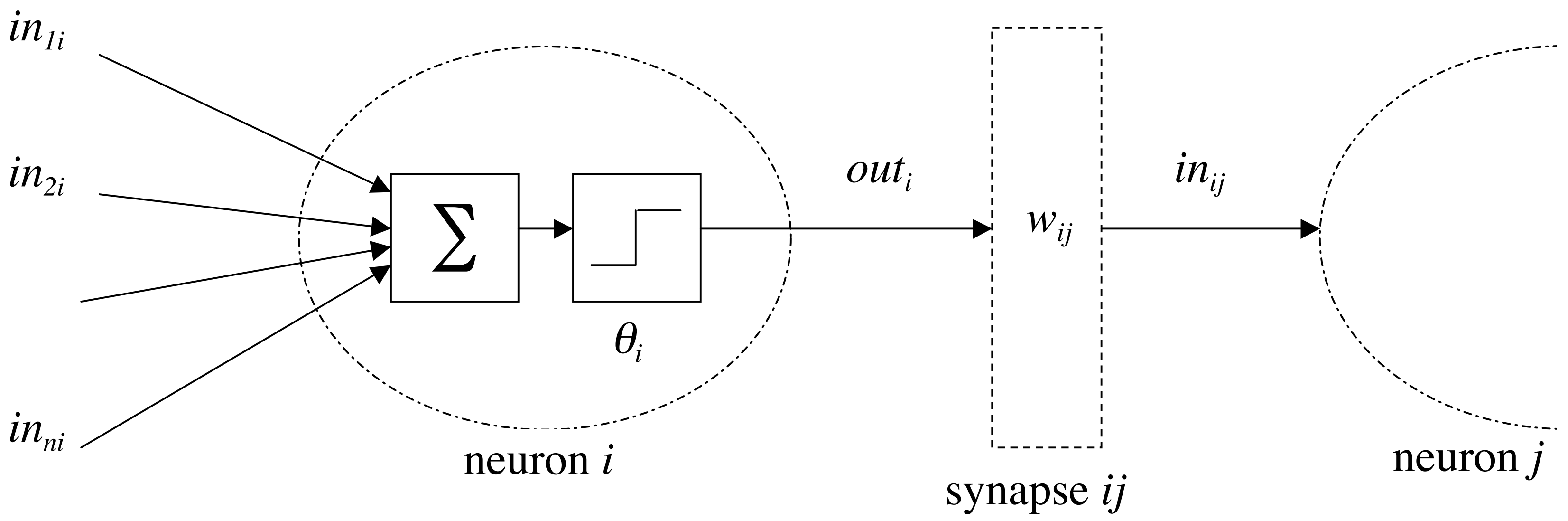
Networks of Artificial Neurons, Single Layer Perceptrons

Neural Computation : Lecture 3

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}:



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

The Need for a Systematic Notation

It requires some care to keep track of all the neural activations and connection weights in a network without introducing confusion or ambiguity.

There are numerous complications that need to be dealt with, for example:

Each neuron has input and output activations, but

the outputs from one neuron provide the inputs to others.

The network has input and output neurons that need special treatment.

Most networks will be built up of layers of neurons, and

it makes sense to number the neurons in each layer separately, but

the weights and activations for different layers must be distinguished.

The letters used for labels/subscripts are arbitrary, and limited in number.

Frequently, the most convenient notation for a new neural network doesn't match that of other networks which have been studied previously.

A Convenient Approach to Notation

To start with, we shall adopt the following conventions for our notation:

The labels “in_i” and “out_i” will now be reserved to indicate the network input and output activations, with other notation used for the activations of other neurons, e.g. “hid_i”.

The network input neurons don’t process information – they simply supply the input activations to the network. Therefore, when counting the layers of neurons within a network, the input layer is not counted, only the layers of processing neurons are.

The labels used to distinguish neurons within a layer (e.g., i = 1, 2, 3, …) are arbitrary, but it helps avoid confusion if they are consistently different for different layers.

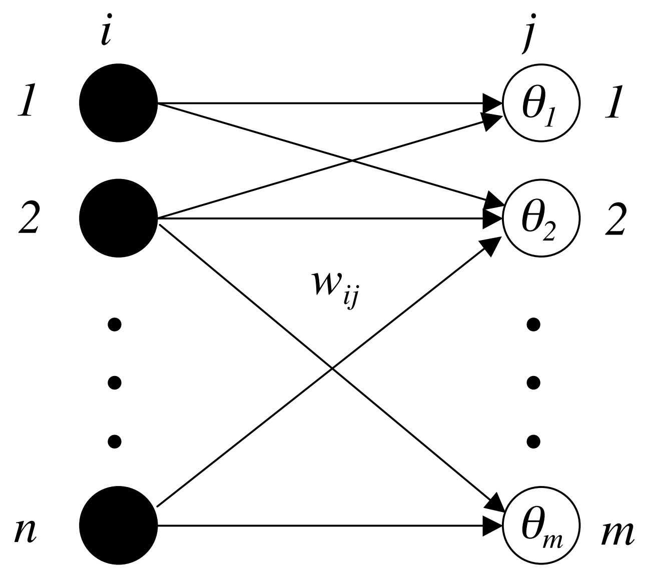
Weights between different layers need to be distinguished, and this can be done most systematically by using some kind of subscript or superscript (e.g., $w_{ij}^{(1)}$, $w_{ij}^{(2)}$, $w_{ij}^{(3)}$).

The term “artificial neuron” is often replaced by “processing unit”, or just “unit”.

The Perceptron

Any number of McCulloch-Pitts neurons can be connected together in any way we like.

The arrangement that has one layer of input neurons feeding forward to one output layer of McCulloch-Pitts neurons, with full connectivity, is known as a ***Perceptron***:



$$out_j = step(\sum_{i=1}^{n} in_i w_{ij} - \theta_j)$$

This is a very simple network, but it is already a powerful computational device. Later we shall see variations of it that make it even more powerful.

Implementing Logic Gates with M-P Neurons

It is possible to implement any form of logic gate using McCulloch-Pitts neurons.

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

The first step to prove this is to show explicitly that it is possible to construct simple networks that perform NOT, AND, and OR. It is then a well known result from logic that one can construct any logical function from these three basic operations. So it follows that a network of McCulloch-Pitts neurons can compute any logical function. This constitutes an ***existence proof*** of the computational power of neural networks.

However, we generally want to avoid decomposing complex problems into basic logic gates, by finding weights and thresholds that work directly in a simple neural network.

The resulting networks will usually have a more complex architectures than simple Perceptrons though, because they require more than a single layer of neurons.

Implementation of Logical NOT, AND, and OR

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

NOT

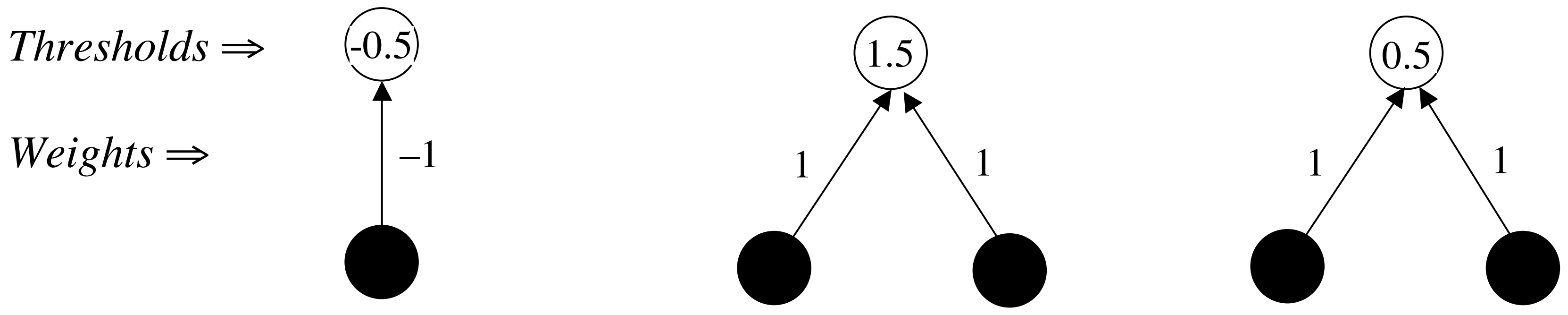
in	out
0	1
1	0

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

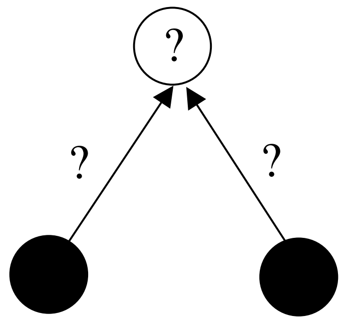


The Need to Find Weights Analytically

Constructing simple networks by hand (e.g., by trial and error) 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 should we keep looking for a solution? We need to be able to calculate appropriate parameter values rather than searching 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:

$$out = step(w_1 in_1 + w_2 in_2 - \theta)$$

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

$$\Rightarrow \quad \begin{array}{l} w_1\,0 + w_2\,0 - \theta < 0 \\ w_1\,0 + w_2\,1 - \theta < 0 \\ w_1\,1 + w_2\,0 - \theta < 0 \\ w_1\,1 + w_2\,1 - \theta \geq 0 \end{array} \quad \Rightarrow \quad \begin{array}{c} \theta > 0 \\ w_2 < \theta \\ w_1 < \theta \\ w_1 + w_2 \geq \theta \end{array}$$

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

$$\Rightarrow \quad \begin{array}{l} w_1 0 + w_2 0 - \theta < 0 \\ w_1 0 + w_2 1 - \theta \geq 0 \\ w_1 1 + w_2 0 - \theta \geq 0 \\ w_1 1 + w_2 1 - \theta < 0 \end{array} \quad \Rightarrow \quad \begin{array}{c} \theta > 0 \\ w_2 \geq \theta \\ w_1 \geq \theta \\ w_1 + w_2 < \theta \end{array}$$

Clearly the first, 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, let us 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. The three most 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.)

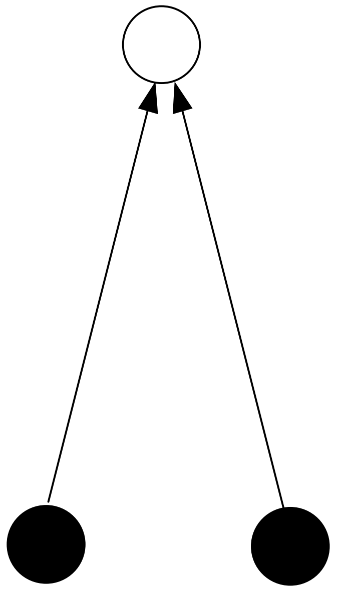
Multi-Layer Feed-forward NNs One input layer, one output layer, and one or more hidden layers of processing units. No feed-back connections. The hidden layers sit in between the input and output layers, and are thus *hidden* from the outside world. (For example, a Multi-Layer Perceptron.)

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, …

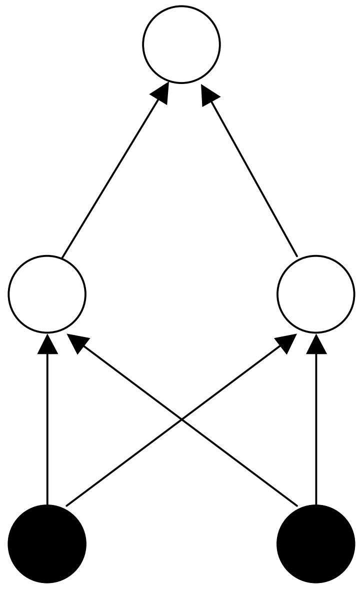
Examples of Network Architecture Types

Single Layer Feed-forward



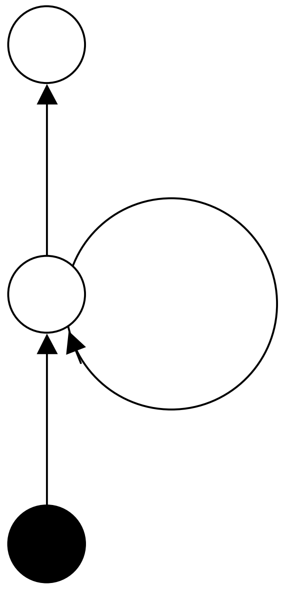
Single-Layer Perceptron

Multi-Layer Feed-forward



Multi-Layer Perceptron

Recurrent Network



Simple Recurrent Network

Types of Neural Network Application

Neural networks perform input-to-output mappings. So far we have looked at simple binary or logic-based mappings, but neural networks are capable of much more than that. The network inputs and outputs can also be real numbers, or integers, or a mixture. The applications fall into two broad categories – classification and regression.

The idea of ***classification*** is to have the neural network decide from a set of inputs which class a given object falls in. The inputs will be suitable "measurements" of the object, and the target outputs represent the associated class, usually as binary vectors. Ideally, the network will output the probability of the object being in each possible class.

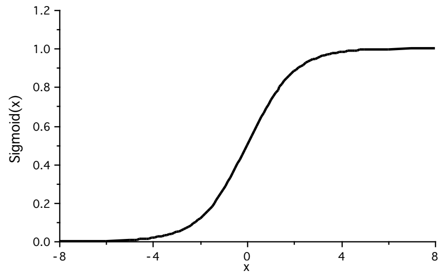
Regression problems are essentially a form of function approximation. The network's linear outputs are real valued numbers corresponding to some underlying function of the inputs that needs to be determined from noisy training data. A special case of this is ***time series prediction*** in which the inputs are measurements at successive points in time, and the output is the prediction of that measurement at later points in time.

Non-Binary Non-Linear Activation/Transfer Function

Sigmoid Functions These are smooth (differentiable) forms of thresholding:

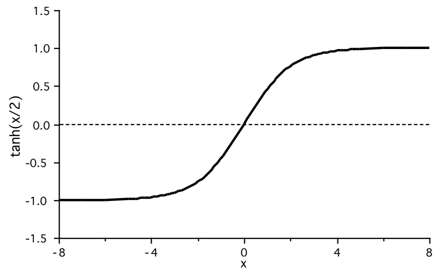
The logistic function

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



Hyperbolic tangent

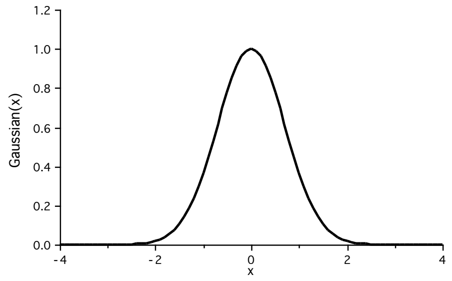
$$\tanh\left(\tfrac{x}{2}\right) = \frac{1-e^{-x}}{1+e^{-x}}$$



Non-Monotonic Functions These tend to lead to less stable mappings, e.g.:

Gaussian function

$$\mathrm{Gaussian}(x) = e^{-x^2}$$



General Procedure 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***, and acquire as much representative training data as you can.
2. Take the ***simplest form of network*** you think might be able to solve the problem, e.g. a single layer Perceptron with step or linear output activation function.
3. Try to find appropriate ***connection weights*** and ***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 performance by checking how well it works on previously unseen ***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. Either the problem is solved or admit defeat – move on to the next problem.

Appropriate Neural Network Inputs

Choosing which “measurements” to use as the inputs to a neural network for a given application is problem dependent and often proves to be rather difficult. This choice is known as ***variable selection*** or ***feature selection***. Clearly, for either classification or regression, the inputs need to be good predictors of the required outputs. Including inputs that have a random relationship with the required outputs will only make the learning harder, and tend to reduce the likelihood of achieving good generalization.

An obvious approach is to simply try all possible combinations of inputs and see which works best. More sophisticated approaches attempt to automate the feature selection using a statistical process of ***automatic relevance determination***, or with some kind of ***evolutionary computation*** approach based on natural selection.

One thing that improves performance in networks with unhelpful inputs is having large amounts of training data, because that makes it easier for the network to learn which inputs are least relevant and reduce the connection weights associated with them.

Neuron Thresholds as a Special Kind of Weight

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

$$\sum_{i=1}^{n} in_i w_{ij} - \theta_j = in_1 w_{1j} + in_2 w_{2j} + \ldots + in_n w_{nj} - \theta_j$$

It is easy to see that if we define $w_{0j} = -\theta_j$ and $in_0 = 1$ then this becomes:

$$\sum_{i=1}^{n} in_i w_{ij} - \theta_j = in_1 w_{1j} + in_2 w_{2j} + \ldots + in_n w_{nj} + in_0 w_{0j} = \sum_{i=0}^{n} in_i w_{ij}$$

This simplifies the basic Perceptron equation so that:

$$out_j = step(\sum_{i=1}^{n} in_i w_{ij} - \theta_j) = step(\sum_{i=0}^{n} in_i w_{ij})$$

We just have to include an extra input unit (a.k.a. the bias unit) with activation $in_0 = 1$ and then we only need to compute "weights", and no explicit thresholds.

Example : A Classification Task

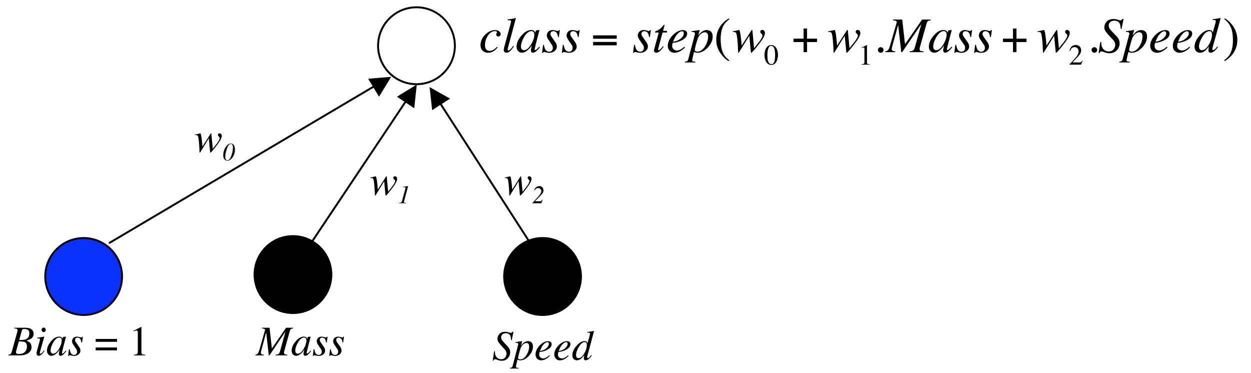
Consider the simple example of classifying aeroplanes. Here, appropriate training data inputs might be a set of measured masses and top speeds of known aeroplanes:

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

Then the aim is to construct a neural network that can classify *any* Bomber or Fighter.

Building a Neural Network for The Example

For the aeroplane classifier example, the 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 would be a simple step-function Perceptron. We can further simplify matters by replacing the threshold by an extra weight w_0 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 – one can vary the architecture and/or the activation function.
4. Finding appropriate connection weights and thresholds will then usually be too hard to do by trial and error or simple computation.

Reading

1. Haykin-2009: Sections 0.4, 0.6
2. Gurney: Sections 3.1, 3.2
3. Callan: Sections 1.1, 1.2