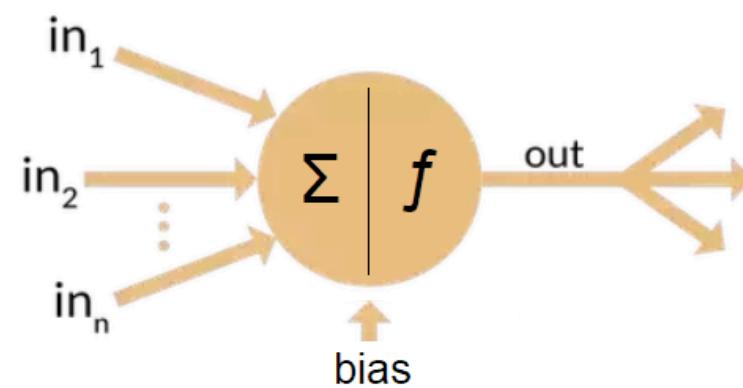
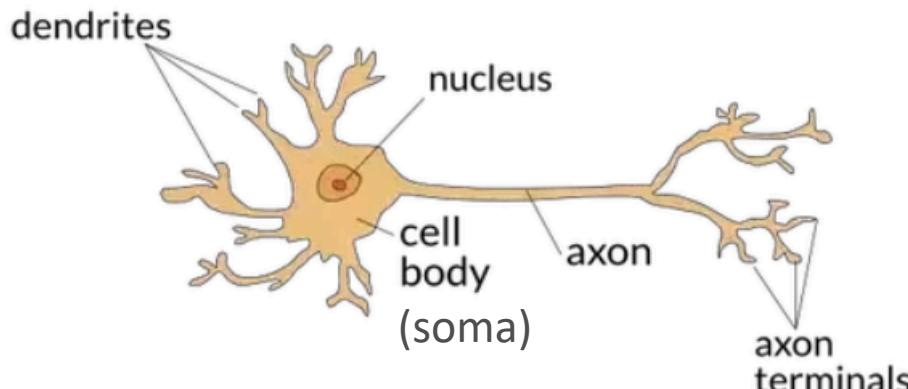
## Deep Learning



# Artificial neural network (ANN)



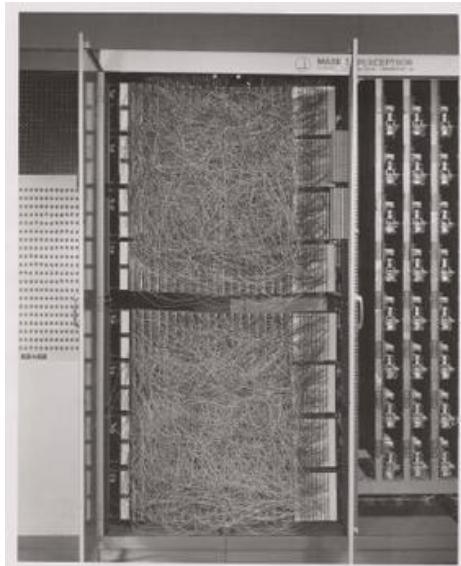
- Machine learning has been around for decades
- First machine learning methods were inspired by how the brain works:





# Perceptron

The first neural network (Frank Rosenblatt, 1957)



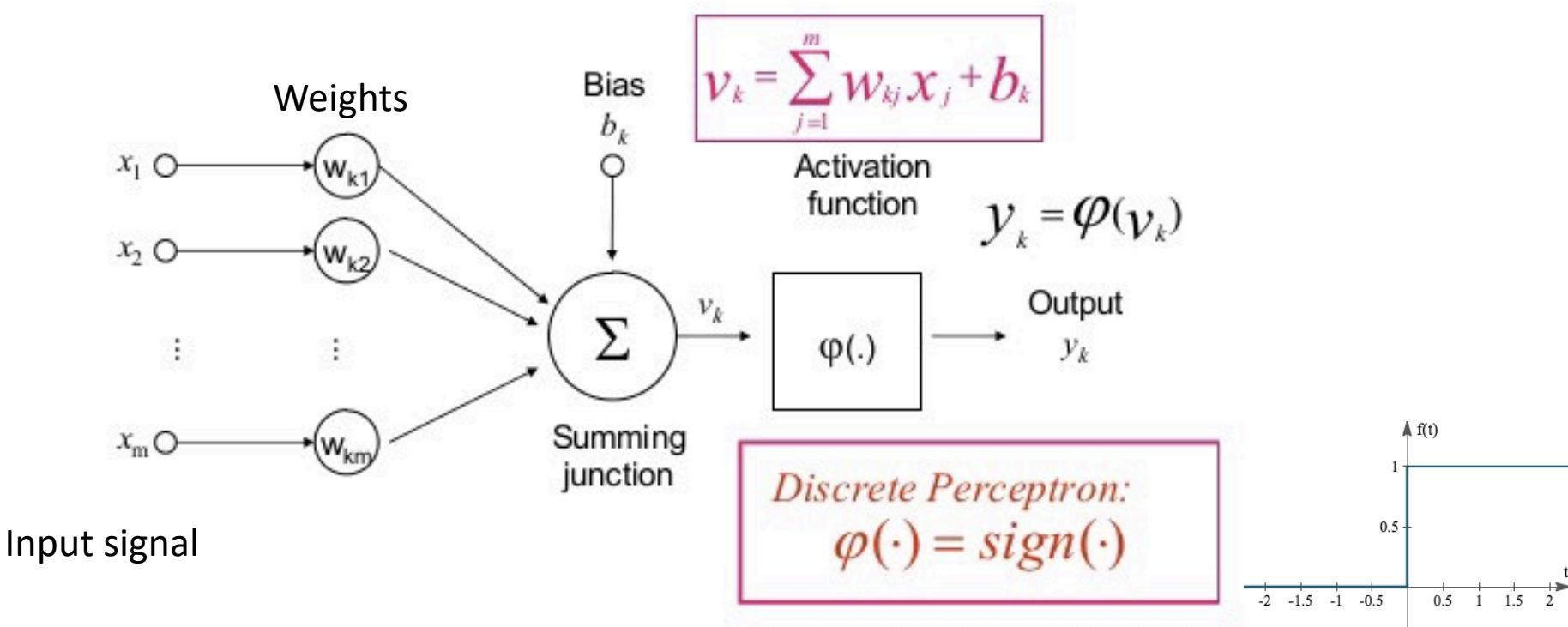
$$f(x) = \begin{cases} 1 & \text{if } \sum_i w_i \cdot x_i > b \\ 0 & \text{otherwise} \end{cases}$$

"Mark 1 perceptron" - machine designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors



# A single-layer perceptron

A single-layer perceptron looks as follows:



\*This and the following slides follow the example on <https://hackernoon.com/a-hands-on-introduction-to-neural-networks-6a03afb468b1>

# A single-layer perceptron

## Input:

- Each input to the neuron ( $x_1, x_2, \dots x_n$ ) is known as a **feature**

## Weights:

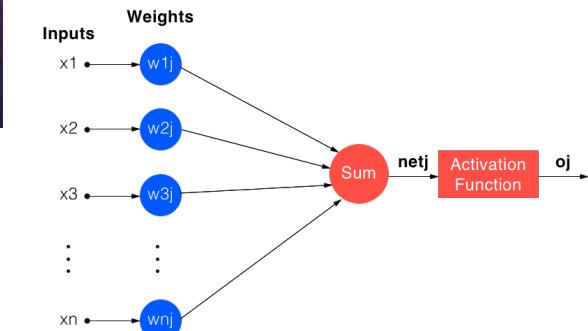
- Each feature is weighted with a number to represent the strength of that input ( $w_{k1}, w_{k2}, \dots w_{km}$ ).

## Bias:

- Additional parameter ( $b_k$ ) which is used to adjust the output along with the weighted sum of the inputs to the neuron.

## Activation function:

- Calculate weighted sum of inputs ( $v_k$ ) , pass through an **activation function** and threshold result  $y_k$  to 0 or 1



# Activation function

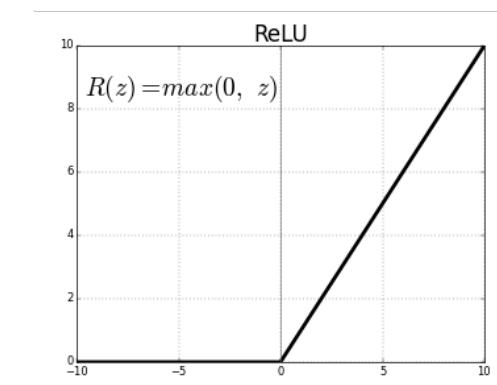
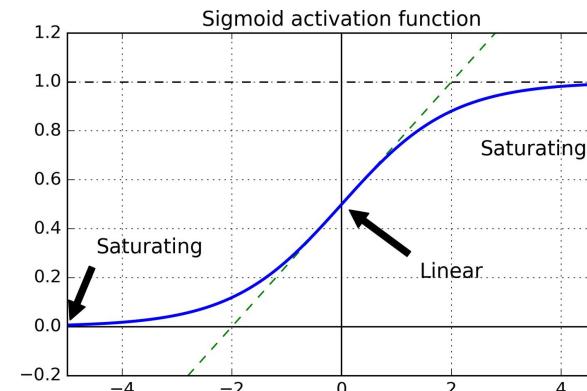
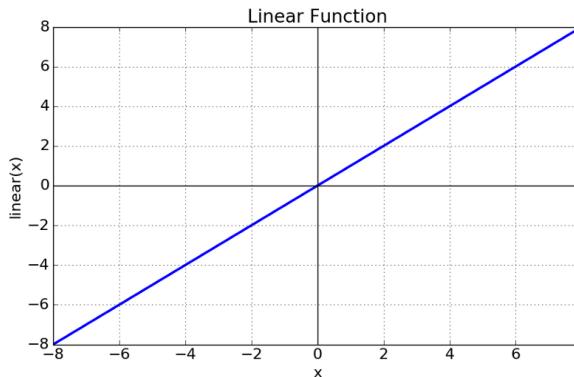
Added to the output end of any neural network

- Can be regarded as a **Transfer Function**

Used to determine whether the output of a neural network is 'yes' or 'no' (or something in between).

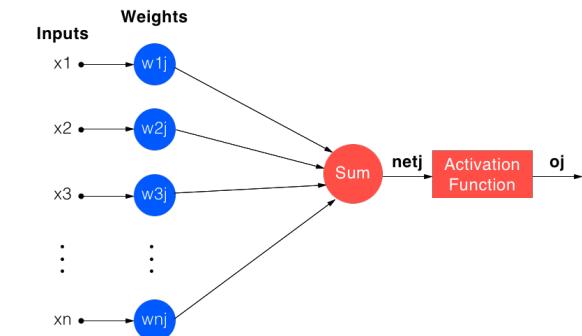
- Maps the resulting values in between 0 to 1 or -1 to 1 (depending upon the activation function)

We distinguish between (piecewise) linear and nonlinear activation functions



# Training a perceptron

- Now that we see how a perceptron works, we need to train it
- Training** a perceptron refers to iteratively updating the weights and bias associated with each of its inputs.
- This allows to progressively approximate the underlying relationship in the given training dataset
- Once properly trained, it can be used to **classify** entirely new samples



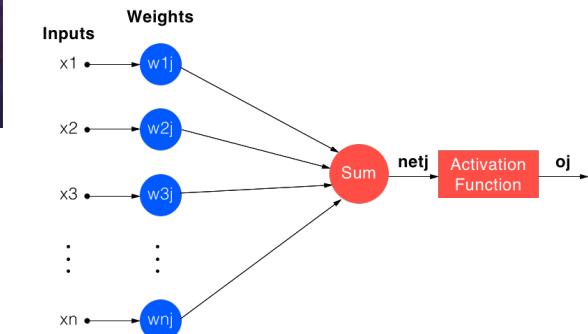
# Training a perceptron

What else do we need?

- We need **an error function (cost function or loss function)**:
  - Measure “how good” a neural network did with respect to its given training sample and the expected output. The cost function must be able to be written as an average over cost functions  $E_i$  for individual training examples  $x_i$ :

$$E = \frac{1}{N} \sum_{i=1}^N E_i$$

- Examples of loss function: L1 norm  $\rightarrow E = \sum_i |E_i|$
- We want to minimize the cost function  $\rightarrow$  need an **optimisation method** (e.g. gradient descent)
- Our activation function should be **differentiable**

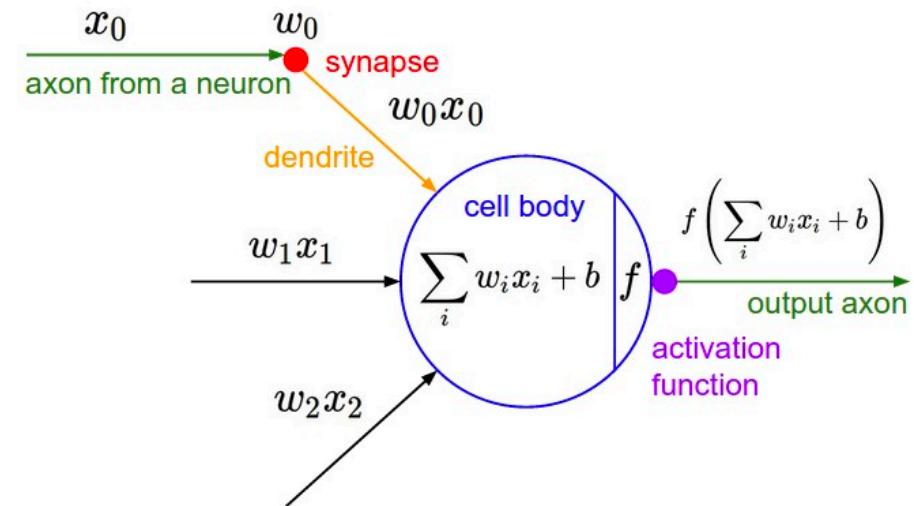




# Training a perceptron – step 1

## 1. Forward Pass

- In the first step of training, we ask the neuron to make a prediction about the training samples.
- This is known as a **forward pass**, and it involves taking a weighted sum of the input features and passing that sum through the activation function.
- Mathematically:
  - $f(\sum_{i=1}^N w_{ij}x_i + b)$





# Training a perceptron – step 2

## 2. Reverse pass:

- The second step to update the neuron's weights using the **Delta-rule** from the gradient descent algorithm:

$$\Delta w_{ij} = -n \frac{\partial E}{\partial w_{ij}}$$

- ie each weight will be updated in the negative direction of the gradient, proportional to an additional term,  $n^*$ .
- This scaling factor, n, determines how large a step we take when updating neuron weights, effectively controlling the rate at which the neuron learns. We call  $n$  the **learning rate**.

\*Often called  $\eta$



# Training a perceptron – step 3

## 3. Repeat for a set number of iterations (=epochs)

- We need to iterate through this several times:
- **Either** until convergence (overall loss is smaller than  $\epsilon$ )
- **Or** until maximum number of epochs



# Training a perceptron - Overview

A single-layer perceptron can be trained as follows:

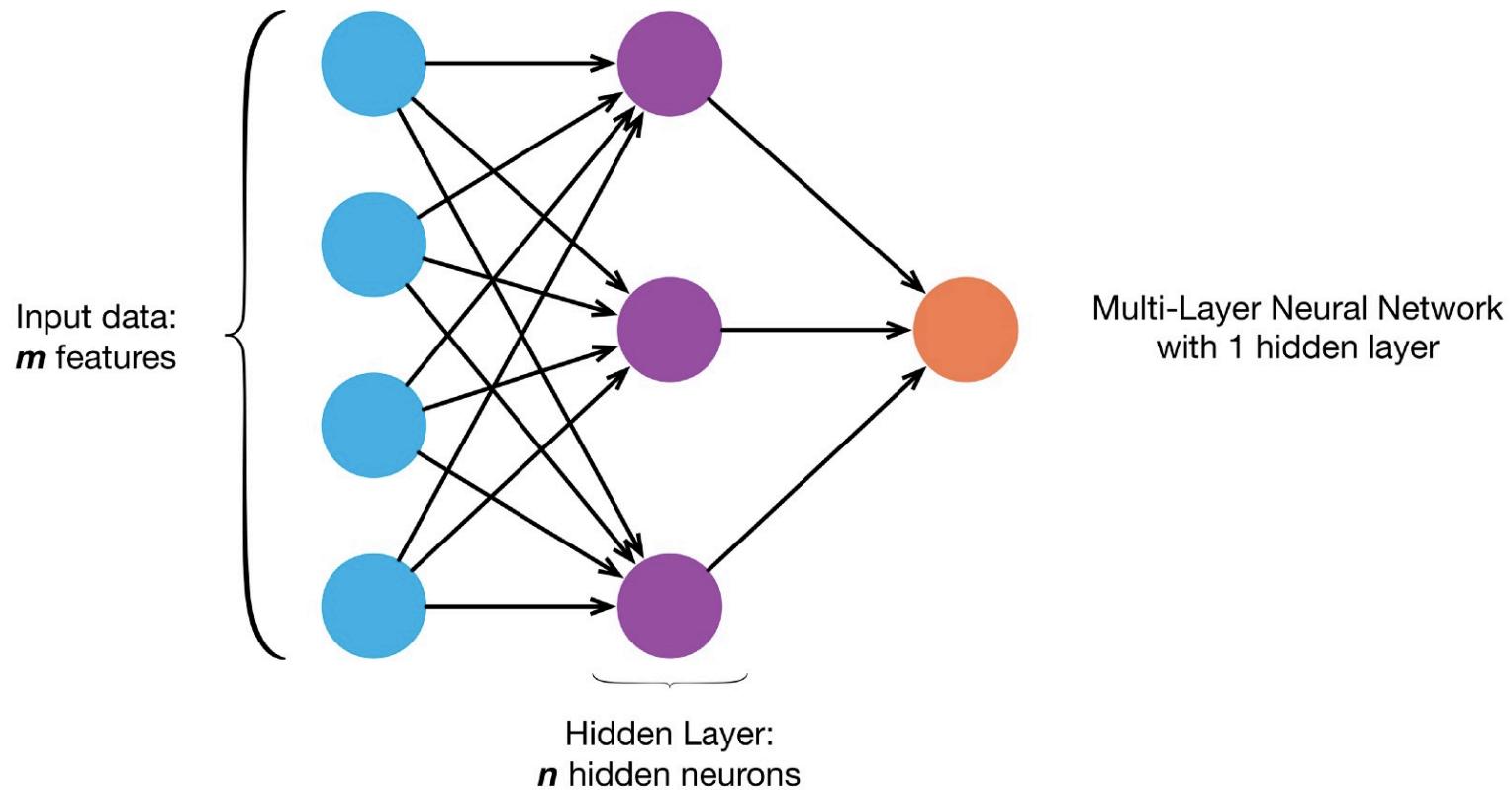
1. Ask the neuron\* to classify a sample (**forward pass**)
2. Update the neuron's weights based on how wrong the prediction is.
3. Repeat for a set number of times (=**epochs**).

\*A perceptron is a neuron with a binary output



# From single-layer to multi-layer perceptron

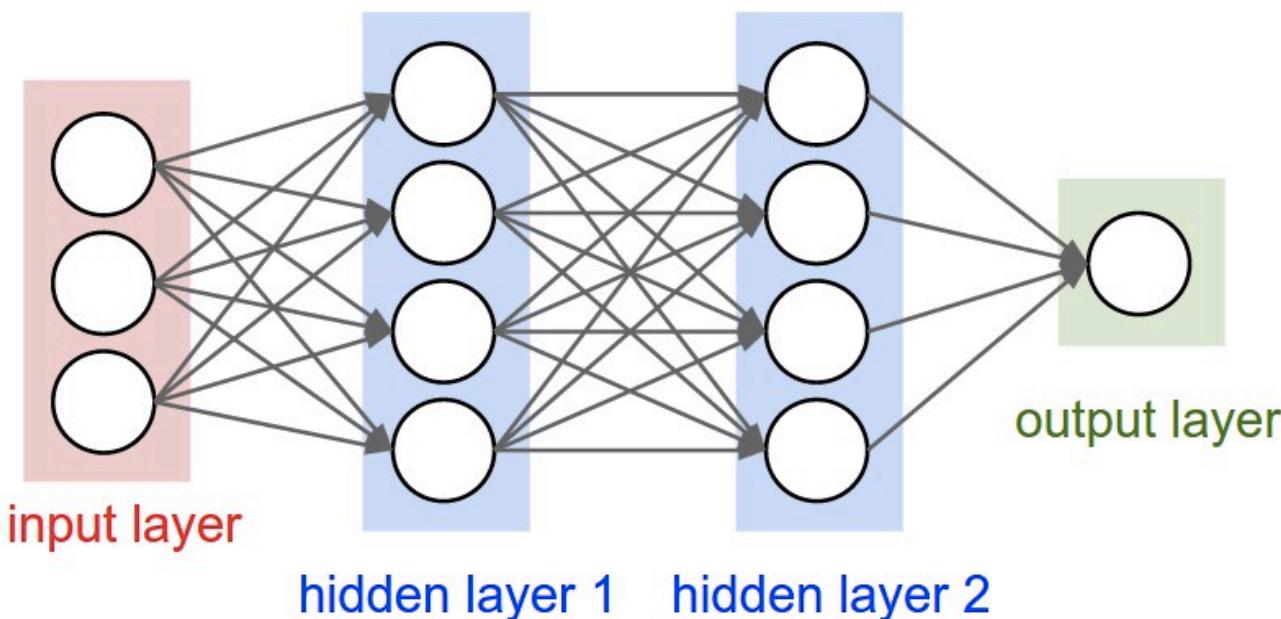
A more complex MLP is shown below (still only 1 hidden layer):





# Multi-layer perceptron

We have stacked multiple perceptrons to generate hidden layers:

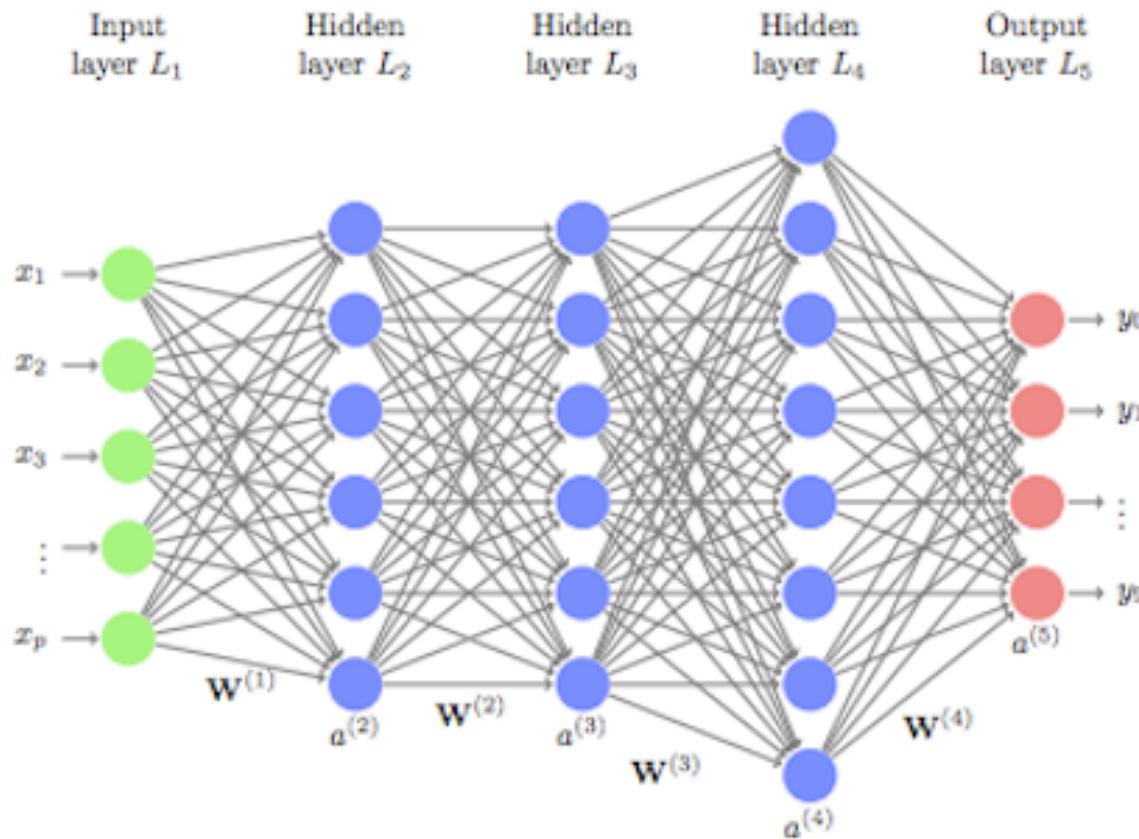


If we have more than one hidden layer, the neural network is considered to be “deep” and we move into **deep learning**



# Deep fully connected networks (FCN)

Compare this to a deep fully-connected network with  $N$  hidden layers:





# Training a FCN: Backpropagation

Backpropagation (BP) is a common method for training a neural network, and it is a generalization of the delta rule to multi-layered feedforward network, by **using the chain rule to iteratively compute gradients for each layer**.

For each training instance, the BP algorithm:

1. First make a prediction (**forward pass**) and measure the error
2. Then go through each layer in reverse to measure the error contribution from each connection (**reverse pass**)
3. And finally slightly tweak the connection weights to reduce the error (**Gradient Descent step**)
4. BP is a generalization of the delta rule to multi-layered feedforward network, by **using the chain rule to iteratively compute gradients for each layer**.



# From FCN to Convolutional neural network

## Disadvantages of FCN:

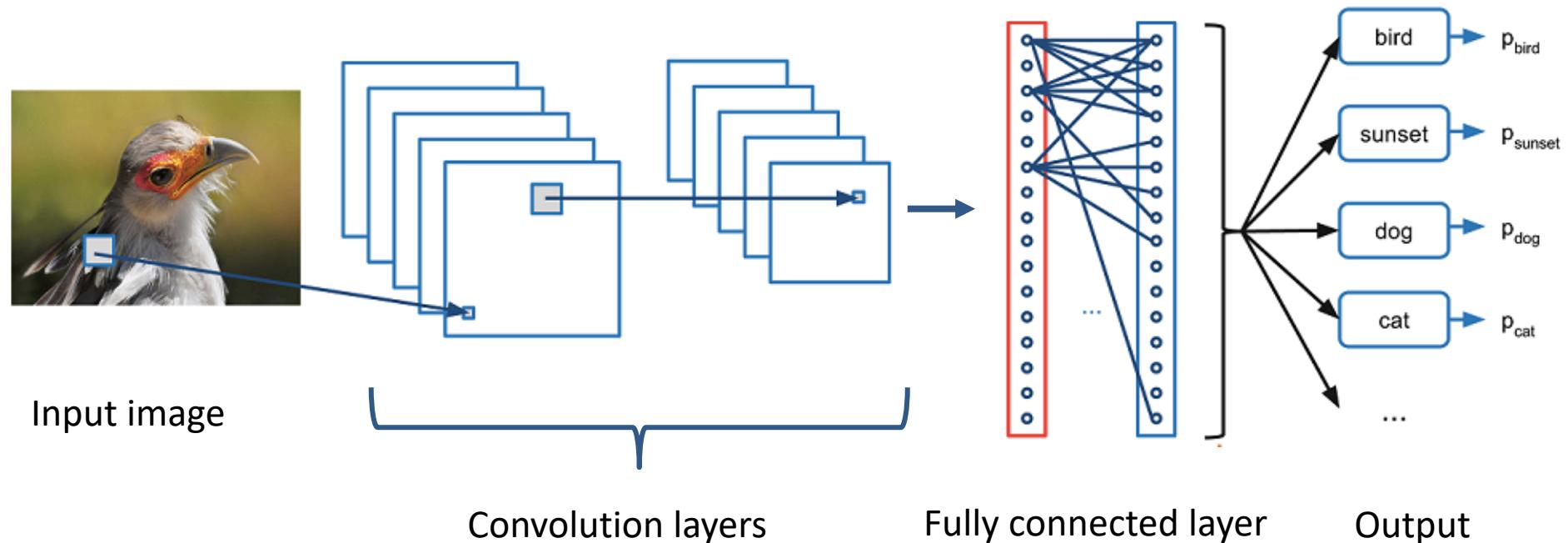
- Large number of parameters to be learned
- Incredibly computationally expensive.
- Slow
- Prone to overfitting data

## Convolutional neural network (CNN):

- Regularized versions of multilayer perceptrons.
- Fully connected layers are replaced by one or more convolutional layers
- Capture spatial and temporal dependencies



# Convolutional Neural Networks (CNNs)





# Convolutional layer

$$s[t] = (x \star w)[t] = \sum_{a=-\infty}^{a=\infty} x[a]w[a+t]$$

Diagram illustrating the convolution operation:

- Feature map (Input)
- Kernel

The convolution layer is the main building block of a convolutional neural network. Stack of convolution layers allow to concentrate from low-level features (first layers) to complex higher-level features (features).

1 <small>x1</small>	1 <small>x0</small>	1 <small>x1</small>	0	0
0 <small>x0</small>	1 <small>x1</small>	1 <small>x0</small>	1	0
0 <small>x1</small>	0 <small>x0</small>	1 <small>x1</small>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved  
feature map

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Weights learned  
by the network

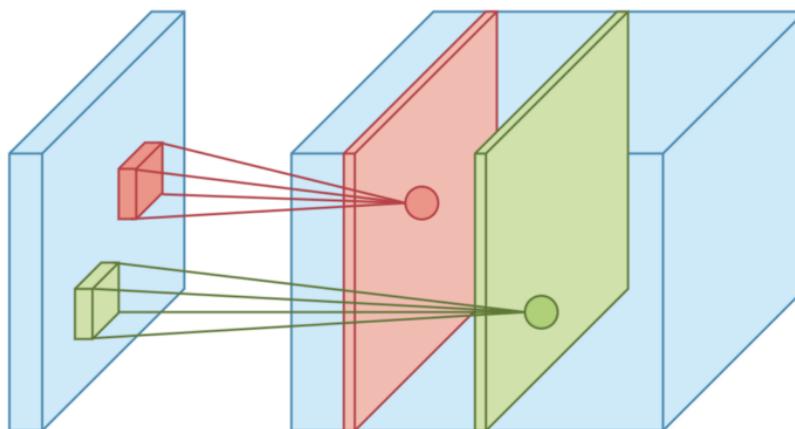
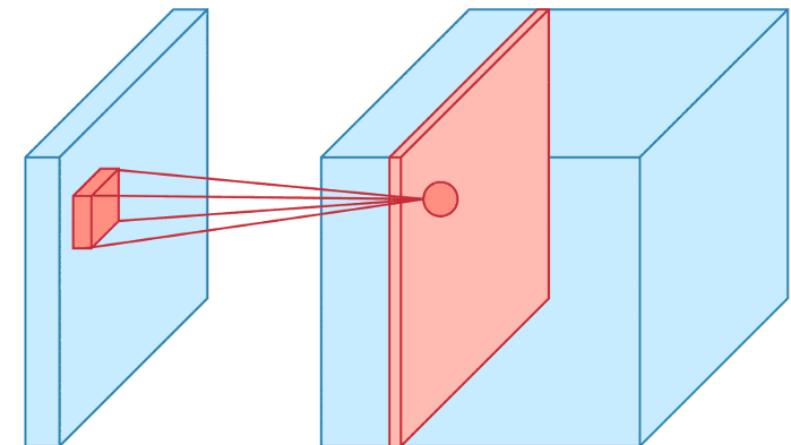
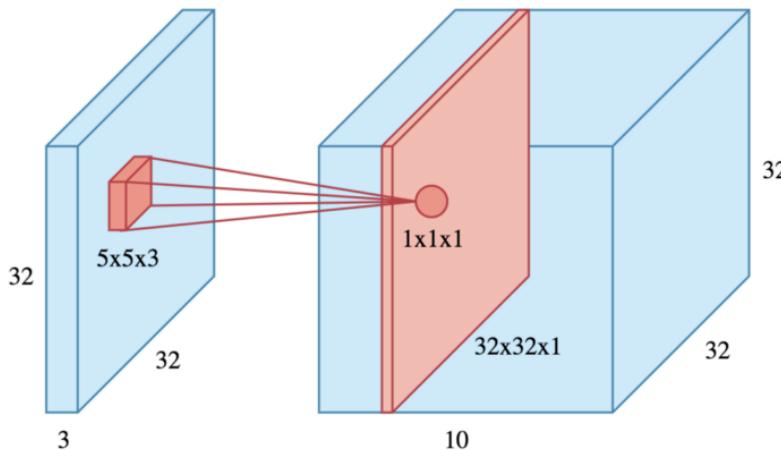
Kernel  
3x3



Receptive field



# Convolutional layer

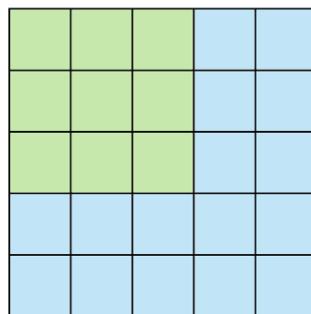


In each convolution layer, we compute different feature maps . Each of them learn different features from the input image.

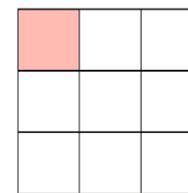


# Convolutional layer: Stride

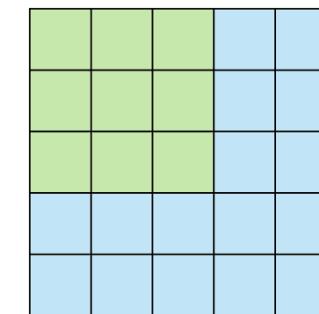
- *Stride* specifies how much we move the convolution filter at each step.



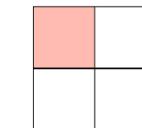
Stride 1



Feature Map

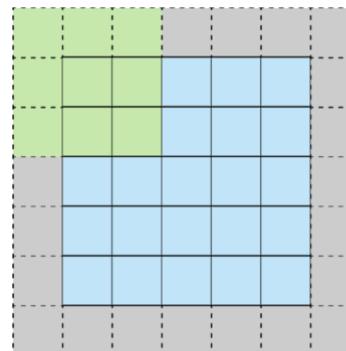


Stride 2

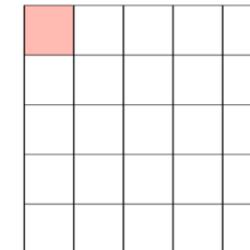


Feature Map

- To maintain the same dimensionality between the input and the output, we can use *zero-padding* to surround the input with zeros.



Stride 1 with Padding

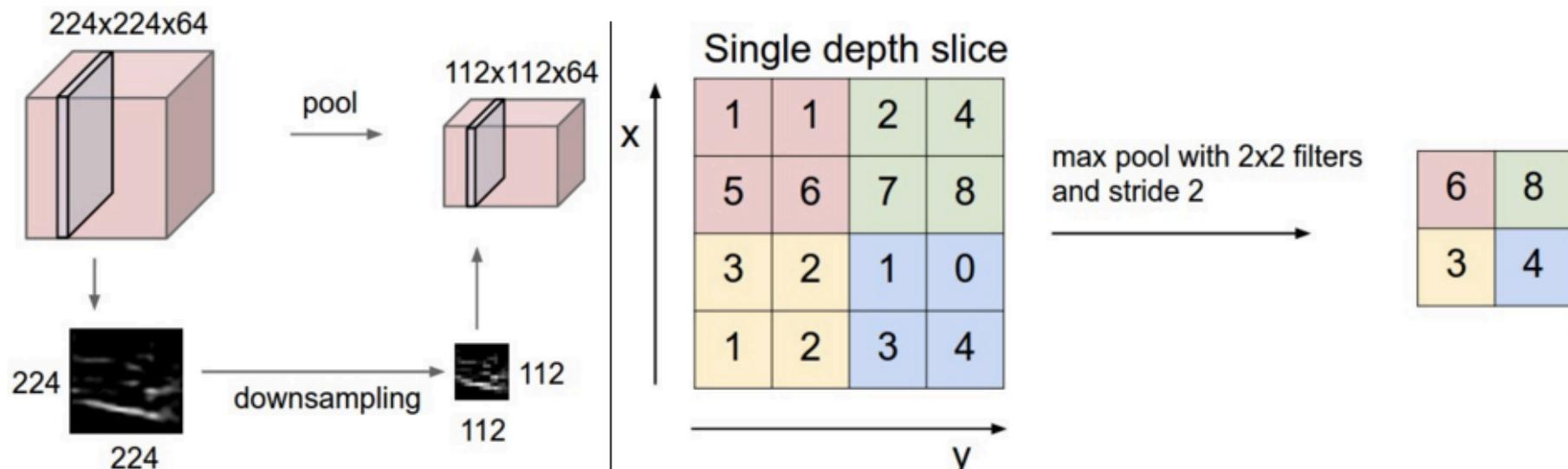


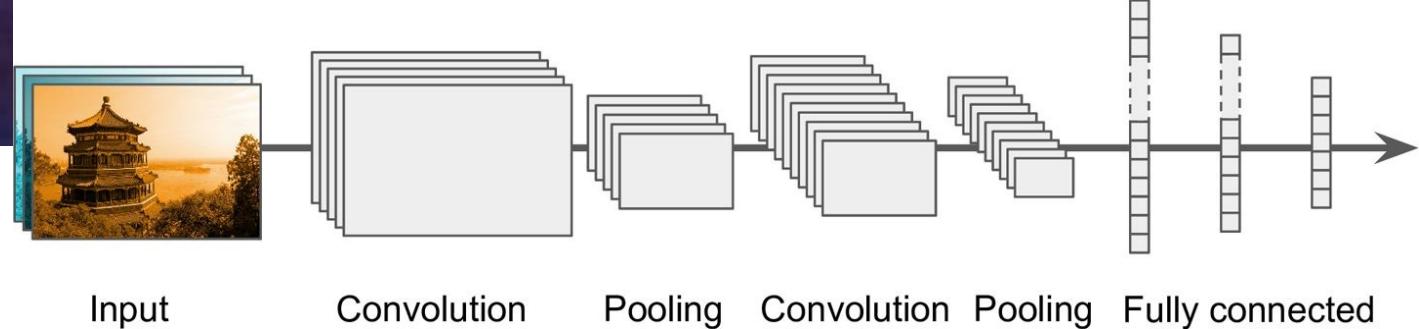
Feature Map



# Pooling layer

- Spatial Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map but retains the most important information.
- Spatial pooling helps with translational invariant features.
- Most common type is **max pooling**





Main **building blocks** of a CNN:

**1. 2D (or 3D) input layer**

**2. Convolutional layer**

- Neurons in each convolutional layer are not connected to every single pixel, **but only to pixels in their receptive fields**

**3. Pooling layer or stride**

- Goal is to **subsample** the input image to reduce computational load, memory usage, numbers of parameters (limits risk of overfitting)

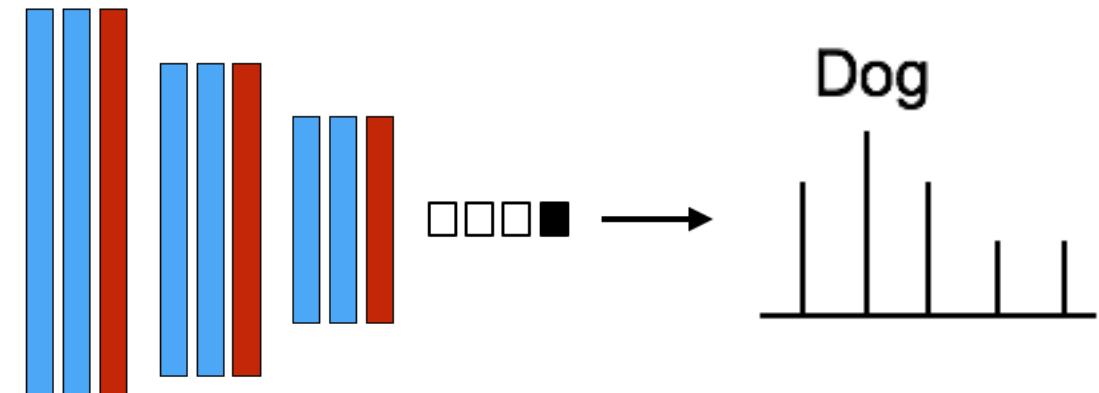
**4. Fully-connected output layer**

- This is a **regular feed-forward network** which produces final output prediction
- E.g. **softmax layer** that outputs estimated class probabilities



# CNNs for image classification

First rewind and look at a simple CNN applied to real image classification

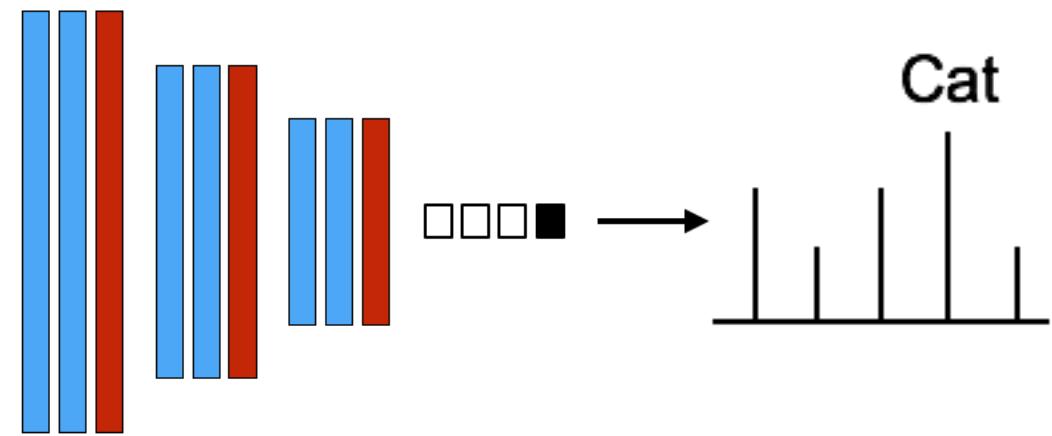


■ Convolution + RELU    □ Fully connected layer  
■ Max pooling              ■ Softmax



# CNNs for image classification

First rewind and look at a simple CNN applied to real image classification



■ Convolution + RELU   □ Fully connected layer

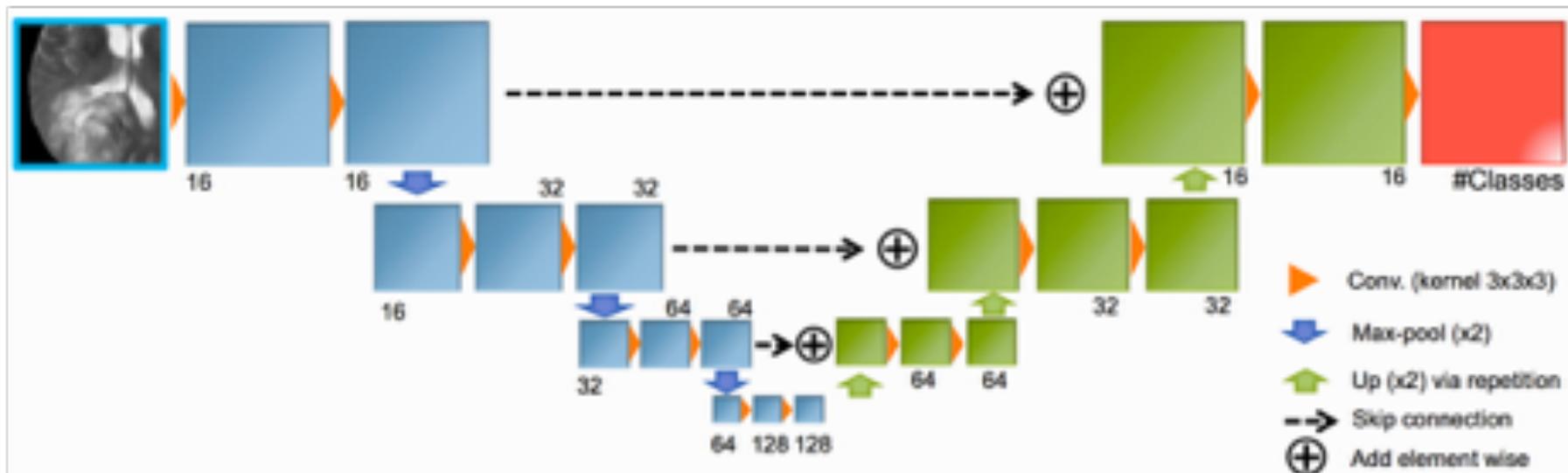
■ Max pooling   ■ Softmax



# CNNs for image segmentation

One major focus in computer vision and medical imaging is on **image segmentation**

- Challenging in medical imaging due to variability in patient anatomy & pathology, patient pose and motion, image artefacts
- U-net is the most common used network , which was proposed by Ronneberger (Google DeepMind) in 2015:





# Bits & Bobs

All of the networks so far (deep neural networks, with or without convolutional layers), need to be carefully designed and trained:

Choice of:

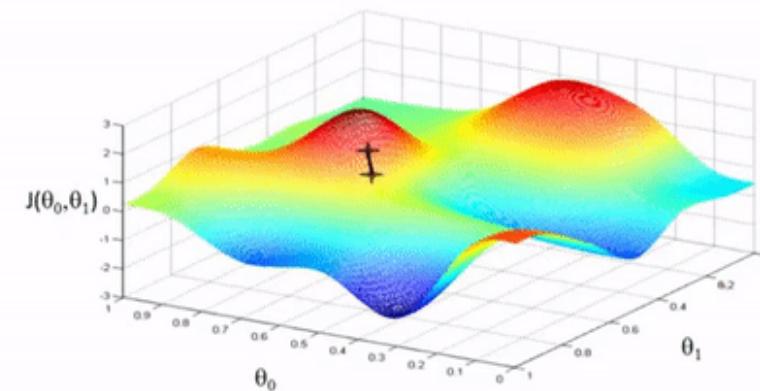
- Number of hidden layers and neurons, stacking (AE)
- Loss function, activation function, learning rate, epochs
- **Optimisation methods**
- **Regularisation methods**



# Optimisation

Choice of faster gradient-based optimisation methods for use with backpropagation:

1. Gradient Descent
2. Momentum optimisation
3. Nesterov Accelerated Gradient
4. AdaGrad
5. RMSProp
6. Adam Optimiser



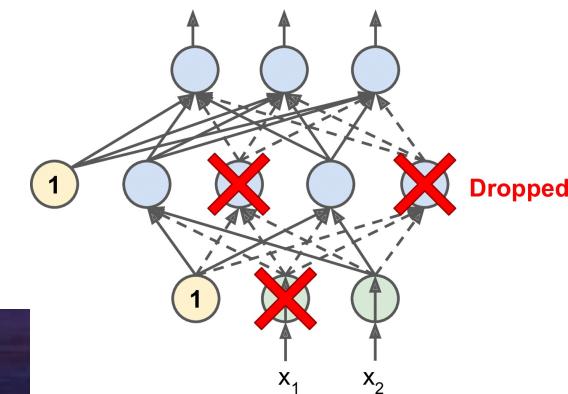
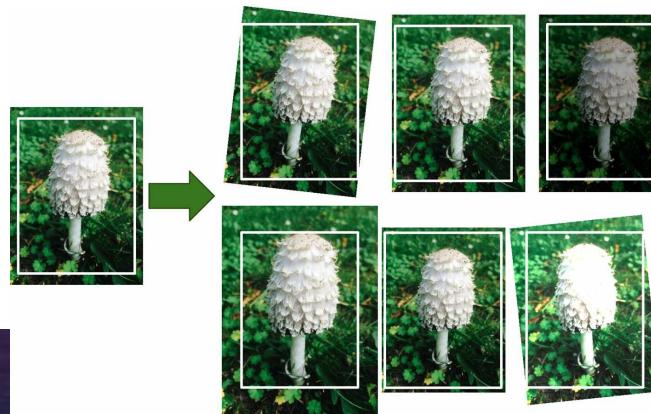


# Regularisation

To avoid **overfitting**, we can do the following:

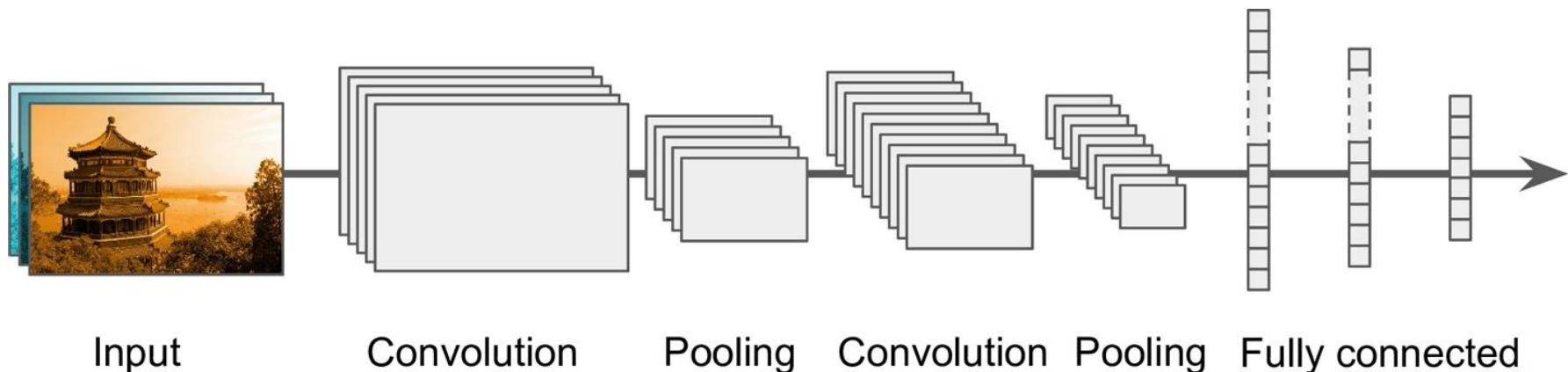
1. **Early stopping:** Interrupt training when its performance on the validation set starts dropping
2.  **$L_1$  and  $L_2$  regularization:** Add a regularization term in the cost function.
3. **Dropout:** At every training step, every neuron (input or hidden) has a probability  $p$  of being temporarily “dropped out”
4. **Data augmentation:** Generate new training instances from existing ones, artificially boosting the size of the training set.

E.g. you can rotate, shift (translate), resize (scale), flip (reflect)





# Summary



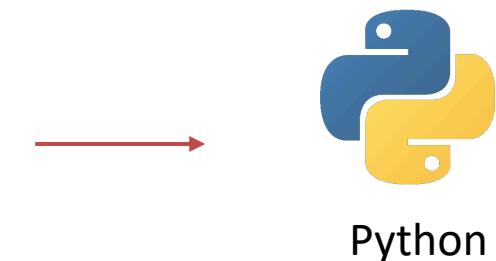
## Hyperparameters:

- Number of layers
- Number of epochs
- Kernel size
- Stride
- Learning rate
- Dropout
- Optimisation method



# Practical point of view

- Programming languages



Python

- Python libraries



Interaction with GPU



# Questions?

Slides adapted from the Machine Learning for  
Biomedical Application course from King's College of  
London