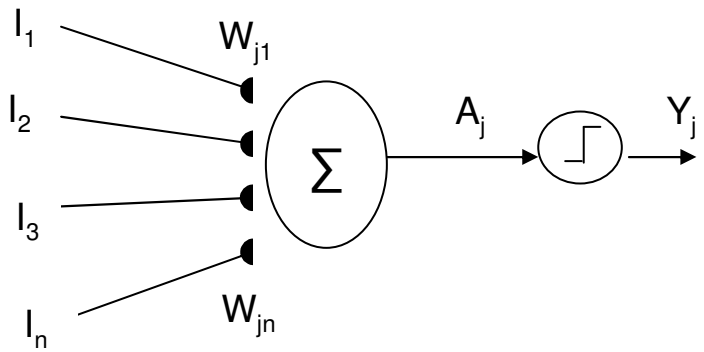
Lecture 2: Single Layer Perceptrons

Kevin Swingler
kms@cs.stir.ac.uk

Recap: McCulloch-Pitts Neuron

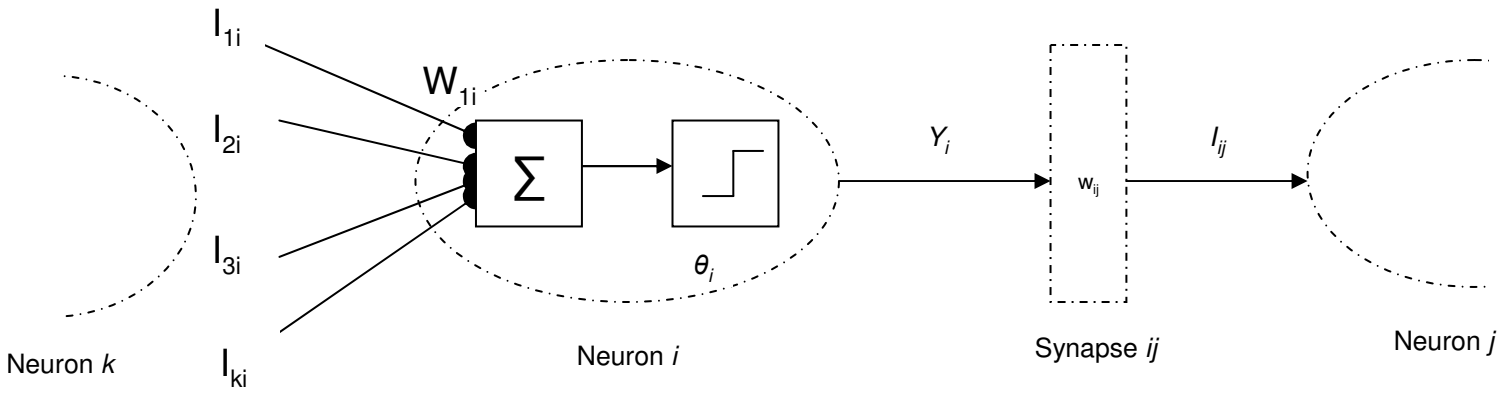
- This vastly simplified model of real neurons is also known as a **Threshold Logic Unit**:



1. A set of synapses (i.e. connections) brings in activations from other neurons
2. A processing unit sums the inputs, and then applies a non-linear activation function
3. An output line transmits the result to other neurons

Networks of McCulloch-Pitts Neurons

One neuron can't do much on its own. Usually we will have many neurons labelled by indices *k*, *i*, *j* and activation flows between via synapses with strengths w_{ki}, w_{ij}:

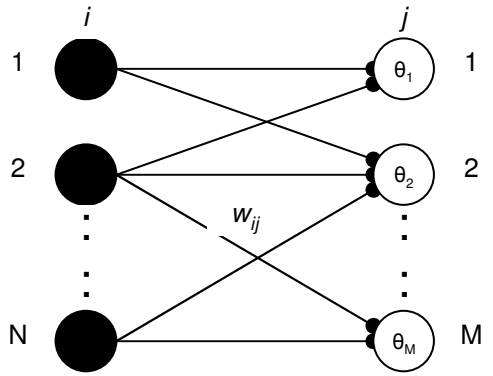


$$I_{ki} = Y_k \cdot w_{ki} \qquad Y_i = \mathrm{sgn}(\sum_{k=1}^{n} I_{ki} - \theta_i) \qquad I_{ij} = Y_i \cdot w_{ij}$$

The Perceptron

We can connect any number of McCulloch-Pitts neurons together in any way we like

An arrangement of one input layer of McCulloch-Pitts neurons feeding forward to one output layer of McCulloch-Pitts neurons is known as a ***Perceptron***.



$$Y_j = \text{sgn}(\sum_{i=1}^{n} Y_i \cdot w_{ij} - \theta_j)$$

Implementing Logic Gates with MP Neurons

We can use McCulloch-Pitts neurons to implement the basic logic gates (e.g. AND, OR, NOT).

It is well known from logic that we can construct any logical function from these three basic logic gates.

All we need to do is find the appropriate connection weights and neuron thresholds to produce the right outputs for each set of inputs.

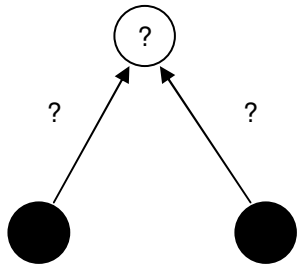
We shall see explicitly how one can construct simple networks that perform NOT, AND, and OR.

Implementation of Logical NOT, AND, and OR

NOT	
in	out
0	1
1	0

AND		
in_1	in_2	out
0	0	0
0	1	0
1	0	0
1	1	1

OR		
in_1	in_2	out
0	0	0
0	1	1
1	0	1
1	1	1

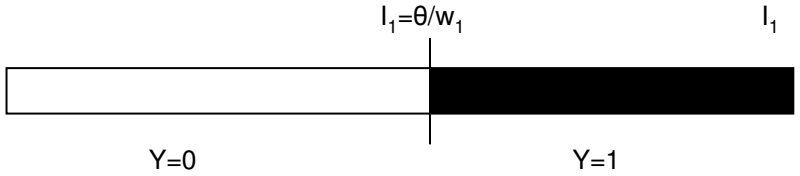


Problem: Train network to calculate the appropriate weights and thresholds in order to classify correctly the different classes (i.e. form **decision boundaries** between classes).

Decision Surfaces

Decision surface is the surface at which the output of the unit is precisely equal to the threshold, i.e. $\sum w_i I_i = \theta$

In **1-D** the surface is just a point:



In **2-D**, the surface is

$$I_1 \cdot w_1 + I_2 \cdot w_2 - \theta = 0$$

which we can re-write as

$$I_2 = \frac{\theta}{w_2} - \frac{w_1}{w_2} I_1$$

So, in 2-D the decision boundaries are **always** straight lines.

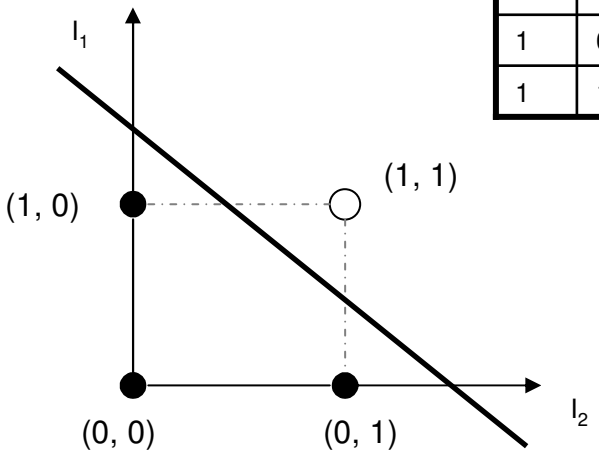
Decision Boundaries for AND and OR

We can now plot the decision boundaries of our logic gates

AND

w1=1, w2=1, θ=1.5

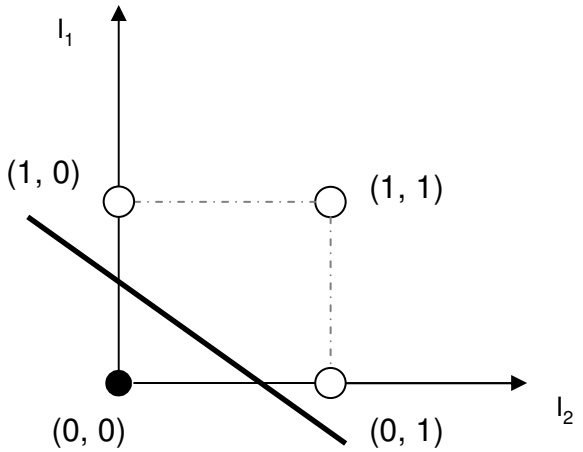
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



OR

w1=1, w2=1, θ=0.5

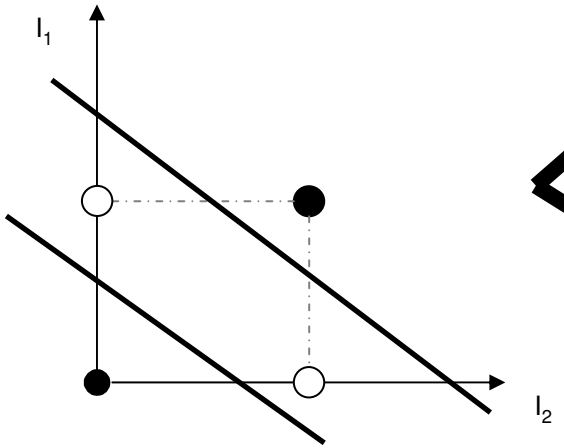
OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1

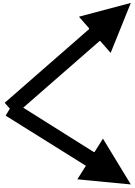


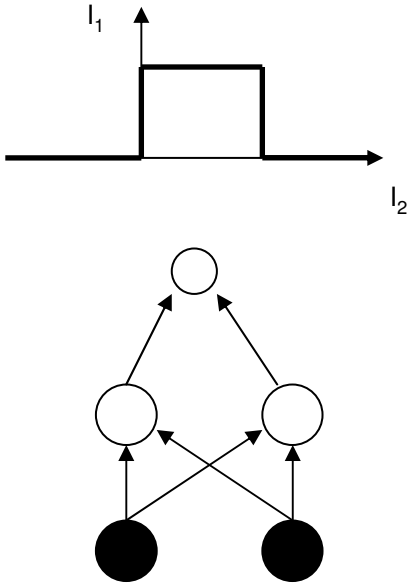
Decision Boundary for XOR

The difficulty in dealing with XOR is rather obvious. We need two straight lines to separate the different outputs/decisions:

XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0







Solution: either change the transfer function so that it has more than one decision boundary, or use a more complex network that is able to generate more complex decision boundaries.

ANN Architectures

Mathematically, ANNs can be represented as weighted directed graphs. The most common ANN architectures are:

Single-Layer Feed-Forward NNs: One input layer and one output layer of processing units. No feedback connections (e.g. a Perceptron)

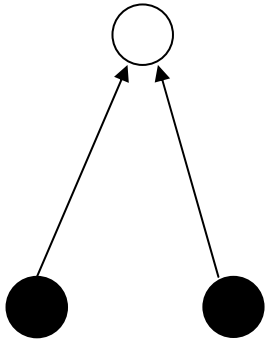
Multi-Layer Feed-Forward NNs: One input layer, one output layer, and one or more hidden layers of processing units. No feedback connections (e.g. a Multi-Layer Perceptron)

Recurrent NNs: Any network with at least one feedback connection. It may, or may not, have hidden units

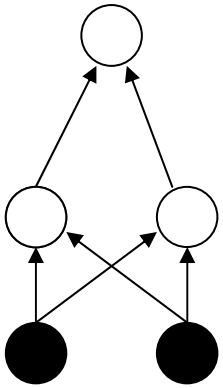
Further interesting variations include: sparse connections, time-delayed connections, moving windows, …

Examples of Network Architectures

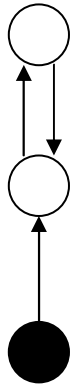
Single Layer Feed-Forward



Multi-Layer Feed-Forward



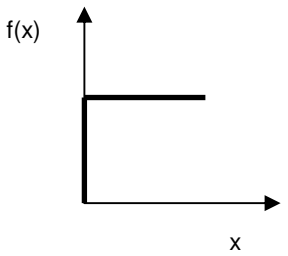
Recurrent Network



Types of Activation/Transfer Function

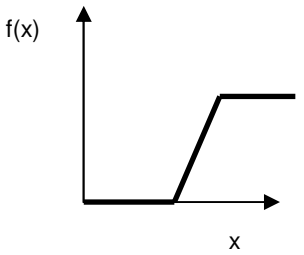
Threshold Function

$$f(x) = \begin{cases} 1 & if \ x \geq 0 \\ 0 & if \ x < 0 \end{cases}$$



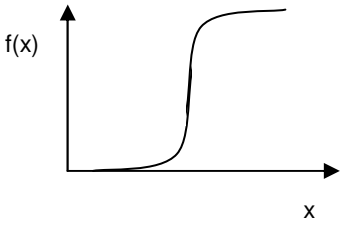
Piecewise-Linear Function

$$f(x) = \begin{cases} 1 & if \ x \geq 0.5 \\ x + 0.5 & if \ -0.5 \leq x \leq 0.5 \\ 0 & if \ x \leq 0.5 \end{cases}$$



Sigmoid Function

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



The Threshold as a Special Kind of Weight

The basic Perceptron equation can be simplified if we consider that the threshold is another connection weight:

$$\sum_{i=1}^{n} I_i \cdot w_{ij} - \theta_j = I_1 \cdot w_{1j} + I_2 \cdot w_{2j} + \ldots + I_n \cdot w_{nj} - \theta_j$$

If we define w_{0j}=-θ_j and I_0=*1* then

$$\sum_{i=1}^{n} I_i \cdot w_{ij} - \theta_j = I_1 \cdot w_{1j} + I_2 \cdot w_{2j} + \ldots + I_n \cdot w_{nj} + I_0 \cdot w_{0j} = \sum_{i=0}^{n} I_i \cdot w_{ij}$$

The Perceptron equation then becomes

$$Y_j = \mathrm{sgn}(\sum_{i=1}^{n} I_i \cdot w_{ij} - \theta_j) = \mathrm{sgn}(\sum_{i=0}^{n} I_i \cdot w_{ij})$$

So, we only have to compute the weights.

Example: A Classification Task

A typical neural network application is classification. Consider the simple example of classifying trucks given their masses and lengths:

Mass	*Length*	*Class*
10.0	6	Lorry
20.0	5	Lorry
5.0	4	Van
2.0	5	Van
2.0	5	Van
3.0	6	Lorry
10.0	7	Lorry
15.0	8	Lorry
5.0	9	Lorry

How do we construct a neural network that can classify any Lorry and Van?

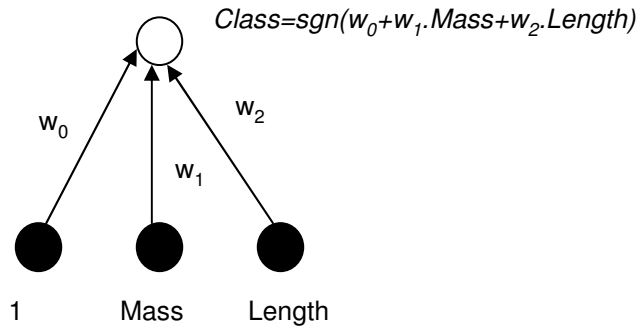
Cookbook Recipe for Building Neural Networks

Formulating neural network solutions for particular problems is a multi-stage process:

1. Understand and specify the problem in terms of inputs and required outputs
2. Take the simplest form of network you think might be able to solve your problem
3. Try to find the appropriate connection weights (including neuron thresholds) so that the network produces the right outputs for each input in its training data
4. Make sure that the network works on its training data and test its generalization by checking its performance on new testing data
5. If the network doesn't perform well enough, go back to stage 3 and try harder
6. If the network still doesn't perform well enough, go back to stage 2 and try harder
7. If the network still doesn't perform well enough, go back to stage 1 and try harder
8. Problem solved – or not

Building a Neural Network (stages 1 & 2)

For our truck example, our inputs can be direct encodings of the masses and lengths. Generally we would have one output unit for each class, with activation 1 for ‘yes’ and 0 for ‘no’. In our example, we still have one output unit, but the activation 1 corresponds to ‘lorry’ and 0 to ‘van’ (or vice versa). The simplest network we should try first is the single layer Perceptron. We can further simplify things by replacing the threshold by an extra weight as we discussed before. This gives us:



Training the Neural Network (stage 3)

Whether our neural network is a simple Perceptron, or a much complicated multi-layer network, we need to develop a systematic procedure for determining appropriate connection weights.

The common procedure is to have the network **learn** the appropriate weights from a representative set of training data.

For classifications a simple Perceptron uses decision boundaries (lines or hyperplanes), which it shifts around until each training pattern is correctly classified.

The process of “shifting around” in a systematic way is called ***learning***.

The learning process can then be divided into a number of small steps.

Supervised Training

1. Generate a training pair or pattern:
   - an input $\mathbf{x} = [\, x_1\ x_2\ \ldots\ x_n]$
   - a target output y_{target} (known/given)
2. Then, present the network with **x** and allow it to generate an output **y**
3. Compare y with y_{target} to compute the error
4. Adjust weights, **w**, to reduce error
5. Repeat 2-4 multiple times

Perceptron Learning Rule

1. Initialize weights at random
2. For each training pair/pattern ($\mathbf{x}$, $\mathbf{y_{target}}$)
   - Compute output y
   - Compute error, $\delta=(y_{target} - y)$
   - Use the error to update weights as follows:

     $\Delta w = w - w_{old}=\eta*\delta*x$ or $w_{new} = w_{old} + \eta*\delta*x$

     where η is called the **learning rate** or **step size** and it determines how smoothly the learning process is taking place.
3. Repeat 2 until convergence (i.e. error δ is zero)

The **Perceptron Learning Rule** is then given by

$$w_{new} = w_{old} + \eta*\delta*x$$

where

$$\delta=(y_{target} - y)$$