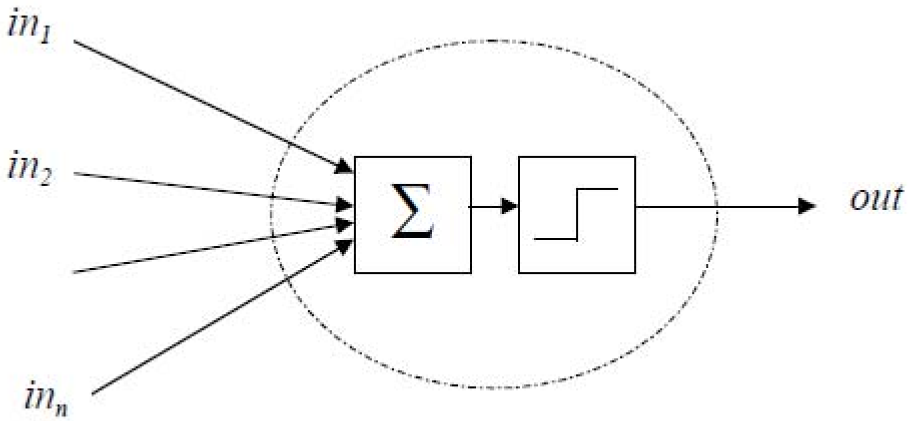
Neural Network Lectures

ME 60033 IMS 2015-16

C S Kumar

The McCulloch-Pitts Neuron

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



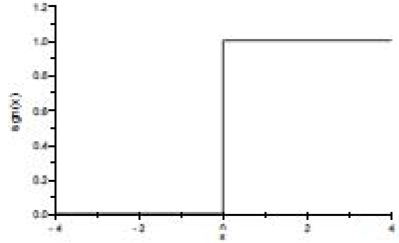
1. A set of synapses (i.e. connections) brings in activations from other neurons.
2. A processing unit sums the inputs, and then applies a non-linear activation function (i.e. squashing/transfer/threshold function).
3. An output line transmits the result to other neurons.

Some Useful Functions

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

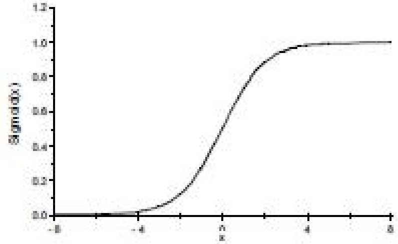
Example 1 The threshold or sign function $\mathrm{sgn}(x)$ is defined as

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



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

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



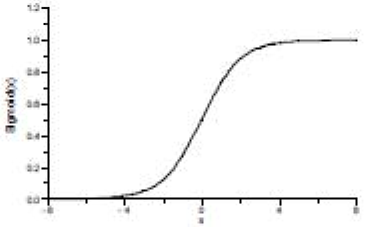
This is a smoothed (differentiable) form of the threshold function.

Other Types of Activation/Transfer Function

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

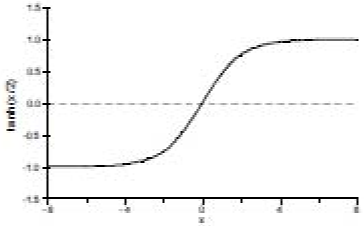
The logistic function

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



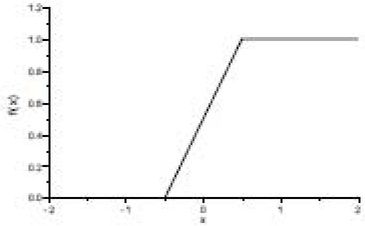
Hyperbolic tangent

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



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 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 :

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

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

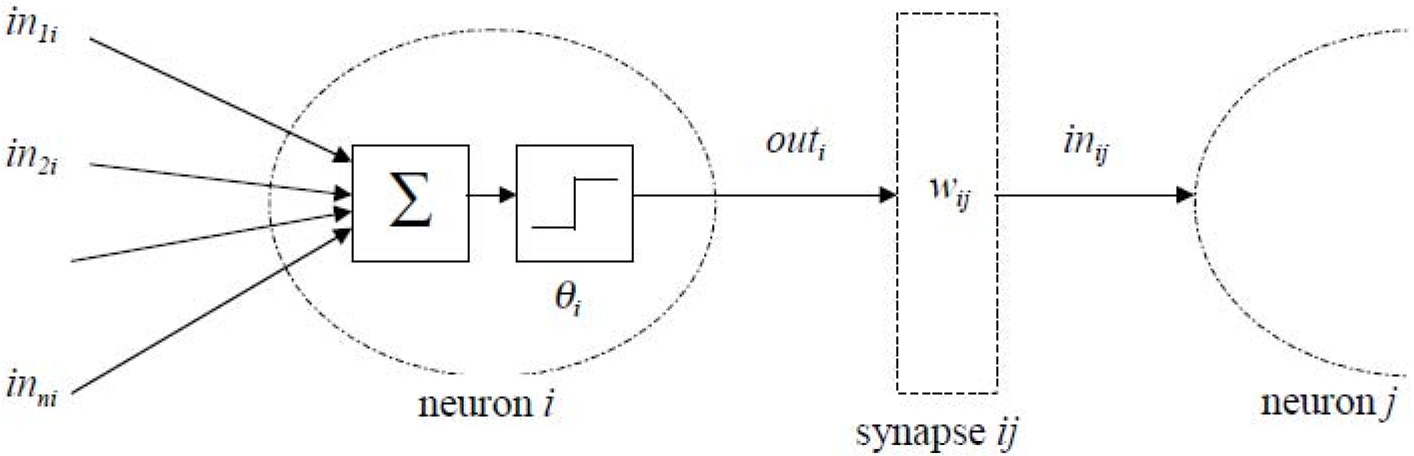
$$out = 1 \quad \text{if } \sum_{k=1}^{n} in_k \geq \theta \qquad\qquad out = 0 \quad \text{if } \sum_{k=1}^{n} in_k < \theta$$

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 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}:

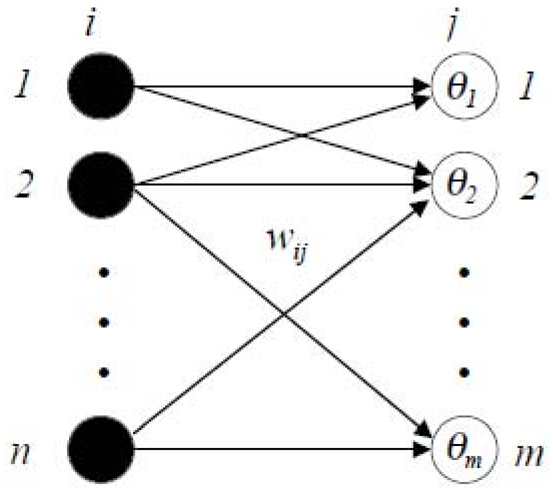


$$in_{ki} = out_k w_{ki} \qquad out_i = \operatorname{sgn}\left(\sum_{k=1}^{n} in_{ki} - \theta_i\right) \qquad in_{ij} = out_i w_{ij}$$

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.

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:

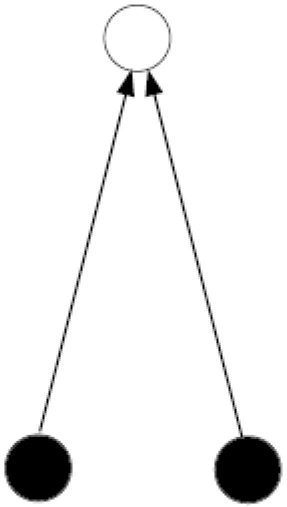
Single-Layer Feed-forward NNs One input layer and one output layer of processing units. No feed-back connections. (For example, a simple Perceptron.)

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.)

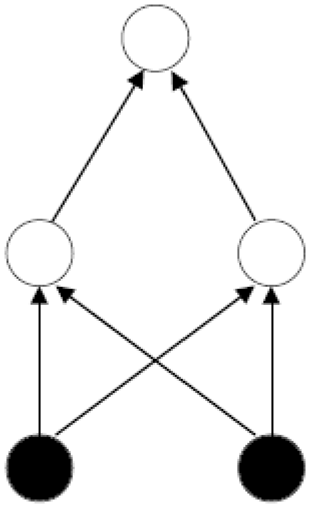
Examples of Network Architectures

Single Layer Feed-forward



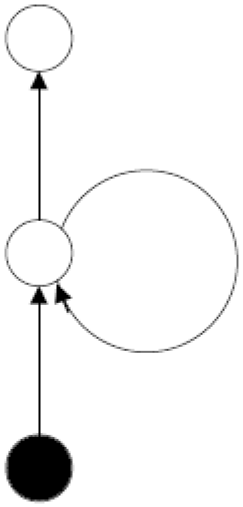
Single-Layer Perceptron

Multi-Layer Feed-forward



Multi-Layer Perceptron

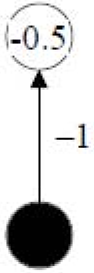
Recurrent Network



Simple Recurrent Network

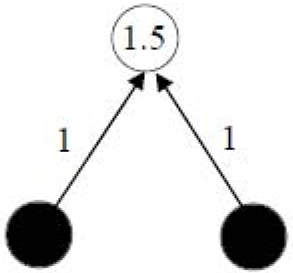
NOT

in	*out*
0	1
1	0



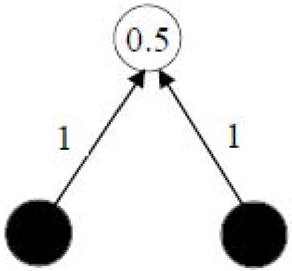
AND

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



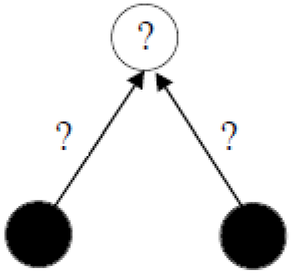
OR

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



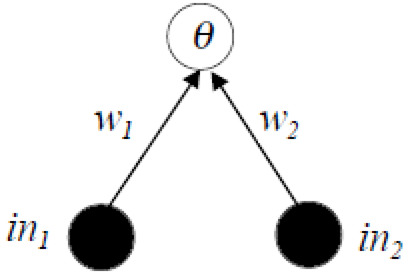
XOR

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



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:



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

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:

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

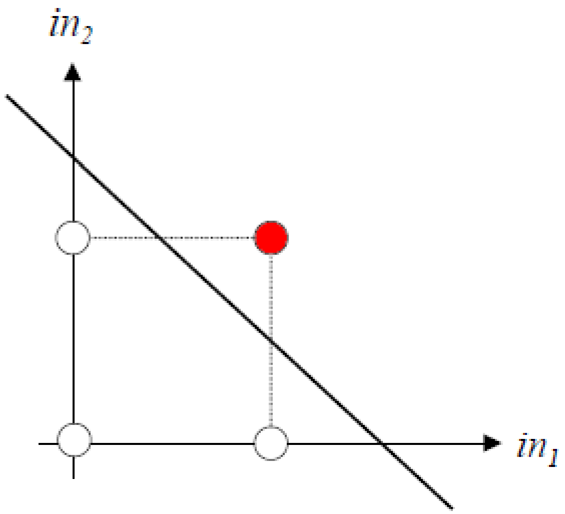
So, in two dimensions the decision boundaries are always straight lines.

Decision Boundaries for AND and OR

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

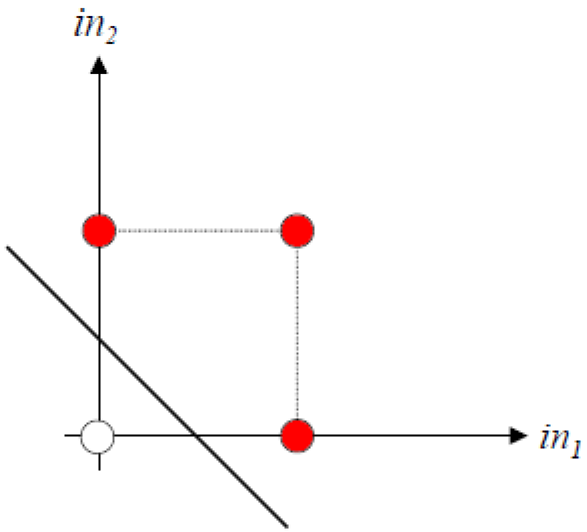
AND

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



OR

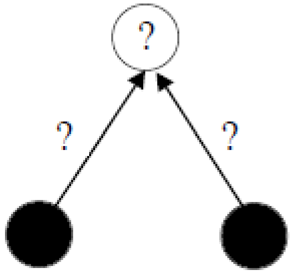
$w_1 = 1, \ w_2 = 1, \ \theta = 0.5$

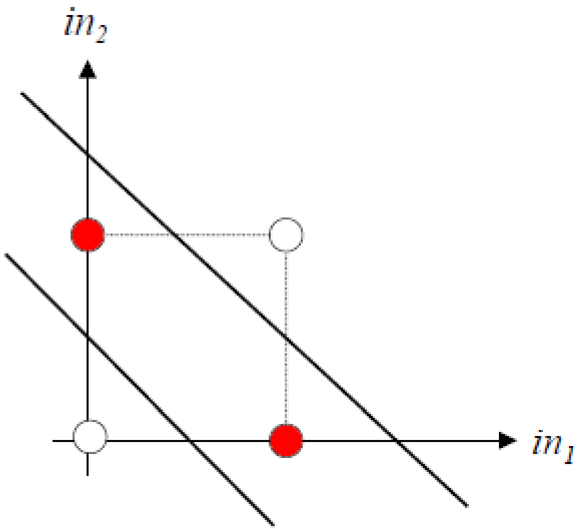


The extent to which we can change the weights and thresholds without changing the output decisions is now clear.

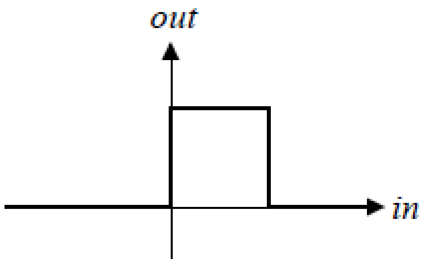
XOR

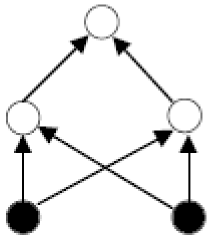
in_1	in_2	*out*
0	0	0
0	1	1
1	0	1
1	1	0





⇒
e.g.





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:

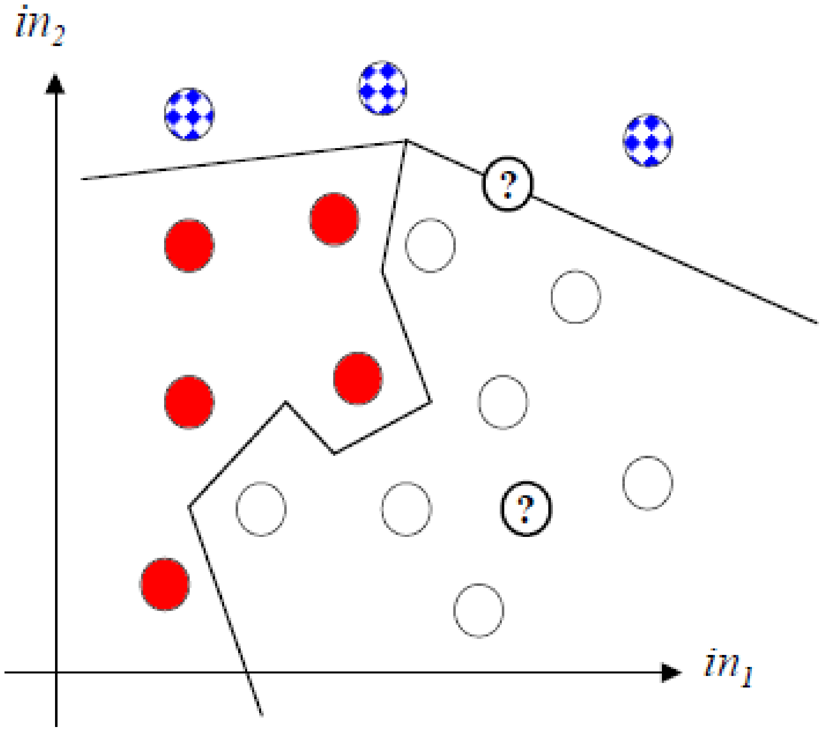
$$w_1 in_1 + w_2 in_2 + \ldots + w_n in_n - \theta = 0$$

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.

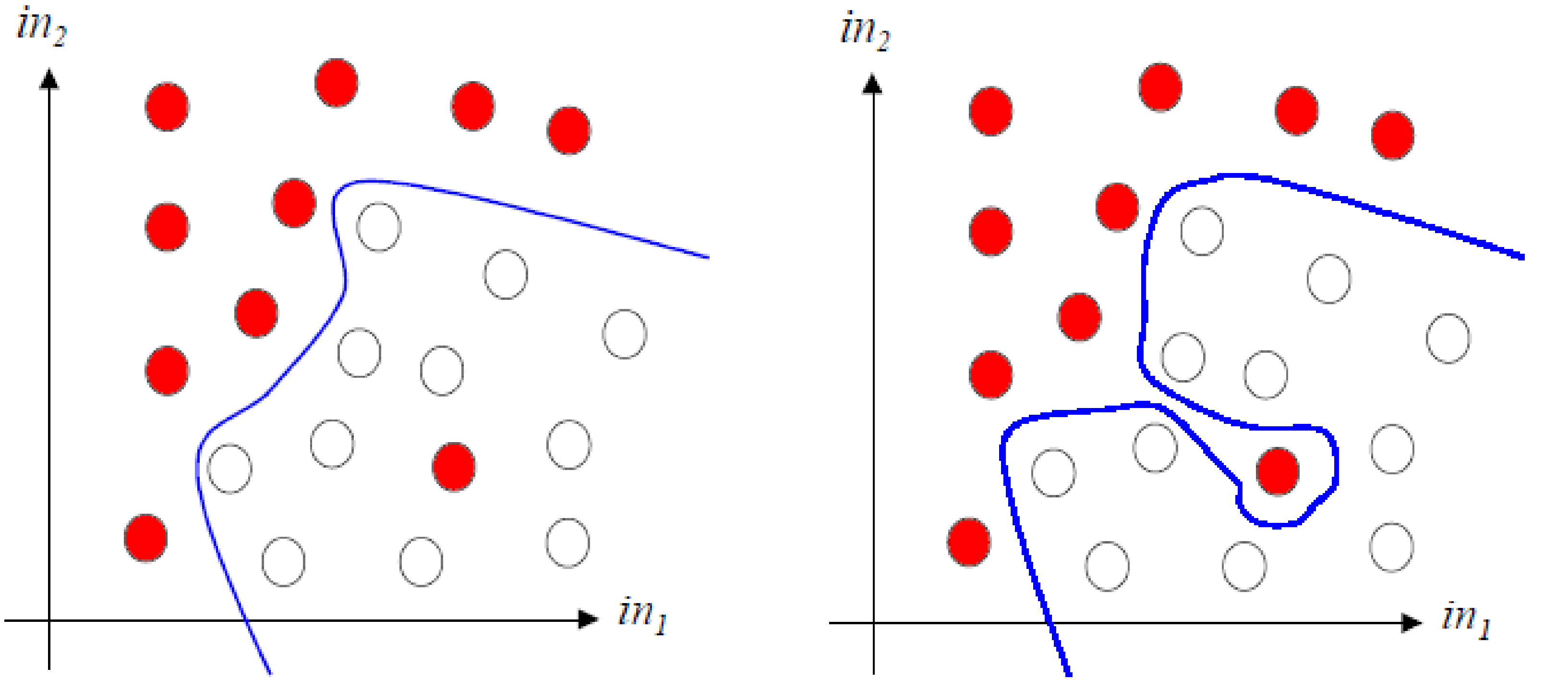
Generalization After training, the network must also be able to generalize, i.e. correctly classify ***test patterns*** it has never seen before.

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

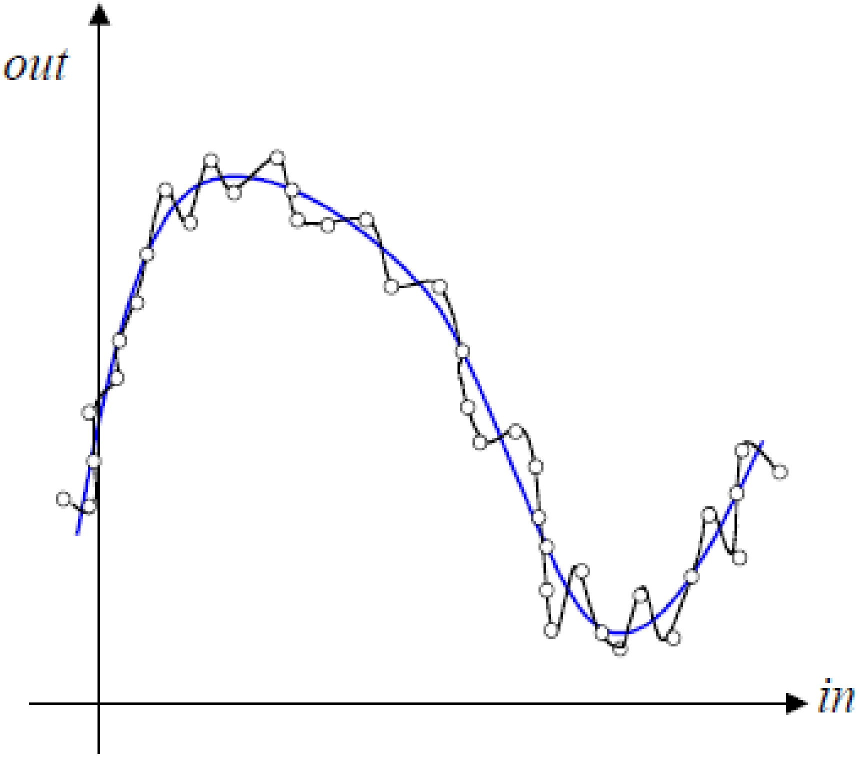
Suppose the task of our network is to learn a classification decision boundary:



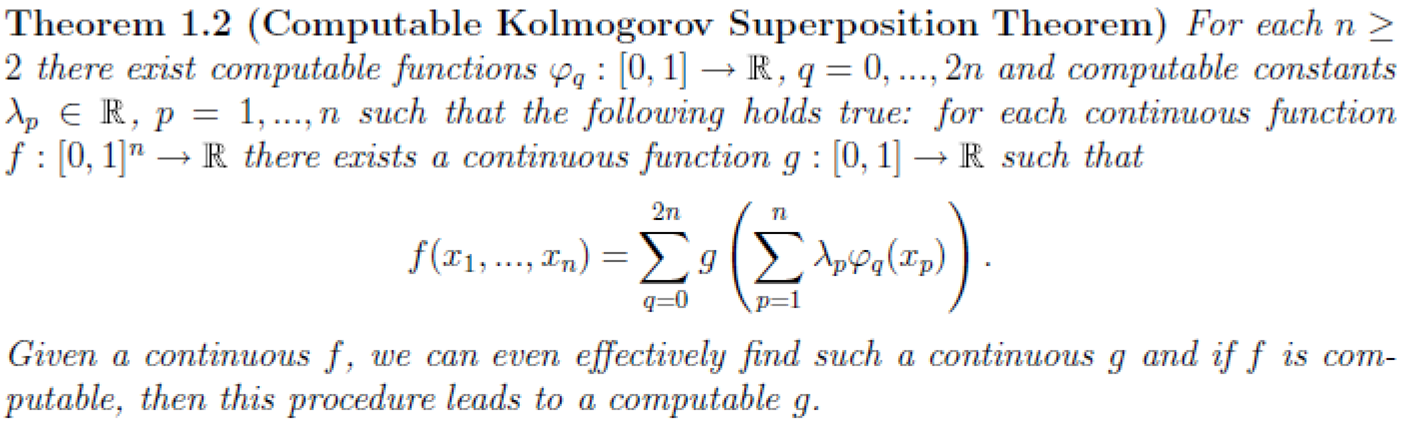
Our aim is for the network to generalize to classify new inputs appropriately. If we know that the training data contains noise, we don't necessarily want the training data to be classified totally accurately, as that is likely to reduce the generalization ability.

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.



Theorem 1.2 (Computable Kolmogorov Superposition Theorem) *For each* $n \geq 2$ *there exist computable functions* $\varphi_q : [0,1] \to \mathbb{R}$, $q = 0, \ldots, 2n$ *and computable constants* $\lambda_p \in \mathbb{R}$, $p = 1, \ldots, n$ *such that the following holds true: for each continuous function* $f : [0,1]^n \to \mathbb{R}$ *there exists a continuous function* $g : [0,1] \to \mathbb{R}$ *such that*

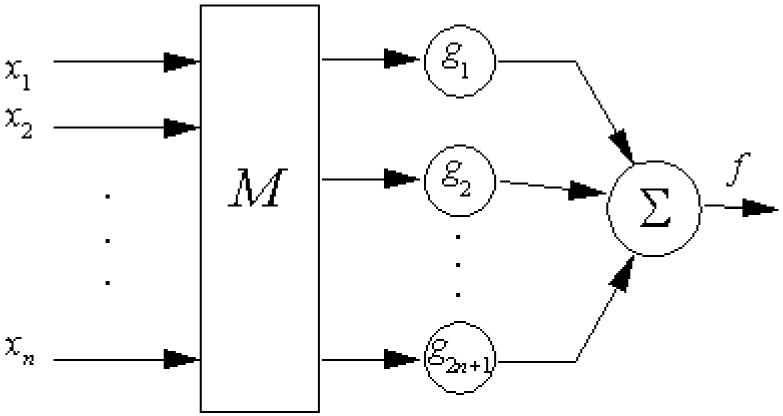
$$f(x_1, \ldots, x_n) = \sum_{q=0}^{2n} g\left(\sum_{p=1}^{n} \lambda_p \varphi_q(x_p)\right).$$

Given a continuous f, *we can even effectively find such a continuous* g *and if* f *is computable, then this procedure leads to a computable* g.

Hecht Nelson 1987,1990; Leppmann 1987 , Spreecher 1993

$$f(x_1,x_2,\ldots,x_n)= \sum_{j=1}^{2n+1} g_j\left(\sum_{i=1}^{n} \phi_{ij}(x_i)\right)$$

Multivariable function



Neural network representation

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.

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

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

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:

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

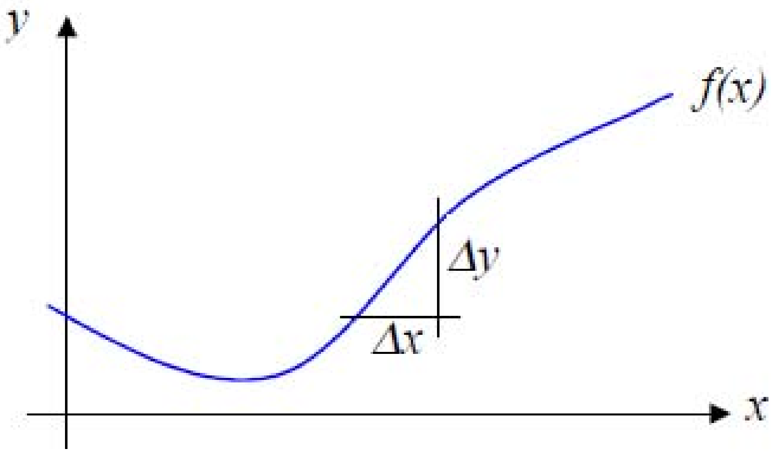
For obvious reasons this is known as the ***Sum Squared Error*** function. It is the total squared error summed over all output units j and all training patterns p.

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:

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

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

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

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

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

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

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

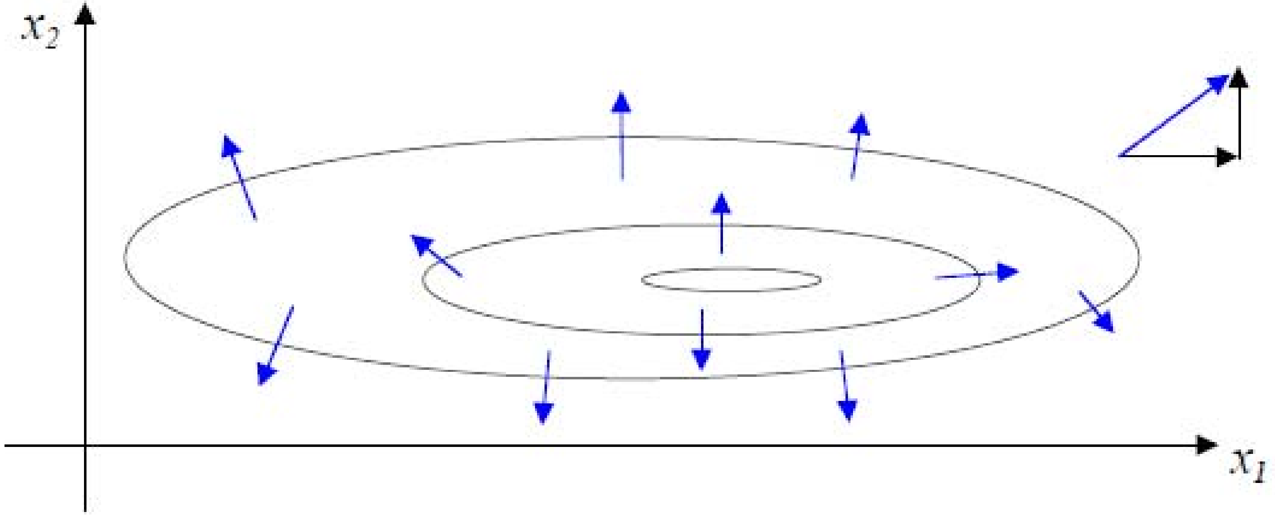
In summary, we can decrease *f(x)* by changing x by the amount:

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

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)*

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

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

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

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

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

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

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:

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

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

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

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

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

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

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

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

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

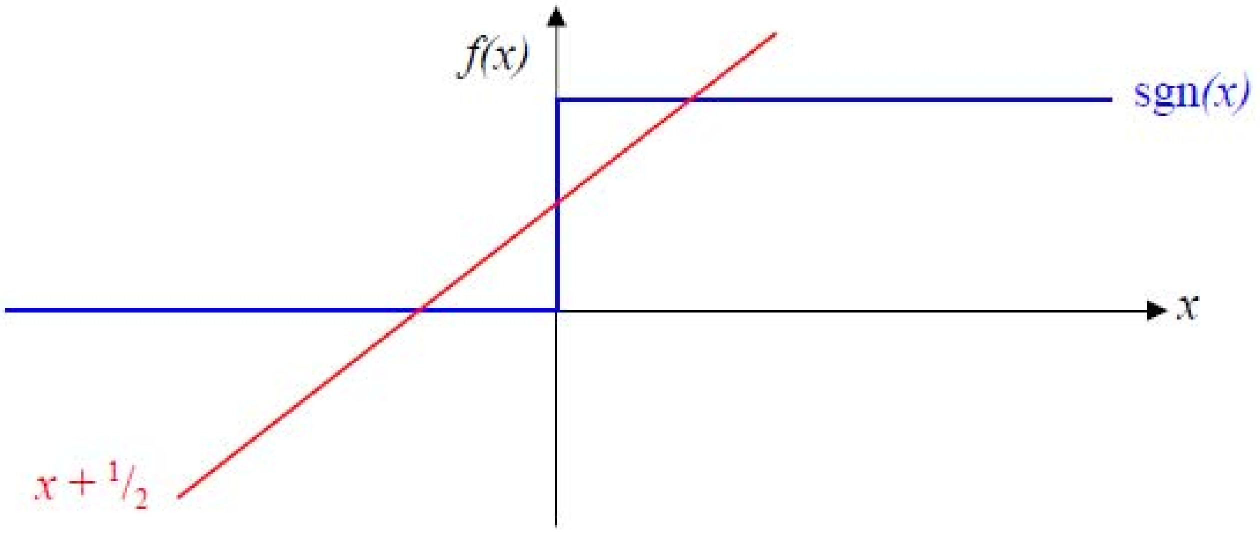
The *prime notation* is defined such that *f′* is the derivative of *f*. We have also used the *Kronecker Delta* symbol δ_{ij} defined such that $\delta_{ij} = 1$ when $i = j$ and $\delta_{ij} = 0$ when $i \neq j$.

The Delta Rule

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

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

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:

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

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

$$\delta_l = targ_l - out_l$$

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.

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.

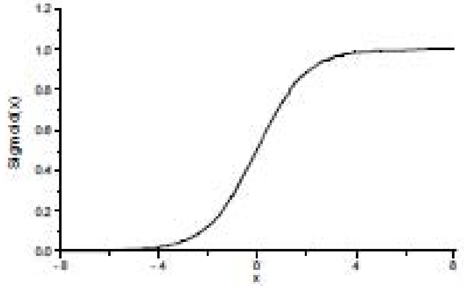
6. Repeat step 5 until the network error function is 'small enough'.

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:

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

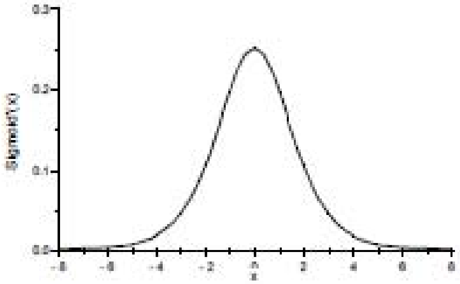


We can use the chain rule by putting $f(x) = g(h(x))$ with $g(h) = h^{-1}$ and $h(x) = 1 + \mathrm{e}^{-x}$ so

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

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

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



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

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

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

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

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?

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

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

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:

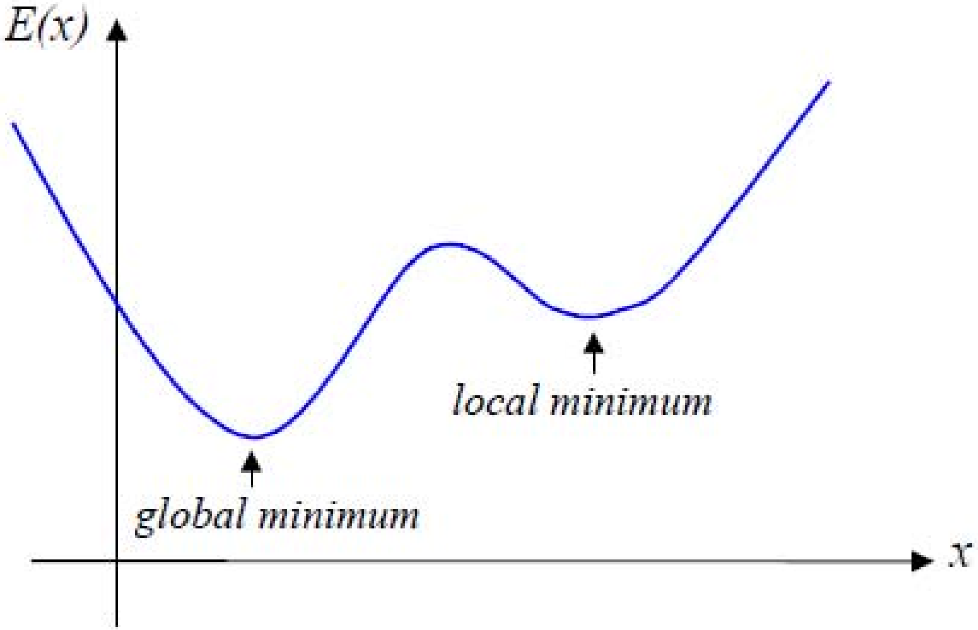
Target Off-sets Use targets of 0.1 and 0.9 (say) instead of 0 and 1. The sigmoids will no longer saturate and the learning will no longer get stuck.

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.