COMP1804

Applied Machine Learning

Lecture 6: Artificial Neural Networks

Dr. Dimitrios Kollias

The Human Brain & Neurons

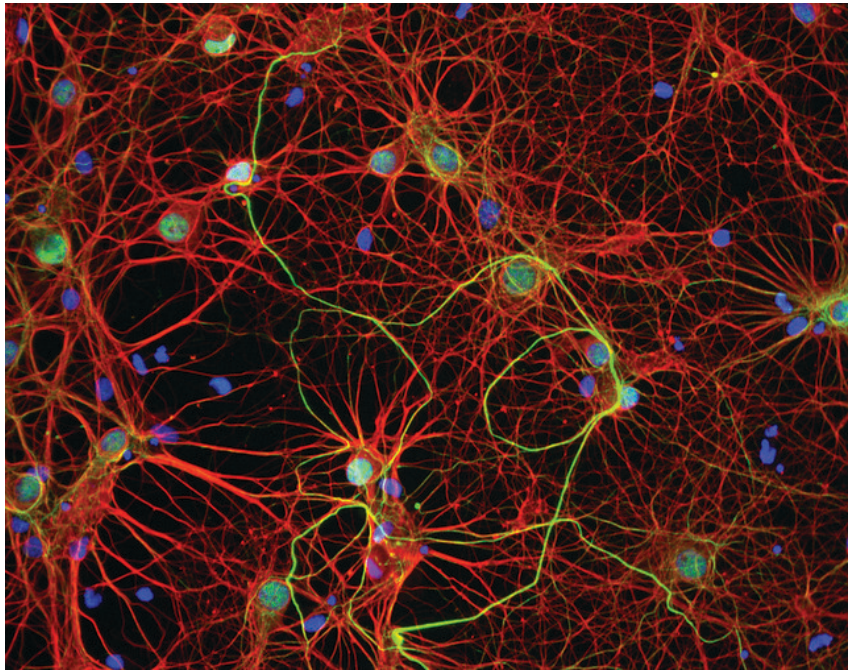
The Neuron

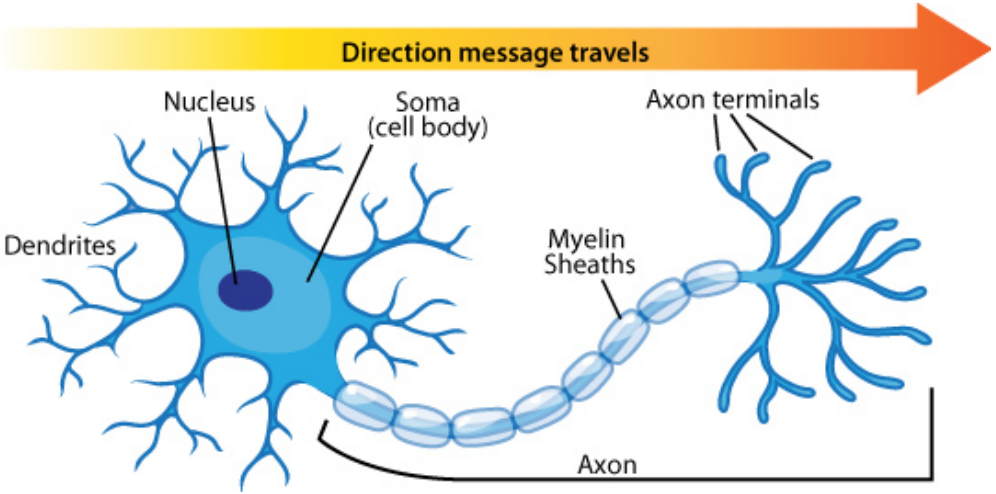
The human nervous system is composed of more than 100 billion cells known as neurons. A neuron is a cell in the nervous system whose function it is to receive and transmit information.

Neurons are made up of three major parts:

- the cell body, or **soma**, which contains the nucleus of the cell and keeps the cell alive
- a branching treelike fiber known as the **dendrite**, which collects information from other cells and sends the information to the soma
- a long, segmented fiber known as the **axon**, which transmits information away from the cell body toward other neurons or to the muscles and glands

Biology of Neuron





Neural Function

- Brain function (thought) occurs as the result of the firing of neurons
- Neurons connect to each other through synapses, which propagate action potential (electrical impulses) by releasing neurotransmiters
  - Synapses can be excitatory (potential-increasing) or inhibitory (potential-decreasing), and have varying activation thresholds
  - Learning occurs as a result of the synapses' plasticity: They exhibit long-term changes in connection strength
- There are about 10^{11} neurons and about 10^{14} synapses in the human brain!

Brain Structure

- Different areas of the brain have different functions
  - Some areas seem to have the same function in all humans (e.g., Broca's region for motor speech); the overall layout is generally consistent
  - Some areas are more plastic, and vary in their function; also, the lower-level structure and function varies greatly

- We don't know how different functions are "assigned" or acquired
  - Partly the result of the physical layout/connection to inputs (sensors) & outputs (effectors)
  - Partly the result of experience (learning)

- We really don't understand how this neural structure leads to what we perceive as "consciousness" or "thought"

Neural Networks

Neural Networks

- Artificial Neural Networks are modeled after biological neural networks and attempt to allow computers to learn in a similar manner to humans.
  In other words, they are algorithms that try to mimic the human brain.

- They have been very widely used in 80s and early 90s;
  their popularity diminished in late 90s

- Recent resurgence: State-of-the-art technique for many applications

Neural Networks:

Comparison of Computing Power

- Computers are way faster than neurons
- There are a lot more neurons than we can reasonably model in modern digital computers, and they all fire in parallel
- Neural networks are designed to be massively parallel

INFORMATION CIRCA 2012	Computer	Human Brain
Computation Units	10-core Xeon: 10^9 Gates	10^{11} Neurons
Storage Units	10^9 bits RAM, 10^{12} bits disk	10^{11} neurons, 10^{14} synapses
Cycle time	10^{-9} sec	10^{-3} sec
Bandwidth	10^9 bits/sec	10^{14} bits/sec

Neural Networks (NN): Why Develop NNs

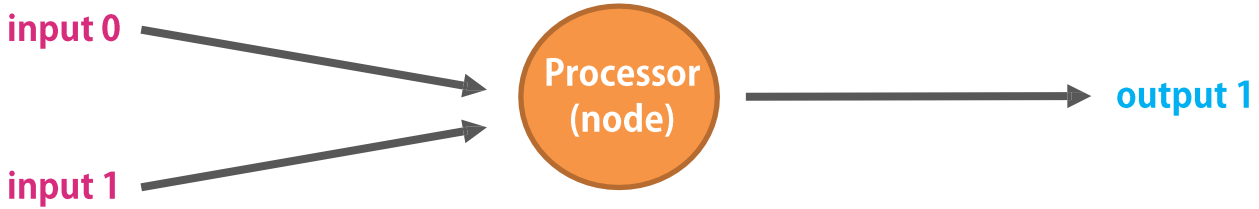
- There are problems that are difficult for humans but easy for computers (e.g. calculating large arithmetic problems).
- Then there are problems easy for humans, but difficult for computers (e.g. recognizing a picture of a person from the side).
- Neural Networks attempt to solve problems that would normally be easy for humans but hard for computers!
- Let's start by looking at the simplest Neural network possible - the perceptron.

The Perceptron

Perceptron

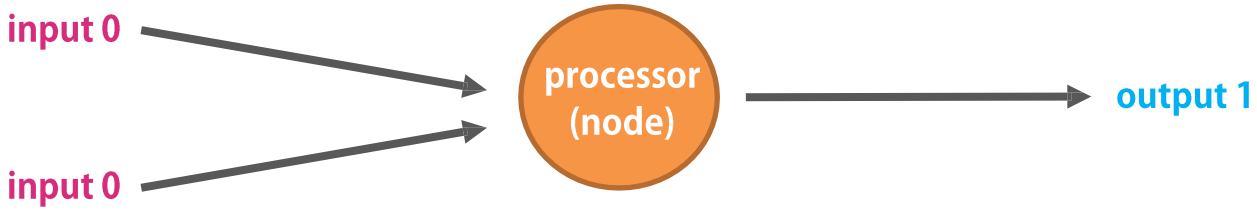
- A perceptron consists of:
  - i) one or more inputs
  - ii) a processor/node
  - iii) a single output

- A perceptron follows the "feed-forward" model
  - meaning inputs are sent into the neuron, are processed, and result in an output



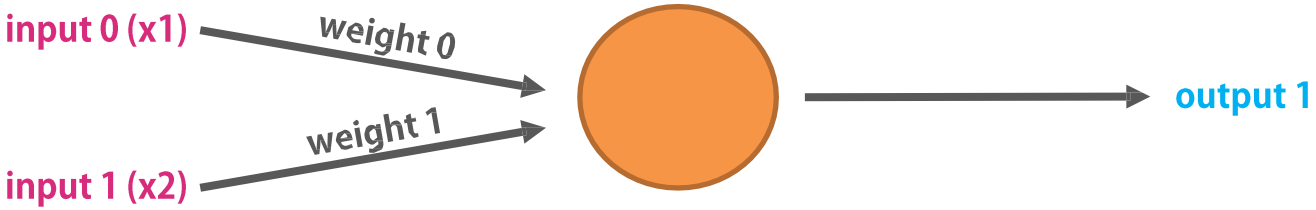
Perceptron: Process

- A perceptron process follows 4 main steps:
  1. Receive Inputs
  2. Weight Inputs
  3. Sum Inputs
  4. Generate Output



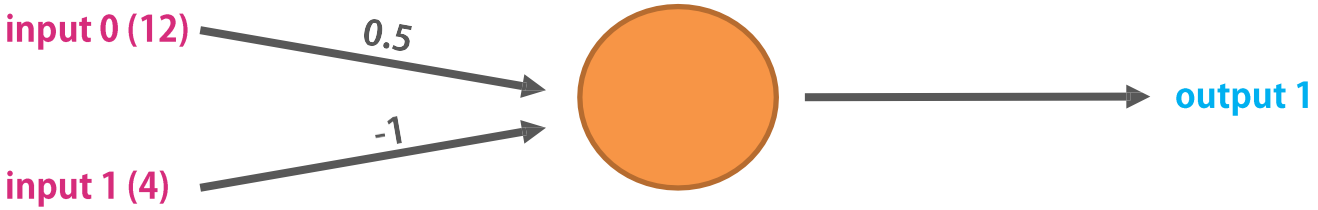
Perceptron: Example (I)

- Say we have a perceptron with two inputs:
  - **Input 0: x1 = 12**
  - **Input 1: x2 = 4**
- Each input that is sent into the neuron must first be weighted,
  - i.e. multiplied by some value



Perceptron: Example (II)

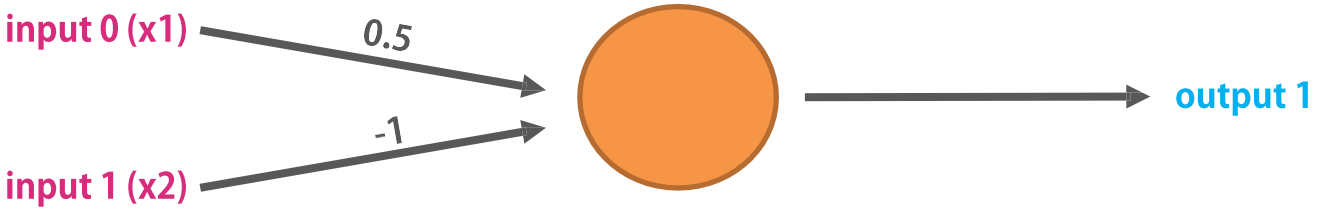
- When creating a perceptron, we'll typically begin by assigning random weights.
  - Weight 0: 0.5
  - Weight 1: -1
- We take each input and multiply it by its weight.
  - Input 0 *Weight 0 ⇒ 12 *0.5 = 6
  - Input 1 *Weight 1 ⇒ 4 *-1 = -4



Perceptron: Example (III)

- The output of a perceptron is generated by passing that sum through an activation function.
  - In the case of a simple binary output, the activation function is what tells the perceptron whether to "fire" or not.

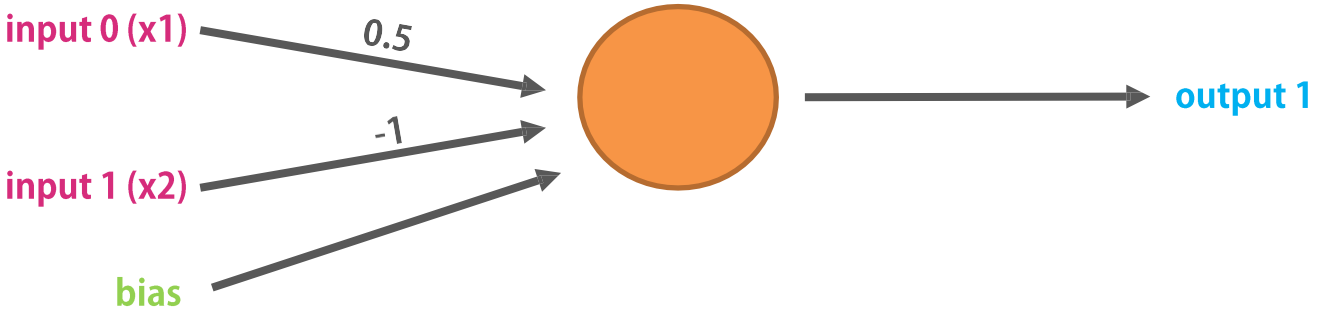
- Many activation functions to choose from.
  - Let's make the activation function the sign of the sum. In other words, if the sum is a positive number, the output is 1; if it is negative, the output is -1.



Perceptron: Example (IV)

- One more thing to consider is Bias.
  - Imagine that both inputs were equal to zero,
  - then any sum no matter what multiplicative weight would also be zero!

- To avoid this problem, we add a third input known as a bias input with a value of e.g. 1.
  - This avoids the zero issue!



Perceptron: Training

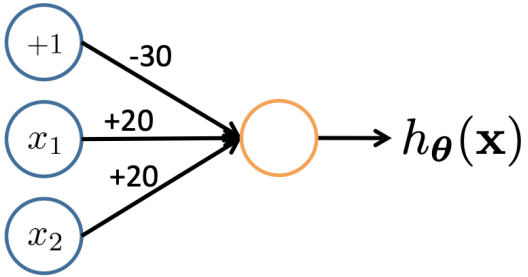
- To actually train the perceptron we use the following steps:
  1. Provide the perceptron with inputs for which there is a known answer.
  2. Ask the perceptron to guess an answer.
  3. Compute the error. (How far off from the correct answer?)
  4. Adjust all the weights according to the error.
  5. Return to Step 1 and repeat!
- We repeat this until we reach an error we are satisfied with (we set this before hand).

Perceptron:
Can it implement the 'AND' Logical Function?

Simple example: AND

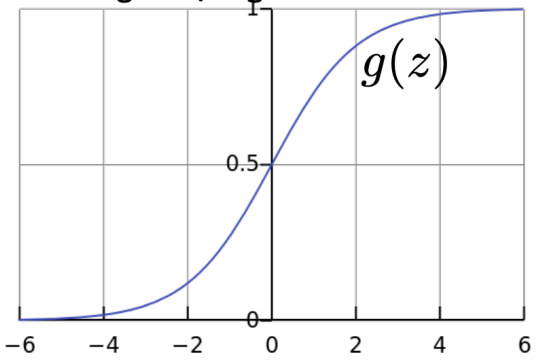
$x_1, x_2 \in \{0, 1\}$

$y = x_1 \text{ AND } x_2$



$$h_\Theta(\mathbf{x}) = g(-30 + 20x_1 + 20x_2)$$

Logistic / Sigmoid Function



x_1	x_2	$h_\Theta(\mathbf{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Perceptron:

Can it implement other Logical Functions?

AND

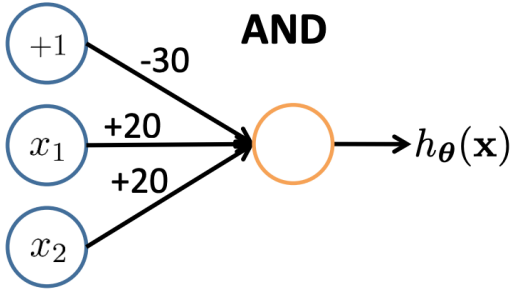
x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

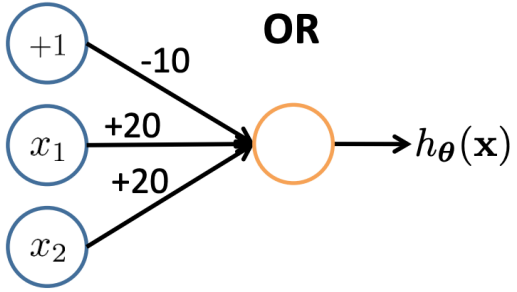
OR

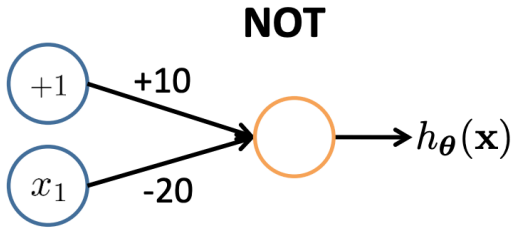
x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

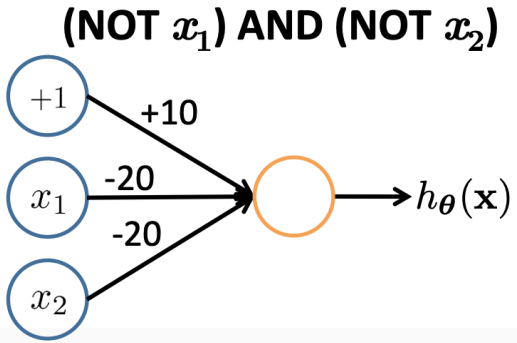
NOT

x	x '
0	1
1	0





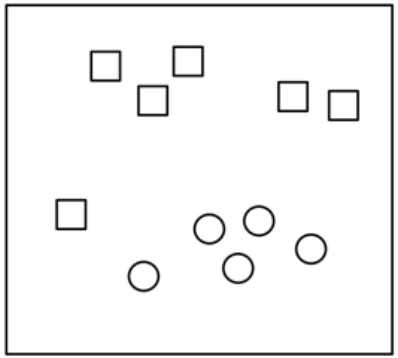




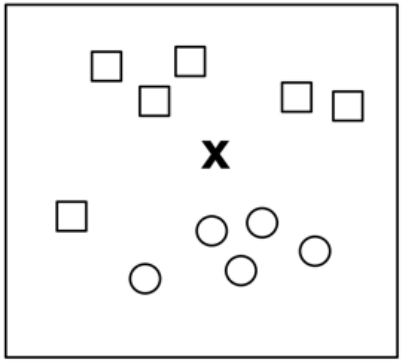
Perceptron:
Seeing it differently (I)

Example:

We have a set of points in 2D Space and each point is labelled as “square” or “circle”:

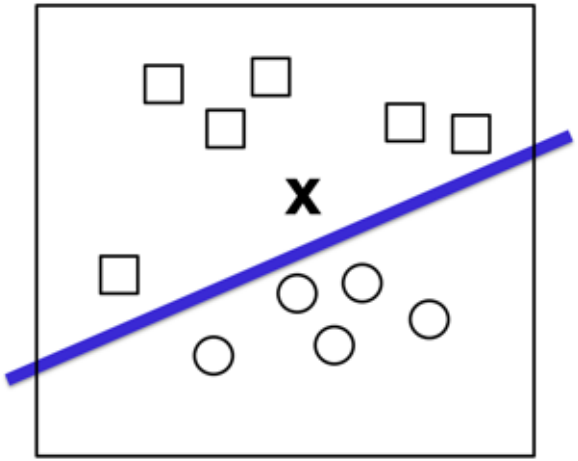


Given a new point “*X*“, we want to know what label corresponds to it:



Perceptron:
Seeing it differently (II)

A common approach is to draw a line that separates the two groups and use this line as a classifier:



Perceptron:
Seeing it differently (III)

To formally state it:
- the input data will be represented by vectors of the form (*x1, x2*) that indicate their coordinates in this 2D Space
- our function will return ‘0’ or ‘1’ (above or below the line) to know if it should be classified as “square” or “circle”.

It is a case of linear regression, where “the line” (the classifier) can be defined by:

$$y = w_1 x_1 + w_2 x_2 + b$$

More generally, we can express the line as:

$$y = W * X + b$$

Perceptron:

Seeing it differently (IV)

To classify input elements X, which in our case are two-dimensional, we must learn:

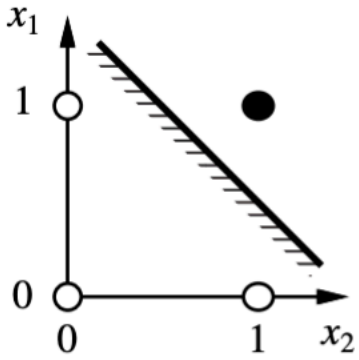
- a vector of weight W of the same dimension as the input vectors, i.e., the vector (*w1, w2*)
- a *b* bias

With these calculated values, we can now construct a perceptron to classify a new element *X*. Basically, it applies this vector *W* of calculated weights on the values in each dimension of the input element *X*, and at the end adds the bias *b.*

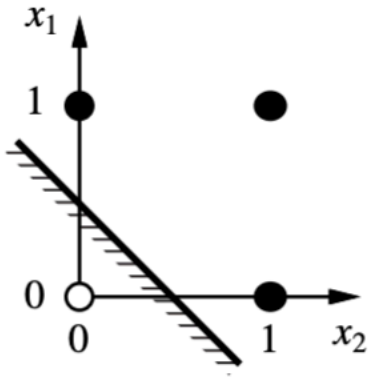
The result of this will be passed through a non-linear “activation” function to produce a result of ‘0’ or ‘1’.

Perceptron:

Seeing it differently in ‘AND’ – ‘OR’



(a) x_1 **and** x_2

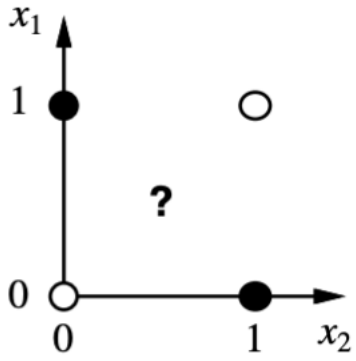


(b) x_1 **or** x_2

Perceptron:
Problem with 'XOR'

XOR

x	*y*	$x \oplus y$
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*0*



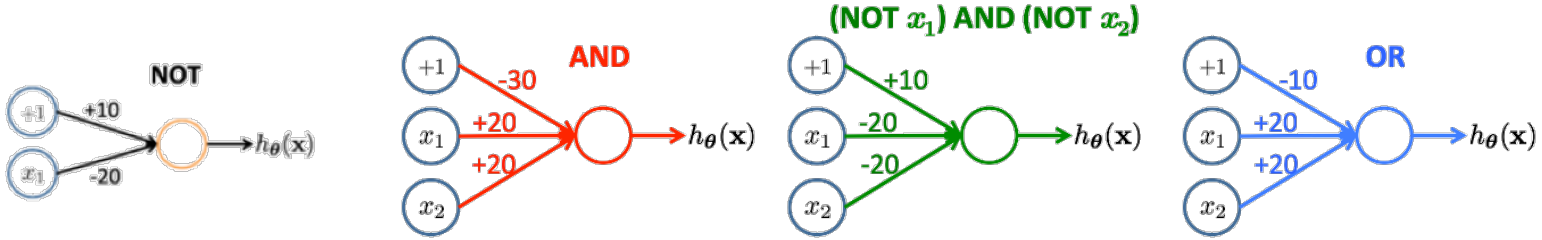
(c) x_1 **xor** x_2

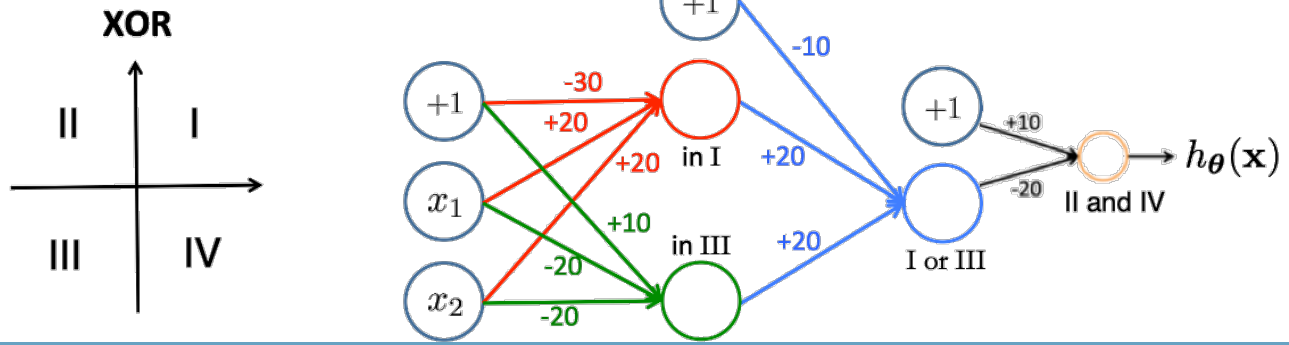
Geometrically, the perceptron can separate its input space with a line → it can only separate linearly separable problems.

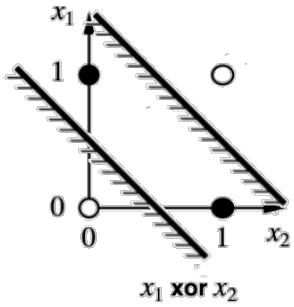
XOR function is not linearly separable, thus it is impossible for a single line to separate it.

Perceptron:
Problem with 'XOR' : Solution

Combine Representations to create Non-Linear Functions:







Multilayer Perceptron
&
Deep Neural Networks

Multilayer Perceptron (MLP)

A multilayer perceptron (MLP) is a group of perceptrons, organized in multiple layers, that can accurately answer complex questions.

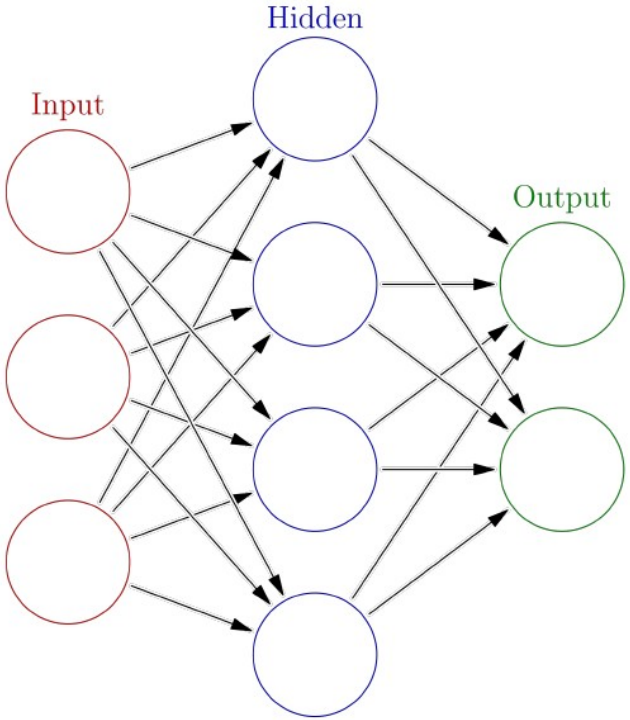
Each perceptron in the first layer sends signals to all the perceptrons in the second layer, and so on.

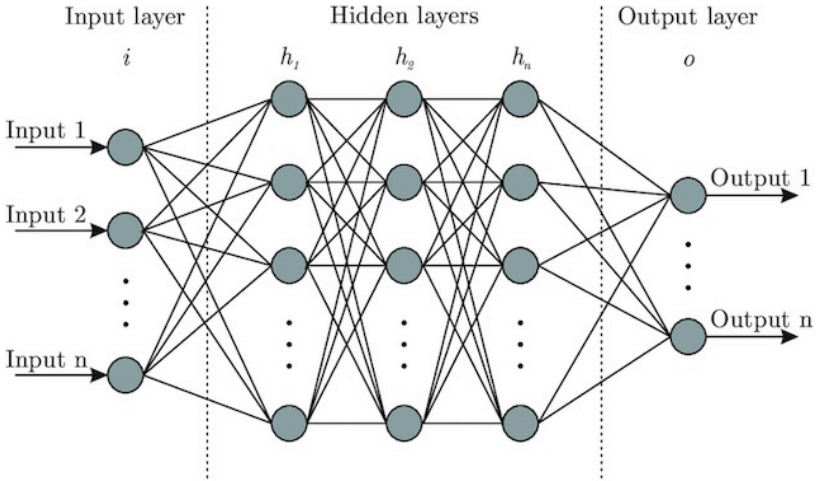
An MLP contains an input layer, at least one hidden layer, and an output layer.
Any layers in between the input and the output ones are known as hidden layers,

- because you don't directly "see" anything but the input or output.

Multilayer Perceptron (MLP):

Examples





Artificial Neural Networks (ANN)

Artificial Neural Networks (ANN) is a supervised learning system built of a large number of simple elements, called neurons or perceptrons.

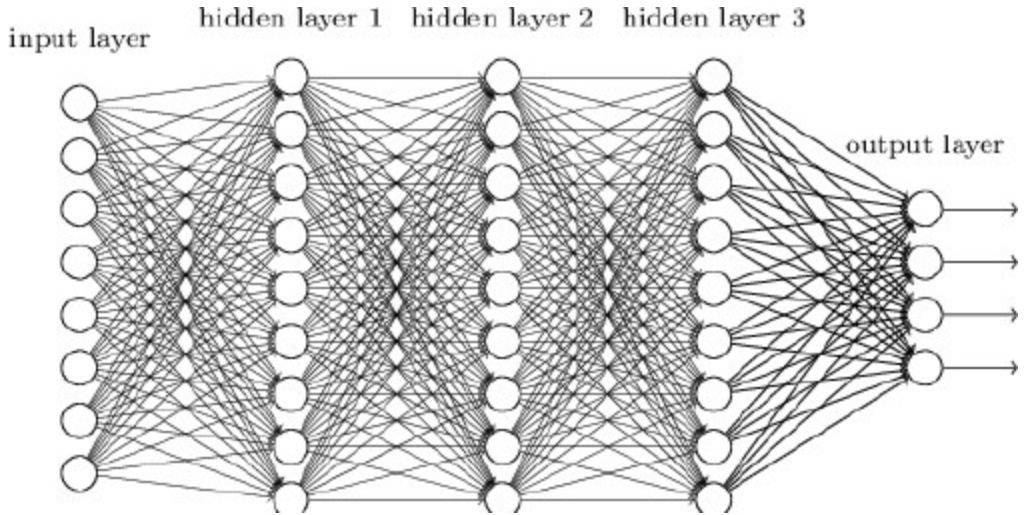
Each neuron can make simple decisions, and feeds those decisions to other neurons, organized in interconnected layers.

Together, the neural network can *emulate almost any function*, and *answer practically any question*, given enough training samples and computing power.

Deep Neural Networks (DNN)

That’s just a Neural Network with many hidden layers, causing it to be "deep"

For example, one of Microsoft’s state of the art vision recognition DNN uses 152 layers



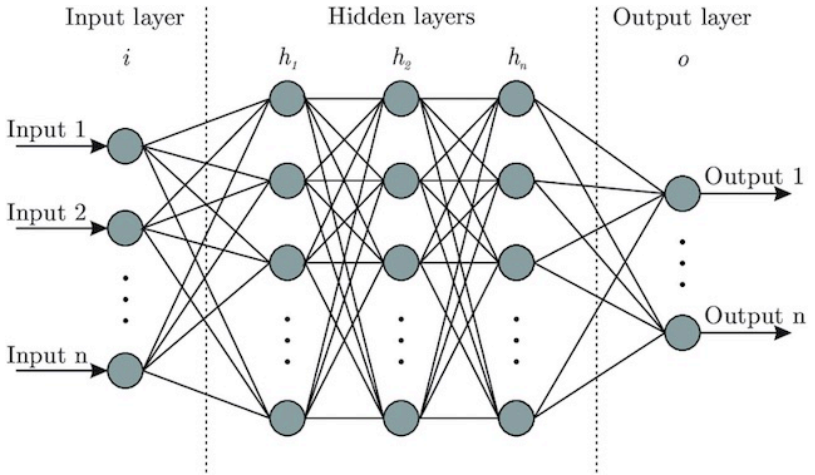
Neural Networks:

Putting it all together (I)

- Neural Networks:

A computer (algorithm) learns to perform some task by analysing, usually hand-labelled, training examples.

e.g. object recognition system: fed thousands of labelled images of cars, houses, coffee cups, etc; it would find visual patterns in the images that consistently correlate with particular labels



Neural Network In more Detail

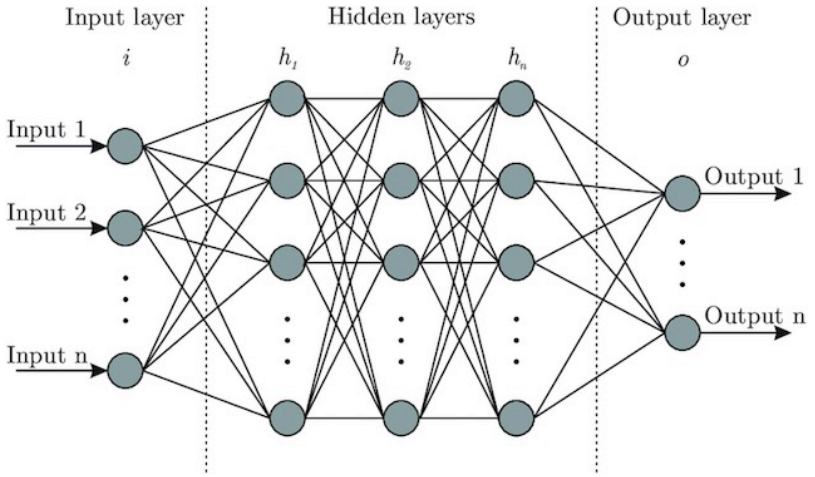
Neural Network

Putting it all together (II)

A Neural Network:

- consists of thousands or even millions of simple processing nodes (neurons) that are densely interconnected
- are organized into layers of nodes/neurons; they're "feed-forward," meaning that data moves through them in only one direction

An individual node might be connected to several nodes in the layer beneath it, from which it receives data, and several nodes in the layer above it, to which it sends data.

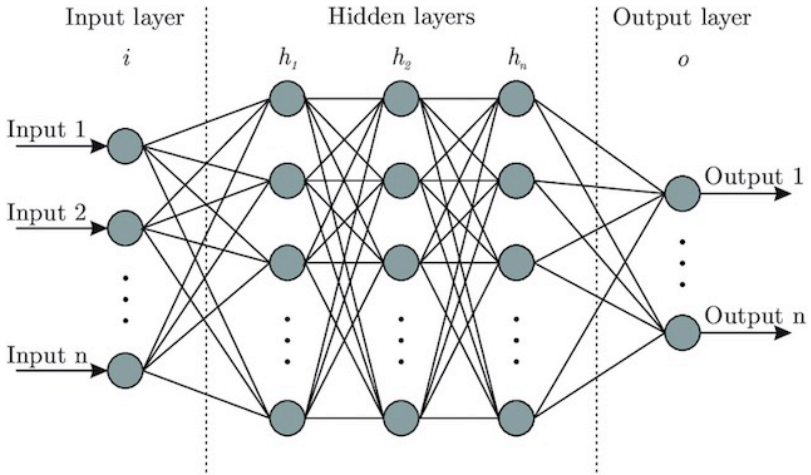


Neural Network

Putting it all together (III)

Inputs:
Source data fed into the neural network,
with the goal of making a decision or prediction about the data.

Inputs to a neural network are typically a set of real values; each value is fed into one of the neurons in the input layer.



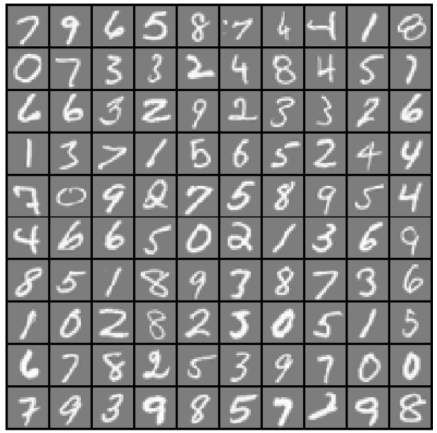
Training Set:
A set of inputs for which the correct outputs are known (it is used to train the neural network).

Neural Network

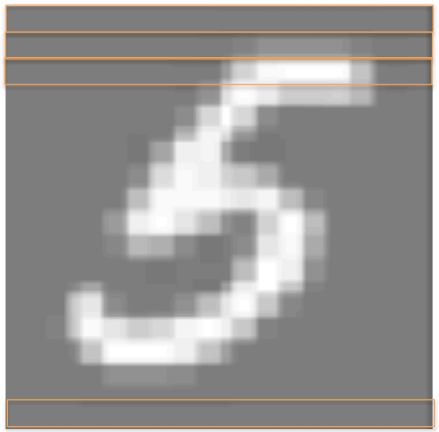
Putting it all together (IV)

Inputs to a neural network are typically a set of real values;
each value is fed into one of the neurons in the input layer.

In the case of images, the image is “unrolled” into a vector x of pixel intensities



20 × 20 pixel images
d = 400



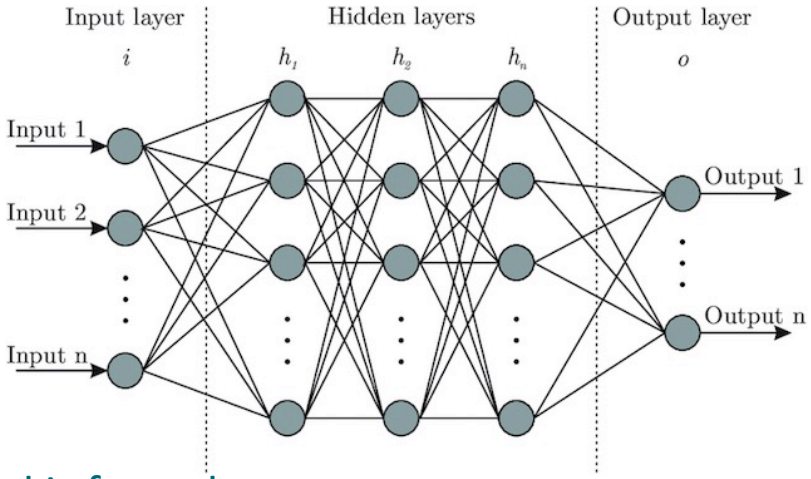
$x_1 \dots x_{20}$
$x_{21} \dots x_{40}$
$x_{41} \dots x_{60}$

$x_{381} \dots x_{400}$

Neural Network

Putting it all together (V)

Outputs:
Neural networks generate their predictions in the form of a set of real values or boolean decisions. Each output value is generated by one of the neurons in the output layer.



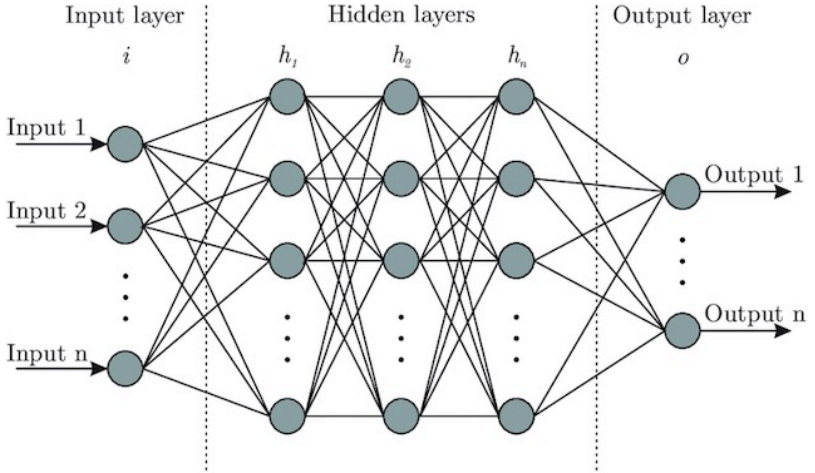
Error Function:
Defines how far the actual output of the current model is from the correct output. When training the model, the objective is to minimize the error function and bring output as close as possible to the correct value.
Examples: MSE/MAE (regression), Cross-Entropy (classification)

Neural Network

Putting it all together (VI)

Hidden Layers and Neurons per Hidden Layer:

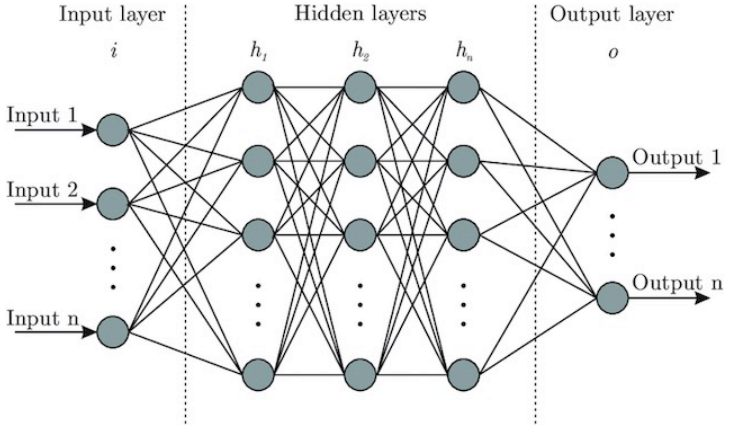
Their numbers are highly dependent on the problem and the architecture of your neural network



Neural Network

Putting it all together (VII)

Weights:
To each of its incoming connections, a node will assign a number known as a “weight”.
When the network is active, the node receives a different data item - a different number - over each of its connections and multiplies it by the associated weight.
It then adds the resulting products together and a bias, yielding a single number.



Activation function:
That number is passed through an activation function = a function that helps the **network learn complex patterns in the data**.
Activation functions help generate output values within an acceptable range, and their non-linear form is crucial for training the network.

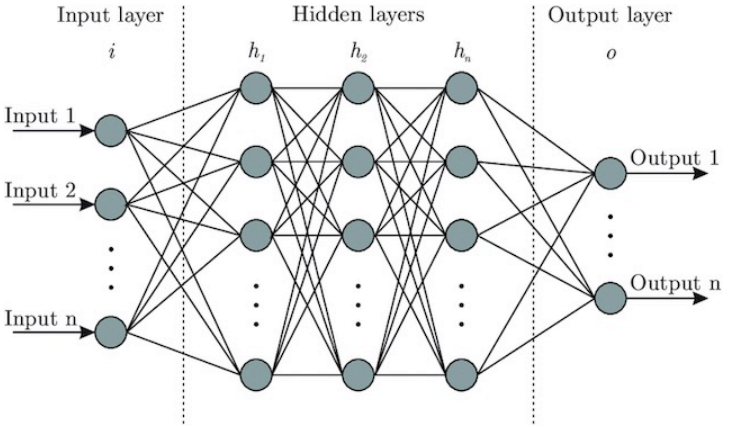
Neural Network

Putting it all together (VIII)

Training

When a neural net is being trained, all of its weights and thresholds are initially set to random values.

Training data is fed to the input layer and passed through the succeeding layers, getting multiplied and added together in complex ways, until it finally arrives, radically transformed, at the output layer, which gives the final predictions.



Then the loss function calculates the error (i.e., how far are these predictions from the correct output/label) which is used to further adjust the weights.

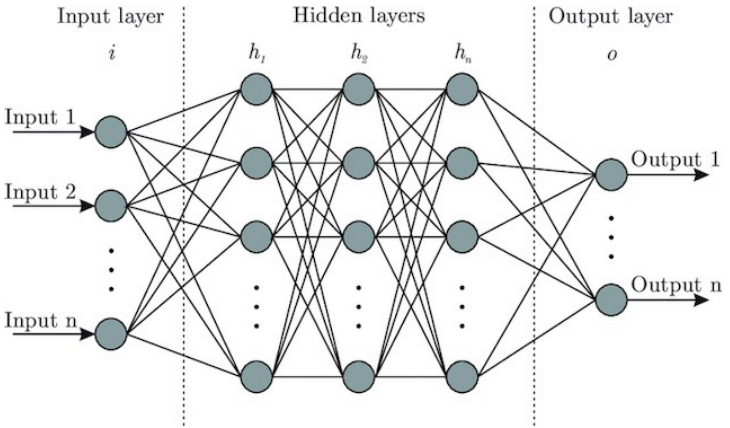
During training, the weights and thresholds are continually adjusted so as to discover the optimal set of weights that training data with the same labels consistently yield similar outputs.

Neural Network

Putting it all together (IX)

Backpropagation Learning Algorithm
In order to discover the optimal weights for the neurons, we perform a backward pass, moving back from the network's prediction to the neurons that generated that prediction.
This is called the backpropagation algorithm.

Backpropagation tracks the derivatives of the activation functions in each successive neuron, to find weights that bring the loss function to a minimum and generate the best prediction.
This mathematical process is called *gradient descent*.



Neural Network

Backpropagation Intuition

Each hidden node is “responsible” for some fraction of the error in each of the output nodes to which it connects

Then, the “blame” is propagated back to provide the error values for the hidden layer

We cycle through our training examples:
– If the output of the network is correct, no changes are made to the weights
– If there is an error, weights are adjusted to reduce the error

The trick is to assess the blame for the error and divide it among the contributing weights

Neural Network

Gradient Descent

Gradient-descent Algorithm: $\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} - \eta \nabla L(\boldsymbol{\theta}^{t-1})$

where:

- η is the learning rate:

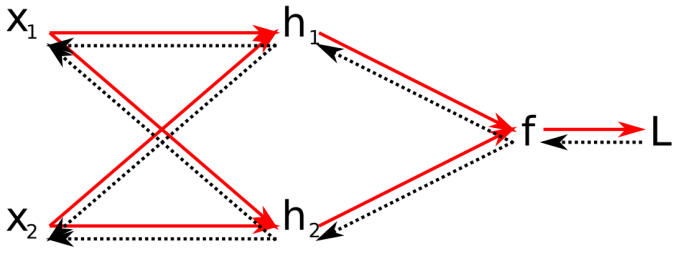
a tuning parameter that determines the step size at each iteration while moving toward a minimum of the loss function. Since it influences to what extent newly acquired information overrides old information, it metaphorically represents the speed at which a machine learning model "learns”

- θ is the weight (can also be referred to as: w)
- t: is the iteration
- ∇L : is the gradient of the loss (del/nabla of the loss function)

Neural Network

Backpropagation/Gradient Descent in more detail

Backpropagation Algorithm – (1) Forward Pass

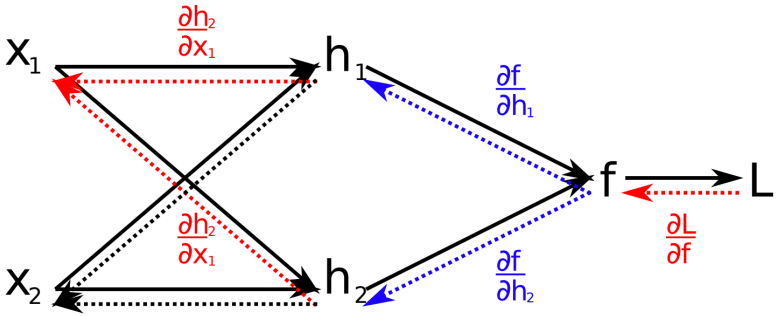


Forward Computation: $L(f(h_1(x_1, x_2, \theta_{h_1}), h_2(x_1, x_2, \theta_{h_2}), \theta_f), y)$

Neural Network

Backpropagation/Gradient Descent in more detail

Backpropagation Algorithm – (2) Chain Rule



Recall the chain rule from calculus:

$$y = f(u)$$
$$u = g(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

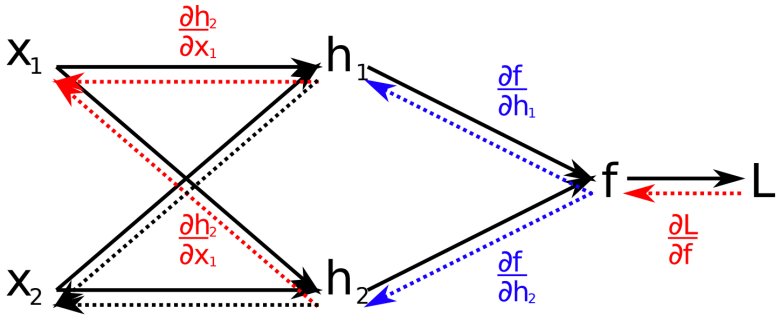
Chain rule of derivatives:

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial x_1} = \frac{\partial L}{\partial f}\left(\frac{\partial f}{\partial h_1}\frac{\partial h_1}{\partial x_1} + \frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial x_1}\right)$$

Neural Network

Backpropagation/Gradient Descent in more detail

Backpropagation Algorithm – (3) Shared Derivatives



Local derivatives are shared:

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial f}\left(\frac{\partial f}{\partial h_1}\frac{\partial h_1}{\partial x_1} + \frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial x_1}\right)$$

$$\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial f}\left(\frac{\partial f}{\partial h_1}\frac{\partial h_1}{\partial x_2} + \frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial x_2}\right)$$

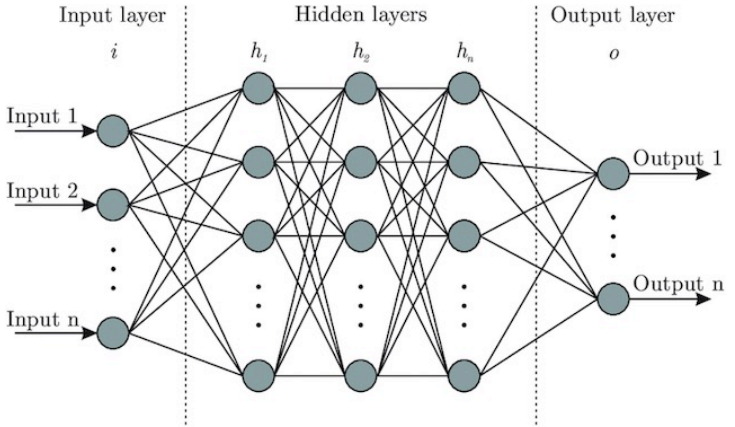
Neural Network

Putting it all together (II)

- Parameters of the model = model weights (& biases)

A model parameter is internal to learn your network and is used to make predictions in a model.
The objective of training is to learn the values of the model parameters.

- A hyperparameter is an external parameter set by the operator of the model; it is a setting that affects the structure or operation of the neural network.



In deep learning projects, tuning hyperparameters is the primary way to build a network that provides accurate predictions for a certain problem. Different values of hyperparameters can have a major impact on the performance of the network.

Neural Network

Putting it all together (II)

Hyperparameters related to neural network structure	Hyperparameters related to the training algorithm
• Number of hidden layers • Number of hidden units/nodes • Regularization techniques • Activation function • Weights initialization	• Learning rate • Number of Epochs • Batch size • Optimizer algorithm

Activation Function

Neural Network

Activation Function: Why use one?

- biological similarity

- keep the value of the output from the neuron restricted to a certain limit as per our requirement. This value if not restricted to a certain limit can go very high/low in magnitude especially in case of very deep neural networks that have millions of parameters. This will lead to computational issues

- the most important feature in an activation function is its ability to add non-linearity into a neural network

Neural Network

Activation Function: Desirable Features (I)

- **Vanishing Gradient problem:**

Neural Networks are trained using gradient descent;

in the backward propagation step we apply the chain rule to get the change in weights to reduce the loss after every epoch.

Now imagine such a chain rule going through multiple layers while backpropagating.

If the value is between 0 and 1, then several such values will get multiplied to calculate the gradient of the initial layers.

This reduces (vanishes) the value of the gradient for the initial layers and those layers are not able to learn properly.

Neural Network

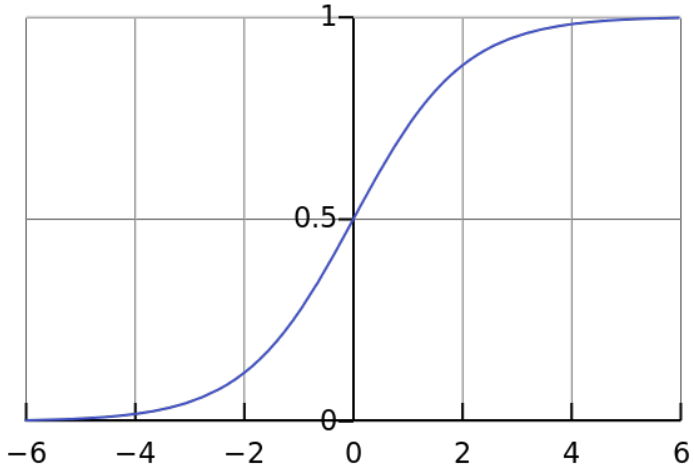
Activation Function: Desirable Features (II)

- **Zero-Centered:** Output of the activation function should be symmetrical at zero so that the gradients do not shift to a particular direction
- **Computational Expense**: Activation functions are applied after every layer and need to be calculated millions of times in deep networks → they should be computationally inexpensive to calculate
- **Differentiable:** Neural networks are trained using gradient descent, hence the layers in the model need to be differentiable. **This is a necessary requirement for a function to work as activation function layer.**

Neural Network

Activation Function: Common cases

Sigmoid: $A = \frac{1}{1+e^{-x}}$



X in [-2,2] → Y values are very steep → any small changes in the values of X in that region will cause values of Y to change significantly.

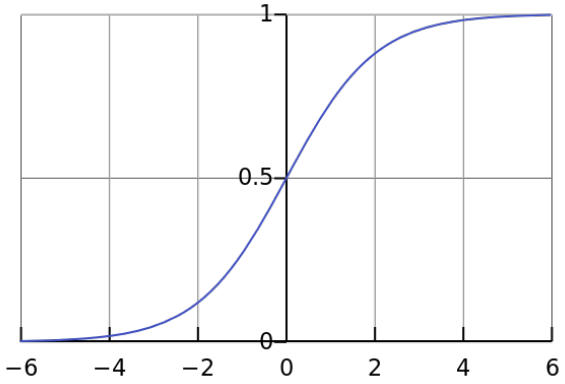
This function has a tendency to bring the Y values to either end of the curve

→It's good for a classifier

Neural Network

Activation Function: Common cases

Sigmoid: $$A = \frac{1}{1+e^{-x}}$$



- Towards either end of the sigmoid function, the Y values tend to respond very less to changes in X. The gradient at that region is going to be small → vanishing gradients
- Computationally expensive
- Not zero-centered
- Is generally used for binary classification problems

Neural Network

Activation Function: Common cases

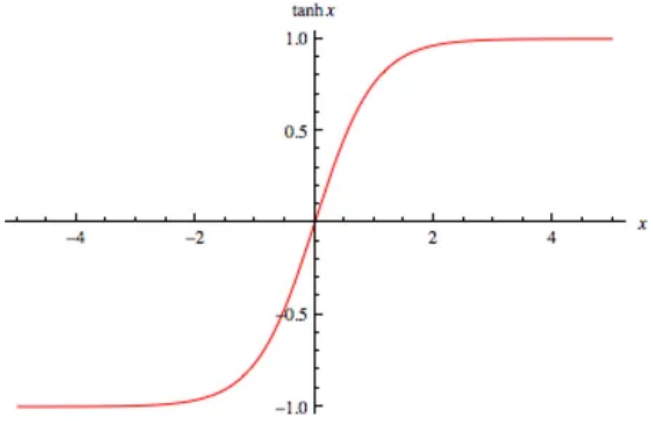
Softmax

- A more generalised form of the sigmoid
- It is used in **multi-class classification problems**
- It produces values in [0,1]
- It normalizes the output of a network to a probability distribution over predicted output classes

Neural Network

Activation Function: Common cases

Tanh: $$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

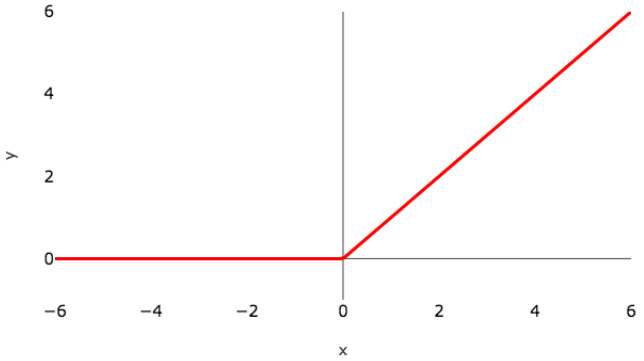


- It is zero-centered
- It is bound in [-1,1]
- Computationally expensive
- Vanishing gradient problem

Neural Network

Activation Function: Common cases

ReLU : $A(x) = \max(0,x)$

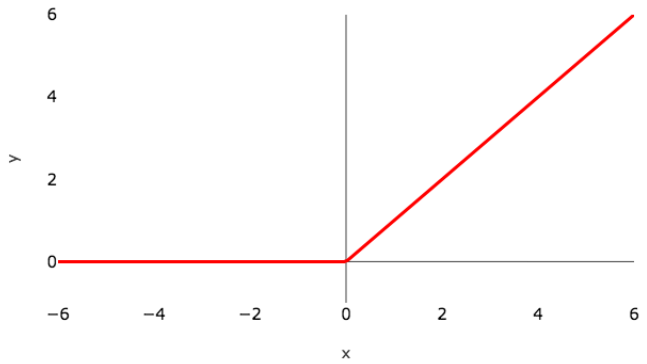


- Widely used activation function
- Easy to compute
- Does not saturate and does not cause the Vanishing Gradient Problem
- Not being zero centred

Neural Network

Activation Function: Common cases

ReLU : $A(x) = \max(0,x)$



Ideally we want a few neurons in the network to not activate
→ activations sparse and efficient

Imagine a network with almost 50% of the neurons yielding 0 activation because of the characteristic of ReLU
→ fewer neurons are firing (sparse activation) and the network is lighter

Overfitting & Underfitting

Neural Network

Overfitting

The concept of overfitting boils down to the fact that the model is unable to generalize well. It has learned the features of the training set extremely well, but if we give the model any data that slightly deviates from the exact data used during training, it's unable to generalize and accurately predict the output.

Weights tuned to fit the idiosyncrasies of the training examples that are not representative of the general distribution.

Neural Network

Dealing with Overfitting

There are two ways to approach an overfit model:

- Reduce overfitting by training the network on more examples:
  - Adding More/New Data To The Training Set
  - Perform Data Augmentation (e.g., add noise to inputs during training)

- Reduce overfitting by changing the complexity of the network:
  - Change network structure (number of weights)
  - Change network parameters (values of weights)

Neural Network

Dealing with Overfitting

Change network parameters (values of weights):

Some techniques that seek to reduce overfitting:

- dropout: probabilistically remove inputs during training
- batch normalization: normalise the input layer by re-centering and re-scaling (zero mean and unit variance)
- early stopping: monitor model performance on a validation set and stop training when performance degrades
- regularization techniques (e.g., L1 and L2): update the general loss function by adding another term known as the regularization term

- multiple neural networks: train several neural network models in parallel, with the same structure but different weights, and average their outputs

Neural Network

Underfitting: What it is & How to deal with it

Underfitting is the phenomenon that a model can neither model the training data nor generalise to new data.

We can address underfitting by increasing the capacity of the model.
Capacity refers to the ability of a model to fit a variety of functions;
more capacity, means that a model can fit more types of functions for mapping inputs to outputs.

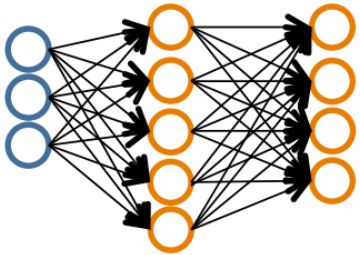
How to tackle underfitting:

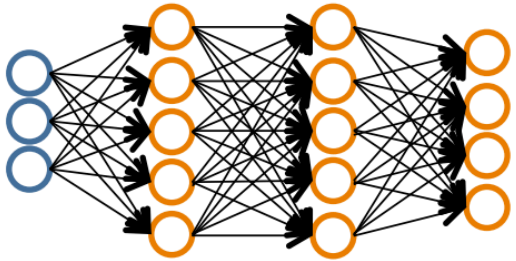
- Adding more layers
- Increasing number of neurons in existing layers
- Training the network for longer

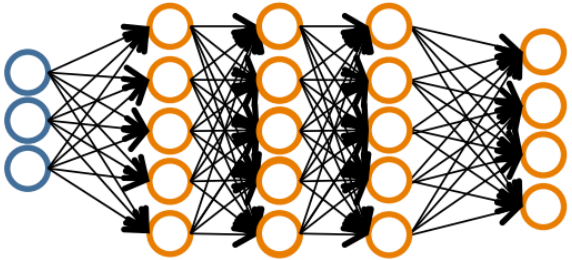
Neural Network

A Few Tips

Pick a network architecture (connectivity pattern between nodes)

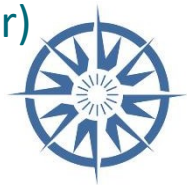






- # input units = # of features in dataset
- # output units = # classes

Reasonable default: 1 hidden layer
if >1 hidden layer, have same # hidden units in every layer (usually the more the better)



TensorFlow

TensorFlow

- TensorFlow tools will be used on this module to implement neural network and deep learning techniques.
- TensorFlow is an open source machine learning platform (software libraries) developed by Google.
- It has become one of the most popular Deep Learning libraries in the field.
- It can run on either CPU or GPU
  - Typically, Deep Neural Networks run much faster on GPUs.

TensorFlow

- The basic idea of TensorFlow is to be able to create data flow graphs.
- These graphs have nodes and edges as seem on the slides above.
- The array (data) passed along from layer of nodes to layer of nodes is known as a Tensor.

Thank you for the attention