Introduction to Artificial Neural Network

Dr. Paramartha Dutta
Professor, Dept. of Computer & System Sciences
Visva-Bharati University
Santiniketan

Introduction to Artificial Neural Network

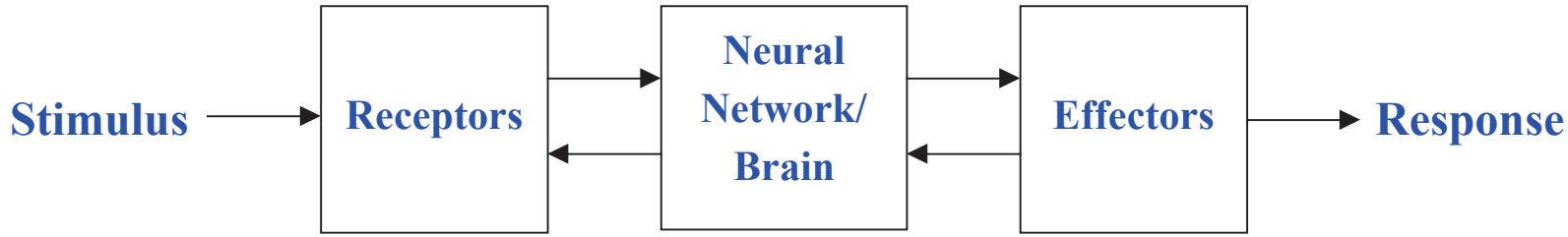
1. Biological Neurons and Neural Networks, Artificial Neurons
2. Networks of Artificial Neurons, Single Layer Perceptrons
3. Learning and Generalization in Single Layer Perceptrons
4. Hebbian Learning and Gradient Descent Learning
5. The Generalized Delta Rule and Practical Considerations
6. Learning in Multi-Layer Perceptrons, Back-Propagation
7. Learning with Momentum, Conjugate Gradient Learning
8. Bias and Variance, Under-Fitting and Over-Fitting
9. Improving Generalization
10. Radial Basis Function Networks: Introduction
11. Radial Basis Function Networks: Applications
12. Self Organizing Maps: Algorithms and Applications

Biological Neurons and Neural Networks, Artificial Neurons

1. Organization of the Nervous System and Brain
2. Brains versus Computers: Some Numbers
3. Biological Neurons and Neural Networks
4. The McCulloch-Pitts Neuron
5. Some Useful Notation: Vectors, Matrices, Functions
6. The McCulloch-Pitts Neuron Equation

The Nervous System

The human nervous system can be broken down into three stages that may be represented in block diagram form as:



The receptors collect information from the environment – e.g. photons on the retina.

The effectors generate interactions with the environment – e.g. activate muscles.

The flow of information/activation is represented by arrows – feedforward and feedback.

Naturally, in this module we will be primarily concerned with the neural network in the middle.

Levels of Brain Organization

The brain contains both large scale and small scale anatomical structures and different functions take place at higher and lower levels.

There is a hierarchy of interwoven levels of organization:

1. Molecules and Ions
2. Synapses
3. Neuronal microcircuits
4. Dendritic trees
5. **Neurons**
6. **Local circuits**
7. Inter-regional circuits
8. Central nervous system

The ANNs we study in this module are crude approximations to levels 5 and 6.

Brains versus Computers : Some numbers

1. There are approximately 10 billion neurons in the human cortex, compared with 10 of thousands of processors in the most powerful parallel computers.

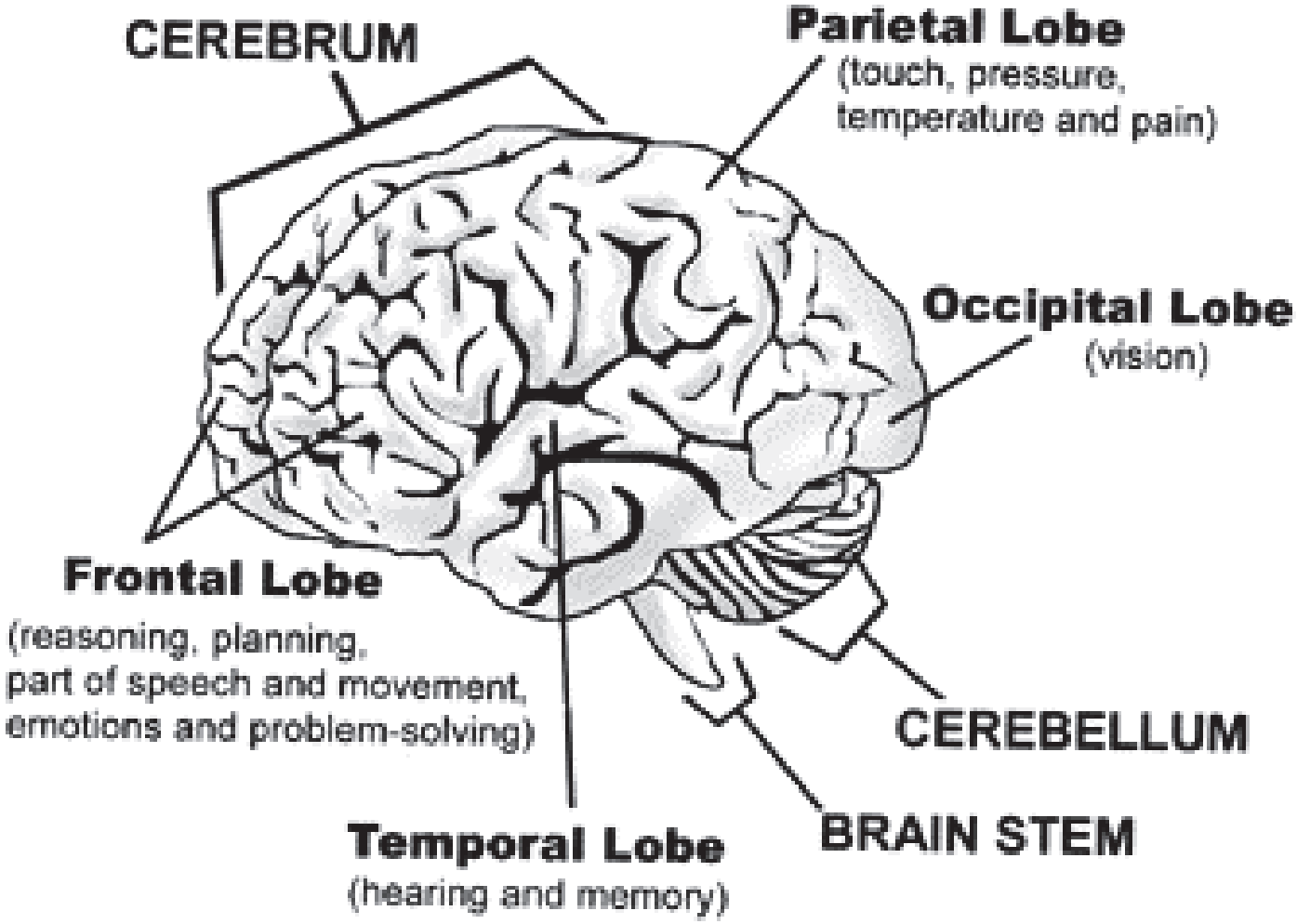
2. Each biological neuron is connected to several thousands of other neurons, similar to the connectivity in powerful parallel computers.

3. Lack of processing units can be compensated by speed. The typical operating speeds of biological neurons is measured in milliseconds (10^{-3} s), while a silicon chip can operate in nanoseconds (10^{-9} s).

4. The human brain is extremely energy efficient, using approximately 10^{-16} joules per operation per second, whereas the best computers today use around 10^{-6} joules per operation per second.

5. Brains have been evolving for tens of millions of years, computers have been evolving for tens of decades.

Structure of a Human Brain



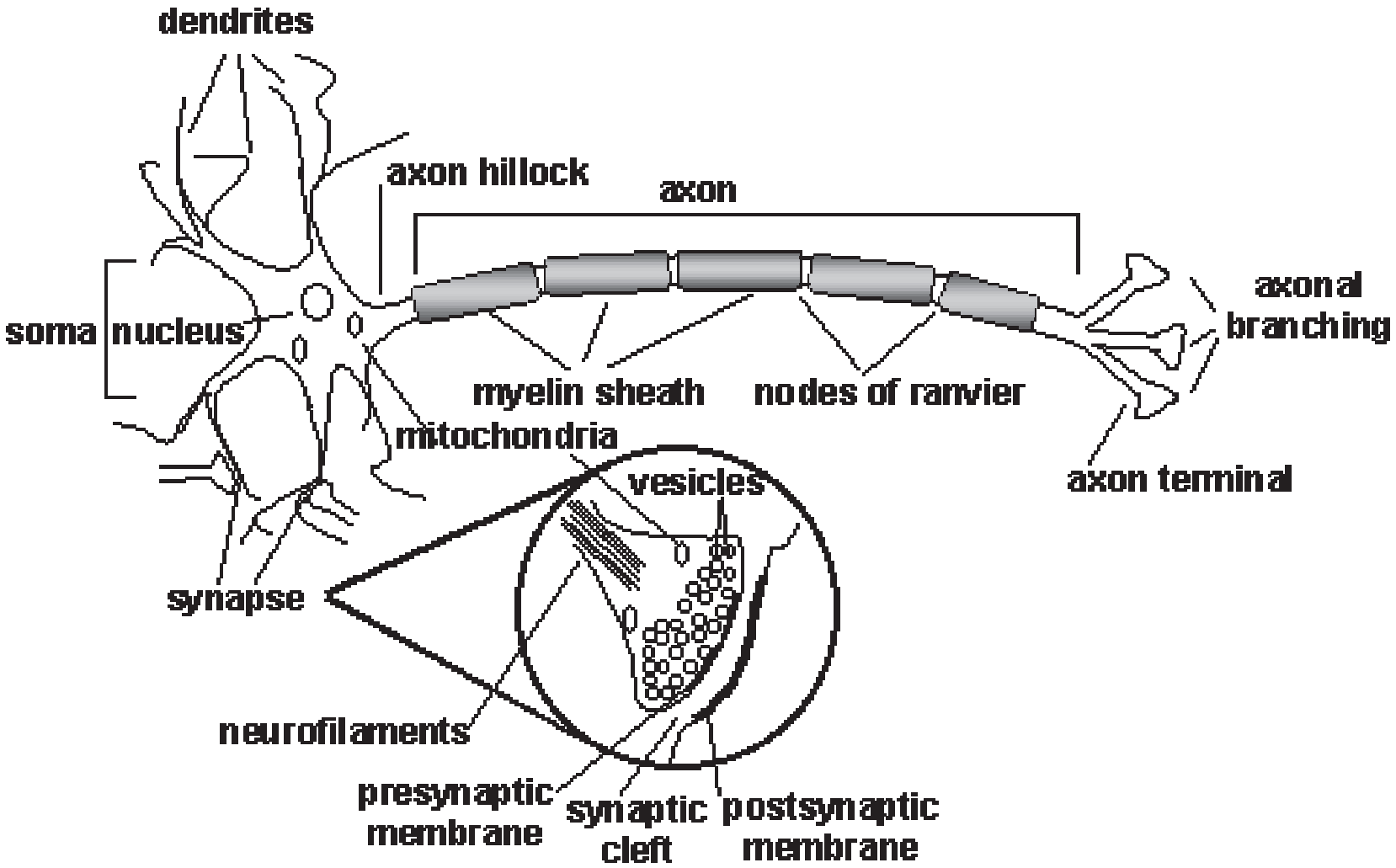
Slice Through a Real Brain

http://medlib.med.utah.edu/WebPath/HISTHTML/NEURANAT/NEURANCA.html

Basic Components of Biological Neurons

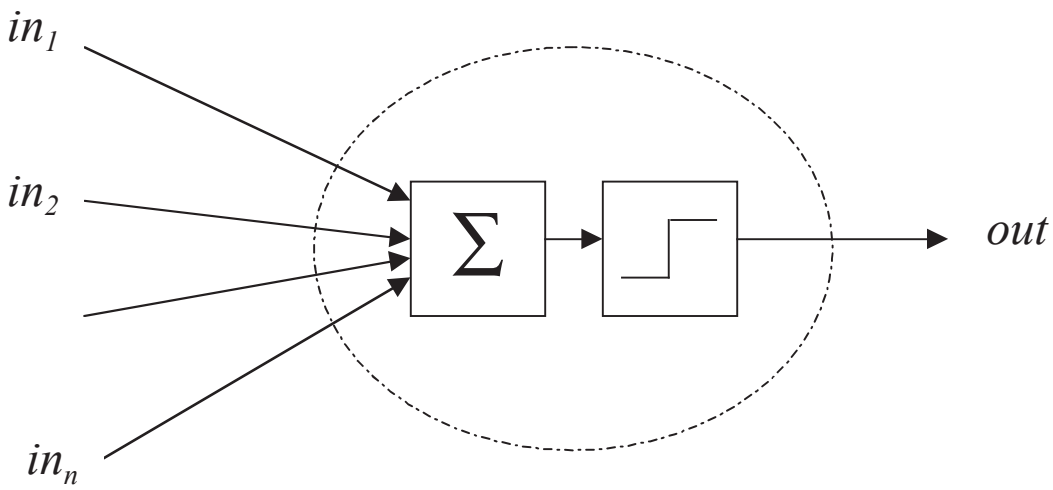
1. The majority of ***neurons*** encode their activations or outputs as a series of brief electrical pulses (i.e. spikes or action potentials).
2. The neuron's ***cell body (soma)*** processes the incoming activations and converts them into output activations.
3. The neuron's ***nucleus*** contains the genetic material in the form of DNA. This exists in most types of cells, not just neurons.
4. ***Dendrites*** are fibres which emanate from the cell body and provide the receptive zones that receive activation from other neurons.
5. ***Axons*** are fibres acting as transmission lines that send activation to other neurons.
6. The junctions that allow signal transmission between the axons and dendrites are called ***synapses***. The process of transmission is by diffusion of chemicals called ***neurotransmitters*** across the synaptic cleft.

Schematic Diagram of a Biological Neuron



The 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 (i.e. squashing/transfer/threshold function).
3. An output line transmits the result to other neurons.

Some Useful Notation

We often need to talk about ordered sets of related numbers – we call them ***vectors***, e.g.

$$\boldsymbol{x} = (x_1, x_2, x_3, \ldots, x_n) \quad , \quad \boldsymbol{y} = (y_1, y_2, y_3, \ldots, y_m)$$

The components x_i can be added up to give a ***scalar*** (number), e.g.

$$s = x_1 + x_2 + x_3 + \ldots + x_n = \sum_{i=1}^{n} x_i$$

Two vectors of the same length may be ***added*** to give another vector, e.g.

$$\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y} = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n)$$

Two vectors of the same length may be ***multiplied*** to give a scalar, e.g.

$$p = \boldsymbol{x}.\boldsymbol{y} = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n = \sum_{i=1}^{n} x_i y_i$$

To any ambiguity/confusion, we will mostly use the component notation (i.e. explicit indices and summation signs) throughout this module.

The Power of the Notation : Matrices

We can use the same vector component notation to represent complex things with many more dimensions/indices. For two indices we have matrices, e.g. an $m \times n$ ***matrix***

$$\mathbf{w} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{1n} \\ \vdots & \vdots & & \vdots \\ w_{m1} & w_{m1} & \cdots & w_{mn} \end{pmatrix}$$

Matrices of the same size can be ***added*** or ***subtracted*** component by component.

An $m \times n$ matrix **a** can be ***multiplied*** with an $n \times p$ matrix **b** to give an $m \times p$ matrix **c**. This becomes straightforward if we write it in terms of components:

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

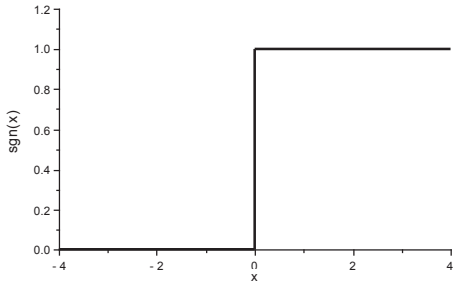
An n component vector can be regarded as a $1 \times n$ or $n \times 1$ matrix.

Some Useful Functions

A function $y = f(x)$ describes a relationship (input-output mapping) from x to y.

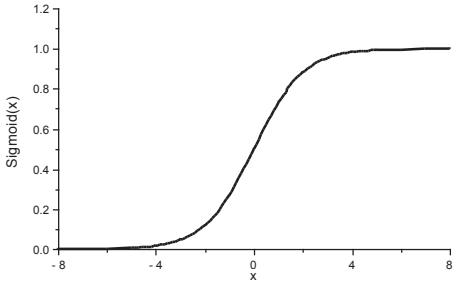
Example 1 The threshold or sign function $\text{sgn}(x)$ is defined as

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



Example 2 The logistic or sigmoid function $\text{Sigmoid}(x)$ is defined as

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



This is a smoothed (differentiable) form of the threshold function.

The McCulloch-Pitts Neuron Equation

Using the above notation, we can now write down a simple equation for the ***output*** *out* of a McCulloch-Pitts neuron as a function of its *n* ***inputs*** in_i :

$$out = \text{sgn}(\sum_{i=1}^{n} in_i - \theta)$$

where θ is the neuron's activation ***threshold***. We can easily see that:

$$out = 1 \quad \text{if} \quad \sum_{k=1}^{n} in_k \geq \theta \qquad\qquad out = 0 \quad \text{if} \quad \sum_{k=1}^{n} in_k < \theta$$

Note that the McCulloch-Pitts neuron is an extremely simplified model of real biological neurons. Some of its missing features include: non-binary inputs and outputs, non-linear summation, smooth thresholding, stochasticity, and temporal information processing.

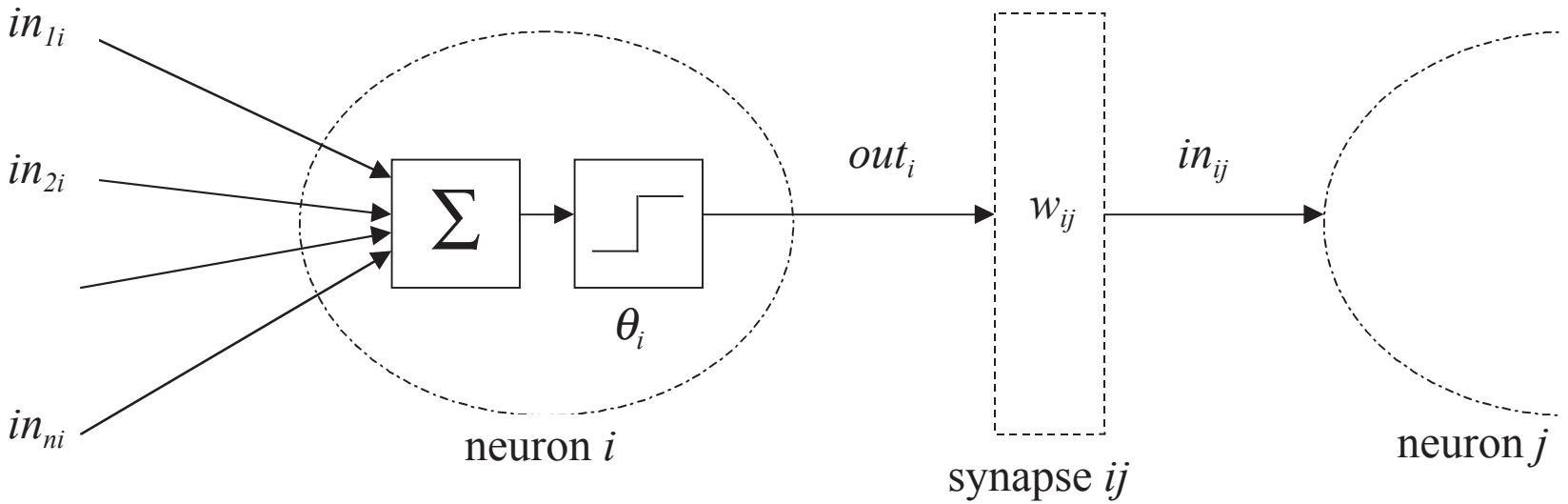
Nevertheless, McCulloch-Pitts neurons are computationally very powerful. One can show that assemblies of such neurons are capable of universal computation.

Networks of Artificial Neurons, Single Layer Perceptrons

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}:

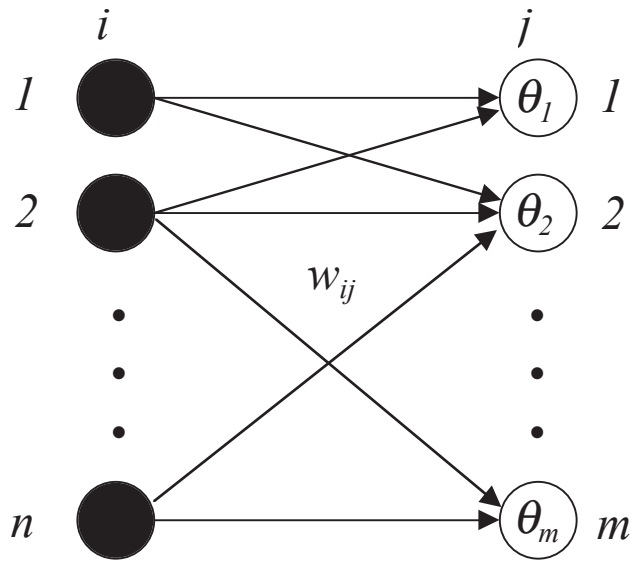


$$in_{ki} = out_k w_{ki} \qquad out_i = \mathrm{sgn}(\sum_{k=1}^{n} in_{ki} - \theta_i) \qquad in_{ij} = out_i w_{ij}$$

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

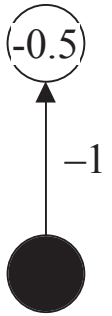
in	*out*
0	1
1	0

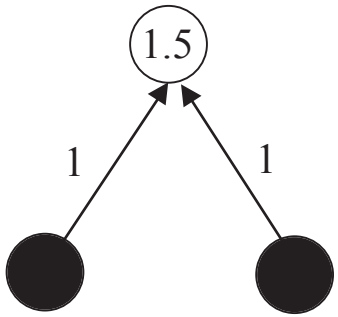
AND

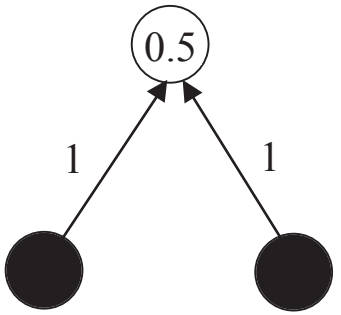
in_1	in_2	*out*
0	0	0
0	1	0
1	0	0
1	1	1

OR

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





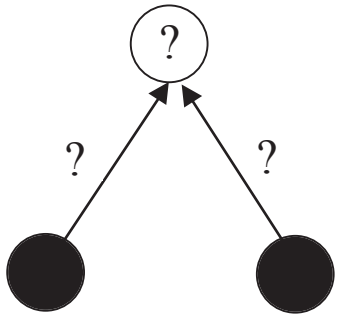


The Need to Find Weights Analytically

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.)

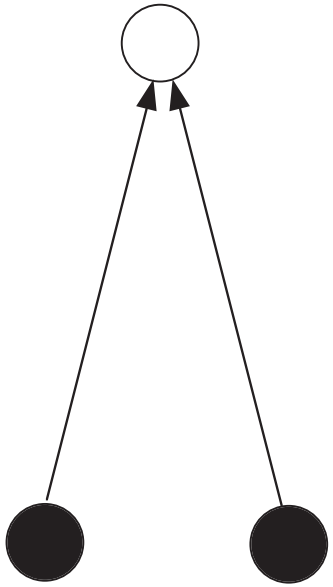
Multi-Layer Feed-forward NNs One input layer, one output layer, and one or more hidden layers of processing units. No feed-back connections. The hidden layers sit in between the input and output layers, and are thus *hidden* from the outside world. (For example, a Multi-Layer Perceptron.)

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, …

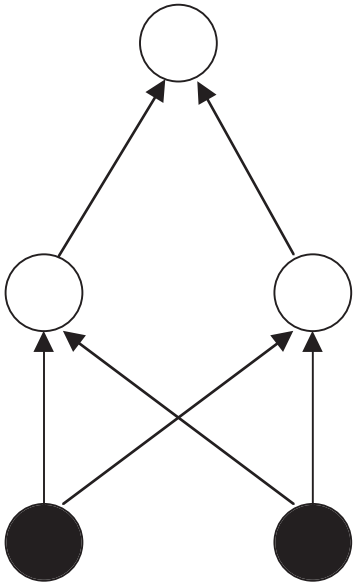
Examples of Network Architectures

Single Layer Feed-forward



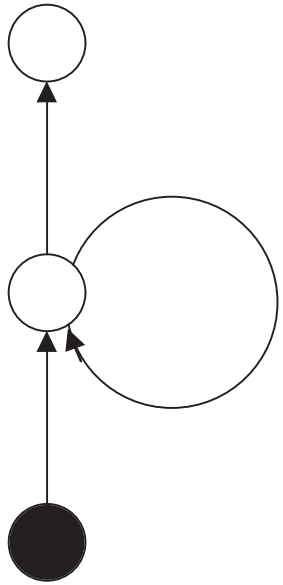
Single-Layer Perceptron

Multi-Layer Feed-forward



Multi-Layer Perceptron

Recurrent Network



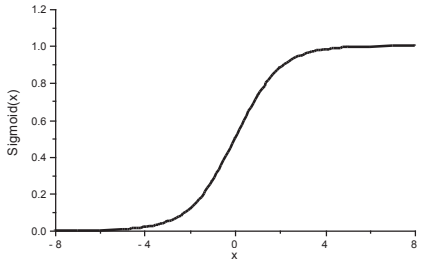
Simple Recurrent Network

Other Types of Activation/Transfer Function

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

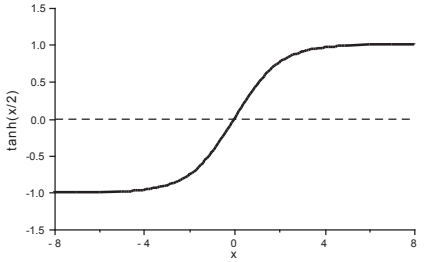
The logistic function

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



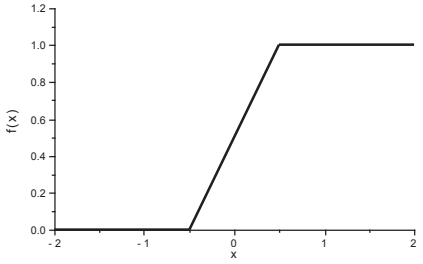
Hyperbolic tangent

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



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:

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

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

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

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?

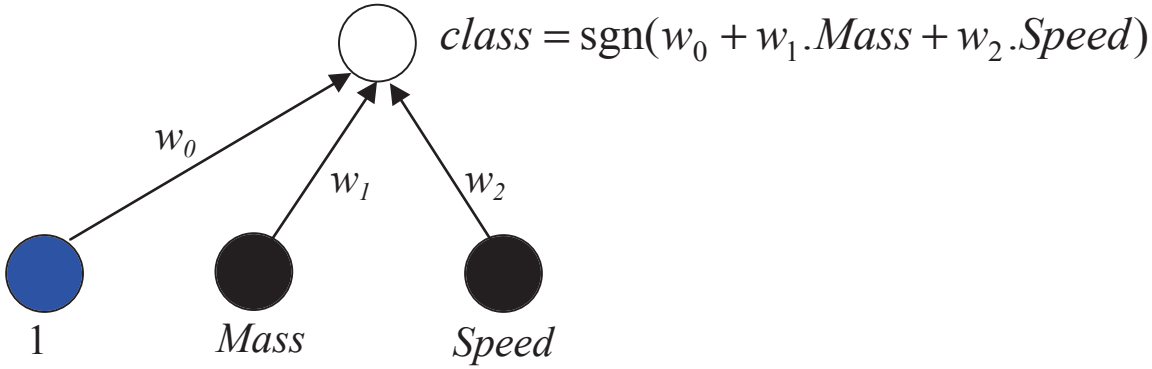
General Procedure for Building Neural Networks

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:



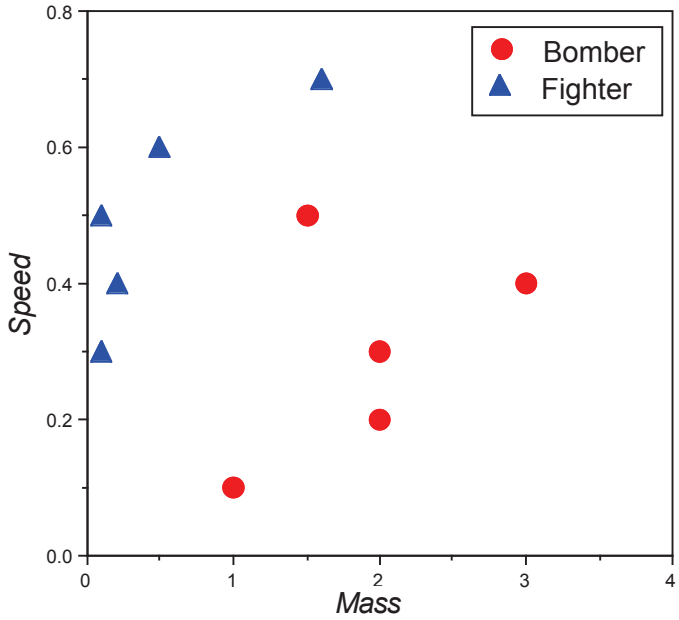
That's stages 1 and 2 done. Next lecture we begin a systematic look at how to proceed with stage 3, first for the Perceptron, and then for more complex types of networks.

Learning and Generalization in Single Layer Perceptrons

1. What Can Perceptrons do?
2. Decision Boundaries – The Two Dimensional Case
3. Decision Boundaries for AND, OR and XOR
4. Decision Hyperplanes and Linear Separability
5. Learning and Generalization
6. Training Networks by Iterative Weight Updates
7. Convergence of the Perceptron Learning Rule

What Can Perceptrons Do?

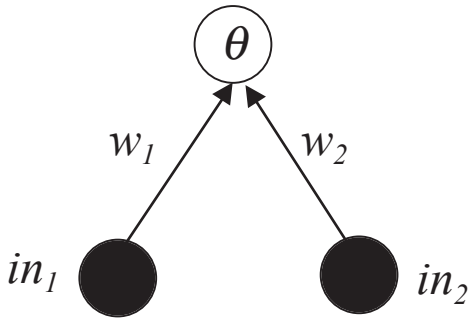
How do we know if a simple Perceptron is powerful enough to solve a given problem? If it can't even do XOR, why should we expect it to be able to deal with our aeroplane classification example, or real world tasks that will be even more complex than that?



In this lecture, we shall look at the limitations of Perceptrons, and how we can find their connection weights without having to compute and solve large numbers of inequalities.

Decision Boundaries in Two Dimensions

For simple logic gate problems, it is easy to visualise what the neural network is doing. It is forming ***decision boundaries*** between classes. Remember, the network output is:



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

The decision boundary (between $out = 0$ and $out = 1$) is at

$$w_1 in_1 + w_2 in_2 - \theta = 0$$

i.e. along the straight line:

$$in_2 = \left(\frac{-w_1}{w_2}\right) in_1 + \left(\frac{\theta}{w_2}\right)$$

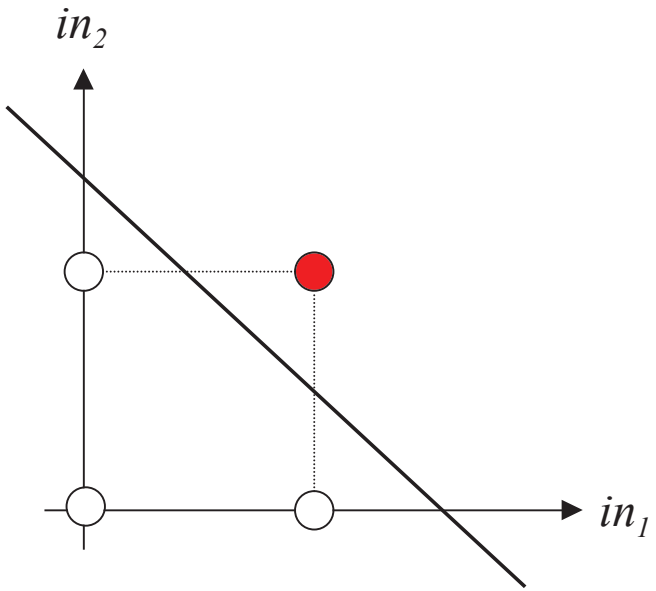
So, in two dimensions the decision boundaries are always straight lines.

Decision Boundaries for AND and OR

We can easily plot the decision boundaries we found by inspection last lecture:

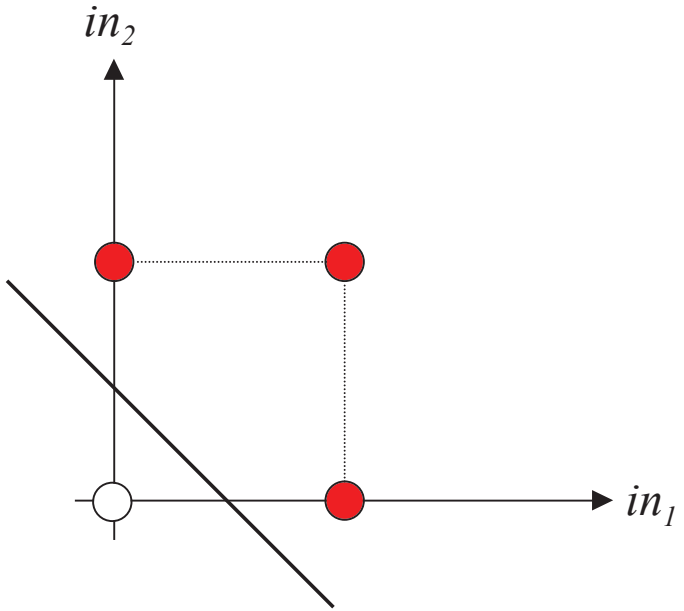
AND

$w_1 = 1,\ w_2 = 1,\ \theta = 1.5$



OR

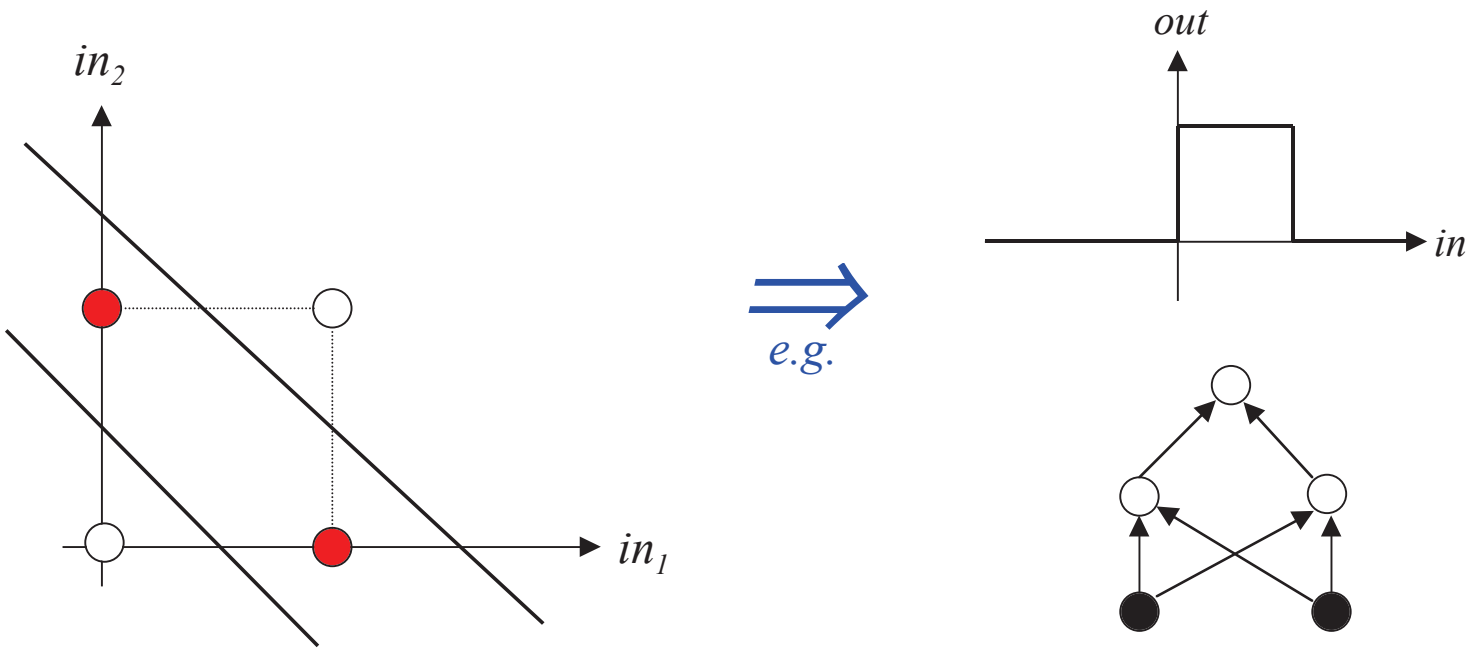
$w_1 = 1,\ w_2 = 1,\ \theta = 0.5$



The extent to which we can change the weights and thresholds without changing the output decisions is now clear.

Decision Boundaries for XOR

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



There are two obvious remedies: 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.

Decision Hyperplanes and Linear Separability

If we have two inputs, then the weights define a decision boundary that is a one dimensional straight line in the two dimensional ***input space*** of possible input values.

If we have *n* inputs, the weights define a decision boundary that is an *n-1* dimensional ***hyperplane*** in the *n* dimensional input space:

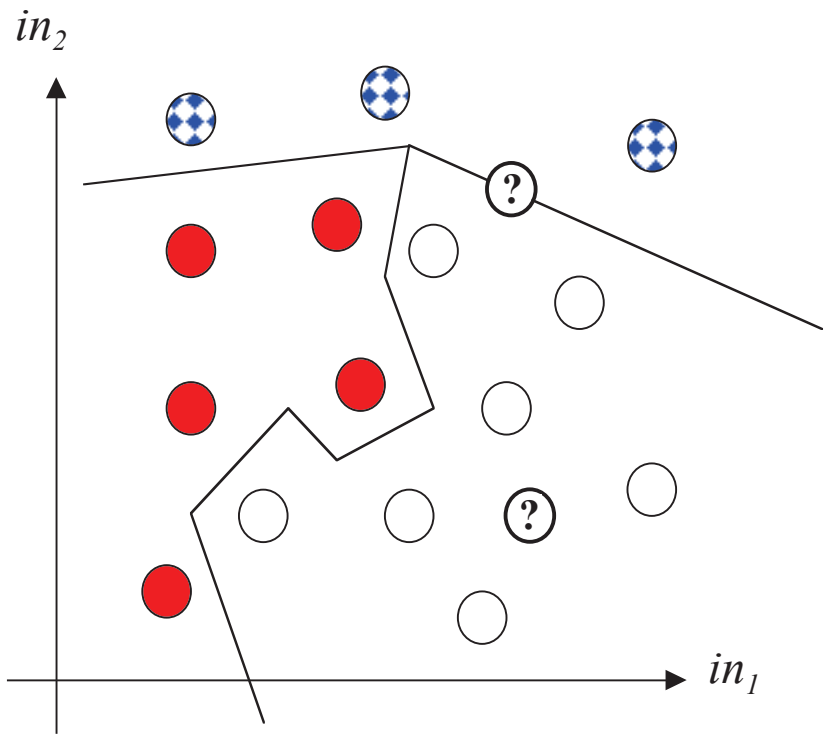
$$w_1 in_1 + w_2 in_2 + \ldots + w_n in_n - \theta = 0$$

This hyperplane is clearly still linear (i.e. straight/flat) and can still only divide the space into two regions. We still need more complex transfer functions, or more complex networks, to deal with XOR type problems.

Problems with input patterns which can be classified using a single hyperplane are said to be ***linearly separable***. Problems (such as XOR) which cannot be classified in this way are said to be ***non-linearly separable***.

General Decision Boundaries

Generally, we will want to deal with input patterns that are not binary, and expect our neural networks to form complex decision boundaries, e.g.



We may also wish to classify inputs into many classes (such as the three shown here).

Learning and Generalization

A network will also produce outputs for input patterns that it was not originally set up to classify (shown with question marks), though those classifications may be incorrect.

There are two important aspects of the network's operation to consider:

Learning The network must learn decision surfaces from a set of ***training patterns*** so that these training patterns are classified correctly.

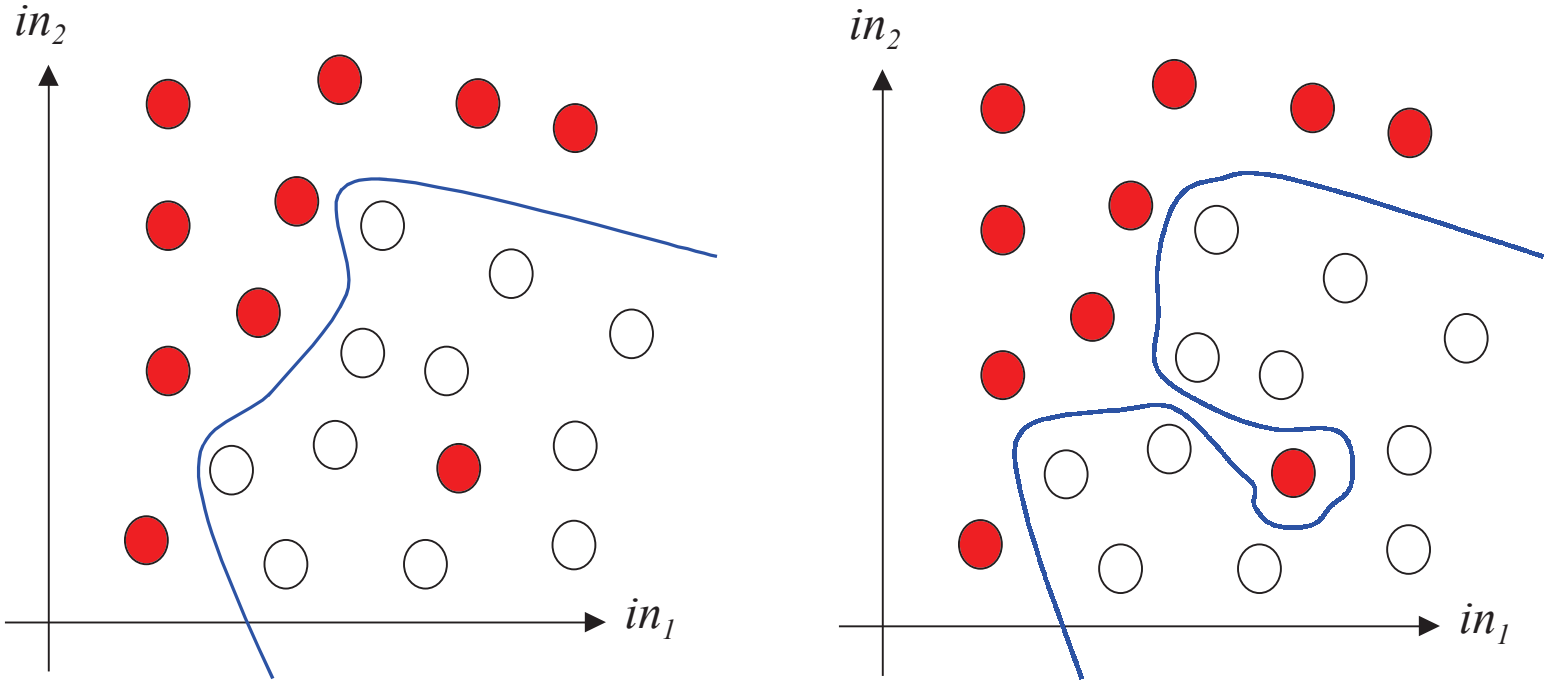
Generalization After training, the network must also be able to generalize, i.e. correctly classify ***test patterns*** it has never seen before.

Usually we want our neural networks to learn well, and also to generalize well.

Sometimes, the training data may contain errors (e.g. noise in the experimental determination of the input values, or incorrect classifications). In this case, learning the training data perfectly may make the generalization worse. There is an important ***trade-off*** between learning and generalization that arises quite generally.

Generalization in Classification

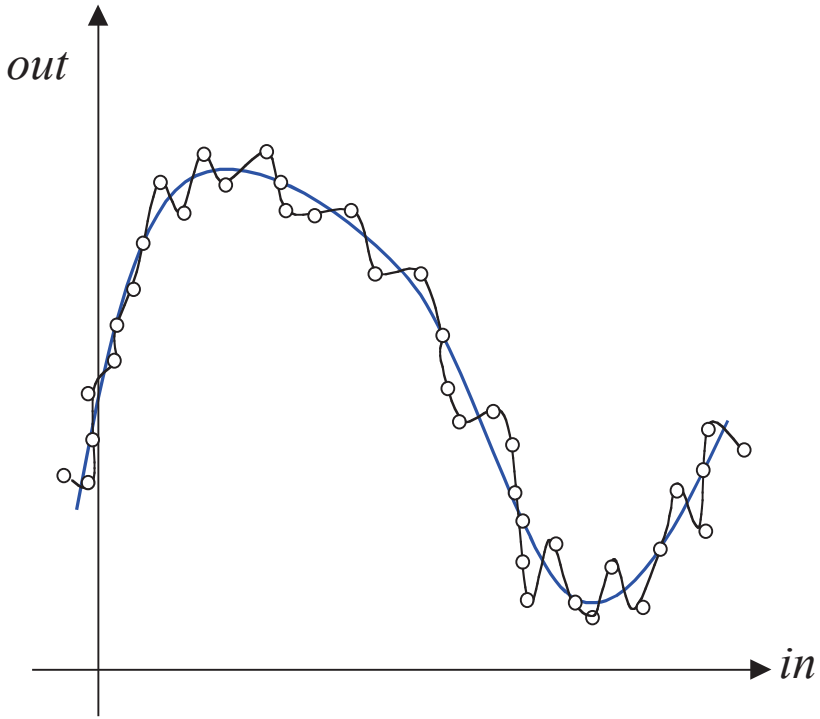
Suppose the task of our network is to learn a classification decision boundary:



Our aim is for the network to generalize to classify new inputs appropriately. If we know that the training data contains noise, we don't necessarily want the training data to be classified totally accurately, as that is likely to reduce the generalization ability.

Generalization in Function Approximation

Suppose we wish to recover a function for which we only have noisy data samples:



We can expect the neural network output to give a better representation of the underlying function if its output curve does not pass through all the data points. Again, allowing a larger error on the training data is likely to lead to better generalization.

Training a Neural Network

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

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

In all but the simplest cases, however, direct computation of the weights is intractable.

Instead, we usually start off with ***random initial weights*** and adjust them in small steps until the required outputs are produced.

We shall now look at a brute force derivation of such an ***iterative learning algorithm*** for simple Perceptrons. Then, in later lectures, we shall see how more powerful and general techniques can easily lead to learning algorithms which will work for neural networks of any specification we could possibly dream up.

Perceptron Learning

For simple Perceptrons performing classification, we have seen that the decision boundaries are hyperplanes, and we can think of ***learning*** as the process of shifting around the hyperplanes until each training pattern is classified correctly.

Somehow, we need to formalise that process of "shifting around" into a systematic algorithm that can easily be implemented on a computer.

The "shifting around" can conveniently be split up into a number of small steps.

If the network weights at time t are $w_{ij}(t)$, then the shifting process corresponds to moving them by an amount $\Delta w_{ij}(t)$ so that at time $t+1$ we have weights

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

It is convenient to treat the thresholds as weights, as discussed previously, so we don't need separate equations for them.

Formulating the Weight Changes

Suppose the target output of unit j is $targ_j$ and the actual output is $out_j = \text{sgn}(\sum in_i w_{ij})$, where in_i are the activations of the previous layer of neurons (e.g. the network inputs). Then we can just go through all the possibilities to work out an appropriate set of small weight changes, and put them into a common form:

If $out_j = targ_j$ do nothing Note $targ_j - out_j = 0$

so $w_{ij} \rightarrow w_{ij}$

If $out_j = 1$ and $targ_j = 0$ Note $targ_j - out_j = -1$

then $\sum in_i w_{ij}$ is too large

first when $in_i = 1$ decrease w_{ij}

so $w_{ij} \rightarrow w_{ij} - \eta = w_{ij} - \eta\, in_i$

and when $in_i = 0$ w_{ij} doesn't matter

so $w_{ij} \rightarrow w_{ij} - 0 = w_{ij} - \eta\, in_i$

so $w_{ij} \rightarrow w_{ij} - \eta\, in_i$

If $out_j = 0$ and $targ_j = 1$ Note $targ_j - out_j = 1$

then $\sum in_i w_{ij}$ is too small

first when $in_i = 1$ increase w_{ij}

so $w_{ij} \rightarrow w_{ij} + \eta = w_{ij} + \eta\, in_i$

and when $in_i = 0$ w_{ij} doesn't matter

so $w_{ij} \rightarrow w_{ij} - 0 = w_{ij} + \eta\, in_i$

so $w_{ij} \rightarrow w_{ij} + \eta\, in_i$

It has become clear that each case can be written in the form:

$$w_{ij} \rightarrow w_{ij} + \eta\, (targ_j - out_j)\, in_i$$

$$\Delta w_{ij} = \eta\, (targ_j - out_j)\, in_i$$

This weight update equation is called the ***Perceptron Learning Rule***. The positive parameter η is called the ***learning rate*** or ***step size*** – it determines how smoothly we shift the decision boundaries.

Convergence of Perceptron Learning

The weight changes Δw_{ij} need to be applied repeatedly – for each weight w_{ij} in the network, and for each training pattern in the training set. One pass through all the weights for the whole training set is called one ***epoch*** of training.

Eventually, usually after many epochs, when all the network outputs match the targets for all the training patterns, all the Δw_{ij} will be zero and the process of training will cease. We then say that the training process has ***converged*** to a solution.

It can be shown that if there does exist a possible set of weights for a Perceptron which solves the given problem correctly, then the Perceptron Learning Rule will find them in a finite number of iterations

Moreover, it can be shown that if a problem is linearly separable, then the Perceptron Learning Rule will find a set of weights in a finite number of iterations that solves the problem correctly.

Hebbian Learning and Gradient Descent Learning

1. Hebbian Learning
2. Learning by Error Minimisation
3. Computing Gradients and Derivatives
4. Gradient Descent Learning
5. Deriving the Delta Rule
6. Delta Rule vs. Perceptron Learning Rule

Hebbian Learning

In 1949 neuropsychologist Donald Hebb postulated how biological neurons learn:

> "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place on one or both cells such that A's efficiency as one of the cells firing B, is increased."

In other words:

1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e. synchronously), then the strength of that synapse is selectively increased.

This rule is often supplemented by:

2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.

so that chance coincidences do not build up connection strengths.

Hebbian versus Perceptron Learning

In the notation used for Perceptrons, the ***Hebbian learning*** weight update rule is:

$$\Delta w_{ij} = \eta \, . \, out_j \, . \, in_i$$

There is strong physiological evidence that this type of learning does take place in the region of the brain known as the *hippocampus*.

Recall that the ***Perceptron learning*** weight update rule we derived was:

$$\Delta w_{ij} = \eta . \, (targ_j - out_j) \, . \, in_i$$

There is some similarity, but it is clear that Hebbian learning is not going to get our Perceptron to learn a set of training data.

The are variations of Hebbian learning that do provide powerful learning techniques for biologically plausible networks, such as ***Contrastive Hebbian Learning***, but we shall adopt another approach for formulating learning algorithms for our networks.

Learning by Error Minimisation

The Perceptron Learning Rule is an algorithm for adjusting the network weights w_{ij} to minimise the difference between the actual outputs out_j and the desired outputs $targ_j$.

We can define an ***Error Function*** to quantify this difference:

$$E(w_{ij}) = \tfrac{1}{2}\sum_p \sum_j \left(targ_j - out_j\right)^2$$

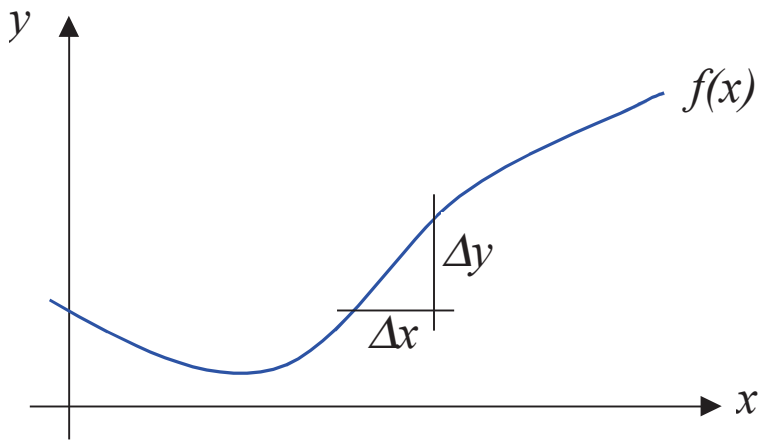
For obvious reasons this is known as the ***Sum Squared Error*** function. It is the total squared error summed over all output units j and all training patterns p.

The aim of ***learning*** is to minimise this error by adjusting the weights w_{ij}. Typically we make a series of small adjustments to the weights $w_{ij} \rightarrow w_{ij} + \Delta w_{ij}$ until the error $E(w_{ij})$ is 'small enough'.

A systematic procedure for doing this requires the knowledge of how the error $E(w_{ij})$ varies as we change the weights w_{ij}, i.e. the ***gradient*** of E with respect to w_{ij}.

Computing Gradients and Derivatives

There is a whole branch of mathematics concerned with computing gradients – it is known as ***Differential Calculus***. The basic idea is simple. Consider a function $y = f(x)$



The gradient, or rate of change, of $f(x)$ at a particular value of x, as we change x can be approximated by $\Delta y/\Delta x$. Or we can write it exactly as

$$\frac{\partial f(x)}{\partial x} = \operatorname*{Lim}_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \operatorname*{Lim}_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

which is known as the ***partial derivative*** of $f(x)$ with respect to x.

Examples of Computing Derivatives Analytically

Some simple examples should make this clearer:

$$f(x) = a.x + b \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{[a.(x+\Delta x)+b]-[a.x+b]}{\Delta x} = a$$

$$f(x) = a.x^2 \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{\left[a.(x+\Delta x)^2\right]-\left[a.x^2\right]}{\Delta x} = 2ax$$

$$f(x) = g(x) + h(x) \Rightarrow \frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{\left(g(x+\Delta x)+h(x+\Delta x)\right)-\left(g(x)+h(x)\right)}{\Delta x} = \frac{\partial g(x)}{\partial x} + \frac{\partial h(x)}{\partial x}$$

Other derivatives can be computed in the same way. Some useful ones are:

$$f(x) = a.x^n \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = nax^{n-1}$$

$$f(x) = \log_e(x) \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = \frac{1}{x}$$

$$f(x) = e^{ax} \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = ae^{ax}$$

$$f(x) = \sin(x) \quad \Rightarrow \quad \frac{\partial f(x)}{\partial x} = \cos(x)$$

Gradient Descent Minimisation

Suppose we have a function *f(x)* and we want to change the value of *x* to minimise *f(x)*. What we need to do depends on the gradient of *f(x)*. There are three cases to consider:

If $\frac{\partial f}{\partial x} > 0$ then *f(x)* increases as *x* increases so we should decrease *x*

If $\frac{\partial f}{\partial x} < 0$ then *f(x)* decreases as *x* increases so we should increase *x*

If $\frac{\partial f}{\partial x} = 0$ then *f(x)* is at a maximum or minimum so we should not change *x*

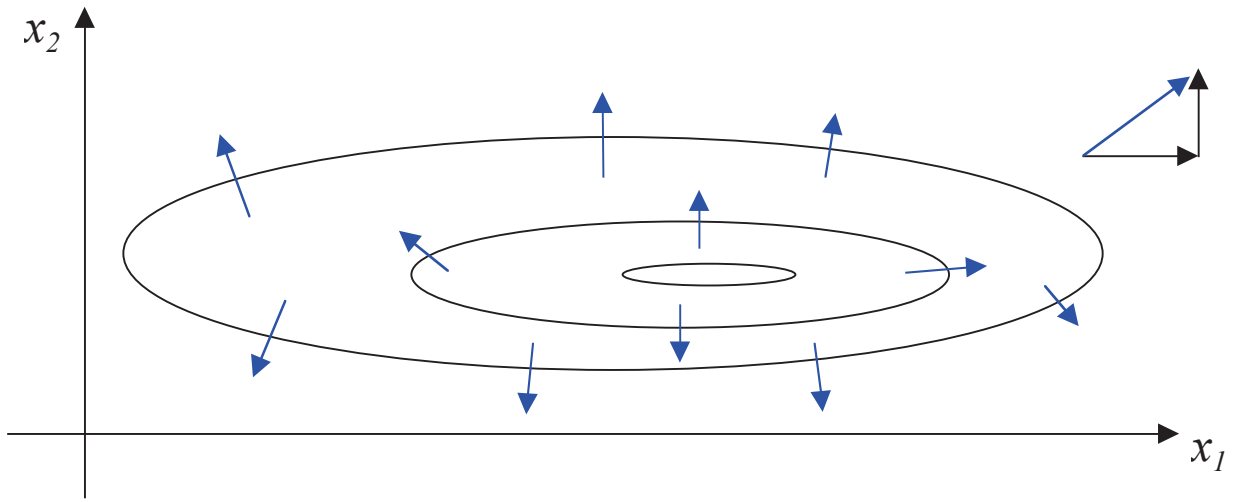
In summary, we can decrease *f(x)* by changing *x* by the amount:

$$\Delta x = x_{new} - x_{old} = -\eta \frac{\partial f}{\partial x}$$

where η is a small positive constant specifying how much we change *x* by, and the derivative $\partial f/\partial x$ tells us which direction to go in. If we repeatedly use this equation, *f(x)* will (assuming η is sufficiently small) keep descending towards its minimum, and hence this procedure is known as ***gradient descent minimisation***.

Gradients in More Than One Dimension

Is it obvious that we need the gradient/derivative itself in the weight update equation, rather than just the sign of the gradient? Consider the two dimensional function shown as a ***contour plot*** with its minimum inside the smallest ellipse:



A few representative gradient vectors are shown. By definition, they will always be perpendicular to the contours, and the closer the contours, the larger the vectors. It is now clear that we need to take the relative magnitudes of the x_1 and x_2 components of the gradient vectors into account if we are to head towards the minimum efficiently.

Gradient Descent Error Minimisation

Remember that we want to train our neural networks by adjusting their weights w_{ij} in order to minimise the error function:

$$E(w_{ij}) = \tfrac{1}{2}\sum_p \sum_j \left(targ_j - out_j\right)^2$$

We now see it makes sense to do this by a series of gradient descent weight updates:

$$\Delta w_{kl} = -\eta \frac{\partial E(w_{ij})}{\partial w_{kl}}$$

If the transfer function for the output neurons is *f(x),* and the activations of the previous layer of neurons are in_i, then the outputs are $out_j = f(\sum_i in_i w_{ij})$, and

$$\Delta w_{kl} = -\eta \frac{\partial}{\partial w_{kl}} \left[\tfrac{1}{2}\sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij}) \right)^2 \right]$$

Dealing with equations like this is easy if we use the chain rules for derivatives.

Chain Rules for Computing Derivatives

Computing complex derivatives can be done in stages. First, suppose $f(x) = g(x).h(x)$

$$\frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{g(x+\Delta x).h(x+\Delta x) - g(x).h(x)}{\Delta x} = \lim_{\Delta x \to 0} \frac{\left(g(x) + \frac{\partial g(x)}{\partial x}\Delta x\right).\left(h(x) + \frac{\partial h(x)}{\partial x}\Delta x\right) - g(x).h(x)}{\Delta x}$$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial g(x)}{\partial x} h(x) + g(x) \frac{\partial h(x)}{\partial x}$$

We can similarly deal with nested functions. Suppose $f(x) = g(h(x))$

$$\frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{g(h(x+\Delta x)) - g(h(x))}{\Delta x} = \lim_{\Delta x \to 0} \frac{g(h(x) + \frac{\partial h(x)}{\partial x}\Delta x) - g(h(x))}{\Delta x}$$

$$\frac{\partial f(x)}{\partial x} = \lim_{\Delta x \to 0} \frac{g(h(x)) + \frac{\partial g(x)}{\partial h(x)}\Delta h(x) - g(h(x))}{\Delta x} = \lim_{\Delta x \to 0} \frac{g(h(x)) + \frac{\partial g(x)}{\partial h(x)}\left(\frac{\partial h(x)}{\partial x}\Delta x\right) - g(h(x))}{\Delta x}$$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial g(h(x))}{\partial h(x)} \cdot \frac{\partial h(x)}{\partial x}$$

Using the Chain Rule on our Weight Update Equation

The algebra gets rather messy, but after repeated application of the chain rule, and some tidying up, we end up with a very simple weight update equation:

$$\Delta w_{kl} = -\eta \frac{\partial}{\partial w_{kl}} \left[\tfrac{1}{2} \sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij}) \right)^2 \right]$$

$$\Delta w_{kl} = -\eta \left[\tfrac{1}{2} \sum_p \sum_j \frac{\partial}{\partial w_{kl}} \left(targ_j - f(\sum_i in_i w_{ij}) \right)^2 \right]$$

$$\Delta w_{kl} = -\eta \left[\tfrac{1}{2} \sum_p \sum_j 2 \left(targ_j - f(\sum_i in_i w_{ij}) \right) \left(-\frac{\partial}{\partial w_{kl}} f(\sum_m in_m w_{mj}) \right) \right]$$

$$\Delta w_{kl} = \eta \left[\sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij}) \right) \left(f'(\sum_n in_n w_{nj}) \frac{\partial}{\partial w_{kl}} (\sum_m in_m w_{mj}) \right) \right]$$

$$\Delta w_{kl} = \eta \left[\sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij}) \right) \left(f'(\sum_n in_n w_{nj})(\sum_m in_m \frac{\partial w_{mj}}{\partial w_{kl}}) \right) \right]$$

$$\Delta w_{kl} = \eta \left[\sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij}) \right) \left(f'(\sum_n in_n w_{ij})(\sum_m in_m \delta_{mk} \delta_{jl}) \right) \right]$$

$$\Delta w_{kl} = \eta \left[\sum_p \sum_j \left(targ_j - f(\sum_i in_i w_{ij}) \right) \left(f'(\sum_n in_n w_{nj})(in_k \delta_{jl}) \right) \right]$$

$$\Delta w_{kl} = \eta \left[\sum_p \left(targ_l - f(\sum_i in_i w_{il}) \right) \left(f'(\sum_n in_n w_{nl})(in_k) \right) \right]$$

$$\Delta w_{kl} = \eta \sum_p (targ_l - out_l).f'(\sum_n in_n w_{nl}).in_k$$

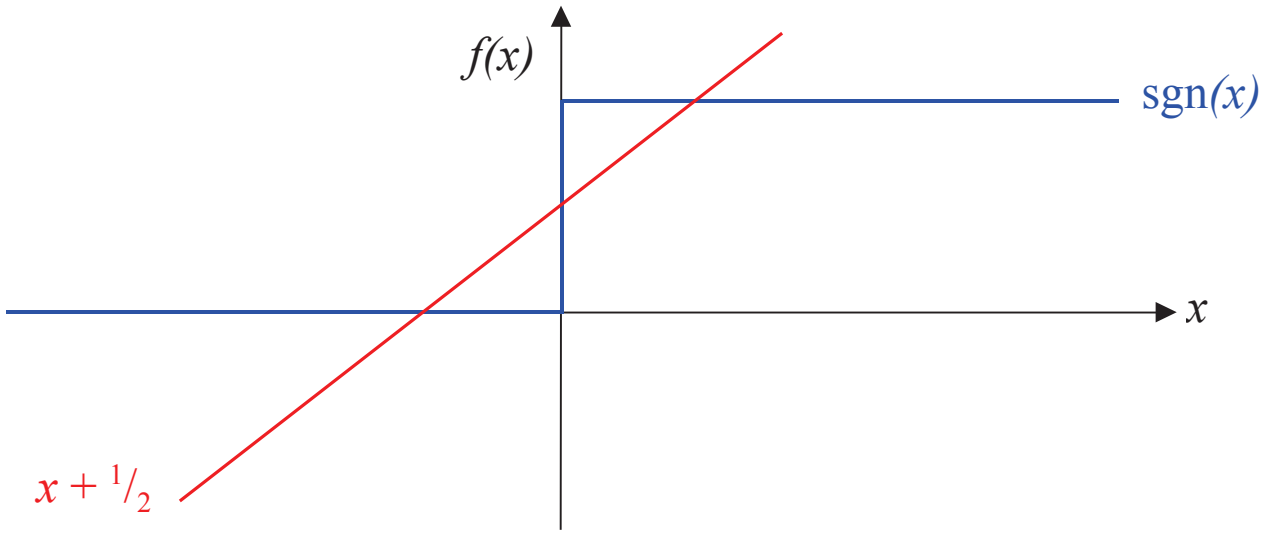
The *prime notation* is defined such that f' is the derivative of f. We have also used the *Kronecker Delta* symbol δ_{ij} defined such that $\delta_{ij} = 1$ when $i = j$ and $\delta_{ij} = 0$ when $i \neq j$.

The Delta Rule

We now have the basic gradient descent learning algorithm for single layer networks:

$$\Delta w_{kl} = \eta \sum_{p} (targ_l - out_l).f'(\sum_{i} in_i w_{il}).in_k$$

Notice that it still involves the derivative of the transfer function $f(x)$. This is clearly problematic for the simple Perceptron that uses the step function sgn(x) as its threshold function, because this has zero derivative everywhere except at $x = 0$ where it is infinite.



Fortunately, there is a ***clever trick*** we can use that will be apparent from the above graph. Suppose we had the transfer $f(x) = x + {}^1/_2$, then when the target is 1 the network will learn $x = {}^1/_2$, and when the target is 0 it will learn $x = -{}^1/_2$. It is clear that these values will also result in the right values of sgn(x), and so the Perceptron will work properly.

In other words, we can use the gradient descent learning algorithm with $f(x) = x + {}^1/_2$ to get our Perceptron to learn the right weights. In this case $f'(x) = 1$ and so the weight update equation becomes:

$$\Delta w_{kl} = \eta \sum_p \left(targ_l - out_l\right).in_k$$

This is known as the ***Delta Rule*** because it depends on the discrepancy

$$\delta_l = targ_l - out_l$$

NOTE: We need to be very careful when using tricks like this. We are using one output function to learn the weights, i.e. $f(x) = x + {}^1/_2$, and a totally different one to produce the required binary outputs of the perceptron, i.e. $f(x) = \text{sgn}(x)$. It is easy to get confused!

Delta Rule vs. Perceptron Learning Rule

We can see that the Delta Rule and the Perceptron Learning Rule for training Single Layer Perceptrons have exactly the same weight update equation:

$$\Delta w_{kl} = \eta \sum_p \left(targ_l - f\left(\sum_i in_i w_{il} \right) \right) . in_k$$

However, there are significant underlying differences. The Perceptron Learning Rule uses the actual activation function $f(x) = \text{sgn}(x)$, whereas the Delta Rule uses the linear function $f(x) = x + {}^1/_2$. The two algorithms were also obtained from very different theoretical starting points. The Perceptron Learning Rule was derived from a consideration of how we should shift around the decision hyper-planes, while the Delta Rule emerged from a gradient descent minimisation of the Sum Squared Error.

The Perceptron Learning Rule will converge to zero error and no weight changes in a finite number of steps if the problem is linearly separable, but otherwise the weights will keep oscillating. On the other hand, the Delta Rule will (for sufficiently small η) always converge to a set of weights for which the error is a minimum, though the convergence to the precise values of $x = \pm {}^1/_2$ will generally proceed at an ever decreasing rate.

The Generalized Delta Rule and Practical Considerations

1. Training a Single Layer Feed-forward Network
2. Deriving the Generalized Delta Rule
3. Practical Considerations for Gradient Descent Learning
   - (*1) Pre-processing of the Training Data*
   - (*2) Choosing the Initial Weights*
   - (*3) Choosing the Learning Rate*
   - (*4) On-line vs. Batch Training*
   - (*5) Choosing the Transfer Function*
   - (*6) Avoiding Flat Spots in the Error Function*
   - (*7) Avoiding Local Minima*
   - (*8) Knowing When to Stop the Training*

Gradient Descent Learning

It is worth summarising all the factors involved in Gradient Descent Learning:

1. The purpose of neural network learning or training is to minimise the output errors on a particular set of training data by adjusting the network weights w_{ij}.
2. We define an Error Function $E(w_{ij})$ that "measures" how far the current network is from the desired (correctly trained) one.
3. Partial derivatives of the error function $\partial E(w_{ij})/\partial w_{ij}$ tell us which direction we need to move in weight space to reduce the error.
4. The learning rate η specifies the step sizes we take in weight space for each iteration of the weight update equation.
5. We keep stepping through weight space until the errors are 'small enough'.
6. If we choose neuron activation functions with derivatives that take on particularly simple forms, we can make the weight update computations very efficient.

These factors lead to powerful learning algorithms for training our neural networks:

Training a Single Layer Feed-forward Network

Now we understand how gradient descent weight update rules can lead to minimisation of a neural network's output errors, it is straightforward to train any network:

1. Take the set of training patterns you wish the network to learn

   $\{in_i^p, out_j^p : i = 1 \ldots ninputs, j = 1 \ldots noutputs, p = 1 \ldots npatterns\}$

2. Set up your network with *ninputs* input units fully connected to *noutputs* output units via connections with weights w_{ij}

3. Generate random initial weights, e.g. from the range [–*smwt*, +*smwt*]

4. Select an appropriate error function $E(w_{ij})$ and learning rate η

5. Apply the weight update $\Delta w_{ij} = -\eta \, \partial E(w_{ij})/\partial w_{ij}$ to each weight w_{ij} for each training pattern *p*. One set of updates of all the weights for all the training patterns is called one ***epoch*** of training.

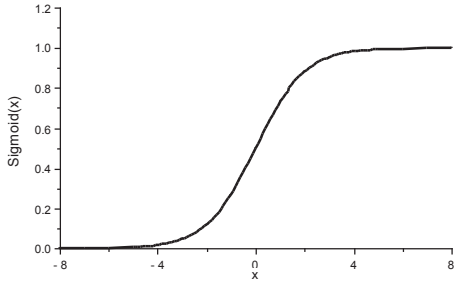
6. Repeat step 5 until the network error function is 'small enough'.

You thus end up with a trained neural network. But step 5 can still be difficult…

The Derivative of a Sigmoid

We noted earlier that the Sigmoid is a smooth (i.e. differentiable) threshold function:

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

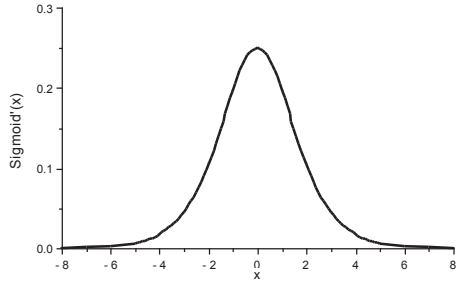


We can use the chain rule by putting $f(x) = g(h(x))$ with $g(h) = h^{-1}$ and $h(x) = 1 + e^{-x}$ so

$$\frac{\partial g(h)}{\partial h} = -\frac{1}{h^2} \quad \text{and} \quad \frac{\partial h(x)}{\partial x} = -e^{-x}$$

$$\frac{\partial f(x)}{\partial x} = -\frac{1}{(1+e^{-x})^2} \cdot (-e^{-x}) = \left(\frac{1}{1+e^{-x}}\right)\left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)$$

$$f'(x) = \frac{\partial f(x)}{\partial x} = f(x).(1-f(x))$$



This simple relation will make our equations much easier and save a lot of computing time!

The Generalised Delta Rule

We can avoid using tricks for deriving gradient descent learning rules, by making sure we use a differentiable activation function such as the Sigmoid. This is also more like the threshold function used in real brains, and has several other nice mathematical properties.

If we use the Sigmoid activation function, a single layer network has outputs given by

$$out_l = \text{Sigmoid}(\sum_i in_i w_{il})$$

and, due to the properties of the Sigmoid derivative, the general weight update equation

$$\Delta w_{kl} = \eta \sum_p (targ_l - out_l).f'(\sum_i in_i w_{il}).in_k$$

simplifies so that it only contains neuron activations and no derivatives:

$$\Delta w_{kl} = \eta \sum_p (targ_l - out_l).out_l.(1 - out_l).in_k$$

This is known as the ***Generalized Delta Rule*** for training sigmoidal networks.

Practical Considerations for Gradient Descent Learning

From the above discussion, it is clear that there remain a number of important questions about training single layer neural networks that still need to be resolved:

1. Do we need to pre-process the training data? If so, how?
2. How do we choose the initial weights from which we start the training?
3. How do we choose an appropriate learning rate η?
4. Should we change the weights after each training pattern, or after the whole set?
5. Are some activation/transfer functions better than others?
6. How can we avoid flat spots in the error function?
7. How can we avoid local minima in the error function?
8. How do we know when we should stop the training?

We shall now consider each of these issues in turn.

Pre-processing the Training Data

In principle, we can just use any raw input-output data to train our networks. However, in practice, it often helps the network to learn appropriately if we carry out some pre-processing of the training data before feeding it to the network.

We should make sure that the training data is representative – it should not contain too many examples of one type at the expense of another. On the other hand, if one class of pattern is easy to learn, having large numbers of patterns from that class in the training set will only slow down the over-all learning process.

If the training data is continuous, rather than binary, it is generally a good idea to re-scale the input values. Simply shifting the zero of the scale so that the mean value of each input is near zero, and normalising so that the standard deviation of the values for each input are roughly the same, can make a big difference. It will require more work, but de-correlating the inputs before normalising is often also worthwhile.

If we are using on-line training rather than batch training, we should usually make sure we shuffle the order of the training data each epoch.

Choosing the Initial Weights

The gradient descent learning algorithm treats all the weights in the same way, so if we start them all off with the same values, all the hidden units will end up doing the same thing and the network will never learn properly.

For that reason, we generally start off all the weights with small random values. Usually we take them from a flat distribution around zero [–*smwt*, +*smwt*], or from a Gaussian distribution around zero with standard deviation *smwt*.

Choosing a good value of *smwt* can be difficult. Generally, it is a good idea to make it as large as you can without saturating any of the sigmoids.

We usually hope that the final network performance will be independent of the choice of initial weights, but we need to check this by training the network from a number of different random initial weight sets.

In networks with hidden layers, there is no real significance to the order in which we label the hidden neurons, so we can expect very different final sets of weights to emerge from the learning process for different choices of random initial weights.

Choosing the Learning Rate

Choosing a good value for the learning rate η is constrained by two opposing facts:

1. If η is too small, it will take too long to get anywhere near the minimum of the error function.
2. If η is too large, the weight updates will over-shoot the error minimum and the weights will oscillate, or even diverge.

Unfortunately, the optimal value is very problem and network dependent, so one cannot formulate reliable general prescriptions. Generally, one should try a range of different values (e.g. $\eta = 0.1, 0.01, 1.0, 0.0001$) and use the results as a guide.

There is no necessity to keep the learning rate fixed throughout the learning process. Typical variable learning rates that prove advantageous are:

$$\eta(t) = \frac{\eta(1)}{t} \qquad\qquad \eta(t) = \frac{\eta(0)}{1 + t/\tau}$$

Similar age dependent learning rates are found to exist in human children.

Batch Training *vs.* On-line Training

The gradient descent learning algorithm contains a sum over all training patterns p

$$\Delta w_{kl} = \eta \sum_{p} (targ_l - out_l).f'(\sum_{i} in_i w_{il}).in_k$$

When we add up the weight changes for all the training patterns like this, and apply them in one go, it is called ***Batch Training***.

A natural alternative is to update all the weights immediately after processing each training pattern. This is called ***On-line Training*** (or ***Sequential Training***).

On-line learning does not perform true gradient descent, and the individual weight changes can be rather erratic. Normally a much lower learning rate η will be necessary than for batch learning. However, because each weight now has *npatterns* updates per epoch, rather than just one, overall the learning is often much quicker. This is particularly true if there is a lot of redundancy in the training data, i.e. many training patterns containing similar information.

Choosing the Transfer Function

We have already seen that having a differentiable transfer/activation function is important for the gradient descent algorithm to work. We have also seen that, in terms of computational effeciency, the standard sigmoid (i.e. logistic function) is a particularly convenient replacement for the step function of the Simple Perceptron.

The logistic function ranges from 0 to 1. There is some evidence that an anti-symmetric transfer function, i.e. one that satisfies $f(-x) = -f(x)$, enables the gradient descent algorithm to learn faster. To do this we must use targets of +1 and –1 rather than 0 and 1. A convenient alternative to the logistic function is then the hyperbolic tangent

$$f(x) = \tanh(x) \qquad f(-x) = -f(x) \qquad f'(x) = 1 - f(x)^2$$

which, like the logistic function, has a particularly simple derivative.

When the outputs are required to be non-binary, i.e. continuous real values, having sigmoidal transfer functions no longer makes sense. In these cases, a simple linear transfer function $f(x) = x$ is appropriate.

Classification Outputs as Probabilities

Another powerful feature of neural network classification systems is that non-binary outputs can be interpreted as the probabilities of the corresponding classifications. For example, an output of 0.9 on a unit corresponding to a particular class would indicate a 90% chance that the input data represents a member of that class.

The mathematics is rather complex, but one can show that for two classes represented as activations of 0 and 1 on a single output unit, the activation function that allows us to do this is none other than our ***Sigmoid*** activation function.

If we have more than two classes, and use one output unit for each class, we should employ a generalization of the Sigmoid known as the ***Softmax*** activation function:

$$out_j = e^{-\sum_i in_i w_{ij}} \Big/ \sum_k e^{-\sum_n in_n w_{nk}}$$

In either case, we use a ***Cross Entropy*** error measure rather than Sum Squared Error.

Flat Spots in the Error Function

The gradient descent weight changes depend on the gradient of the error function. Consequently, if the error function has flat spots, the learning algorithm can take a long time to pass through them.

A particular problem with the sigmoidal transfer functions is that the derivative tends to zero as it saturates (i.e. gets towards 0 or 1). This means that if the outputs are totally wrong (i.e. 0 instead of 1, or 1 instead of 0), the weight updates are very small and the learning algorithm cannot easily correct itself. There are two simple solutions:

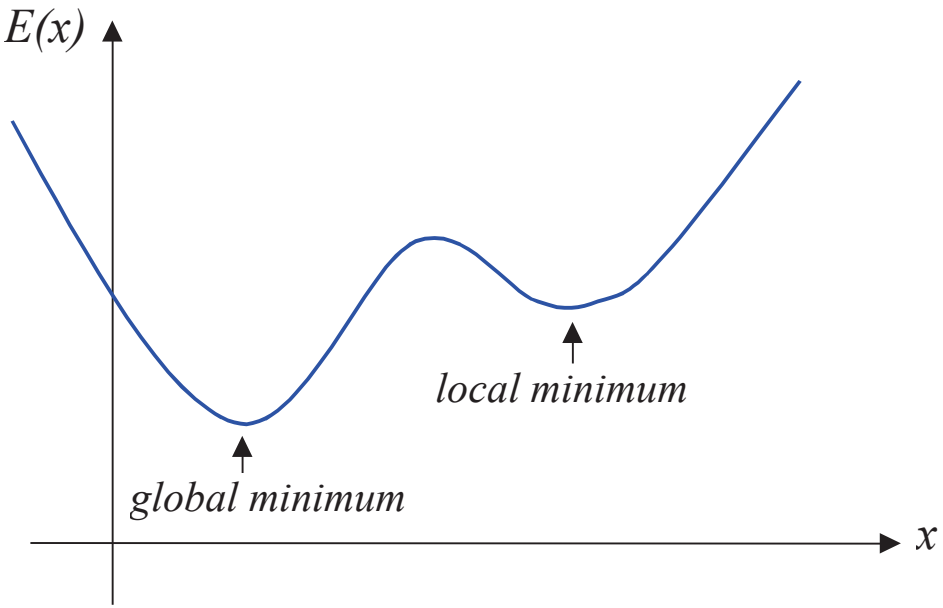
Target Off-sets Use targets of 0.1 and 0.9 (say) instead of 0 and 1. The sigmoids will no longer saturate and the learning will no longer get stuck.

Sigmoid Prime Off-set Add a small off-set (of 0.1 say) to the sigmoid prime (i.e. the sigmoid derivative) so that it is no longer zero when the sigmoids saturate.

We can now see why we should keep the initial network weights small enough that the sigmoids are not saturated before training. Off-setting the targets also has the effect of stopping the network weights growing too large.

Local Minima

Error functions can quite easily have more than one minimum:



If we start off in the vicinity of the local minimum, we may end up at the local minimum rather than the global minimum. Starting with a range of different initial weight sets increases our chances of finding the global minimum. Any variation from true gradient descent will also increase our chances of stepping into the deeper valley.

When to Stop Training

The Sigmoid(x) function only takes on its extreme values of 0 and 1 at $x = \pm\infty$. In effect, this means that the network can only achieve its binary targets when at least some of its weights reach $\pm\infty$. So, given finite gradient descent step sizes, our networks will never reach their binary targets.

Even if we off-set the targets (to 0.1 and 0.9 say) we will generally require an infinite number of increasingly small gradient descent steps to achieve those targets.

Clearly, if the training algorithm can never actually reach the minimum, we have to stop the training process when it is 'near enough'. What constitutes 'near enough' depends on the problem. If we have binary targets, it might be enough that all outputs are within 0.1 (say) of their targets. Or, it might be easier to stop the training when the sum squared error function becomes less than a particular small value (0.2 say).

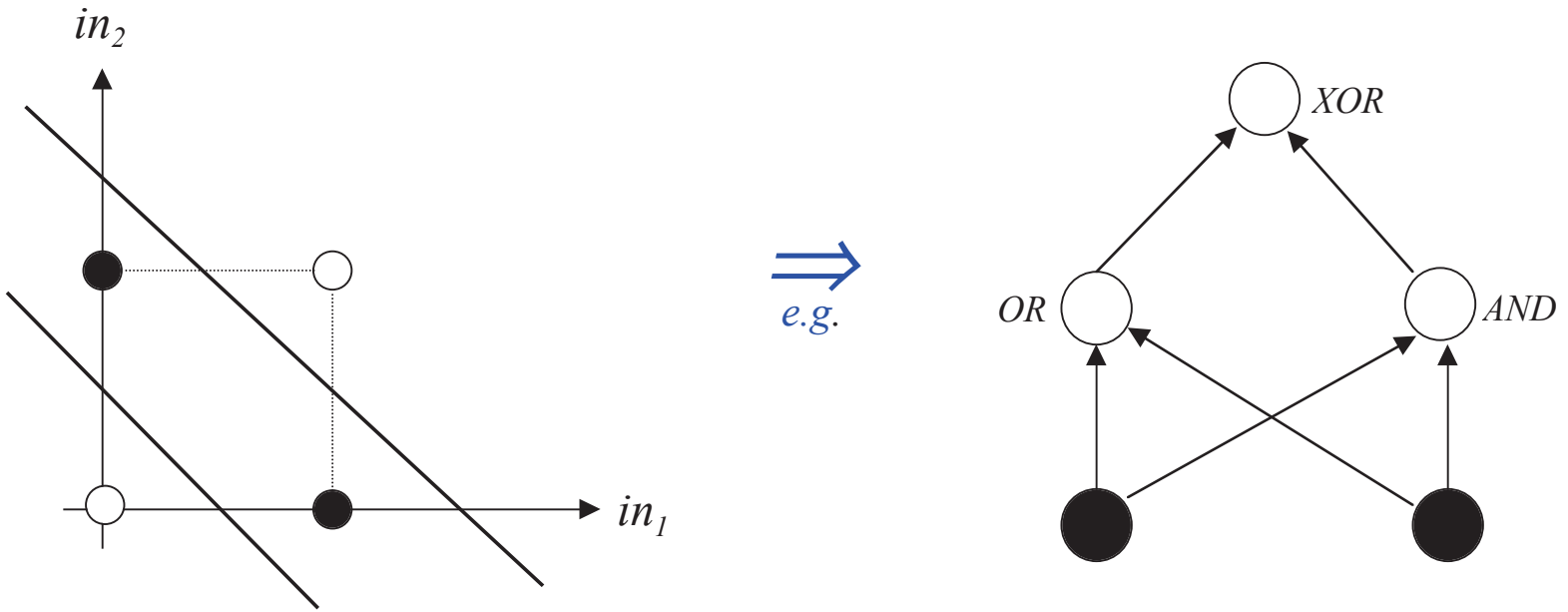
We shall see later that, when we have noisy training data, the training set error and the generalization error are related, and an appropriate stopping criteria will emerge in order to optimize the network's generalization ability.

Learning in Multi-Layer Perceptrons, Back-Propagation

1. XOR and Linear Separability Revisited
2. Notation for Multi-Layer Networks
3. Multi-Layer Perceptrons (MLPs)
4. Learning in Multi-Layer Perceptrons
5. Deriving the Back-Propagation Algorithm
6. Training a Multi-Layer Feed-forward Network
7. Further Practical Considerations for Training MLPs
8. How Many Hidden Units?
9. Different Learning Rates for Different Layers?

XOR and Linear Separability Revisited

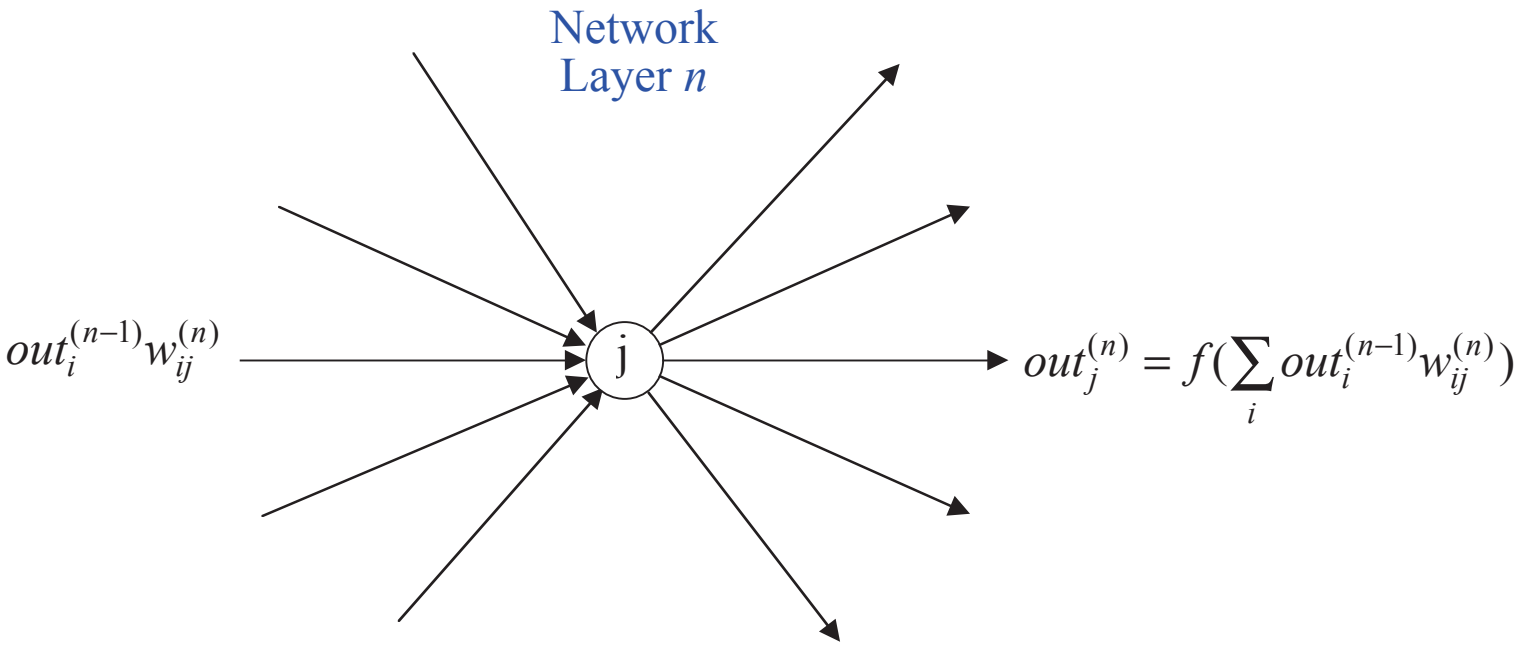
Remember that it is not possible to find weights that enable Single Layer Perceptrons to deal with non-linearly separable problems like XOR:



However, Multi-Layer Perceptrons (MLPs) are able to cope with non-linearly separable problems. Historically, the problem was that there were no learning algorithms for training MLPs. Actually, it is now quite straightforward.

Notation for Multi-Layer Networks

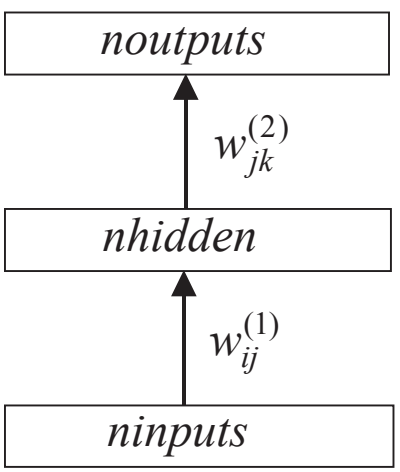
Dealing with multi-layer networks is easy if we use a sensible notation. We simply need another label *(n)* to tell us which layer in the network we are dealing with:



Each unit j in layer n receives activations $out_i^{(n-1)} w_{ij}^{(n)}$ from the previous layer of processing units and sends activations $out_j^{(n)}$ to the next layer of units.

Multi-Layer Perceptrons (MLPs)

Conventionally, the input layer is layer 0, and when we talk of an N layer network we mean there are N layers of weights and N non-input layers of processing units. Thus a two layer Multi-Layer Perceptron takes the form:



$$out_k^{(2)} = f(\sum_j out_j^{(1)} w_{jk}^{(2)})$$

$$out_j^{(1)} = f(\sum_i out_i^{(0)} w_{ij}^{(1)})$$

$$out_i^{(0)} = in_i$$

It is clear how we can add in further layers, though for most practical purposes two layers will be sufficient. Note that there is nothing stopping us from having different activation functions $f(x)$ for different layers, or even different units within a layer.

Learning in Multi-Layer Perceptrons

We can use the same ideas as before to train our N-layer neural networks. We want to adjust the network weights $w_{ij}^{(n)}$ in order to minimise the sum-squared error function

$$E(w_{ij}^{(n)}) = \tfrac{1}{2}\sum_p \sum_j \left(targ_j^p - out_j^{(N)}(in_i^p) \right)^2$$

and again we can do this by a series of gradient descent weight updates

$$\Delta w_{kl}^{(m)} = -\eta \frac{\partial E(w_{ij}^{(n)})}{\partial w_{kl}^{(m)}}$$

Note that it is only the outputs $out_j^{(N)}$ of the final layer that appear in the error function. However, the final layer outputs will depend on all the earlier layers of weights, and the learning algorithm will adjust them all.

The learning algorithm automatically adjusts the outputs $out_j^{(n)}$ of the earlier (hidden) layers so that they form appropriate intermediate (hidden) representations.

Computing the Partial Derivatives

For a two layer network, the final outputs can be written:

$$out_k^{(2)} = f\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right) = f\left(\sum_j f\left(\sum_i in_i w_{ij}^{(1)}\right).w_{jk}^{(2)}\right)$$

We can then use the chain rules for derivatives, as for the Single Layer Perceptron, to give the derivatives with respect to the two sets of weights $w_{hl}^{(1)}$ and $w_{hl}^{(2)}$:

$$\frac{\partial E(w_{ij}^{(n)})}{\partial w_{hl}^{(m)}} = -\sum_p \sum_k \left(targ_k - out_k^{(2)}\right).\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(m)}}$$

$$\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(2)}} = f'\left(\sum_j out_j^{(1)} w_{jk}^{(2)}\right).out_h^{(1)}.\delta_{kl}$$

$$\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(1)}} = f'\left(\sum_j out_j^{(1)} w_{jk}^{(2)}\right).f'\left(\sum_i in_i w_{il}^{(1)}\right).w_{lk}^{(2)}.in_h$$

Deriving the Back Propagation Algorithm

All we now have to do is substitute our derivatives into the weight update equations

$$\Delta w_{hl}^{(2)} = \eta \sum_{p} \left(targ_l - out_l^{(2)} \right) . f' \left(\sum_j out_j^{(1)} w_{jl}^{(2)} \right) . out_h^{(1)}$$

$$\Delta w_{hl}^{(1)} = \eta \sum_{p} \sum_{k} \left(targ_k - out_k^{(2)} \right) . f' \left(\sum_j out_j^{(1)} w_{jk}^{(2)} \right) . f' \left(\sum_i in_i w_{il}^{(1)} \right) . w_{lk}^{(2)} . in_h$$

Then if the transfer function $f(x)$ is a Sigmoid we can use $f'(x) = f(x).(1 - f(x))$ to give

$$\Delta w_{hl}^{(2)} = \eta \sum_{p} \left(targ_l - out_l^{(2)} \right) . out_l^{(2)} . \left(1 - out_l^{(2)} \right) . out_h^{(1)}$$

$$\Delta w_{hl}^{(1)} = \eta \sum_{p} \sum_{k} \left(targ_k - out_k^{(2)} \right) . out_k^{(2)} . \left(1 - out_k^{(2)} \right) . w_{lk}^{(2)} . out_l^{(1)} . \left(1 - out_l^{(1)} \right) . in_h$$

These equations constitute the basic Back-Propagation Learning Algorithm.

Simplifying the Computation

When implementing the Back-Propagation algorithm it is convenient to define

$$delta_k^{(2)} = \left(targ_k - out_k^{(2)}\right).f'\left(\sum_j out_j^{(1)} w_{jk}^{(2)}\right) = \left(targ_k - out_k^{(2)}\right).out_k^{(2)}.\left(1 - out_k^{(2)}\right)$$

which is a generalised output deviation. We can then write the weight update rules as

$$\Delta w_{hl}^{(2)} = \eta \sum_p delta_l^{(2)}.out_h^{(1)}$$

$$\Delta w_{hl}^{(1)} = \eta \sum_p \left(\sum_k delta_k^{(2)}.w_{lk}^{(2)}\right).out_l^{(1)}.\left(1 - out_l^{(1)}\right).in_h$$

So the weight $w_{hl}^{(2)}$ between units h and l is changed in proportion to the output of unit h and the *delta* of unit l. The weight changes at the first layer now take on the same form as the final layer, but the 'error' at each unit l is ***back-propagated*** from each of the output units k via the weights $w_{lk}^{(2)}$.

Networks With Any Number of Hidden Layers

It is now becoming clear that, with the right notation, it is easy to extend our gradient descent algorithm to work for any number of hidden layers. We define

$$delta_k^{(N)} = \left(targ_k - out_k^{(N)}\right).f'\left(\sum_j out_j^{(1)} w_{jk}^{(N)}\right) = \left(targ_k - out_k^{(N)}\right).out_k^{(N)}.\left(1 - out_k^{(N)}\right)$$

as the delta for the output layer, and then back-propagate the deltas to earlier layers using

$$delta_k^{(n)} = \left(\sum_k delta_k^{(n+1)}.w_{lk}^{(n+1)}\right).f'\left(\sum_j out_j^{(n-1)} w_{jk}^{(n)}\right) = \left(\sum_k delta_k^{(n+1)}.w_{lk}^{(n+1)}\right).out_k^{(n)}.\left(1 - out_k^{(n)}\right)$$

Then each weight update equation can be written as:

$$\Delta w_{hl}^{(n)} = \eta \sum_p delta_l^{(n)}.out_h^{(n-1)}$$

Suddenly the Back-Propagation Algorithm looks very simple and easily programmable!

The Cross Entropy Error Function

We have talked about using the ***Sum Squared Error (SSE)*** function as a measure of network performance, and as a basis for gradient descent learning algorithms. A useful alternative for classification networks is the ***Cross Entropy (CE)*** error function

$$E_{ce}(w_{ij}) = -\sum_p \sum_j \left[targ_j^p . \log\left(out_j(in_i^p)\right) + \left(1 - targ_j^p\right) . \log\left(1 - out_j(in_i^p)\right) \right]$$

This is appropriate if we want to interpret the network outputs as probabilities, and has several advantages over the *SSE* function. When we compute the partial derivatives for the gradient descent weight update equations, the sigmoid derivative cancels out leaving

$$\Delta w_{hl}^{(N)} = \eta \sum_p \left(targ_l - out_l^{(N)} \right) . out_h^{(N-1)}$$

which is easier to compute than the *SSE* equivalent, and no longer has the property of going to zero when the outputs are totally wrong (so no need for offsets, etc.).

The Need For Non-Linearity

We have noted that if the network outputs are non-binary, then it is appropriate to have a linear output activation function rather than a Sigmoid. So why not use linear activation functions on the hidden layers as well?

Recall that for an activation functions $f^{(n)}(x)$ at layer n, the outputs are given by

$$out_k^{(2)} = f^{(2)}\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right) = f^{(2)}\left(\sum_j f^{(1)}\left(\sum_i in_i w_{ij}^{(1)}\right).w_{jk}^{(2)}\right)$$

so if the hidden layer activation is linear, i.e. $f^{(1)}(x) = x$, this simplifies to

$$out_k^{(2)} = f^{(2)}\left(\sum_i in_i.\left(\sum_j w_{ij}^{(1)} w_{jk}^{(2)}\right)\right)$$

But this is equivalent to a single layer network with weights $w_{ik} = \sum_j w_{ij}^{(1)} w_{jk}^{(2)}$ and we know that such a network cannot deal with non-linearly separable problems.

Training a Two-Layer Feed-forward Network

The training procedure for two layer networks is similar to that for single layer networks:

1. Take the set of training patterns you wish the network to learn

   $$\{in_i^p, out_j^p : i = 1 \ldots ninputs, j = 1 \ldots noutputs, p = 1 \ldots npatterns\} .$$

2. Set up your network with *ninputs* input units fully connected to *nhidden* non-linear hidden units via connections with weights $w_{ij}^{(1)}$, which in turn are fully connected to *noutputs* output units via connections with weights $w_{jk}^{(2)}$.

3. Generate random initial weights, e.g. from the range [–*smwt*, +*smwt*]

4. Select an appropriate error function $E(w_{jk}^{(n)})$ and learning rate η.

5. Apply the weight update equation $\Delta w_{jk}^{(n)} = -\eta \partial E(w_{jk}^{(n)}) \big/ \partial w_{jk}^{(n)}$ to each weight $w_{jk}^{(n)}$ for each training pattern *p*. One set of updates of all the weights for all the training patterns is called one *epoch* of training.

6. Repeat step 5 until the network error function is 'small enough'.

The extension to networks with more hidden layers should be obvious.

Practical Considerations for Back-Propagation Learning

Most of the practical considerations necessary for general Back-Propagation learning were already covered when we talked about training single layer Perceptrons:

1. Do we need to pre-process the training data? If so, how?
2. How do we choose the initial weights from which we start the training?
3. How do we choose an appropriate learning rate η?
4. Should we change the weights after each training pattern, or after the whole set?
5. Are some activation/transfer functions better than others?
6. How can we avoid flat spots in the error function?
7. How can we avoid local minima in the error function?
8. How do we know when we should stop the training?

However, there are also two important issues that were not covered before:

9. How many hidden units do we need?
10. Should we have different learning rates for the different layers?

How Many Hidden Units?

The best number of hidden units depends in a complex way on many factors, including:

1. The number of training patterns
2. The numbers of input and output units
3. The amount of noise in the training data
4. The complexity of the function or classification to be learned
5. The type of hidden unit activation function
6. The training algorithm

Too few hidden units will generally leave high training and generalisation errors due to under-fitting. Too many hidden units will result in low training errors, but will make the training unnecessarily slow, and will result in poor generalisation unless some other technique (such as *regularisation*) is used to prevent over-fitting.

Virtually all "rules of thumb" you hear about are actually nonsense. A sensible strategy is to try a range of numbers of hidden units and see which works best.

Different Learning Rates for Different Layers?

A network as a whole will usually learn most efficiently if all its neurons are learning at roughly the same speed. So maybe different parts of the network should have different learning rates η. There are a number of factors that may affect the choices:

1. The later network layers (nearer the outputs) will tend to have larger local gradients (*deltas*) than the earlier layers (nearer the inputs).
2. The activations of units with many connections feeding into or out of them tend to change faster than units with fewer connections.
3. Activations required for linear units will be different for Sigmoidal units.
4. There is empirical evidence that it helps to have different learning rates η for the thresholds/biases compared with the real connection weights.

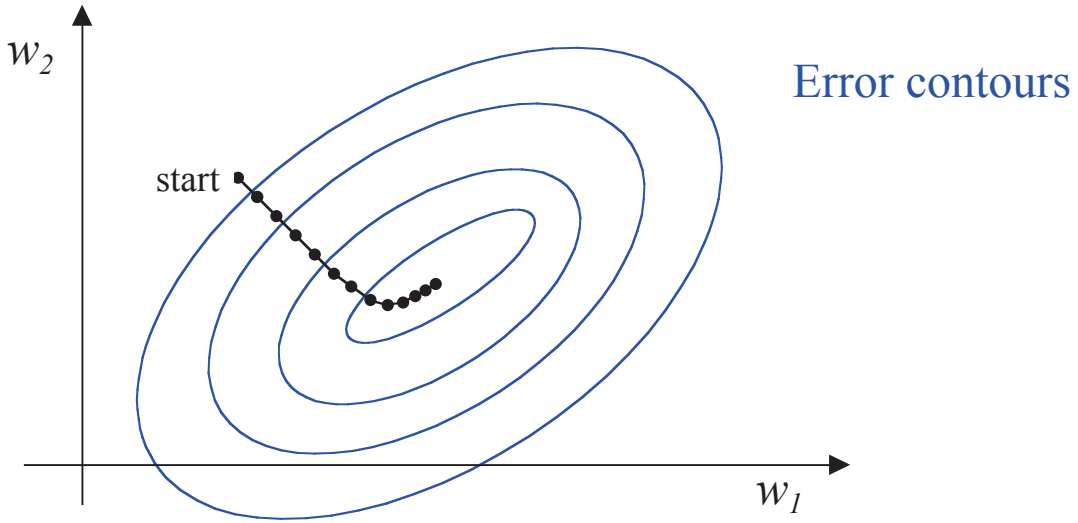
In practice, it is often quicker to just use the same rates η for all the weights and thresholds, rather than spending time trying to work out appropriate differences. A very powerful approach is to use evolutionary strategies to determine good learning rates.

Learning with Momentum, Conjugate Gradient Learning

1. Visualising Learning
2. Learning with Momentum
3. Learning with Line Searches
4. Parabolic Interpolation
5. Conjugate Gradient Learning

Visualising Learning

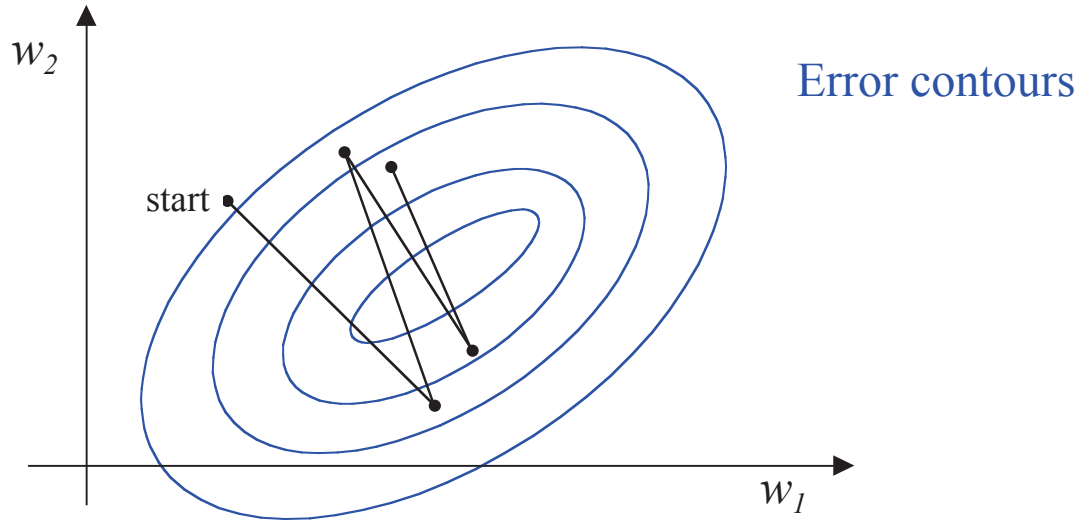
Visualising neural network learning is difficult because there are so many weights being updated at once. However, we can plot error function contours for pairs of weights to get some idea of what is happening. The weight update equations will produce a series of steps in weight space from the starting position to the error minimum:



True gradient descent produces a smooth curve perpendicular to the contours. Weight updates with a small step size η will result in the good approximation to it shown.

Step Size Too Large

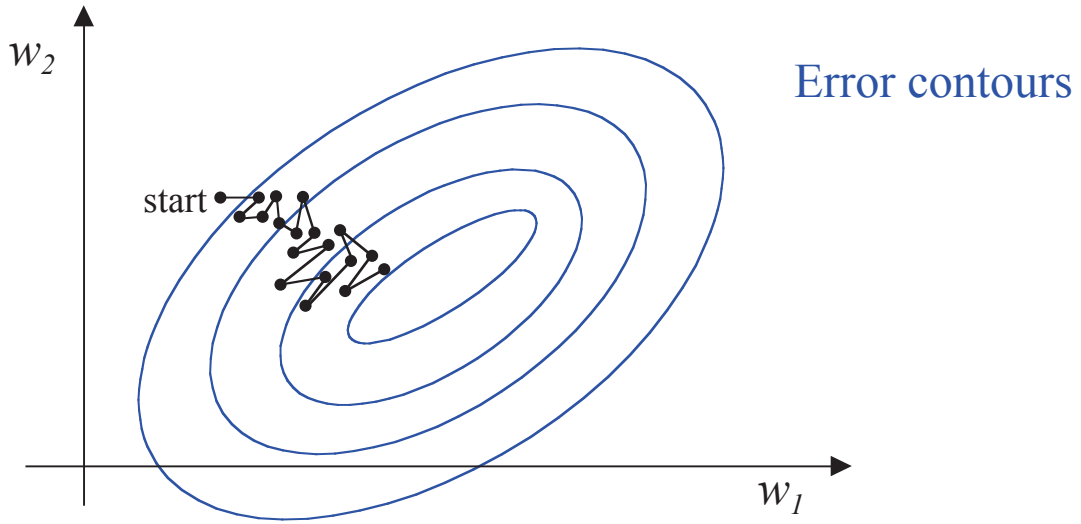
If the our learning rate step size is set too large, the approximation to true gradient descent will be poor, and we will end up with overshoots, or even divergence:



In practice, we don't need to plot error contours to see if this is happening. It is obvious that, if the error function is fluctuating, we should reduce the step size. It is also worth checking individual weights and output activations as well. On the other hand, if everything is smooth, it is worth trying to increase η and seeing if it stays smooth.

On-line Learning

If we update the weights after each training pattern, rather than adding up the weight changes for all the patterns before applying them, the learning algorithm is no longer true gradient descent, and the weight changes will not be perpendicular to the contours:



If we keep the step sizes small enough, the erratic behaviour of the weight updates will not be too much of a problem, and the increased number of weight changes will still get us to the minimum quicker than true gradient descent (i.e. batch learning).

Learning with Momentum

A compromise that will smooth out the erratic behaviour of the on-line updates, without slowing down the learning too much, is to update the weights with the moving average of the individual weight changes corresponding to single training patterns.

If we label everything by the time t (which we can conveniently measure in weight update steps), then implementing a moving average is easy:

$$\Delta w_{hl}^{(n)}(t) = \eta \sum_{p} delta_{l}^{(n)}(t).out_{h}^{(n-1)}(t) + \alpha.\Delta w_{hl}^{(n)}(t-1)$$

We simply add a ***momentum*** term $\alpha.\Delta w_{hl}^{(n)}(t-1)$ which is the weight change of the previous step times a momentum parameter α. If α is zero, then we have the standard on-line training algorithm used before. As we increase α towards one, each step includes increasing contributions from previous training patterns. Obviously it makes no sense to have α less than zero, or greater than one. Good sizes of α depend on the size of the training data set and how variable it is. Usually, we will need to decrease η as we increase α, so that the total step sizes don't get too large.

Learning with Line Searches

Learning algorithms which work by taking a sequence of steps in weight space all have two basic components: the ***step size*** $size(t)$ and the ***direction*** $dir_{hl}^{(n)}(t)$ such that

$$\Delta w_{hl}^{(n)}(t) = size(t).dir_{hl}^{(n)}(t)$$

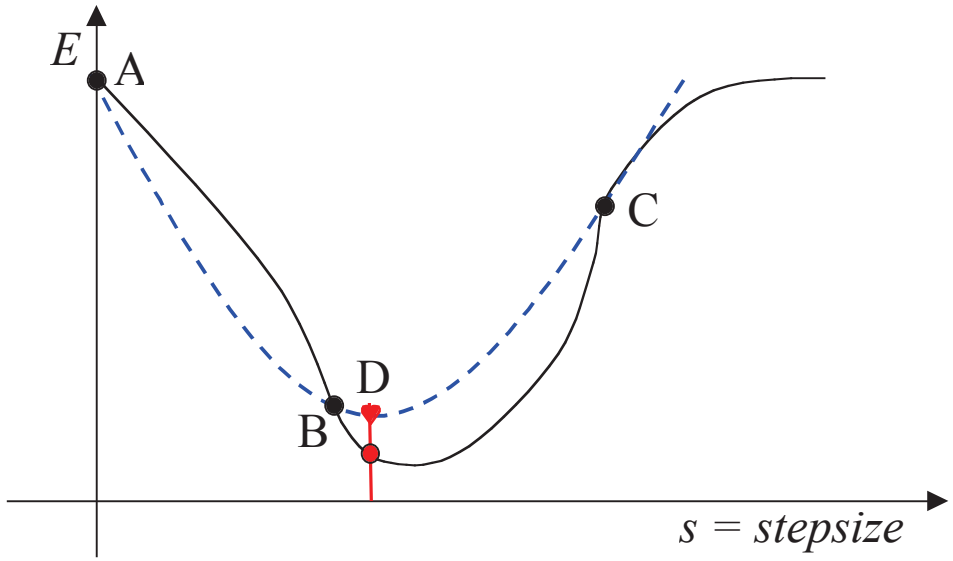
For gradient descent algorithms, such as Back-Propagation, the chosen direction is given by the partial derivatives of the error function $dir_{hl}^{(n)}(t) = -\partial E(w_{jk}^{(m)}) / \partial w_{hl}^{(n)}$, and the step size is simply the small constant learning rate parameter $size(t)=\eta$.

A better procedure might be to carry on along in the chosen direction until the error starts rising again. This involves performing a ***line search*** to determine the step size.

The simplest procedure for performing a line search would be to take a series of small steps along the chosen direction until the error increases, and then go back one step. However, this is not likely to be much more efficient than standard gradient descent. Fortunately, there are many better procedures for performing line searches.

Parabolic Interpolation

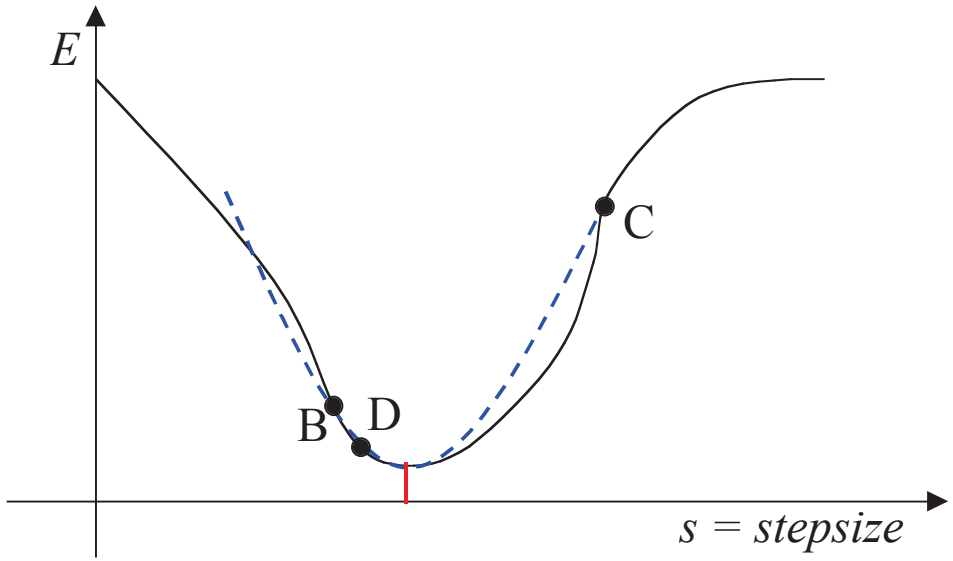
We can plot the variation of the error function *E(s)* as we increase the step *s* taken along our chosen direction. Suppose we can find three points A, B, C such that $E(A) > E(B)$ and $E(B) < E(C)$. If A is where we are, and C is far away, this should not be difficult.



The three points are sufficient to define the parabola that passes through them, and we can then easily compute the minimum point D of that parabola. The step size corresponding to that point D is a good guess for the appropriate step size for the actual error function.

Repeated Parabolic Interpolation

The process of parabolic interpolation can easily be repeated. We take our guess of the appropriate step size given by point D, together with the two original points of lowest error (i.e. B and C) to make up a new set of three points, and repeat the procedure:



Each iteration of this process brings us closer to the minimum. However, the gradients change as we take each step, so it is not computationally efficient to get each step size too accurately. Often it is better to get it roughly right and move on to the next direction.

Problems using Gradient Descent with Line Search

We have seen how we can determine appropriate step sizes by doing line searches. It is not obvious, though, that using the gradient descent direction is still the best thing to do. Remember that we chose our step size $s(t-1)$ to minimise the new error

$$E\left(w_{ij}(t)\right) = E\left(w_{ij}(t-1) - s(t-1)\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)} \right)$$

But we know the derivative of this with respect to $s(t-1)$ must be zero at the minimum, and we can use the chain rule for derivatives to compute that derivative, so we have

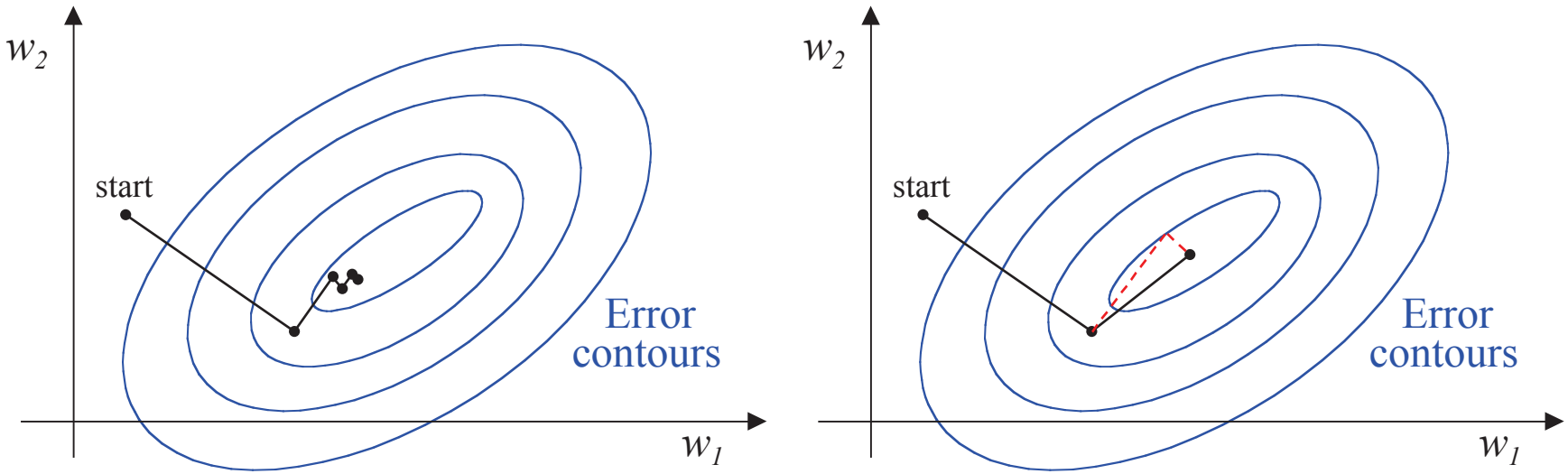
$$\frac{\partial E(w_{ij}(t))}{\partial s(t-1)} = \sum_{i,j} \frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} \cdot \frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)} = 0$$

The vector product of the old direction $-\partial E(w_{ij}(t-1))/\partial w_{ij}(t-1)$ and the new direction $-\partial E(w_{ij}(t))/\partial w_{ij}(t)$ is zero, which means the directions are orthogonal (perpendicular). This will result in a less than optimal zig-zag path through weight space.

Finding a Better Search Direction

An obvious approach to avoid the zig-zagging that occurs when we use line searches and gradient directions is to make the new step direction $dir_{ij}(t)$ a compromise between the new gradient direction $-\partial E(w_{ij}(t))/\partial w_{ij}(t)$ and the previous step direction $dir_{ij}(t-1)$:

$$dir_{ij}(t) = -\frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)} + \beta . dir_{ij}(t-1)$$



Conjugate Gradient Learning

The basis of ***Conjugate Gradient Learning*** is to find a value for β in the last equation so that each new search direction spoils as little as possible the minimisation achieved by the previous one. We thus want to find the new direction $dir_{ij}(t)$ such that the gradient $-\partial E(w_{ij}(t))/\partial w_{ij}(t)$ at the new point $w_{ij}(t)+s.dir_{ij}(t)$ in the old direction is zero, i.e.

$$\sum_{i,j} dir_{ij}(t-1).\frac{\partial E(w_{ij}(t)+s.dir_{ij}(t))}{\partial w_{ij}(t)}=0$$

An appropriate value of β that satisfies this is given by the ***Polak-Ribiere rule***

$$\beta=\frac{\sum_{i,j}\left(\frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)}-\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}\right).\frac{\partial E(w_{ij}(t))}{\partial w_{ij}(t)}}{\sum_{i,j}\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}.\frac{\partial E(w_{ij}(t-1))}{\partial w_{ij}(t-1)}}$$

For most practical applications this is the fastest way to train our neural networks.

Bias and Variance, Under-Fitting and Over-Fitting

1. The Computational Power of MLPs
2. Learning and Generalization Revisited
3. A Statistical View of Network Training
4. Bias and Variance
5. Under-fitting, Over-fitting and the Bias/Variance Trade-off
6. Preventing Under-fitting and Over-fitting

Computational Power of MLPs

The ***universal approximation theorem*** can be stated as:

Let $\varphi(\cdot)$ be a non-constant, bounded, and monotone-increasing continuous function. Then for any continuous function $f(\mathbf{x})$ with $\mathbf{x} = \{x_i \in [0,1] : i = 1, \ldots, m\}$ and $\varepsilon > 0$, there exists an integer M and real constants $\{\alpha_j, b_j, w_{jk} : j = 1, \ldots, M, k = 1, \ldots, m\}$ such that

$$F(x_1, \ldots, x_m) = \sum_{j=1}^{M} \alpha_j \varphi\left(\sum_{k=1}^{m} w_{jk} x_k - b_j \right)$$

is an approximate realisation of $f(\cdot)$, that is

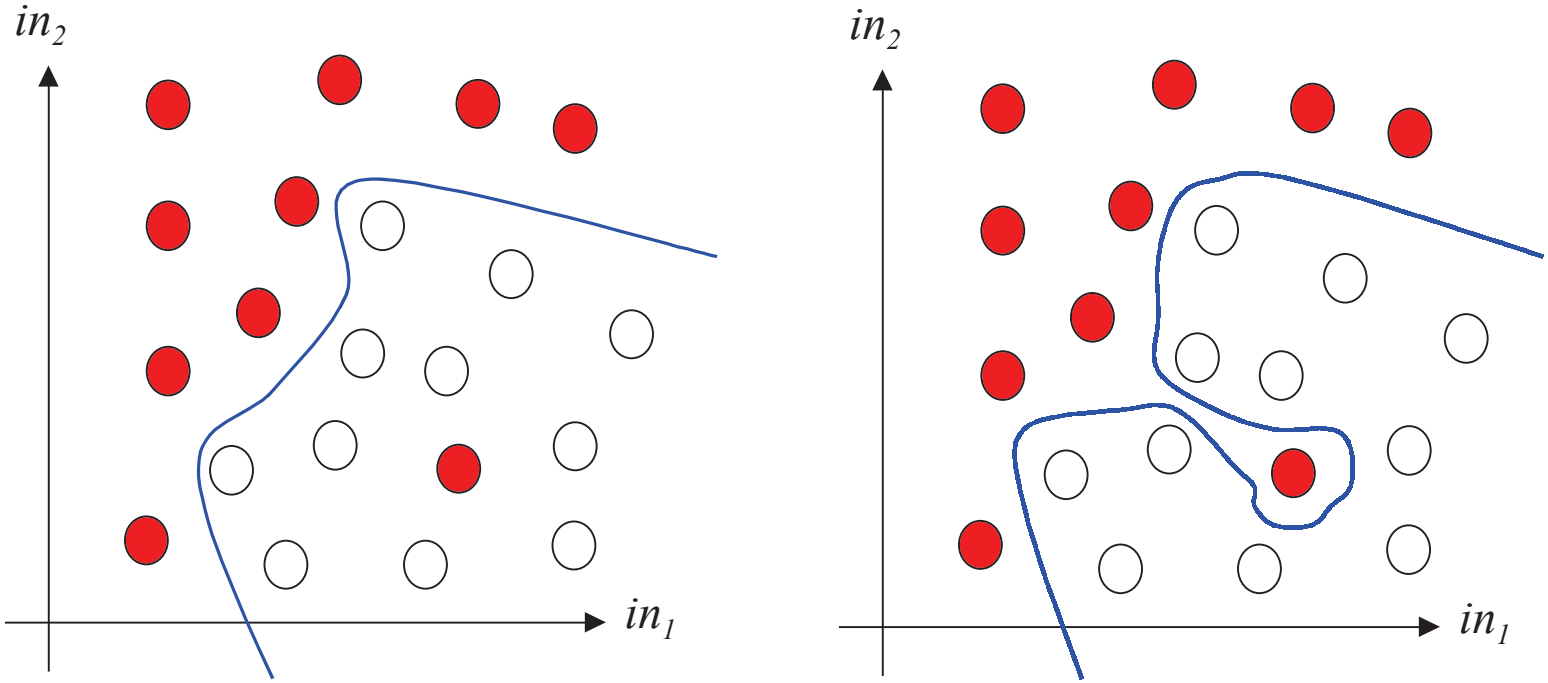
$$\left| F(x_1, \ldots, x_m) - f(x_1, \ldots, x_m) \right| < \varepsilon$$

for all $\mathbf{x}$ that lie in the input space.

Clearly this applies to an MLP with M hidden units, since $\varphi(\cdot)$ can be a sigmoid, w_{jk}, b_j can be hidden layer weights and biases, and α_j can be output weights. It follows that, given enough hidden units, a two layer MLP can approximate any continuous function.

Learning and Generalization Revisited

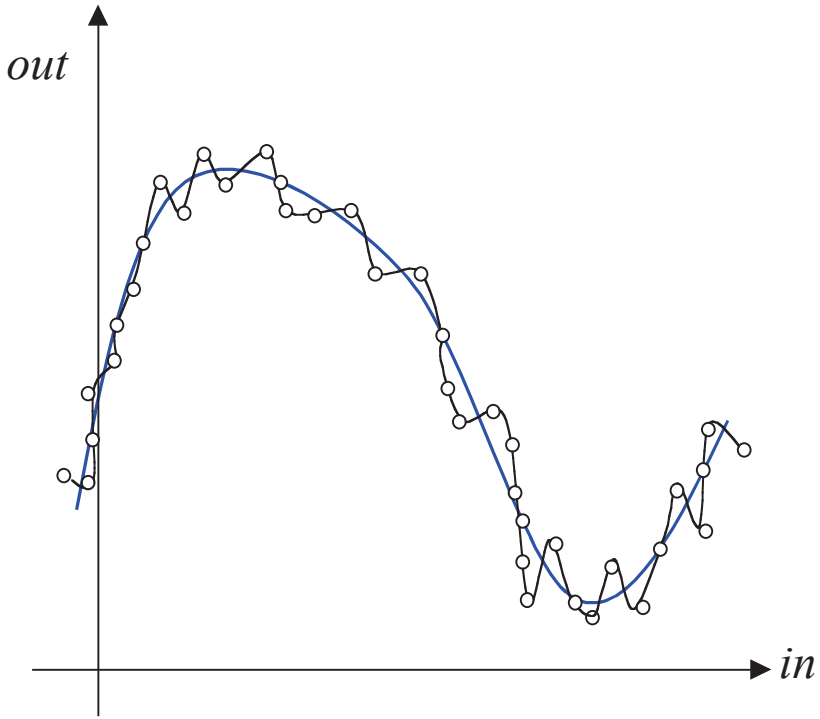
Recall the idea of getting a neural network to learn a classification decision boundary:



Our aim is for the network to generalize to classify new inputs appropriately. If we know that the training data contains noise, we don't necessarily want the training data to be classified totally accurately as that is likely to reduce the generalisation ability.

Generalization in Function Approximation

Similarly if our network is required to recover an underlying function from noisy data:



We can expect the network to give a more accurate generalization to new inputs if its output curve does not pass through all the data points. Again, allowing a larger error on the training data is likely to lead to better generalization.

A Statistical View of the Training Data

Suppose we have a ***training data set*** D for our neural network:

$$D = \{ x_i^p, y^p : i = 1 \ldots ninputs, p = 1 \ldots npatterns \}$$

This consists of an output y^p for each input pattern x_i^p. To keep the notation simple we shall assume we only have one output unit – the extension to many outputs is obvious.

Generally, the training data will be generated by some actual function $g(x_i)$ plus random noise ε^p (which may, for example, be due to data gathering errors), so

$$y^p = g(x_i^p) + \varepsilon^p$$

We call this a ***regressive model*** of the data. We can define a statistical expectation operator $\mathcal{E}$ that averages over all possible training patterns, so

$$g(x_i) = \mathcal{E}[y \mid x_i]$$

We say that the regression function $g(x_i)$ is the *conditional mean of the model output y given the inputs* x_i.

A Statistical View of Network Training

The neural network training problem is to construct an output function $net(x_i, W, D)$ of the network weights $W = \{w_{ij}^{(n)}\}$, based on the data D, that best approximates the regression model, i.e. the underlying function $g(x_i)$.

We have seen how to train a network by minimising the sum-squared error cost function:

$$E(W) = \tfrac{1}{2} \sum_{p \in D} \left(y^p - net(x_i^p, W, D) \right)^2$$

with respect to the network weights $W = \{w_{ij}^{(n)}\}$. However, we have also observed that, to get good generalisation, we do not necessarily want to achieve that minimum. What we really want to do is minimise the difference between the network's outputs $net(x_i, W, D)$ and the underlying function $g(x_i) = \mathcal{E}[y \mid x_i]$.

The natural sum-squared error function, i.e. $\left(\mathcal{E}[y \mid x_i] - net(x_i, W, D) \right)^2$, depends on the specific training set D, and we really want our network training regime to produce good results averaged over all possible noisy training sets.

Bias and Variance

If we define the expectation or average operator $\mathcal{E}_D$ which takes the ***ensemble average*** over all possible training sets D, then some rather messy algebra allows us to show that:

$$\mathcal{E}_D\left[\left(\mathcal{E}[y \mid x_i] - net(x_i, W, D)\right)^2\right]$$

$$= \left(\mathcal{E}_D\left[net(x_i, W, D)\right] - \mathcal{E}[y \mid x_i]\right)^2 + \mathcal{E}_D\left[\left(net(x_i, W, D) - \mathcal{E}_D\left[net(x_i, W, D)\right]\right)^2\right]$$

$$= \quad \text{(bias)}^2 \quad + \quad \text{(variance)}$$

This error function consists of two positive components:

(bias)^2 the difference between the average network output $\mathcal{E}_D[net(x_i, W, D)]$ and the regression function $g(x_i) = \mathcal{E}[y \mid x_i]$. This can be viewed as the ***approximation error***.

(variance) the variance of the approximating function $net(x_i, W, D)$ over all the training sets D. It represents the ***sensitivity*** of the results on the particular choice of data D.

In practice there will always be a trade-off between these two error components.

The Extreme Cases of Bias and Variance

We can best understand the concepts of ***bias*** and ***variance*** by considering the two extreme cases of what the network might learn.

Suppose our network is lazy and just generates the same constant output whatever training data we give it, i.e. $net(x_i, W, D) = c$. In this case the variance term will be zero, but the bias will be large, because the network has made no attempt to fit the data.

Suppose our network is very hard working and makes sure that it fits every data point:

$$\mathcal{E}_D\left[net(x_i, W, D)\right] = \mathcal{E}_D\left[y(x_i)\right] = \mathcal{E}_D\left[g(x_i) + \varepsilon\right] = \mathcal{E}[y \mid x_i]$$

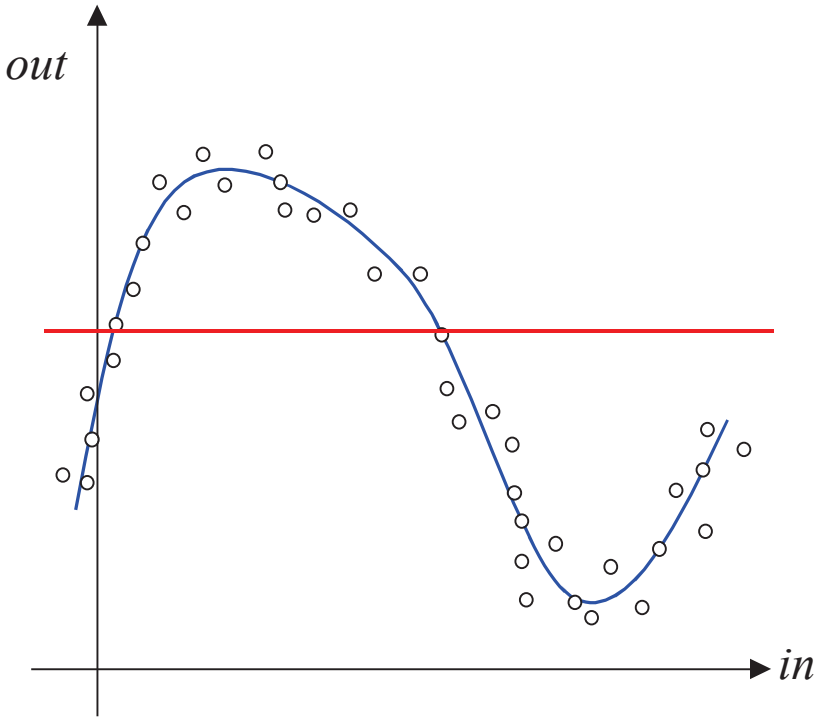
so the bias is zero, but the variance is:

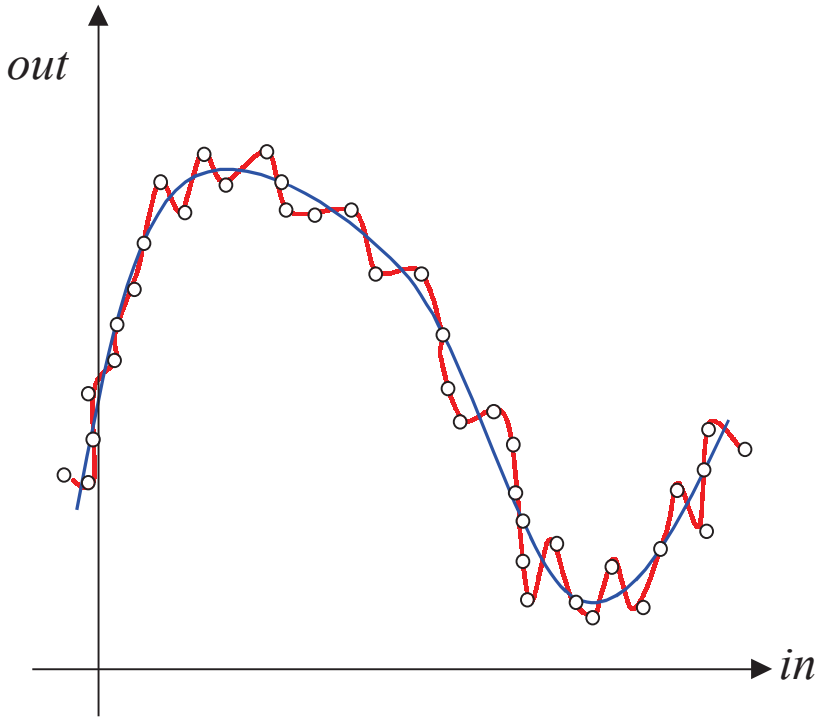
$$\mathcal{E}_D\left[\left(net(x_i, W, D) - \mathcal{E}_D\left[net(x_i, W, D)\right]\right)^2\right] = \mathcal{E}_D\left[\left(g(x_i) + \varepsilon - \mathcal{E}_D\left[g(x_i) + \varepsilon\right]\right)^2\right] = \mathcal{E}_D\left[(\varepsilon)^2\right]$$

i.e. the variance of the noise on the data, which could be substantial.

Examples of the Two Extreme Cases

The lazy and hard-working networks approach our function approximation as follows:





Ignore the data ⇒

Big approximation errors (high bias)

No variation between data sets (no variance)

Get every data point ⇒

No approximation errors (zero bias)

Variation between data sets (high variance)

Under-fitting, Over-fitting and the Bias/Variance Trade-off

If our network is to generalize well to new data, we obviously need it to generate a good approximation to the underlying function $g(x_i) = \mathcal{E}[y \mid x_i]$, and we have seen that to do this we must minimise the sum of the bias and variance terms. There will clearly have to be a ***trade-off*** between minimising the bias and minimising the variance.

A network which is too closely fitted to the data will tend to have a large variance and hence give a large expected generalization error. We then say that ***over-fitting*** of the training data has occurred.

We can easily decrease the variance by smoothing the network outputs, but if this is taken too far, then the bias becomes large, and the expected generalization error is large again. We then say that ***under-fitting*** of the training data has occurred.

This trade-off between bias and variance plays a crucial role in the application of neural network techniques to practical applications.

Preventing Under-fitting and Over-fitting

To ***prevent under-fitting*** we need to make sure that:

1. The network has enough hidden units to represent to required mappings.
2. We train the network for long enough so that the sum squared error cost function is sufficiently minimised.

To ***prevent over-fitting*** we can:

1. Stop the training early – before it has had time to learn the training data too well.
2. Restrict the number of adjustable parameters the network has – e.g. by reducing the number of hidden units, or by forcing connections to share the same weight values.
3. Add some form of *regularization* term to the error function to encourage smoother network mappings.
4. Add noise to the training patterns to smear out the data points.

Next lecture will be dedicated to looking at these approaches to improving generalization.

Improving Generalization

1. Improving Generalization
2. Training, Validation and Testing Data Sets
3. Cross-Validation
4. Weight Restriction and Weight Sharing
5. Stopping Training Early
6. Regularization and Weight Decay
7. Adding Noise / Jittering
8. Which is the Best Approach ?

Improving Generalization

We have seen that, for our networks to generalize well, we need to avoid both under-fitting of the training data (which corresponds to high statistical bias) and over-fitting of the training data (which corresponds to high statistical variance).

There are a number of approaches for improving generalization – we can:

1. Arrange to have the optimum number of free parameters (independent connection weights) in our model.
2. Stop the gradient descent training at the appropriate point.
3. Add a regularization term to the error function to smooth out the mappings that are learnt.
4. Add noise to the training patterns to smooth out the data points.

To employ these effectively we need to be able to estimate from our training data what the generalization is likely be.

Training, Validation and Testing Data Sets

We have talked about ***training data*** sets – the data used for training our networks. The ***testing data*** set is the unseen data that is used to test the network's generalization.

We usually want to optimize our network's training procedures to result in the best generalization, but using the testing data to do this would clearly be cheating. What we can do is assume that the training data and testing data are drawn randomly from the same data set, and then any sub-set of the training data that we do not train the network on can be used to estimate what the performance on the testing set will be, i.e. what the generalization will be. The portion of the data we have available for training that is withheld from the network training is called the ***validation data*** set, and the remainder of the data is called the ***training data*** set. This approach is called the ***hold out method***.

The idea is that we split the available data into training and validation sets, train various networks using the training set, test each one on the validation set, and the network which is best is likely to provide the best generalization to the testing set.

Cross Validation

Often the availability of training data is limited, and using part of it as a validation set is not practical. An alternative is to use the procedure of ***cross-validation***.

In ***K-fold cross-validation*** we divide all the training data at random into K distinct subsets, train the network using K–1 subsets, and test the network on the remaining subset. The process of training and testing is then repeated for each of the K possible choices of the subset omitted from the training. The average performance on the K omitted subsets is then our estimate of the generalization performance.

This procedure has the advantage that is allows us to use a high proportion of the available training data (a fraction $1-{}^{1}/_{K}$) for training, while making use of all the data points in estimating the generalization error. The disadvantage is that we need to train the network K times. Typically $K \sim 10$ is considered reasonable.

If K is made equal to the full sample size, it is called ***leave-one-out cross validation***.

Weight Restriction and Weight Sharing

Perhaps the most obvious way to prevent over-fitting in our models (i.e. neural networks) is to restrict the number of free parameters they have.

The simplest way we can do this is to restrict the number of hidden units, as this will automatically reduce the number of weights. We can use some form of validation or cross-validation scheme to find the best number for each given problem.

An alternative is to have many weights in the network, but constrain certain groups of them to be equal. If there are symmetries in the problem, we can enforce ***hard weight sharing*** by building them into the network in advance. In other problems we can use ***soft weight sharing*** where sets of weights are encouraged to have similar values by the learning algorithm.

One way to do this is to add an appropriate term to the error/cost function. This method can then be seen as a particular form of ***regularization***.

Stopping Training Early

Neural networks are often set up with more than enough parameters for over-fitting to occur, and so other procedures have to be employed to prevent it.

For the iterative gradient descent based network training procedures we have considered (such as batch back-propagation and conjugate gradients), the training set error will naturally decrease with increasing numbers of epochs of training.

The error on the unseen validation and testing data sets, however, will start off decreasing as the under-fitting is reduced, but then it will eventually begin to increase again as over-fitting occurs.

The natural solution to get the best generalization, i.e. the lowest error on the test set, is to use the procedure of ***early stopping***. One simply trains the network on the training set until the error on the validation set starts rising again, and then stops. That is the point at which we expect the generalization error to start rising as well.

Practical Aspects of Stopping Early

One potential problem with the idea of stopping early is that the validation error may go up and down numerous times during training. The safest approach is generally to train to convergence (or at least until it is clear that the validation error is unlikely to fall again), saving the weights at each epoch, and then go back to weights at the epoch with the lowest validation error.

There is an approximate relationship between stopping early and a particular form of regularization which indicates that it will work best if the training starts off with very small random initial weights.

There are general practical problems concerning how to best split the available training data into distinct training and validation data sets. For example: What fraction of the patterns should be in the validation set? Should the data be split randomly, or by some systematic algorithm? As so often, these issues are very problem dependent and there are no simple general answers.

Regularization

The general technique of ***regularization*** encourages smoother network mappings by adding a penalty term Ω to the standard (e.g. sum squared error) cost function

$$E_{reg} = E_{sse} + \lambda\Omega$$

where the regularization parameter λ controls the trade-off between reducing the error E_{sse} and increasing the smoothing. This modifies the gradient descent weight updates so

$$\Delta w_{kl}^{(m)} = -\eta \frac{\partial E_{sse}(w_{ij}^{(n)})}{\partial w_{kl}^{(m)}} - \eta\lambda \frac{\partial \Omega(w_{ij}^{(n)})}{\partial w_{kl}^{(m)}}$$

For example, for the case of soft weight sharing we can use the regularization function

$$\Omega = -\sum_{k,l,m} \ln\left(\sum_{j=1}^{M} \frac{\alpha_j}{\sqrt{2\pi\sigma_j^2}} \exp\left\{ -\frac{(w_{kl}^{(m)} - \mu_j)^2}{2\sigma_j^2} \right\} \right)$$

in which the $3M$ parameters α_j, μ_j, σ_j are optimised along with the weights.

Weight Decay

One of the simplest forms of regularization has a regularization function which is just the sum of the squares of the network weights (not including the thresholds):

$$\Omega = -\tfrac{1}{2} \sum_{k,l,m} (w_{kl}^{(m)})^2$$

In conventional curve fitting this regularizer is known as ***ridge regression***. We can see why it is called ***weight decay*** when we observe the extra term in the weight updates:

$$-\eta\lambda \frac{\partial\Omega(w_{ij}^{(n)})}{\partial w_{kl}^{(m)}} = -\eta\lambda w_{kl}^{(m)}$$

In each epoch the weights decay in proportion to their size, i.e. exponentially.

Empirically, this leads to significant improvements in generalization. This is because producing over-fitted mappings requires high curvature and hence large weights. Weight decay keeps the weights small and hence the mappings are smooth.

Adding Noise / Jittering

Adding ***noise*** or ***jitter*** to the inputs during training is also found empirically to improve network generalization. This is because the noise will 'smear out' each data point and make it difficult for the network to fit the individual data points precisely, and consequently reduce over-fitting.

Actually, if the added noise is ξ with variance λ, the error function can be written:

$$E_{sse}(\xi_i) = \tfrac{1}{2}\sum_p \sum_\xi \sum_j \left(y_j^p - net_j(x_i^p + \xi_i)\right)^2$$

and after some tricky mathematics one can show that

$$\lambda\Omega = E_{sse}(\xi_i) - E_{sse}(0) = \tfrac{1}{2}\lambda \sum_p \sum_j \left(\frac{\partial net_j(x_k^p)}{\partial x_i^p} \right)^2$$

which is a standard ***Tikhonov regularizer*** minimising curvature. The gradient descent weight updates can then be performed with an extended back-propagation algorithm.

Which is the Best Approach ?

To get an optimal balance between bias and variance, we need to find a way of optimising the effective complexity of the model, which for neural networks means optimising the effective number of adaptable parameters.

One way to optimise that number is to use the appropriate number of connection weights, which can easily be adjusted by changing the number of hidden units. The other approaches we have looked at can all be seen to be effectively some form of regularization, which involves adding a penalty term to the standard gradient descent error function. The degree of regularization, and hence the effective complexity of the model, is controlled by adjusting the regularization scale λ.

In practice, we find the best number of hidden units, or degree of regularization, using a validation data set or the procedure of cross-validation. The approaches all work well, and which we choose will ultimately depend on which is most convenient for the particular problem in hand. Unfortunately, there is no overall best approach!

Radial Basis Function Networks: Introduction

1. Introduction to Radial Basis Functions
2. Exact Interpolation
3. Common Radial Basis Functions
4. Radial Basis Function (RBF) Networks
5. Problems with Exact Interpolation Networks
6. Improving RBF Networks
7. The Improved RBF Network

Introduction to Radial Basis Functions

The idea of ***Radial Basis Function (RBF) Networks*** derives from the theory of function approximation. We have already seen how Multi-Layer Perceptron (MLP) networks with a hidden layer of sigmoidal units can learn to approximate functions. RBF Networks take a slightly different approach. Their main features are:

1. They are two-layer feed-forward networks.
2. The hidden nodes implement a set of radial basis functions (e.g. Gaussian functions).
3. The output nodes implement linear summation functions as in an MLP.
4. The network training is divided into two stages: first the weights from the input to hidden layer are determined, and then the weights from the hidden to output layer.
5. The training/learning is very fast.
6. The networks are very good at interpolation.

We'll spend the next three lectures studying the details…

Exact Interpolation

The ***exact interpolation*** of a set of N data points in a multi-dimensional space requires every one of the D dimensional input vectors $\mathbf{x}^p = \{x_i^p : i = 1,..., D\}$ to be mapped onto the corresponding target output t^p. The goal is to find a function $f(\mathbf{x})$ such that

$$f(\mathbf{x}^p) = t^p \qquad \forall\, p = 1,..., N$$

The radial basis function approach introduces a set of N ***basis functions***, one for each data point, which take the form $\phi\left(\left\|\mathbf{x} - \mathbf{x}^p\right\|\right)$ where $\phi(\cdot)$ is some non-linear function whose form will be discussed shortly. Thus the pth such function depends on the distance $\left\|\mathbf{x} - \mathbf{x}^p\right\|$, usually taken to be Euclidean, between $\mathbf{x}$ and $\mathbf{x}^p$. The output of the mapping is then taken to be a linear combination of the basis functions, i.e.

$$f(\mathbf{x}) = \sum_{p=1}^{N} w_p \phi\left(\left\|\mathbf{x} - \mathbf{x}^p\right\|\right)$$

The idea is to find the "weights" w_p such that the function goes through the data points.

Determining the Weights

It is easy to determine equations for the weights by combining the above equations:

$$f(\mathbf{x}^q) = \sum_{p=1}^{N} w_p \phi\left(\left\|\mathbf{x}^q - \mathbf{x}^p\right\|\right) = t^q$$

We can write this in matrix form by defining the vectors $\mathbf{t} = \{t^p\}$ and $\mathbf{w} = \{w_p\}$, and the matrix $\mathbf{\Phi} = \left\{\Phi_{pq} = \phi\left(\left\|\mathbf{x}^q - \mathbf{x}^p\right\|\right)\right\}$. This simplifies the equation to $\mathbf{\Phi}\,\mathbf{w} = \mathbf{t}$. Then, provided the inverse of $\mathbf{\Phi}$ exists, we can use any standard matrix inversion techniques to give

$$\mathbf{w} = \mathbf{\Phi}^{-1}\,\mathbf{t}$$

It can be shown that, for a large class of basis functions $\phi(\cdot)$, the matrix $\mathbf{\Phi}$ is indeed non-singular (and hence invertable) providing the data points are distinct.

Once we have the weights, we have a function $f(\mathbf{x})$ that represents a continuous differentiable surface that passes exactly through each data point.

Commonly Used Radial Basis Functions

A range of theoretical and empirical studies have indicated that many properties of the interpolating function are relatively insensitive to the precise form of the basis functions $\phi(r)$. Some of the most commonly used basis functions are:

1. Gaussian Functions:

$$\phi(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right) \qquad \text{width parameter } \sigma > 0$$

2. Multi-Quadric Functions:

$$\phi(r) = \left(r^2 + \sigma^2\right)^{1/2} \qquad \text{parameter } \sigma > 0$$

3. Generalized Multi-Quadric Functions:

$$\phi(r) = \left(r^2 + \sigma^2\right)^{\beta} \qquad \text{parameters } \sigma > 0,\ 1 > \beta > 0$$

4. **Inverse Multi-Quadric Functions**:

$$\phi(r) = \left(r^2 + \sigma^2\right)^{-1/2} \qquad \text{parameter } \sigma > 0$$

5. **Generalized Inverse Multi-Quadric Functions**:

$$\phi(r) = \left(r^2 + \sigma^2\right)^{-\alpha} \qquad \text{parameters } \sigma > 0,\ \alpha > 0$$

6. **Thin Plate Spline Function**:

$$\phi(r) = r^2 \ln(r)$$

7. **Cubic Function**:

$$\phi(r) = r^3$$

8. **Linear Function**:

$$\phi(r) = r$$

Properties of the Radial Basis Functions

The Gaussian and Inverse Multi-Quadric Functions are ***localised*** in the sense that

$$\phi(r) \to 0 \quad \text{as} \quad |\mathrm{r}| \to \infty$$

but this is not strictly necessary. All the other functions above have the property

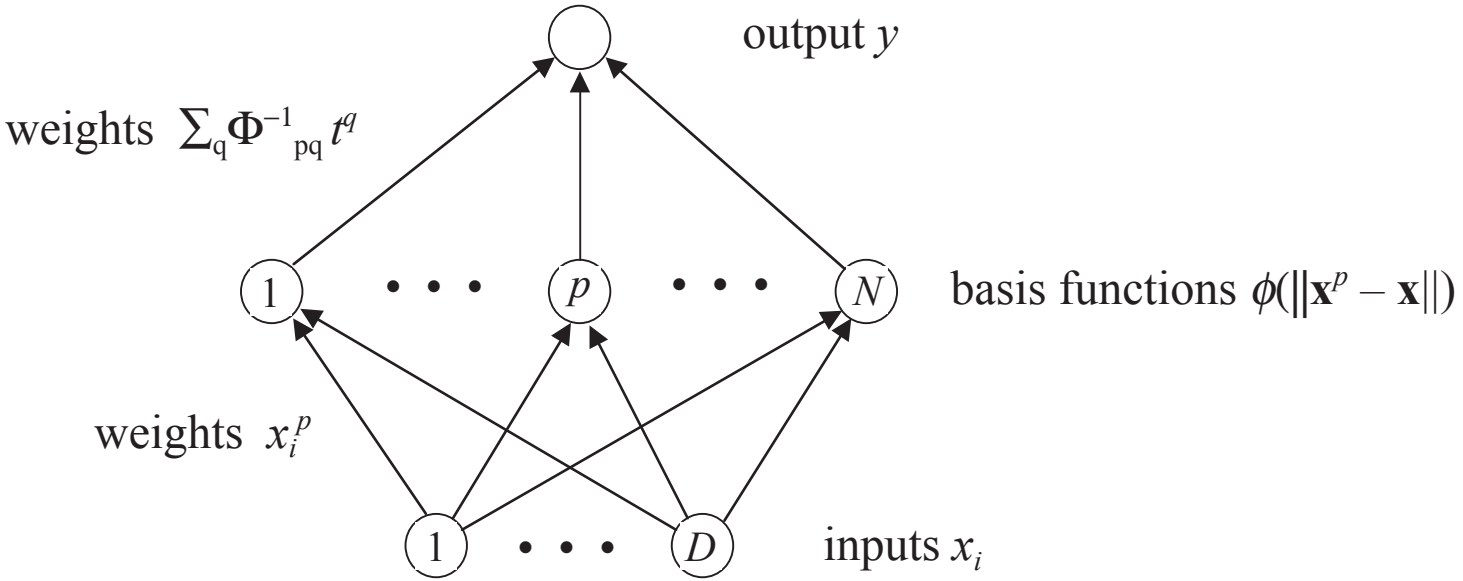
$$\phi(r) \to \infty \quad \text{as} \quad |\mathrm{r}| \to \infty$$

Note that even the Linear Function $\phi(r) = r = \left\| \mathbf{x} - \mathbf{x}^p \right\|$ is still non-linear in the components of $\mathbf{x}$. In one dimension, this leads to a piecewise-linear interpolating function which performs the simplest form of exact interpolation.

For neural network mappings, there are good reasons for preferring localised basis functions. We shall focus our attention on ***Gaussian basis functions*** since, as well as being localised, they have a number of other useful analytic properties. We can also see intuitively how to set their widths σ and build up function approximations with them.

Radial Basis Function Networks

You might think that what we have just described isn't really a neural network. And a lot of people would agree with you! However, we can see how to make it look like one:



Note that the N training patterns $\{ x_i^p, t^p \}$ determine the weights directly. The hidden layer to output weights multiply the hidden unit activations in the conventional manner, but the input to hidden layer weights are used in a very different fashion.

Problems with Exact Interpolation

We have seen how we can set up RBF networks that perform exact interpolation, but there are two serious problems with these exact interpolation networks:

1. They perform poorly with noisy data

As we have already seen for Multi-Layer Perceptrons (MLPs), we do not usually want the network's outputs to pass through all the data points when the data is noisy, because that will be a highly oscillatory function that will not provide good generalization.

2. They are not computationally efficient

The network requires one hidden unit (i.e. one basis function) for each training data pattern, and so for large data sets the network will become very costly to evaluate.

With MLPs we can improve generalization by using more training data – the opposite happens in RBF networks, and they take longer to compute as well.

Improving RBF Networks

We can take the basic structure of the RBF networks that perform exact interpolation and improve upon them in a number of ways:

1. The number M of basis functions (hidden units) need not equal the number N of training data points. In general it is better to have M much less than N.
2. The centres of the basis functions do not need to be defined as the training data input vectors. They can instead be determined by a training algorithm.
3. The basis functions need not all have the same width parameter σ. These can also be determined by a training algorithm.
4. We can introduce bias parameters into the linear sum of activations at the output layer. These will compensate for the difference between the average value over the data set of the basis function activations and the corresponding average value of the targets.

Of course, these will make analysing and optimising the network much more difficult.

The Improved RBF Network

When these changes are made to the exact interpolation formula, and we allow the possibility of more than one output unit, we arrive at the RBF network mapping

$$y_k(\mathbf{x}) = w_{k0} + \sum_{j=1}^{M} w_{kj}\phi_j(\mathbf{x})$$

which we can simplify by introducing an extra basis function $\phi_0 = 1$ to give

$$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x})$$

For the case of Gaussian basis functions we have

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \boldsymbol{\mu}_j\|^2}{2\sigma_j^2}\right)$$

in which we have $M \times D$ basis centres $\{\boldsymbol{\mu}_j\}$ and M widths $\{\sigma_j\}$. Next lecture we shall see how to determine all the network parameters $\{w_{kj}, \boldsymbol{\mu}_j, \sigma_j\}$.

Radial Basis Function Networks: Applications

1. Supervised RBF Network Training
2. Regularization Theory for RBF Networks
3. RBF Networks for Classification
4. The XOR Problem in RBF Form
5. Comparison of RBF Networks with MLPs
6. Real World Application – EEG Analysis

Supervised RBF Network Training

Supervised training of the basis function parameters will generally give better results than unsupervised procedures, but the computational costs are usually enormous.

The obvious approach is to perform gradient descent on a sum squared output error function as we did for our MLPs. The error function would be

$$E = \sum_p \sum_k (y_k(\mathbf{x}^p) - t_k^p)^2 = \sum_p \sum_k (\sum_{j=0}^{M} w_{kj}\phi_j\left(\mathbf{x}^p, \boldsymbol{\mu}_j, \sigma_j\right) - t_k^p)^2$$

and we would iteratively update the weights/basis function parameters using

$$\Delta w_{jk} = -\eta_w \frac{\partial E}{\partial w_{jk}} \qquad \Delta \mu_{ij} = -\eta_\mu \frac{\partial E}{\partial \mu_{ij}} \qquad \Delta \sigma_j = -\eta_\sigma \frac{\partial E}{\partial \sigma_i}$$

We have all the problems of choosing the learning rates η, avoiding local minima and so on, that we had for training MLPs by gradient descent. Also, there is a tendency for the basis function widths to grow large leaving non-localised basis functions.

Regularization Theory for RBF Networks

Instead of restricting the number of hidden units, an alternative approach for preventing overfitting in RBF networks comes from the theory of ***regularization***, which we saw previously was a method of controlling the smoothness of mapping functions.

We can have one basis function centred on each training data point as in the case of exact interpolation, but add an extra term to the error measure which penalizes mappings which are not smooth. If we have network outputs $y_k(\mathbf{x}^p)$ and sum squared error measure, we can introduce some appropriate differential operator $\mathbf{P}$ and write

$$E = \tfrac{1}{2}\sum_p \sum_k (y_k(\mathbf{x}^p) - t_k^p)^2 + \lambda \sum_k \int |\mathbf{P} y_k(\mathbf{x})|^2 \, d\mathbf{x}$$

where λ is the regularization parameter which determines the relative importance of smoothness compared with error. There are many possible forms for $\mathbf{P}$, but the general idea is that mapping functions $y_k(\mathbf{x})$ which have large curvature should have large values of $|\mathbf{P} y_k(\mathbf{x})|^2$ and hence contribute a large penalty in the total error function.

Computing the Regularized Weights

Provided the regularization term is quadratic in $y_k(\mathbf{x})$, the second layer weights can still be found by solving a set of linear equations. For example, the regularizer

$$\lambda \sum_k \left[\sum_p \sum_i \frac{1}{2} \left(\frac{\partial^2 y_k(\mathbf{x}^p)}{\partial x_i^{\,2}} \right)^2 \right]$$

certainly penalizes large output curvature, and minimizing the error function E now leads to linear equations for the output weights that are no harder to solve than we had before

$$\mathbf{M}\mathbf{W}^{\mathsf{T}} - \mathbf{\Phi}^{\mathsf{T}}\mathbf{T} = 0$$

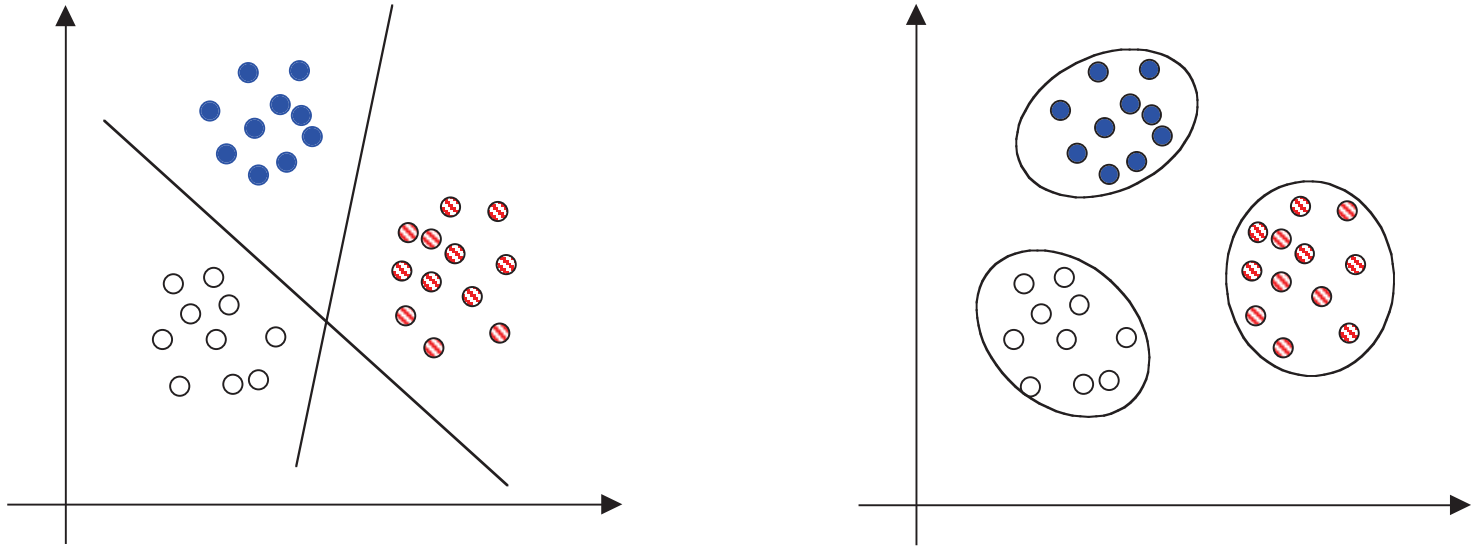
In this we have defined the same matrices with components $(\mathbf{W})_{kj} = w_{kj}$, $(\mathbf{\Phi})_{pj} = \phi_j(\mathbf{x}^p)$, and $(\mathbf{T})_{pk} = \{t_k^p\}$ as before, and also the regularized version of $\mathbf{\Phi}^{\mathsf{T}}\mathbf{\Phi}$

$$\mathbf{M} = \mathbf{\Phi}^{\mathsf{T}}\mathbf{\Phi} + \lambda \sum_i \frac{\partial^2 \mathbf{\Phi}^{\mathsf{T}}}{\partial x_i^{\,2}} \frac{\partial^2 \mathbf{\Phi}}{\partial x_i^{\,2}}$$

Clearly for $\lambda = 0$ this reduces to the un-regularized result we derived in the last lecture.

RBF Networks for Classification

So far we have concentrated on RBF networks for function approximation. They are also useful for classification problems. Consider a data set that falls into three classes:



An MLP would naturally separate the classes with hyper-planes in the input space (as on the left). An alternative approach would be to model the separate class distributions by localised radial basis functions (as on the right).

Implementing RBF Classification Networks

In principle, it is easy to have an RBF network perform classification – we simply need to have an output function $y_k(\mathbf{x})$ for each class k with appropriate targets

$$t_k^p = \begin{cases} 1 & \text{if pattern } p \text{ belongs to class } k \\ 0 & \text{otherwise} \end{cases}$$

and, when the network is trained, it will automatically classify new patterns.

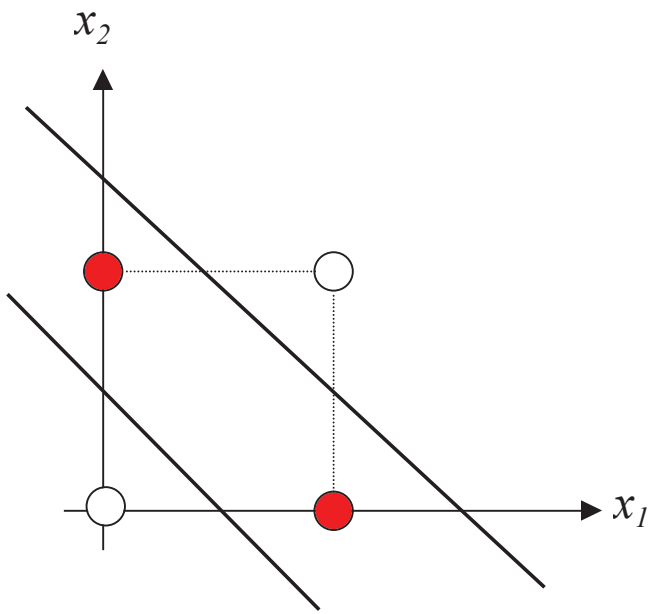
The underlying justification is found in ***Cover's theorem*** which states that "A complex pattern classification problem cast in a high dimensional space non-linearly is more likely to be linearly separable than in a low dimensional space". We know that once we have linear separable patterns, the classification problem is easy to solve.

In addition to the RBF network outputting good classifications, it can be shown that the outputs of such a regularized RBF network classifier will also provide estimates of the ***posterior class probabilities***.

The XOR Problem Revisited

We are already familiar with the non-linearly separable XOR problem:

p	x_1	x_2	t
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0



We know that Single Layer Perceptrons with step or sigmoidal activation functions cannot generate the right outputs, because they are only able to form a single decision boundary. To deal with this problem using Perceptrons we needed to either change the activation function, or introduce a non-linear hidden layer to give an Multi Layer Perceptron (MLP).

The XOR Problem in RBF Form

Recall that sensible RBFs are M Gaussians $\phi_j(\mathbf{x})$ centred at random training data points:

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{M}{d_{\max}^2}\left\|\mathbf{x}-\boldsymbol{\mu}_j\right\|^2\right) \qquad \text{where} \quad \{\boldsymbol{\mu}_j\} \subset \{\mathbf{x}^p\}$$

To perform the XOR classification in an RBF network, we start by deciding how many basis functions we need. Given there are four training patterns and two classes, $M = 2$ seems a reasonable first guess. We then need to decide on the basis function centres. The two separated zero targets seem a good bet, so we can set $\boldsymbol{\mu}_1 = (0,0)$ and $\boldsymbol{\mu}_2 = (1,1)$ and the distance between them is $d_{max} = \sqrt{2}$. We thus have the two basis functions

$$\phi_1(\mathbf{x}) = \exp\left(-\left\|\mathbf{x}-\boldsymbol{\mu}_1\right\|^2\right) \qquad \text{with} \quad \boldsymbol{\mu}_1 = (0,0)$$

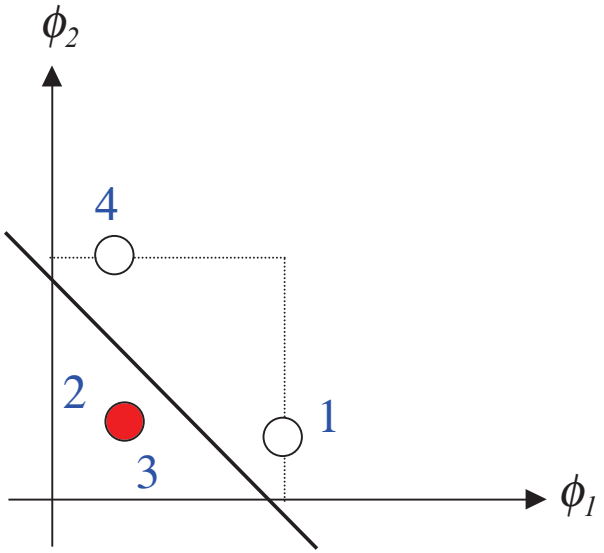
$$\phi_2(\mathbf{x}) = \exp\left(-\left\|\mathbf{x}-\boldsymbol{\mu}_2\right\|^2\right) \qquad \text{with} \quad \boldsymbol{\mu}_2 = (1,1)$$

This will hopefully transform the problem into a linearly separable form.

The XOR Problem Basis Functions

Since the hidden unit activation space is only two dimensional we can easily plot how the input patterns have been transformed:

p	x_1	x_2	ϕ_1	ϕ_2
1	0	0	1.0000	0.1353
2	0	1	0.3678	0.3678
3	1	0	0.3678	0.3678
4	1	1	0.1353	1.0000



We can see that the patterns are now linearly separable. Note that in this case we did not have to increase the dimensionality from the input space to the hidden unit/basis function space – the non-linearity of the mapping was sufficient. **Exercise**: check what happens if you chose two different basis function centres.

The XOR Problem Output Weights

In this case we just have one output $y(\mathbf{x})$, with one weight w_j to each hidden unit j, and one bias $-\theta$. This gives us the network's input-output relation for each input pattern $\mathbf{x}$

$$y(\mathbf{x}) = w_1\phi_1(\mathbf{x}) + w_2\phi_2(\mathbf{x}) - \theta$$

Then, if we want the outputs $y(\mathbf{x}^p)$ to equal the targets t^p, we get the four equations

$$\begin{aligned}
1.0000w_1 + 0.1353w_2 - 1.0000\theta &= 0 \\
0.3678w_1 + 0.3678w_2 - 1.0000\theta &= 1 \\
0.3678w_1 + 0.3678w_2 - 1.0000\theta &= 1 \\
0.1353w_1 + 1.0000w_2 - 1.0000\theta &= 0
\end{aligned}$$

Three are different, and we have three variables, so we can easily solve them to give

$$w_1 = w_2 = -2.5018 \quad , \quad \theta = -2.8404$$

This completes our "training" of the RBF network for the XOR problem.

Comparison of RBF Networks with MLPs

There are clearly a number of similarities between RBF networks and MLPs:

Similarities

1. They are both non-linear feed-forward networks
2. They are both universal approximators
3. They are used in similar application areas

It is not surprising, then, to find that there always exists an RBF network capable of accurately mimicking a specified MLP, or vice versa. However the two networks do differ from each other in a number of important respects:

Differences

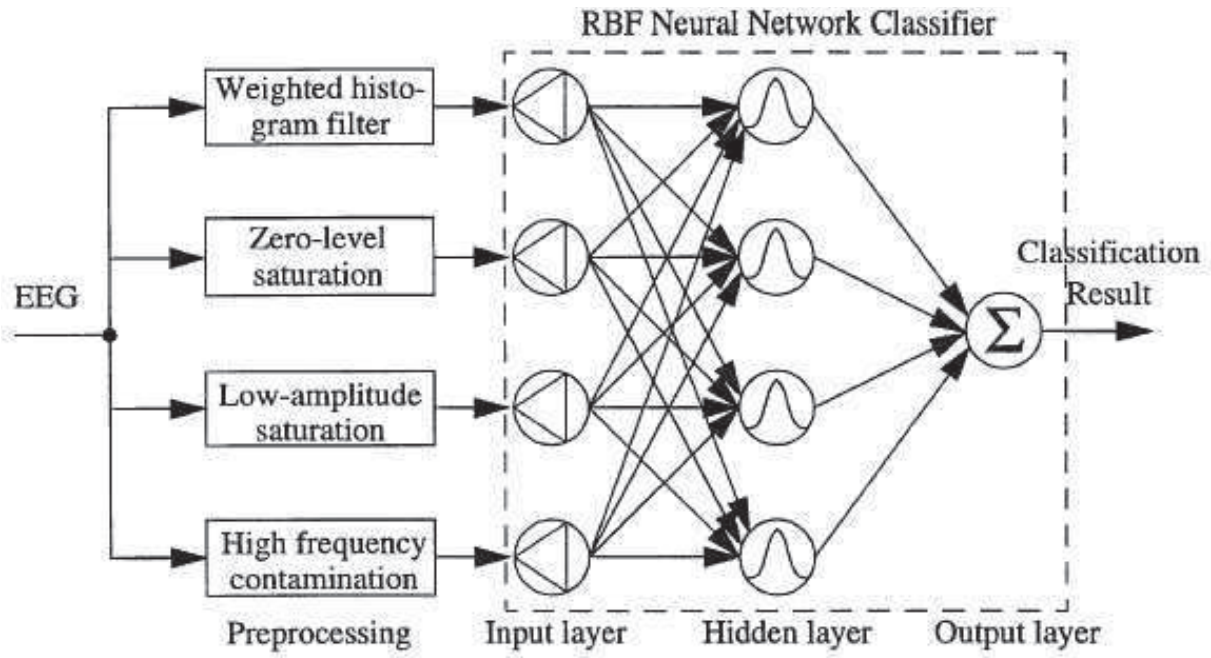
1. An RBF network (in its natural form) has a single hidden layer, whereas MLPs can have any number of hidden layers.

2. RBF networks are usually fully connected, whereas it is common for MLPs to be only partially connected.
3. In MLPs the computation nodes (processing units) in different layers share a common neuronal model, though not necessarily the same activation function. In RBF networks the hidden nodes (basis functions) operate very differently, and have a very different purpose, to the output nodes.
4. In RBF networks, the argument of each hidden unit activation function is the ***distance*** between the input and the "weights" (RBF centres), whereas in MLPs it is the ***inner product*** of the input and the weights.
5. MLPs are usually trained with a single global supervised algorithm, whereas RBF networks are usually trained one layer at a time with the first layer unsupervised.
6. MLPs construct ***global*** approximations to non-linear input-output mappings with ***distributed*** hidden representations, whereas RBF networks tend to use ***localised*** non-linearities (Gaussians) at the hidden layer to construct ***local*** approximations.

Although, for approximating non-linear input-output mappings, the RBF networks can be trained much faster, MLPs may require a smaller number of parameters.

Real World Application – EEG Analysis

One successful RBF network detects epileptiform artefacts in EEG recordings:



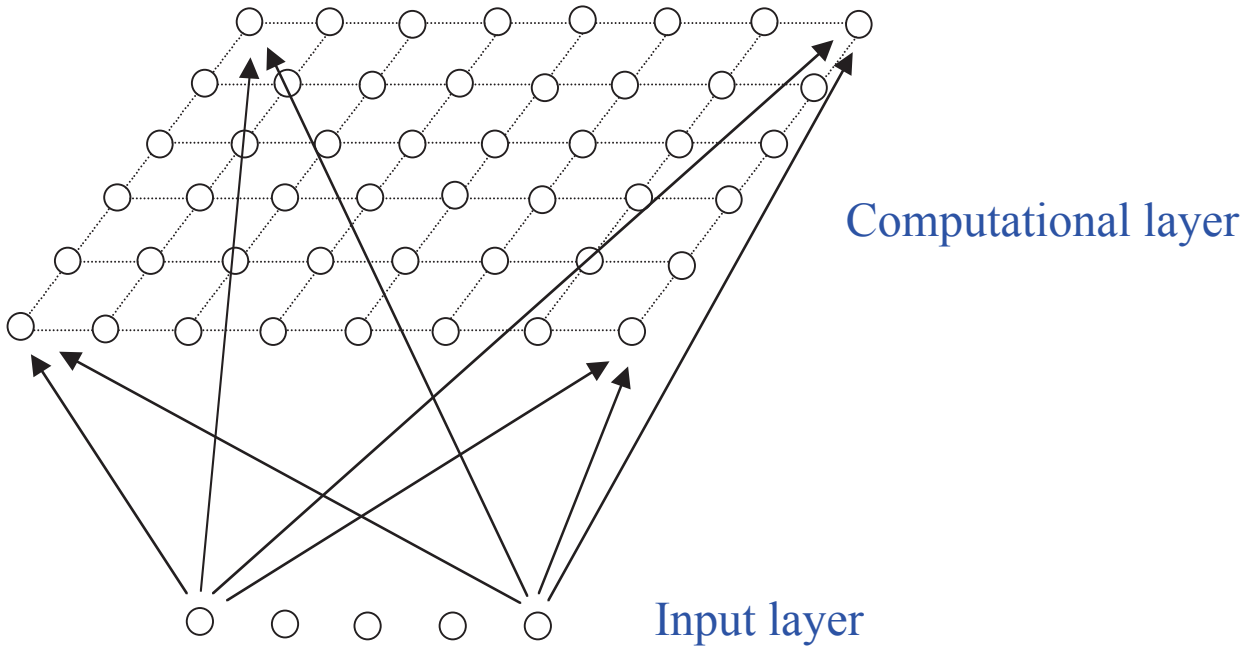
For full details see the original paper by: A. Saastamoinen, T. Pietilä, A. Värri, M. Lehtokangas, & J. Saarinen, (1998). Waveform detection with RBF network – Application to automated EEG analysis. *Neurocomputing,* vol. **20**, pp. 1-13

Self Organizing Maps: Algorithms and Applications

1. The SOM Architecture and Algorithm
2. Properties of the Feature Map
    1. Approximation of the Input Space
    2. Topological Ordering
    3. Density Matching
    4. Feature Selection
3. Application: The Phonetic Typewriter

The Architecture a Self Organizing Map

We shall concentrate on the SOM system known as a ***Kohonen Network***. This has a feed-forward structure with a single computational layer of neurons arranged in rows and columns. Each neuron is fully connected to all the source units in the input layer:



A one dimensional map will just have a single row or column in the computational layer.

The SOM Algorithm

The aim is to learn a ***feature map*** from the spatially ***continuous input space***, in which our input vectors live, to the low dimensional spatially ***discrete output space*****,** which is formed by arranging the computational neurons into a grid.

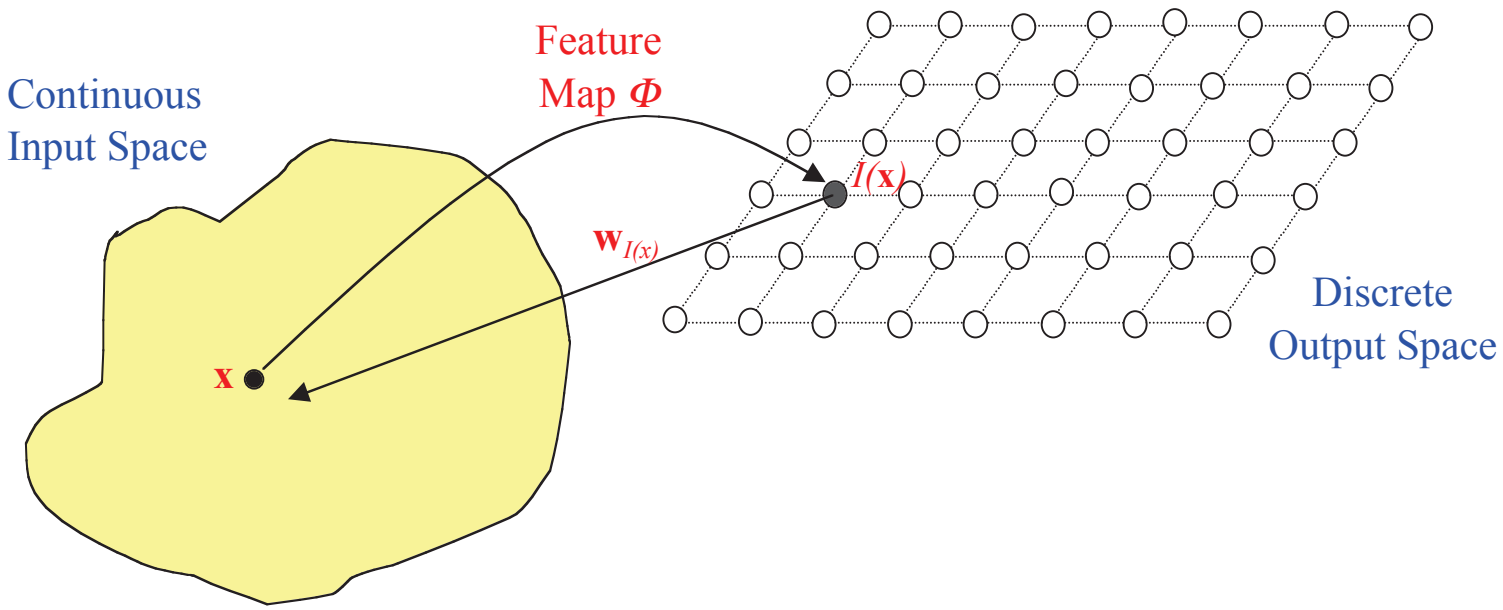
The stages of the SOM algorithm that achieves this can be summarised as follows:

1. ***Initialization*** – Choose random values for the initial weight vectors $\mathbf{w}_j$.
2. ***Sampling*** – Draw a sample training input vector $\mathbf{x}$ from the input space.
3. ***Matching*** – Find the winning neuron $I(\mathbf{x})$ that has weight vector closest to the input vector, i.e. the minimum value of $d_j(\mathbf{x}) = \sum_{i=1}^{D}(x_i - w_{ji})^2$.
4. ***Updating*** – Apply the weight update equation $\Delta w_{ji} = \eta(t)\, T_{j,I(\mathbf{x})}(t)\, (x_i - w_{ji})$ where $T_{j,I(\mathbf{x})}(t)$ is a Gaussian neighbourhood and $\eta(t)$ is the learning rate.
5. ***Continuation*** – keep returning to step 2 until the feature map stops changing.

We shall now explore the properties of the feature map and look at some examples.

Properties of the Feature Map

Once the SOM algorithm has converged, the feature map displays important statistical characteristics of the input space. Given an input vector $\mathbf{x}$, the feature map Φ provides a winning neuron $I(\mathbf{x})$ in the output space, and the weight vector $\mathbf{w}_{I(\mathbf{x})}$ provides the coordinates of the image of that neuron in the input space.



The feature map has four important properties that we shall look at in turn:

Property 1 : Approximation of the Input Space

The feature map Φ represented by the set of weight vectors $\{\mathbf{w}_i\}$ in the output space, provides a good approximation to the input space.

We can state the aim of the SOM as storing a large set of input vectors $\{\mathbf{x}\}$ by finding a smaller set of prototypes $\{\mathbf{w}_i\}$ so as to provide a good approximation to the original input space. The theoretical basis of this idea is rooted in ***vector quantization theory***, the motivation of which is dimensionality reduction or data compression.

In effect, the goodness of the approximation is given by the total squared distance

$$D = \sum_{\mathbf{x}} \left\| \mathbf{x} - \mathbf{w}_{I(\mathbf{x})} \right\|^2$$

which we wish to minimize. If we work through gradient descent style mathematics we do end up with the SOM weight update algorithm, which confirms that it is generating a good approximation to the input space.

Property 2 : Topological Ordering

The feature map Φ computed by the SOM algorithm is topologically ordered in the sense that the spatial location of a neuron in the output lattice/grid corresponds to a particular domain or feature of the input patterns.

The topological ordering property is a direct consequence of the weight update equation that forces the weight vector $\mathbf{w}_{I(\mathbf{x})}$ of the winning neuron $I(\mathbf{x})$ to move toward the input vector $\mathbf{x}$. The crucial factor is that the weight updates also move the weight vectors $\mathbf{w}_j$ of the closest neighbouring neurons j along with the winning neuron $I(\mathbf{x})$. Together these weight changes cause the whole output space to become appropriately ordered.

We can visualise the feature map Φ as an ***elastic or virtual net*** with a grid like topology. Each output node can be represented in the input space at coordinates given by their weights. Then if the neighbouring nodes in output space have their corresponding points in input space connected together, the resulting image of the output grid reveals directly the topological ordering at each stage of the network training.

Property 3 : Density Matching

The feature map Φ reflects variations in the statistics of the input distribution: regions in the input space from which the sample training vectors $\mathbf{x}$ are drawn with high probability of occurrence are mapped onto larger domains of the output space, and therefore with better resolution than regions of input space from which training vectors are drawn with low probability.

We need to relate the input vector probability distribution $p(\mathbf{x})$ to the ***magnification factor*** $m(\mathbf{x})$ of the feature map. Generally, for two dimensional feature maps the relation cannot be expressed as a simple function, but in one dimension we can show that

$$m(\mathbf{x}) \propto p^{2/3}(\mathbf{x})$$

So the SOM algorithm doesn't match the input density exactly, because of the power of 2/3 rather than 1. Computer simulations indicate similar approximate density matching in general, always with the low input density regions slightly over-represented.

Property 4 : Feature Selection

Given data from an input space with a non-linear distribution, the self organizing map is able to select a set of best features for approximating the underlying distribution.

This property is a natural culmination of properties 1 through 3.

Remember how Principal Component Analysis (PCA) is able to compute the input dimensions which carry the most variance in the training data. It does this by computing the eigenvector associated with the largest eigenvalue of the correlation matrix.

PCA is fine if the data really does form a line or plane in input space, but if the data forms a curved line or surface (e.g. a semi-circle), linear PCA is no good, but a SOM will overcome the approximation problem by virtue of its topological ordering property.

The SOM provides a discrete approximation of finding so-called ***principal curves*** or ***principal surfaces***, and may therefore be viewed as a non-linear generalization of PCA.

Application: The Phonetic Typewriter

One of the earliest and well known applications of the SOM is the phonetic typewriter of Kohonen. It is set in the field of speech recognition, and the problem is to classify phonemes in real time so that they could be used to drive a typewriter from dictation.

The real speech signals obviously needed pre-processing before being applied to the SOM. A combination of filtering and Fourier transforming of data sampled every 9.83 ms from spoken words provided a set of 16 dimensional spectral vectors. These formed the input space of the SOM, and the output space was an 8 by 12 grid of nodes.

Though the network was effectively trained on time-sliced speech waveforms, the output nodes became sensitised to phonemes and the relations between, because the network inputs were real speech signals which are naturally clustered around ideal phonemes. As a spoken word is processed, a path through output space maps out a phonetic transcription of the word. Some post-processing was required because phonemes are typically 40-400 ms long and span many time slices, but the system was surprisingly good at producing sensible strings of phonemes from real speech.