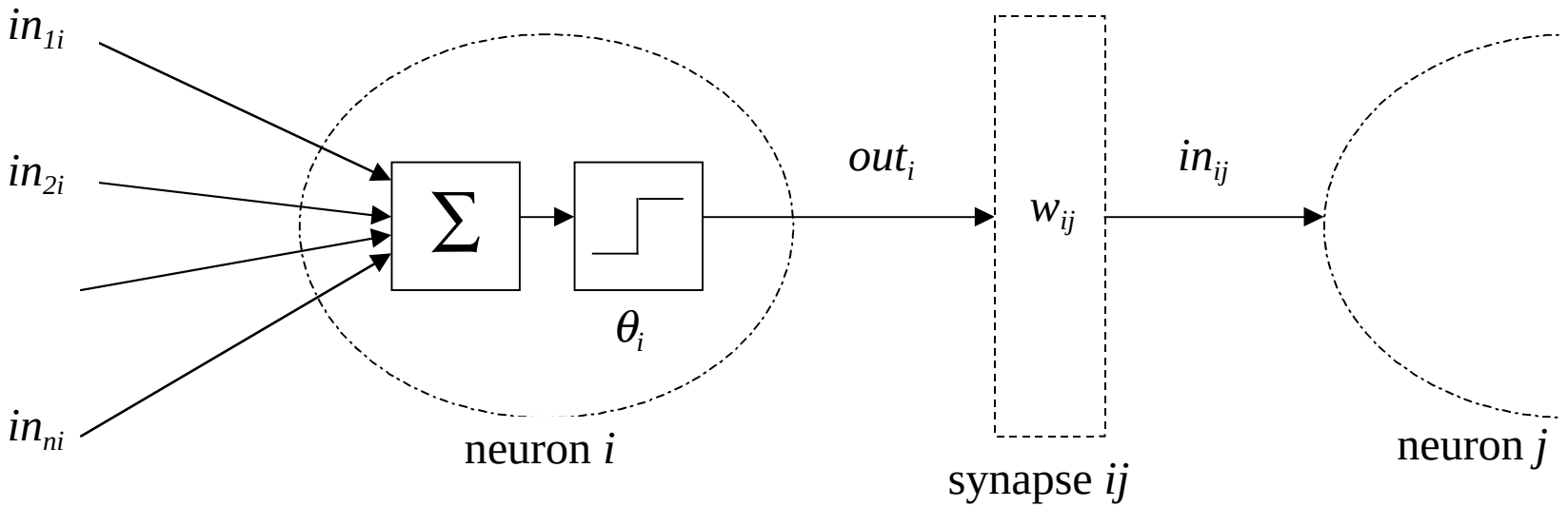
Networks of Artificial Neurons, Single Layer Perceptrons

Introduction to Neural Networks : Lecture 3

1. Networks of McCulloch-Pitts Neurons
2. Single Layer Feed-Forward Neural Networks: The Perceptron
3. Implementing Logic Gates with McCulloch-Pitts Neurons
4. Finding Weights Analytically
5. Limitations of Simple Perceptrons
6. Introduction to More Complex Neural Networks
7. General Procedure for Building Neural Networks

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 them via synapses with strengths w_{ki}, w_{ij}:

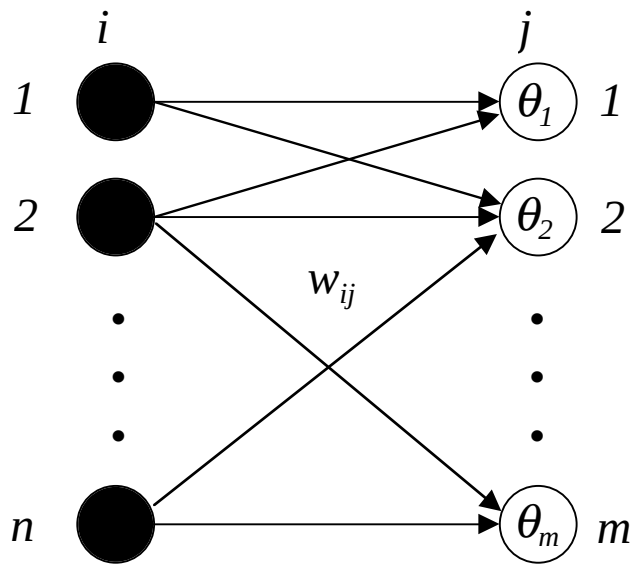


$$in_{ki} = out_k w_{ki} \qquad out_i = \mathrm{sgn}(\sum_{k=1}^{n} in_{ki} - \theta_i) \qquad in_{ij} = out_i w_{ij}$$

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***.



$$out_j = \mathrm{sgn}(\sum_{i=1}^{n} out_i w_{ij} - \theta_j)$$

Already this is a powerful computational device. Later we shall see variations that make it even more powerful.

Implementing Logic Gates with M-P Neurons

We can use McCulloch-Pitts neurons to implement the 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.

It is then a well known result from logic that we can construct any logical function from these three operations.

The resulting networks, however, will usually have a much more complex architecture than a simple Perceptron.

We generally want to avoid decomposing complex problems into simple logic gates, by finding the weights and thresholds that work directly in a Perceptron architecture.

Implementation of Logical NOT, AND, and OR

In each case we have inputs in_i and outputs *out*, and need to determine the weights and thresholds. It is easy to find solutions by inspection:

NOT

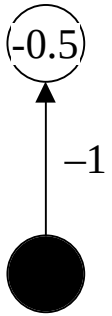
in	*out*
0	1
1	0

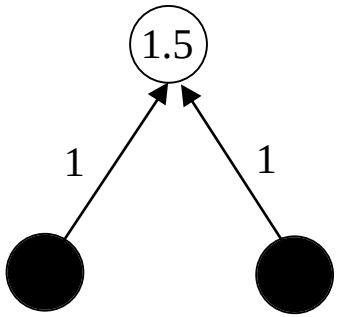
AND

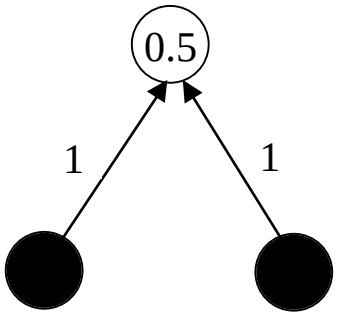
in_1	in_2	*out*
0	0	0
0	1	0
1	0	0
1	1	1

OR

in_1	in_2	*out*
0	0	0
0	1	1
1	0	1
1	1	1





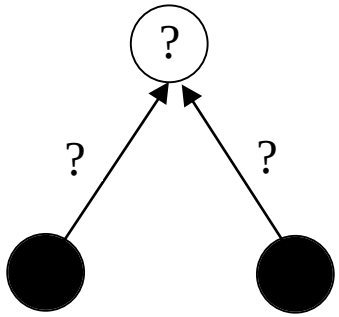


The Need to Find Weights Analytically

Constructing simple networks by hand is one thing. But what about harder problems? For example, what about:

XOR

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



How long do we keep looking for a solution? We need to be able to calculate appropriate parameters rather than looking for solutions by trial and error.

Each training pattern produces a linear inequality for the output in terms of the inputs and the network parameters. These can be used to compute the weights and thresholds.

Finding Weights Analytically for the AND Network

We have two weights w_1 and w_2 and the threshold θ, and for each training pattern we need to satisfy

$$out = \text{sgn}(w_1 in_1 + w_2 in_2 - \theta)$$

So the training data lead to four inequalities:

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

$\Rightarrow$

$$\begin{aligned} w_1\,0 + w_2\,0 - \theta &< 0 \\ w_1\,0 + w_2\,1 - \theta &< 0 \\ w_1\,1 + w_2\,0 - \theta &< 0 \\ w_1\,1 + w_2\,1 - \theta &\geq 0 \end{aligned}$$

$\Rightarrow$

$$\begin{aligned} \theta &> 0 \\ w_2 &< \theta \\ w_1 &< \theta \\ w_1 + w_2 &\geq \theta \end{aligned}$$

It is easy to see that there are an infinite number of solutions. Similarly, there are an infinite number of solutions for the NOT and OR networks.

Limitations of Simple Perceptrons

We can follow the same procedure for the XOR network:

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

$\Rightarrow$

$$
\begin{aligned}
w_1\,0 + w_2\,0 - \theta &< 0 \\
w_1\,0 + w_2\,1 - \theta &\geq 0 \\
w_1\,1 + w_2\,0 - \theta &\geq 0 \\
w_1\,1 + w_2\,1 - \theta &< 0
\end{aligned}
$$

$\Rightarrow$

$$
\begin{aligned}
\theta &> 0 \\
w_2 &\geq \theta \\
w_1 &\geq \theta \\
w_1 + w_2 &< \theta
\end{aligned}
$$

Clearly the second and third inequalities are incompatible with the fourth, so there is in fact no solution. We need more complex networks, e.g. that combine together many simple networks, or use different activation/thresholding/transfer functions.

It then becomes much more difficult to determine all the weights and thresholds by hand. Next lecture we shall see how a neural network can ***learn*** these parameters.

First, we need to consider what these more complex networks might involve.

ANN Architectures/Structures/Topologies

Mathematically, ANNs can be represented as ***weighted directed graphs***. For our purposes, we can simply think in terms of activation flowing between processing units via one-way connections. Three common ANN architectures are:

Single-Layer Feed-forward NNs One input layer and one output layer of processing units. No feed-back connections. (For example, a simple Perceptron.)

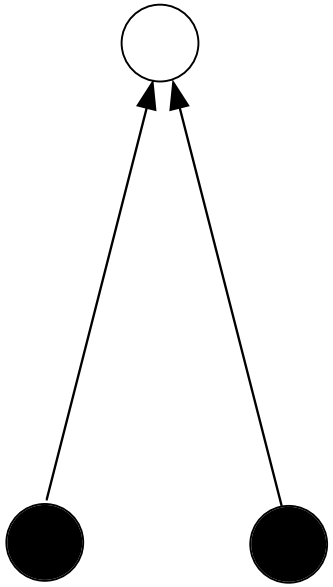
Multi-Layer Feed-forward NNs One input layer, one output layer, and one or more hidden layers of processing units. No feed-back connections. The hidden layers sit in between the input and output layers, and are thus *hidden* from the outside world. (For example, a Multi-Layer Perceptron.)

Recurrent NNs Any network with at least one feed-back connection. It may, or may not, have hidden units. (For example, a Simple Recurrent Network.)

Further interesting variations include: short-cut connections, partial connectivity, time-delayed connections, Elman networks, Jordan networks, moving windows, …

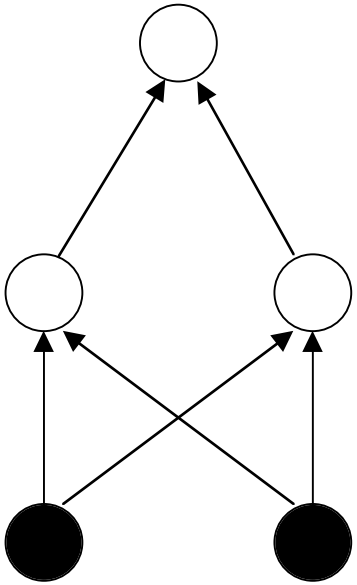
Examples of Network Architectures

Single Layer Feed-forward



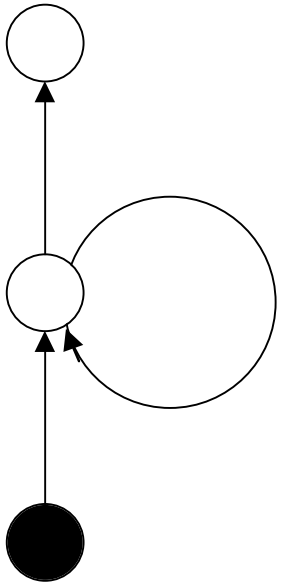
Single-Layer Perceptron

Multi-Layer Feed-forward



Multi-Layer Perceptron

Recurrent Network



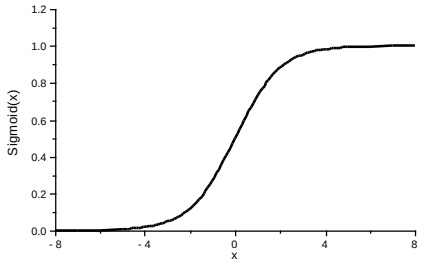
Simple Recurrent Network

Other Types of Activation/Transfer Function

Sigmoid Functions These are smooth (differentiable) and monotonically increasing.

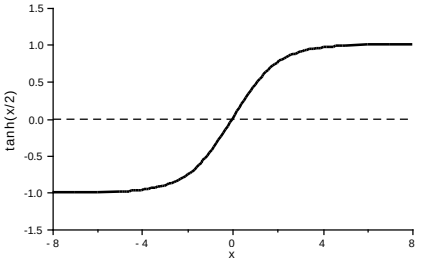
The logistic function

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



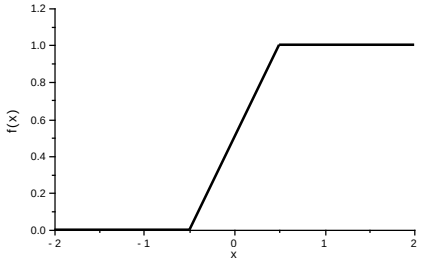
Hyperbolic tangent

$$\tanh\left(\tfrac{x}{2}\right) = \frac{1-e^{-x}}{1+e^{-x}}$$



Piecewise-Linear Functions Approximations of a sigmoid functions.

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



The Threshold as a Special Kind of Weight

It would simplify the mathematics if we could treat the neuron threshold as if it were just another connection weight. The crucial thing we need to compute for each unit *j* is:

$$\sum_{i=1}^{n} out_i w_{ij} - \theta_j = out_1 w_{1j} + out_2 w_{2j} + ... + out_n w_{nj} - \theta_j$$

It is easy to see that if we define $w_{0j} = -\theta_j$ and $out_0 = 1$ then this becomes:

$$\sum_{i=1}^{n} out_i w_{ij} - \theta_j = out_1 w_{1j} + out_2 w_{2j} + ... + out_n w_{nj} + out_0 w_{0j} = \sum_{i=0}^{n} out_i w_{ij}$$

This simplifies the basic Perceptron equation so that:

$$out_j = \text{sgn}(\sum_{i=1}^{n} out_i w_{ij} - \theta_j) = \text{sgn}(\sum_{i=0}^{n} out_i w_{ij})$$

We just have to include an extra input unit with activation $out_0 = 1$ and then we only need to compute "weights", and no explicit thresholds.

Example : A Classification Task

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

Mass	*Speed*	*Class*
1.0	0.1	Bomber
2.0	0.2	Bomber
0.1	0.3	Fighter
2.0	0.3	Bomber
0.2	0.4	Fighter
3.0	0.4	Bomber
0.1	0.5	Fighter
1.5	0.5	Bomber
0.5	0.6	Fighter
1.6	0.7	Fighter

How do we construct a neural network that can classify *any* Bomber and Fighter?

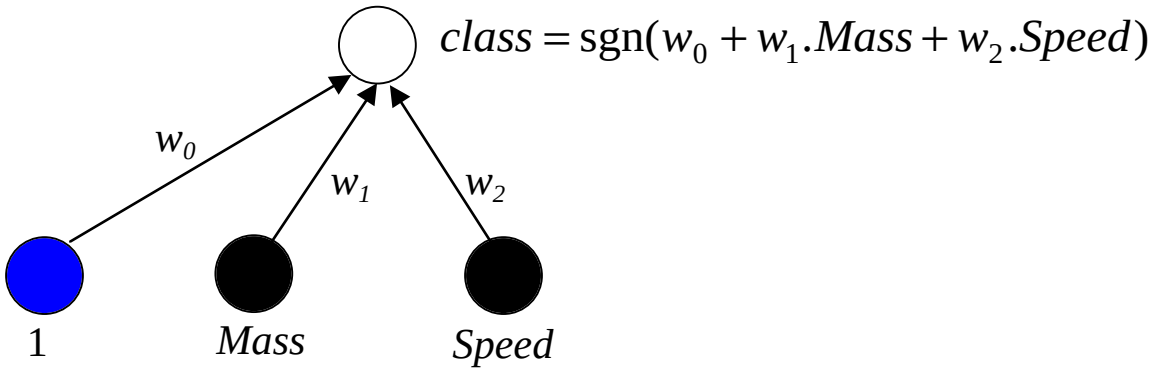
General Procedure for Building Neural Networks

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

1. Understand and specify your problem in terms of ***inputs and required outputs***, e.g. for classification the outputs are the classes usually represented as binary vectors.
2. Take the ***simplest form of network*** you think might be able to solve your problem, e.g. a simple Perceptron.
3. Try to find 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 – move on to next problem.

Building a Neural Network for Our Example

For our aeroplane classifier example, our inputs can be direct encodings of the masses and speeds. Generally we would have one output unit for each class, with activation 1 for 'yes' and 0 for 'no'. With just two classes here, we can have just one output unit, with activation 1 for 'fighter' and 0 for 'bomber' (or vice versa). The simplest network to try first is a simple Perceptron. We can further simplify matters by replacing the threshold by an extra weight as discussed above. This gives us:



That's stages 1 and 2 done. Next lecture we begin a systematic look at how to proceed with stage 3, first for the Perceptron, and then for more complex types of networks.

Overview and Reading

1. Networks of McCulloch-Pitts neurons are powerful computational devices, capable of performing *any* logical function.
2. However, simple Single-Layer Perceptrons with step-function activation functions are limited in what they can do (e.g. they can't do XOR).
3. Many more powerful neural network variations are possible – we can vary the architecture and/or the activation function.
4. Finding the appropriate connection weights and thresholds will then usually be too hard to do by trial and error or simple computation.

Reading

1. Haykin: Sections 1.4, 1.6
2. Gurney: Sections 3.1, 3.2
3. Callan: Sections 1.1, 1.2