QUIZ 1

McCulloch-Pitts nöronu için

A- yapısını çizerek gösterin

B- Çıkış fonksiyonu nasıl hesaplanır ?

C- Hangi problemleri çözebilir/çözemez ?

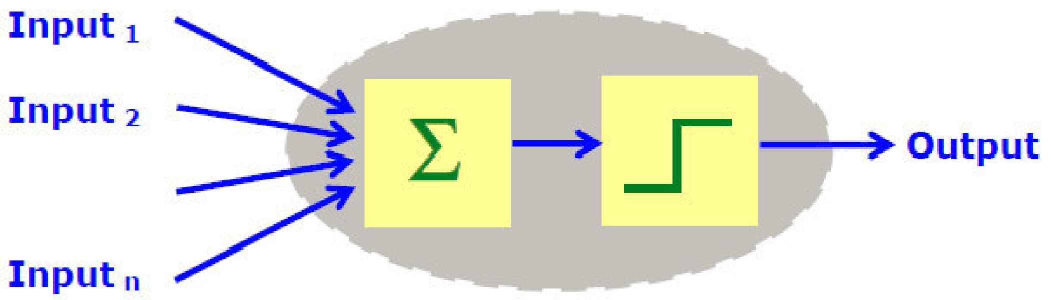
McCulloch-Pitts Neuron

A very simplified model of real neurons is known as a Threshold Logic Unit (TLU). The model is said to have :

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

McCulloch-Pitts (M-P) Neuron Equation

McCulloch-Pitts neuron is a simplified model of real biological neuron.



Simplified Model of Real Neuron (Threshold Logic Unit)

The equation for the output of a McCulloch-Pitts neuron as a function of **1** to **n** inputs is written as

$$\text{Output} = \text{sgn}\left(\sum_{i=1}^{n} \text{Input}_i - \Phi\right)$$

where Φ is the neuron's activation threshold.

$$\text{If } \sum_{i=1}^{n} \text{Input}_i \geq \Phi \text{ then Output} = 1$$

$$\text{If } \sum_{i=1}^{n} \text{Input}_i < \Phi \text{ then Output} = 0$$

In this McCulloch-Pitts neuron model, the missing features are :

- Non-binary input and output,
- Non-linear summation,
- Smooth thresholding,
- Stochastic, and
- Temporal information processing.

Artificial Neuron - Basic Elements

Neuron consists of three basic components - weights, thresholds, and a single activation function.

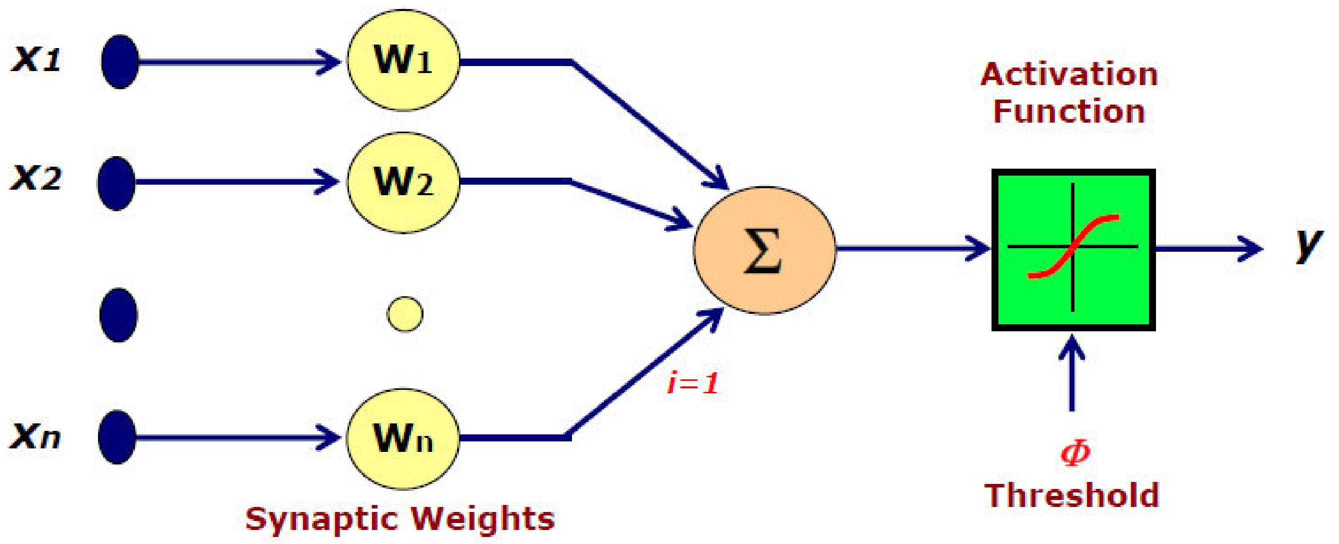


Fig Basic Elements of an Artificial Linear Neuron

- **Weighting Factors w**

  The values $w_1 , w_2 , . . . w_n$ are weights to determine the strength of input vector $X = [x_1 , x_2 , . . . , x_n]^T$. Each input is multiplied by the associated weight of the neuron connection $X^T W$. The +ve weight excites and the -ve weight inhibits the node output.

$$I = X^T.W = x_1 w_1 + x_2 w_2 + ... + x_n w_n = \sum_{i=1}^{n} x_i w_i$$

- **Threshold Φ**

The node's internal threshold Φ is the magnitude offset. It affects the activation of the node output **y** as:

$$Y = f(I) = f\left\{ \sum_{i=1}^{n} x_i w_i - \Phi_k \right\}$$

To generate the final output **Y** , the sum is passed on to a non-linear filter **f** called Activation Function or Transfer function or Squash function which releases the output **Y**.

- **Threshold for a Neuron**

  In practice, neurons generally do not fire (produce an output) unless their total input goes above a threshold value.

  The total input for each neuron is the sum of the weighted inputs to the neuron minus its threshold value. This is then passed through the sigmoid function. The equation for the transition in a neuron is :

  $a = 1/(1 + \exp(-x))$ where

  $x = \sum_i a_i w_i - Q$

a	is the activation for the neuron
a_i	is the activation for neuron *i*
w_i	is the weight
Q	is the threshold subtracted

- **Activation Function**

  An activation function **f** performs a mathematical operation on the signal output. The most common activation functions are:

  - Linear Function,
  - Threshold Function,
  - Piecewise Linear Function,
  - Sigmoidal (S shaped) function,
  - Tangent hyperbolic function

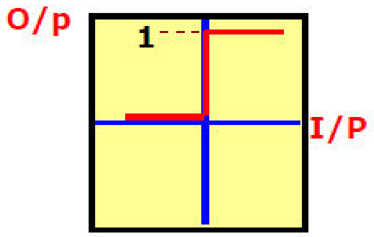
  The activation functions are chosen depending upon the type of problem to be solved by the network.

- **I/P** Horizontal axis shows sum of inputs .
- **O/P** Vertical axis shows the value the function produces ie output.
- All functions **f** are designed to produce values between **0** and **1**.

- **Threshold Function**

  A threshold (hard-limiter) activation function is either a binary type or a bipolar type as shown below.

binary threshold Output of a binary threshold function produces :

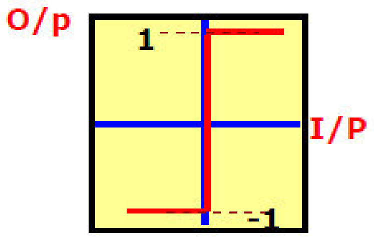


1 if the weighted sum of the inputs is +ve,

0 if the weighted sum of the inputs is -ve.

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

bipolar threshold Output of a bipolar threshold function produces :



1 if the weighted sum of the inputs is +ve,

-1 if the weighted sum of the inputs is -ve.

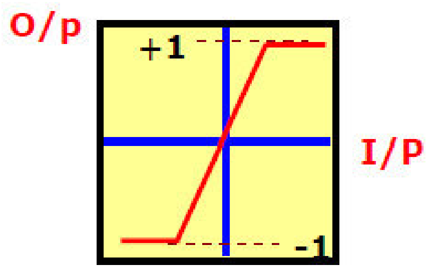
$$Y = f(I) = \begin{cases} 1 & \text{if } I \geq 0 \\ -1 & \text{if } I < 0 \end{cases}$$

Neuron with hard limiter activation function is called McCulloch-Pitts model.

- **Piecewise Linear Function**

  This activation function is also called saturating linear function and can have either a binary or bipolar range for the saturation limits of the output. The mathematical model for a symmetric saturation function is described below.

Piecewise Linear



This is a sloping function that produces :

-1 for a -ve weighted sum of inputs,

1 for a +ve weighted sum of inputs.

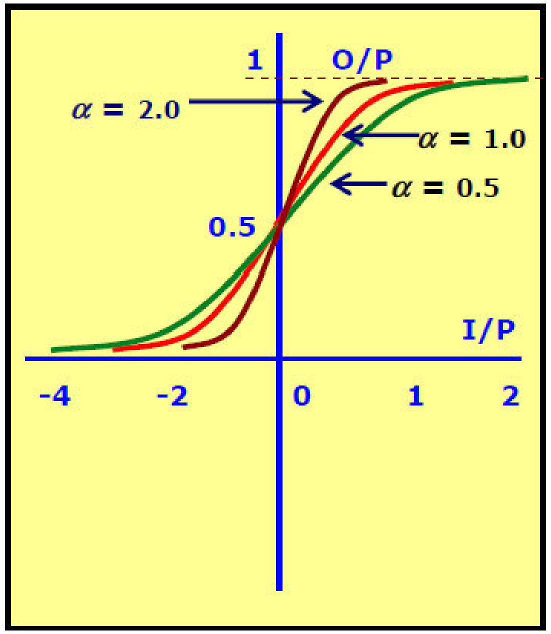
$\propto$ **I** proportional to input for values between **+1** and **-1** weighted sum,

$$Y = f(I) = \begin{cases} 1 & \text{if } I \geq 0 \\ I & \text{if } -1 \geq I \geq 1 \\ -1 & \text{if } I < 0 \end{cases}$$

Sigmoidal Function (S-shape function)

The nonlinear curved S-shape function is called the sigmoid function. This is most common type of activation used to construct the neural networks. It is mathematically well behaved, differentiable and strictly increasing function.

Sigmoidal function



A sigmoidal transfer function can be written in the form:

$$Y = f(I) = \frac{1}{1 + e^{-\alpha I}} \;, \quad 0 \le f(I) \le 1$$

$$= 1/(1 + \exp(-\alpha I)) \;, \quad 0 \le f(I) \le 1$$

This is explained as

$\approx$ **0** for large -ve input values,

1 for large +ve values, with

a smooth transition between the two.

α is slope parameter also called shape parameter; symbol the λ is also used to represented this parameter.

The sigmoidal function is achieved using exponential equation. By varying α different shapes of the function can be obtained which adjusts the abruptness of the function as it changes between the two asymptotic values.

Example :

The neuron shown consists of four inputs with the weights.

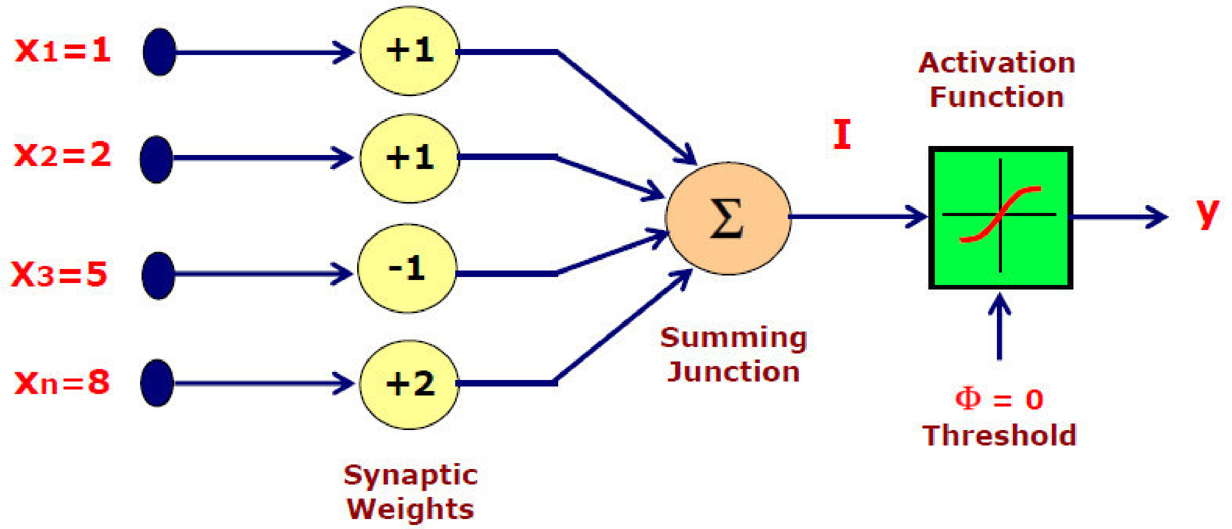


Fig Neuron Structure of Example

The output **I** of the network, prior to the activation function stage, is

$$I = X^T . W = \begin{bmatrix} 1 & 2 & 5 & 8 \end{bmatrix} \bullet \begin{pmatrix} +1 \\ +1 \\ -1 \\ +2 \end{pmatrix} = 14$$

$$= (1 \times 1) + (2 \times 1) + (5 \times -1) + (8 \times 2) = 14$$

With a binary activation function the outputs of the neuron is:

y (threshold) = 1;

An Artificial Neural Network (ANN) is a data processing system, consisting large number of simple highly interconnected processing elements as artificial neuron in a network structure that can be represented using a directed graph **G**, an ordered 2-tuple **(V, E)** , consisting a set **V** of vertices and a set **E** of edges.

- The vertices may represent neurons (input/output) and
- The edges may represent synaptic links labeled by the weights attached.

Example :

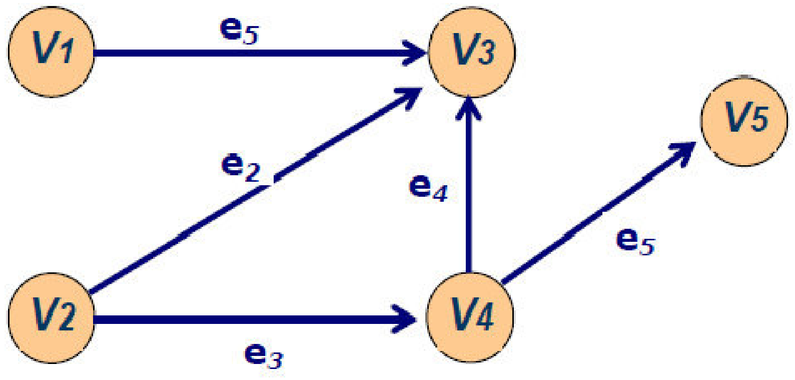


Fig. Directed Graph

Vertices $V = \{ v_1 , v_2 , v_3 , v_4, v_5 \}$

Edges $E = \{ e_1 , e_2 , e_3 , e_4, e_5 \}$

The Threshold as a Special Kind of Weight

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

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

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

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

This simplifies the basic Perceptron equation so that:

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

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

Example : A Classification Task

A typical neural network application is classification. Consider the simple example of classifying aeroplanes given a known set of 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 or Fighter?

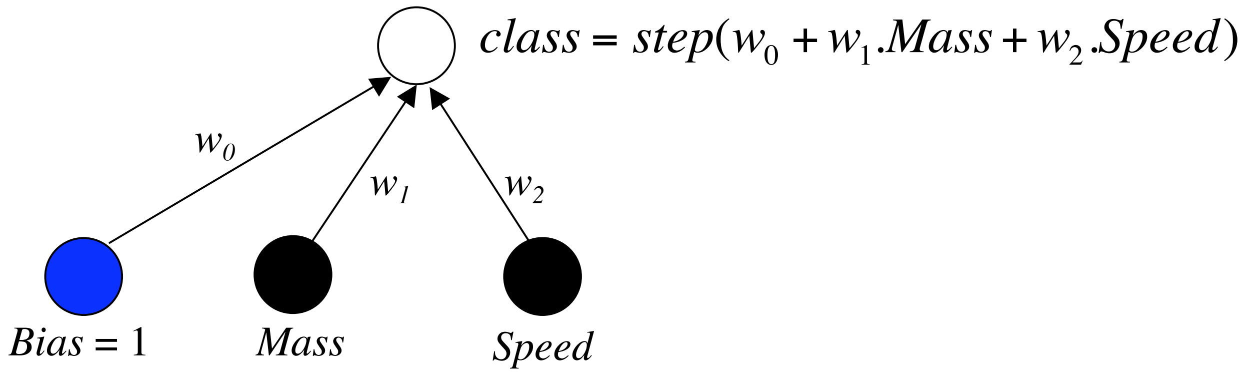
General Procedure for Building Neural Networks

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

1. Understand and specify the problem in terms of ***inputs and required outputs***, 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 the 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 performance by checking how well it works on previously unseen ***testing data***.
5. If the network doesn't perform well enough, go back to stage 3 and try harder.
6. If the network still doesn't perform well enough, go back to stage 2 and try harder.
7. If the network still doesn't perform well enough, go back to stage 1 and try harder.
8. Either the problem is solved or admit defeat – move on to the next problem.

Building a Neural Network for The Example

For the aeroplane classifier example, the inputs can be direct encodings of the masses and speeds. Generally we would have one output unit for each class, with activation 1 for 'yes' and 0 for 'no'. With just two classes here, we can have just one output unit, with activation 1 for 'fighter' and 0 for 'bomber' (or vice versa). The simplest network to try first is a simple Perceptron. We can further simplify matters by replacing the threshold by an extra weight w_0 as discussed above. This gives us:



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

Overview and Reading

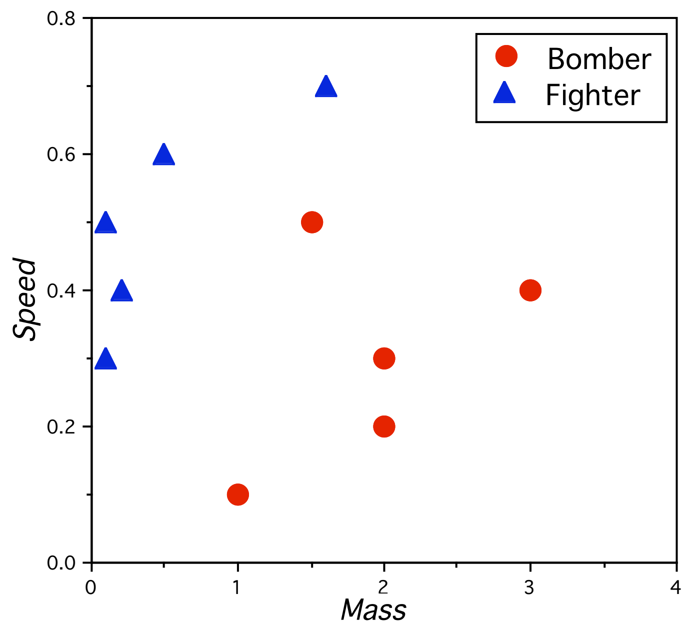
1. Networks of McCulloch-Pitts neurons are powerful computational devices, capable of performing *any* logical function.
2. However, simple Single-Layer Perceptrons with step-function activation functions are limited in what they can do (e.g., they can't do XOR).
3. Many more powerful neural network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

Reading

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

What Can Perceptrons Do?

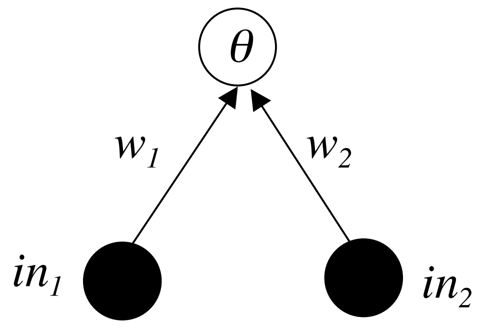
How do we know if a simple Perceptron is powerful enough to solve a given problem? If it can't even do XOR, why should we expect it to be able to deal with the aeroplane classification example, or real world tasks that will be even more complex than that?



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:



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

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:

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

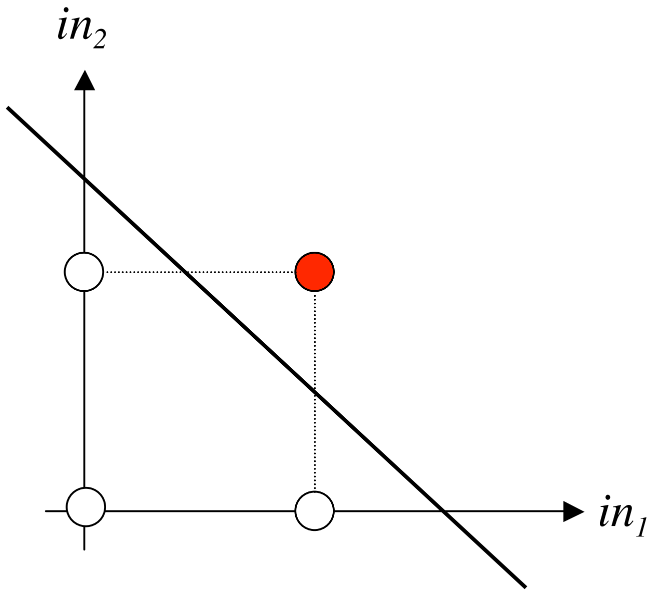
So, in two dimensions the decision boundaries are always straight lines.

Decision Boundaries for AND and OR

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

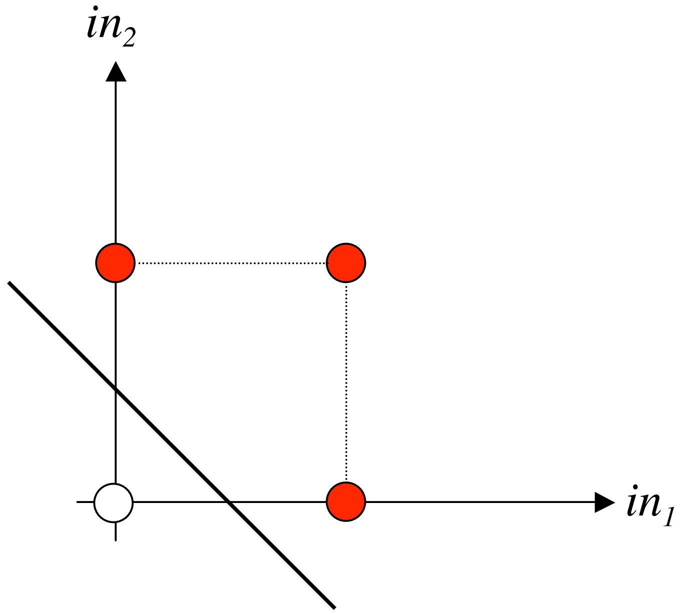
AND

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



OR

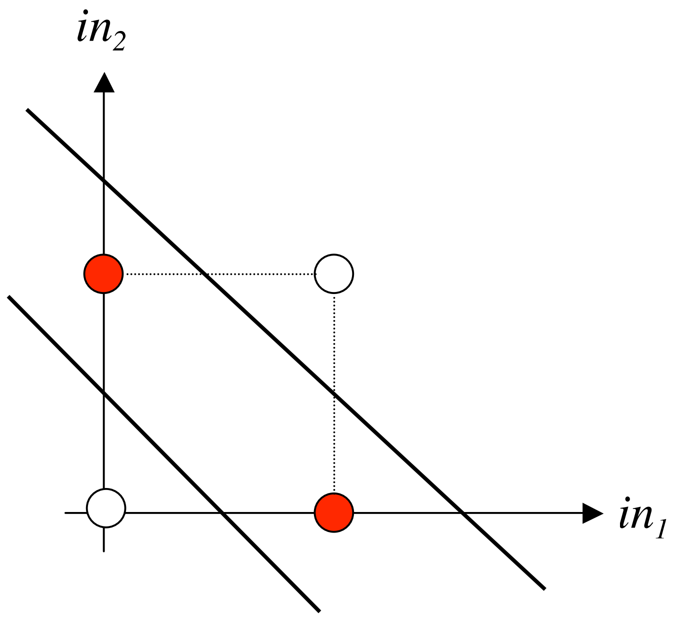
$w_1 = 1, \ w_2 = 1, \ \theta = 0.5$



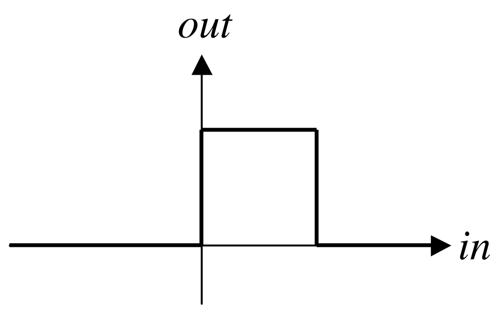
The extent to which we can change the weights and thresholds without changing the output decisions is now clear.

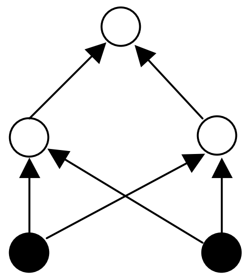
Decision Boundaries for XOR

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



$\Longrightarrow$ *e.g.*





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:

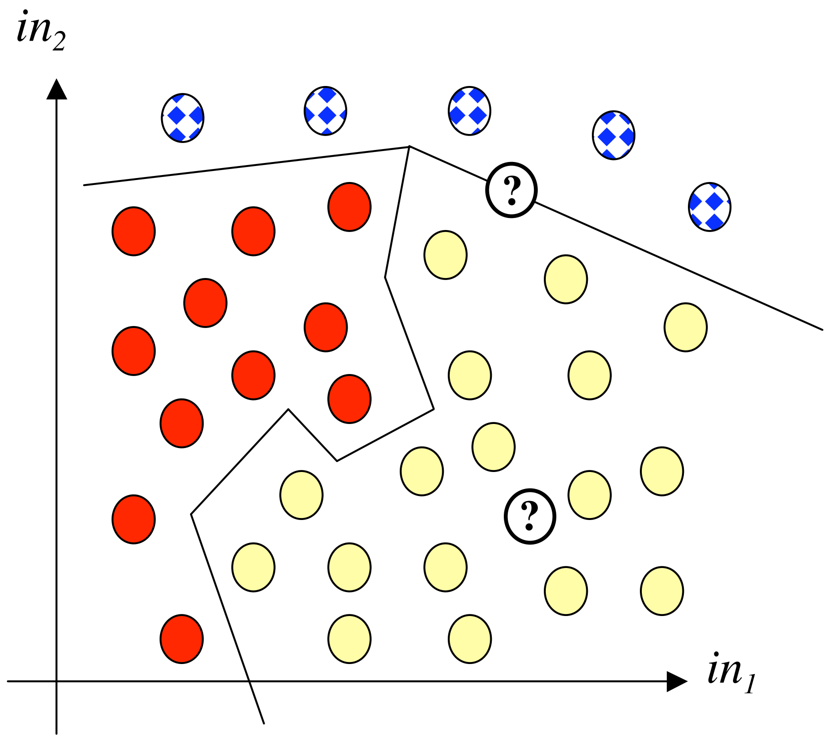
$$w_1 in_1 + w_2 in_2 + \ldots + w_n in_n - \theta = 0$$

This hyperplane is clearly still linear (i.e., straight or flat or non-curved) 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 that 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 often 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 boundaries from a set of ***training patterns*** so that these training patterns are classified correctly.

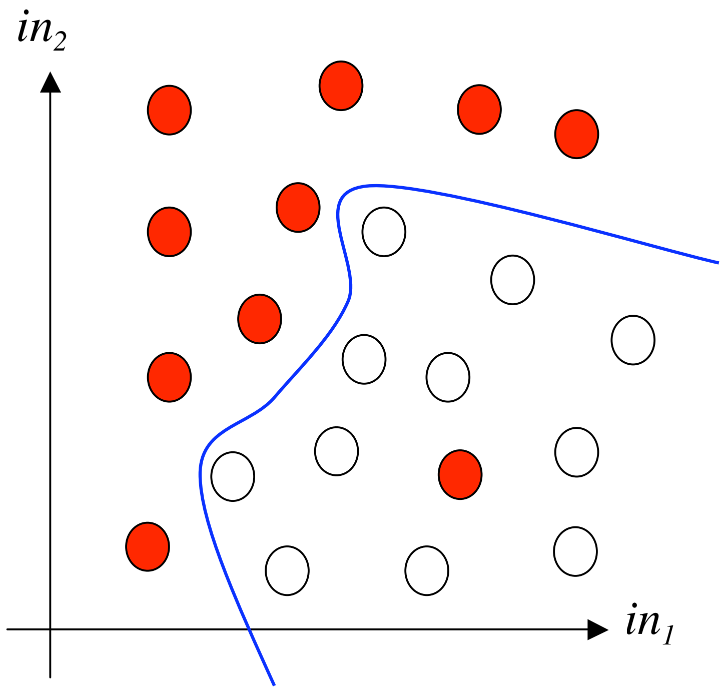
Generalization : After training, the network must also be able to generalize, i.e. correctly classify ***test patterns*** it has never seen before.

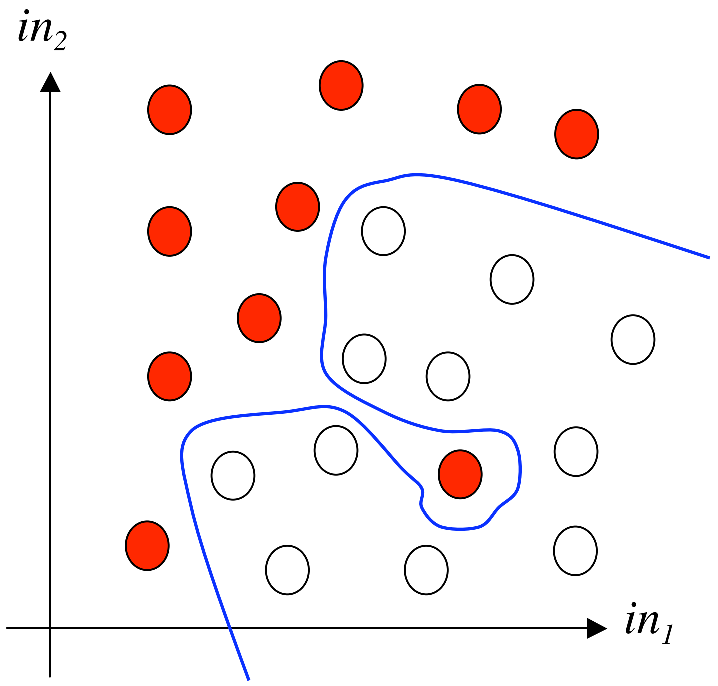
Usually we want the neural network to learn in a way that produces good generalization.

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 the neural network is to learn a classification decision boundary:

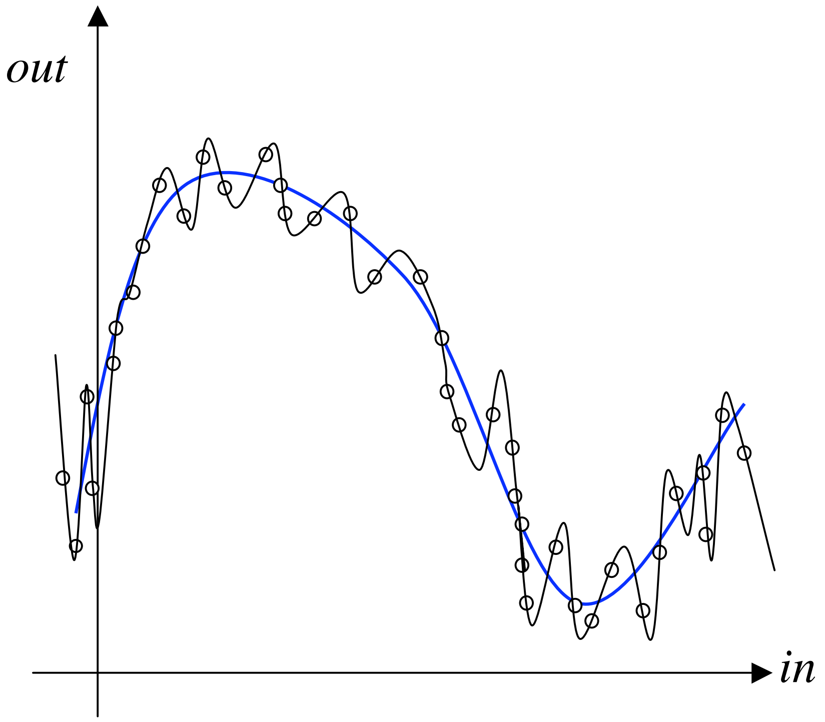




The aim is to get the network to generalize to classify new inputs appropriately. If we know that the training data contains noise, we don't necessarily want the training data to be classified totally accurately, as that is likely to reduce the generalization ability.

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 the 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.

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

Instead, a good all-purpose process is to start off with ***random initial weights*** and adjust them in small steps until the required outputs are produced.

We shall first 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 a small 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 $out_j = \text{step}(\sum in_i w_{ij})$, where in_i are the activations of the previous layer of neurons (i.e. the network inputs for a Perceptron). 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 is possible to prove 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.

Overview and Reading

1. Neural network classifiers learn decision boundaries from training data.
2. Simple Perceptrons can only cope with linearly separable problems.
3. Trained networks are expected to generalize, i.e. deal appropriately with input data they were not trained on.
4. One can train networks by iteratively updating their weights.
5. The Perceptron Learning Rule will find weights for linearly separable problems in a finite number of iterations.

Reading

1. Haykin-2009: Sections 1.1, 1.2, 1.3
2. Gurney: Sections 3.3, 3.4, 3.5, 4.1, 4.2, 4.3, 4.4.
3. Beale & Jackson: Sections 3.3, 3.4, 3.5, 3.6, 3.7
4. Callan: Sections 1.2, 1.3, 2.2, 2.3

Hebbian Learning

The neuropsychologist Donald Hebb postulated in 1949 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 more familiar terminology, that can be stated as the ***Hebbian Learning*** rule:

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.

Then, in the notation used for Perceptrons, that can be written as the weight update:

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

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

Modified Hebbian Learning

An obvious problem with the above rule is that it is unstable – chance coincidences will build up the connection strengths, and all the weights will tend to increase indefinitely. Consequently, the basic learning rule (1) 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.

Another way to stop the weights increasing indefinitely involves normalizing them so they are constrained to lie between 0 and 1. This is preserved by the weight update

$$\Delta w_{ij} = \frac{w_{ij} + \eta.out_j.in_i}{\left(\sum_k (w_{kj} + \eta.out_j.in_k)^2\right)^{1/2}} - w_{ij}$$

which, using a small η and linear neuron approximation, leads to ***Oja's Learning Rule***

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

which is a useful stable form of Hebbian Learning.

Hebbian versus Perceptron Learning

It is instructive to compare the Hebbian and Oja learning rules with the ***Perceptron learning*** weight update rule we derived previously, namely:

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

There is clearly some similarity, but the absence of the target outputs $targ_j$ means that Hebbian learning is never going to get a Perceptron to learn a set of training data.

There exist variations of Hebbian learning, such as ***Contrastive Hebbian Learning***, that do provide powerful supervised learning for biologically plausible networks.

However, it has been shown that, for many relevant cases, much simpler non-biologically plausible algorithms end up producing the same functionality as these biologically plausible Hebbian-type learning algorithms.

For the purposes of this module, we shall therefore pursue simpler non-Hebbian approaches for formulating learning algorithms for our artificial neural networks.

Learning by Error Minimisation

The general requirement for learning is an algorithm that adjusts the network weights w_{ij} to minimise the difference between the actual outputs out_j and the desired outputs $targ_j$.

It is natural to define an ***Error Function*** E to quantify this difference, for example:

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

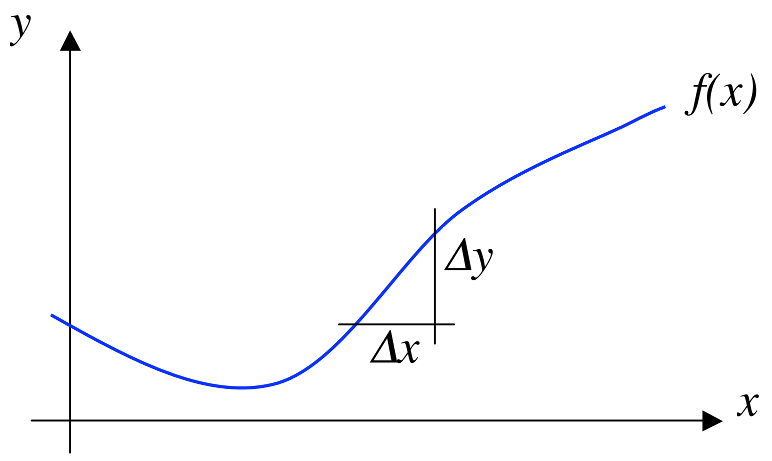
For obvious reasons this is known as the ***Sum Squared Error*** (SSE) function. It is the total squared error summed over all output units j and all training patterns p.

The aim of the ***learning algorithm*** is to minimise such an error measure by making appropriate adjustments to the weights w_{ij}. Typically we apply a series of small updates to the weights $w_{ij} \to w_{ij} + \Delta w_{ij}$ 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

The branch of mathematics concerned with computing gradients is called ***Differential Calculus***. The relevant general idea is straightforward. Consider a function $y = f(x)$:



The gradient of $f(x)$, at a particular value of x, is the rate of change of $f(x)$ as we change x, and that can be approximated by $\Delta y/\Delta x$ for small Δx. It can be written 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 illustrate how derivatives can be computed:

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

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

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

Other derivatives can be found in the same way. Some particularly 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

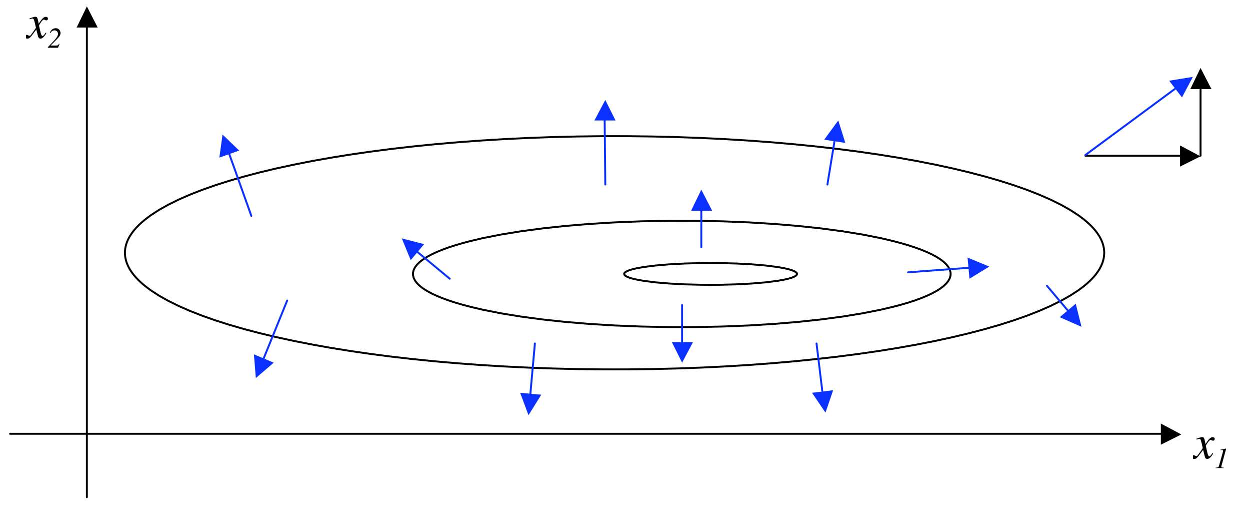
In summary, we can decrease $f(x)$ by changing x by the amount:

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

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

It might not be obvious that one needs the gradient/derivative itself in the weight update equation, rather than just the sign of the gradient. So, 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.

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

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

2. Set up the 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".

This will produce a trained neural network, but steps 4 and 5 can still be difficult…

Gradient Descent Error Minimisation

We will look at how to choose the error function E next lecture. Suppose, for now, that we want to train a neural network by adjusting its weights w_{ij} to minimise the SSE:

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

We have seen that we can do this by making 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.