Layered Perceptron Networks and the Error Back Propagation Algorithm

M. Brown

October 30, 1996

One of the main factors behind the recent revival of interest in neural computing was the derivation of a learning algorithm for training so-called Multi-Layered Perceptron (MLP) networks. It was Minsky and Papert's analysis of the computational properties of a Single-Layer Perceptron (SLP) network in 1969 [MinPap69] which led to the decline of these techniques, as they pointed out the limitations of these simple networks. SLP networks had been trained to perform many interesting and useful pattern classification tasks, and the associated training algorithm was guaranteed to converge given that the classification problem was solvable. However, no training algorithm was available for the more complex, and powerful, MLP networks, and such was the standing of Minsky and Papert in the AI community, that their thorough research essentially killed off the field of neural computing for two decades, while research into symbolic computing flourished.

In 1986, Rumelhart, Hinton and McClelland published the book "Parallel Distributed Processing" [Rumetal86a] which contained a learning algorithm which could be used to train MLP networks. Unlike the single-layer version of this rule, it was not guaranteed to converge, but it provided a partial answer to Minsky and Papert's criticisms and despite the many limitations of this training algorithm, it is still probably the simplest (you may disagree with this when you see its derivation!) and most widely used technique in the field of neural computation. It turned out that this learning algorithm had been proposed several times prior to this date (by Werbos 1974, Parker 1985 etc.), but Rumelhart and McClelland's authoritative description is still widely regarded as the seminal text and was responsible for the resurgance of interest in the field of neural computing.

These MLP networks are *universal adaptive systems* as they are trained in a *supervised manner* and they can be shown to have the ability to approximate any continuous nonlinear function (over a compact domain) to an arbitrary degree of accuracy. They have been used:

1. in autonomous vehicles to fuse pixel-based vision and laser ranger finder data, thus steering a van along a road,
2. to model, identify and control nonlinear plants,
3. to synthesise text to speech mappings,

and for numerous other applications.

This section of the course describes both the SLP and MLP networks and outlines the corresponding learning rules which made them famous during the Sixties and Nineties, respectively. It concentrates on the underlying networks' structures and their learning rules, but it also discusses how a network *generalises*[1], as this is central to the successful development and deployment of neural systems. To begin with, the basic architecture and notation used for the Perceptron networks is described, then the structure and training procedures used with the SLP networks are outlined. The computational possibilities and limitations associated with SLP networks are then discussed and the more complex MLP networks are described.

1 Architecture and Notation

A natural way to describe neural networks is as directed graphs, in which the nodes are processing elements (neurons) and the arcs depict connections between nodes, i and j say. Connections are labelled with their corresponding weights, w_{ij}, which are adjustable in that they are modified during training to produce the desired network behaviour. In addition, an extra *threshold* value is fed into each node which can be treated as just another adjustable weight, although in Kröse and van der Smagt [KroSma93], the threshold (bias) of the i^{th} node is denoted by a separate parameter θ_i whereas in this set of notes it is denoted by an appended weight, w_{i0}.

Feedforward nets are those whose graph is *acyclic*, all connection paths are directed from input to output, and there is no feedback. This means that such nets are unconditionally stable.

Feedforward networks are *layered* as shown in figure 1, i.e. the set of nodes can be grouped into subsets such that connections from nodes in a particular subset N_i are incident only on nodes in another particular subset N_j, and on no other nodes (where the subset N_* in this case are layers of the MLP or SLP). All nodes in a given subset are at the same 'path distance' from the input (in terms of the number of arcs which have to be traversed to reach the node) and at the same distance from the output.

There is some confusion in the literature over the exact meaning of the term *layer*. Strictly, it means a layer of adjustable connection weights, but it is also used less precisely to mean a "layer of nodes". Obviously, a network having L layers of connection weights has $L+1$ 'layers' (subsets in the above sense) of nodes. This is consistent with the notation employed by Kröse and van der Smagt.

It is useful to classify feedforward networks into the following two categories:

Single-Layer Perceptrons (SLPs) which have a single layer of adjustable weights.

Multi-Layer Perceptrons (MLPs) which have multiple layers of adjustable weights.

Because $L = 1$ for an SLP , there are just $L + 1 = 2$ subsets of nodes (in the above sense) corresponding to the input and the output nodes, respectively. The value of L also denotes the "path distance" from all input nodes to the output layer. MLPs have nodes which are neither input nor output nodes and these are known as *hidden* units. The input and output

[1]Generalisation is the ability of the system to respond appropriately to inputs which are not contained in the training set. Often this relies on the fact that the underlying, unknown function is smooth.

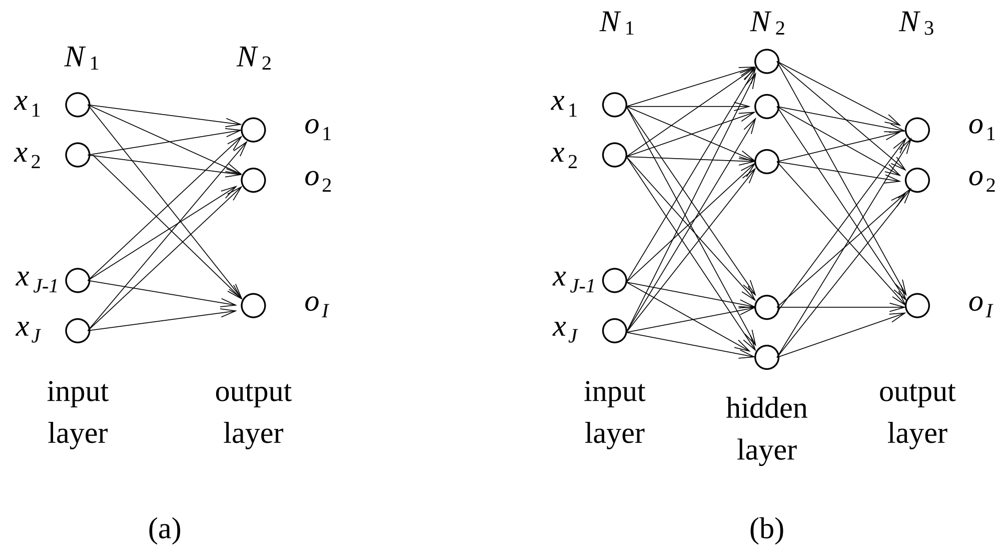


Figure 1: A single-layered ($L = 1$) feedforward network (a) and a multi-layered ($L = 2$) feedforward network (b).

layers are the network's interface with its environment as the input nodes' output values are set equal to the current input pattern ($a_i^p = x_i^p$) and the network's output is simply the output of the output nodes ($o_i^p = a_i^p$).

Some authors reserve the term "Perceptron" for SLPs alone. This usage is a little old-fashioned, and we will not respect it here. We use the term to refer both to SLPs and MLPs.

1.1 Nodes

The knowledge and processing abilities of a layered Perceptron network are stored in the arrangement of the nodes (number of layers and the number of nodes in each layer), the transfer function associated with each node and the value of the weighted links. The network's architecture (structure) defines the nodes' arrangement and the weights are determined from the training data set using a supervised learning algorithm such as Error Back-Propagation (EBP). Therefore, to understand how a network calculates its output, the transfer function of each node must be hard-wired into the network's design. One of the fundamental ideas behind neural computing is that intelligence and knowledge should arise from the interactions between simple processes (nodes) arranged in a distributed manner, hence the functionality of each of the individual nodes is fairly basic.

As the network operates in a feedforward, layered manner, the output of all the nodes on the previous layer (starting with the input layer) should be calculated before proceeding onto the next layer. Therefore, consider the operations of node i which has an associated weight vector $\mathbf{w}_i$ leading into it, then its output a_i is defined by:

$$a_i = f(i_i) \tag{1}$$

$$i_i \;=\; \sum_{j=0}^{J} w_{ij}\, a_j \tag{2}$$

where the terms a_j refer to the outputs of the previous layer of nodes, the bias term has been incorporated into the input vector in the form of an extra input and weight, a_0 and w_{i0}, respectively, and $f(.)$ is a simple, often nonlinear, univariate transfer function, ie. the output of node i is a function, $f(.)$, of the weighted sum of the outputs of the nodes of the previous layer. The node projects the incoming input vector onto its weight vector and then passes this scalar quantity through the transfer function. This is illustrated in figure 2.

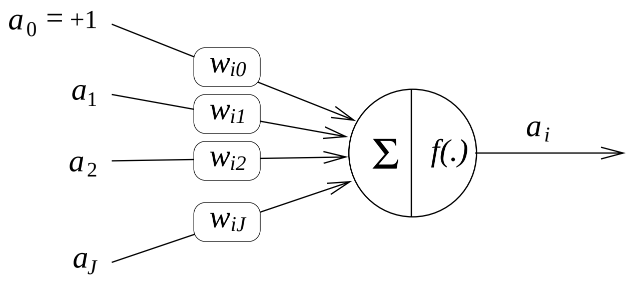


Figure 2: The operation of a single node with an output a_i, input vector $\mathbf{a}_j$ and weight vector $\mathbf{w}_i$.

The transfer function can take many forms from a simple linear scaling:

$$a_i = k \times i_i \tag{3}$$

where k is usually 1, to more complex, bounded, nonlinear functions such as the sigmoid:

$$a_i = \frac{1}{1 + \exp(-k \times i_i)} \tag{4}$$

and its hard-limiting (classification) form:

$$a_i = \begin{cases} 1 & \text{if } i_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

As the gain k in equation 4 approaches ∞, the continuous sigmoid becomes closer to the discontinuous hard-limiter tranfer function, although it is always differentiable. All three of these functions are commonly employed in Perceptron networks, and they are illustrated in figure 3.

1.2 Inputs and Outputs

Perceptron networks can be used with binary or real-valued input and output data, although the first networks were designed to work with only binary input *and* output data.

When the output of a Perceptron network is *binary* by using a hard limiting transfer function on the output, the system operates as a classification algorithm. Classes are encoded using a *1-of-n* scheme where the i^{th} of n class is represented by an identically zero vector of length

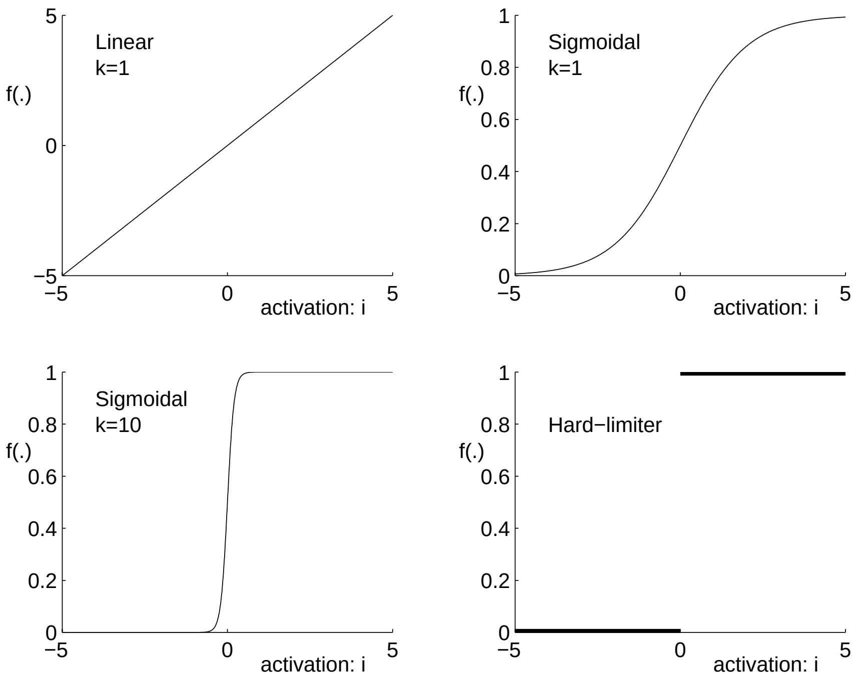


Figure 3: Different transfer functions employed in layered Perceptron networks.

n except for a unity element in the i^{th} place. This is illustrated in figure 4 for a 3 class classification problem. Each hyperplane $\mathbf{h}_i$ represents the *decision boundary* for the i^{th} class which is the partitioning of the input into regions which correspond to a particular class label.

SLP and MLP networks are often used as adaptive logic functions as they can be regarded as a particular 2 class mapping (true or false) which works on binary inputs. Indeed, when Parker re-discovered the EBP algorithm in 1982, he described it as a method for learning logical functions. However, serious questions about the application of MLPs to learning logical functions can be raised. Unless the training data contains enough information about the form of the discrimination function, the generalisation ability of these Perceptron networks will be arbitrary. This would be true of *any* learning system for a data set which is lacking discriminatory information for logical functions.

Real-valued outputs occur when the output nodes have a linear or continuous sigmoidal transfer function. Thus the network can be used for function approximation, system identification, data fusion etc. When sigmoidal transfer functions are used in the output nodes, the network's output is restricted to lie in the interval (0,1) and the training data should also be scaled to lie in this range. Using a linear transfer function allows the network's output to lie in an arbitrary range, although for an MLP with sigmoidal nodes in the hidden layer, the output is still bounded (where the bounds are adapted to the training data) due to the hidden layer nodes being bounded.

Inputs to the Perceptron networks can be either binary $\in \{0,1\}$ or bipolar $\in \{-1,1\}$ or real-valued. Typically, the inputs are restricted to be binary or bipolar when the unknown mapping is a logical function. However, for most other tasks, the inputs can be assumed to

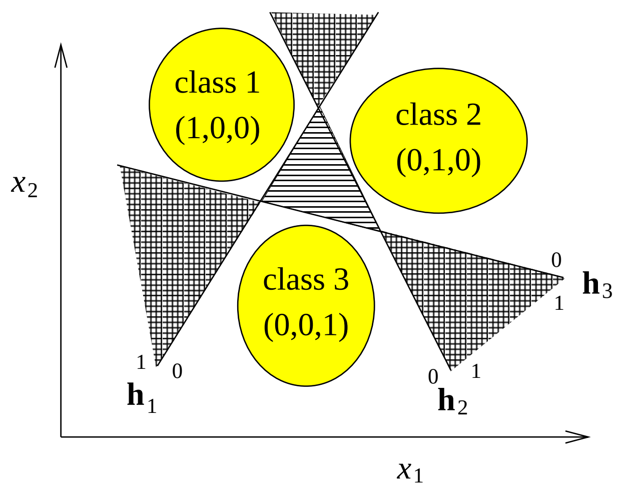


Figure 4: Separating decision boundaries $\mathbf{h}_i$ (representing the i^{th} output node) for multi-class decisions with a one-of-n encoding with 3 outputs and 3 classes. The patterned areas represent inconsistent network outputs: no class prediction (horizontal lines) and two class prediction (grid).

be real-valued.

2 Single-Layer Perceptron

The SLP was originally developed by Rosenblatt (1958; 1962) who carried out a lot of important work on (single-layer) Perceptrons and introduced a training rule for such networks as well as building a trainable pattern-recognition machine. SLPs are pattern classification devices (binary outputs) which can accept either binary/bipolar or real-valued inputs. This device was used extensively during the Sixties and it was largely its limitations which caused the decline of research into neural computation at the start of the Seventies.

Figure 5 depicts an array of McCulloch and Pitts neurons[2] forming an SLP. The network input is the vector $\mathbf{x} = (x_0, x_1, x_2, \ldots, x_J)$, the output is the vector $\mathbf{o} = (o_1, o_2, \ldots, o_I)$ and the weight vectors $\mathbf{w}_i = (w_{i0}, w_{i1}, \ldots, w_{iJ})$ are simply a series of weights which are multiplying the corresponding inputs. An I-output network is equivalent to I single-output networks, as each of the weight vectors associated with the output nodes are *independent* and can be trained separately. Therefore, the description of the learning rule and the analysis of its convergence can be restricted to single-output SLPs , and the rate of convergence of the overall I-output network is as slow as the slowest output node.

2.1 Classes of Perceptron Networks

Strictly speaking, the SLP developed by Rosenblatt and his co-workers in the Sixties did not have a single layer, but it did have a single layer of *adaptive weights*. Preceding this layer of

[2]A McCulloch and Pitts neuron is simply a node whose activation value is formed from the linearly weighted sum of the input signals, and the transfer function is a binary, hard limiter.

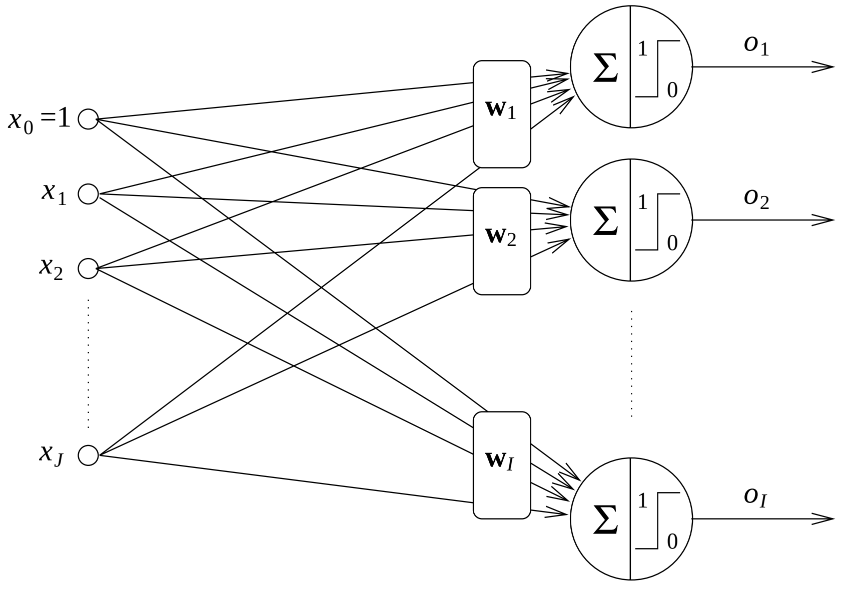


Figure 5: Single layer of McCulloch and Pitts neurons forming an J-input, I-output Perceptron network.

adaptive weights was a *fixed*, nonlinear mapping which transformed the input presented to the network. Different Perceptron networks were distinguished by the type of nonlinear mapping, but the important restriction of these type of networks was that this initial processing stage was fixed and not adapted during the training procedure. Therefore, a network's computational complexity was determined prior to the learning procedure and *without* reference to the training set.

One class of SLP network had a fixed, nonlinear mapping in the "hidden layer", where the McCulloch and Pitts neurons were randomly connected. This type of SLP was very popular for pixel-based vision classification tasks where the input space was high dimensional, and some form of compression was required. The transfer functions in the hidden layer were hard-limiting binary or bipolar mappings, so the "inputs" to the layer of adaptive weights had two states.

Another type of SLP network was a diameter-limited network, where instead of randomly connecting the hidden layer with the input space, each node represented small, overlapping regions.

2.2 Training Rule

The Perceptron training rule is specified in Algorithm 1. It relies on having labelled training data, i.e. learning is supervised. The basic idea is that if one of the bits in the output vector, o_i^p, is in error, the weights contributing to that error, w_{ij}^p, are adjusted to make the Perceptron more likely to produce the correct output, for a training pattern $\{\mathbf{x}^p, \mathbf{d}^p\}$, where $\mathbf{d}^p$ is a vector of desired (or ideal) outputs for that training pattern. That is, if the output value is 1 but it ought to be 0, decrease the relevant weights; whereas if the output value is 0 but it ought to be 1, increase these weights. Therefore, the *classification error* for each output node $(d_i^p - o_i^p)$

can be used to direct the weight updates as:

$$d_i^p - o_i^p = \begin{cases} 1 & \text{if } d_i^p = 1 \text{ and } o_i^p = 0 \\ -1 & \text{if } d_i^p = 0 \text{ and } o_i^p = 1 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Two points about this weight updating are:

- Deciding which weights are contributing to the error, and by how much, is a manifestation of a classic problem in artificial intelligence, called the *credit assignment problem* (although we are actually seeking to apportion *blame* here rather than credit as the weights are only updated if the output is incorrect).
- While it is clear in this case that the weights $\mathbf{w}_i$ are to blame if o_i^p is in error, we don't know by how much to blame each individual weight. The solution adopted in the Perceptron training rule is to apportion blame according to the input value x_j^p; by adding it to/subtracting it from each w_{ij}. The rationale is that if the input value is 0, the corresponding weight cannot be contributing to the error. (Note that, strictly speaking, the network's inputs should be denoted as $a_j^p = x_j^p$ although this may be confusing, so the network's input is considered to be the original input vector $\mathbf{x}^p$.)

```
1. start at $t = 0$ with small random initial weights
   (and thresholds) and error criterion, $\epsilon$
2. For all input patterns $p$ from 1 to $P$
   (a) apply a new input pattern and calculate output, $\mathbf{o}^p$
   (b) for each output $o_i^p$ from 1 to $I$, update the weights according to:
       $w_{ij}(t+1) = w_{ij}(t) + (d_i^p - o_i^p)\, x_j^p$   for all $j$, $0 \le j \le J$
3. IF total sum of squared errors $\sum_{p=1}^{P}\sum_{i=1}^{I}(d_i^p - o_i^p)^2 > \epsilon$, THEN
   increment $t$ and iterate from 2.
```

Algorithm 1: Single-Layer Perceptron learning rule.

To evaluate the effect of this learning rule, consider the i^{th} output node's activation value immediately after adaptation:

$$\begin{aligned} i_i^p(t+1) &= \sum_{j=0}^{J} w_{ij}(t+1)\, x_j^p && (7) \\ &= \sum_{j=0}^{J} \Big(w_{ij}(t) + (d_i^p - o_i^p)\, x_j^p \Big)\, x_j^p && (8) \\ &= \sum_{j=0}^{J} w_{ij}(t)\, x_j^p + (d_i^p - o_i^p) \sum_{j=0}^{J} x_j^2 && (9) \\ &= i_i^p(t) + (d_i^p - o_i^p)\, \|\mathbf{x}^p\|_2^2 && (10) \end{aligned}$$

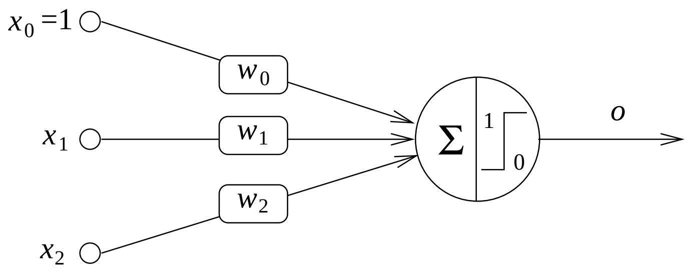


Figure 6: Simple 2-input, 1-output single-node Perceptron. (Note that the bias term is not considered as a "true" input as it is fixed.

so i_i^p has been changed by a positive amount $\|\mathbf{x}^p\|_2^2$ (see A.1), and this is in a direction to make the thresholded output o_i^p more like d_i^p. After several applications of this training strategy, the thresholded output, o_i^p, will be correct. Note that $\|\mathbf{x}^p\|_2^2$ is *always* at least 1 due to the prescence of the extra bias input x_0 which is always switched on.

3 What Can SLPs Do?

What can a network like that in figure 5 do in computational terms? To help answer this question, consider a very simple 2-input, 1-output SLP consisting of a single McCulloch and Pitts neuron, as depicted in figure 6. In this figure, the network is depicted as a directed graph as described earlier. The restriction to just two inputs allows us to visualise the operation of the SLP in a 2-dimensional (x_1, x_2) input space.

Now, the boundary between the output conditions $o = 1$ and $o = 0$ is given when the node's activation value is zero, ie:

$$w_0 + x_1 w_1 + x_2 w_2 = 0$$

and, hence:

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2} \tag{11}$$

which can be written as:

$$x_2 = m x_1 + c \tag{12}$$

which has a gradient $m = -w_1/w_2$ intercept $c = -w_0/w_2$.

Clearly, this specifies a straight-line relationship in the (x_1, x_2) input space, as depicted in figure 7. This line is called a *linear discriminant* since it separates points in the input space which produce an output of $o = 0$ from those which produce an output of $o = 1$. (You may like to think about what it is that determines which side of line is the $o = 1$ region and which is the $o = 0$ region.) These different outputs determine whether an input is a member of class 1 or 2.

Now the slope, m, and intercept, c, of the linear discriminant in equation 11 depend upon the weights (and thresholds) of the Perceptron. But these are modifiable via the training rule. Hence, the purpose of the Perceptron training rule is to adjust the linear discriminant to achieve the desired network behaviour; as embodied in the training data. However, there is a "compression" of information as the *two* quantities m and c are determined by *three* weights

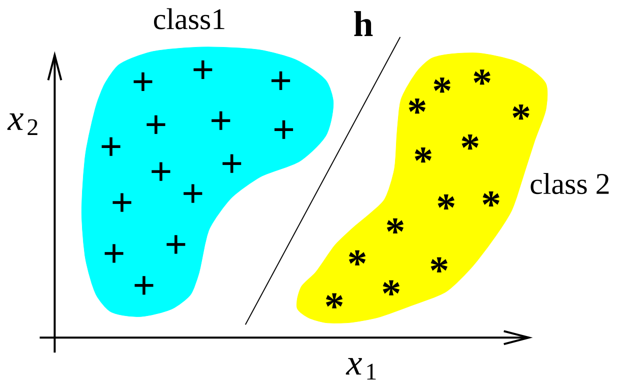


Figure 7: The linear discriminant for a single-layer, single-output Perceptron which divides the input space linearly into two regions. The line is described by $\mathbf{h}: \ \mathbf{x}^T\mathbf{w} = 0$.

w_0, w_1 and w_2. This is because the weight vector can be any size and the corresponding decision boundary will be unchanged as long as the ratios $-w_1/w_2$ and $-w_0/w_2$ stay the same. However, if the weights are multiplied by a negative scalar, the class regions will be swapped over.

So, the answer to the question "what can SLPs do?" is: they can compute linearly-separable functions of their inputs. For logical (binary) problems with a small number of inputs (2 or 3) there are only a small number of nonlinearly separable problems which SLP networks cannot solve. However when there exists a large number of inputs, the class of linearly separable functions is small in comparison to the number of nonlinearly separable ones. Figure 8 shows how simple AND, OR and NOT operators can be implemented as SLPs.

It is important to realise that an SLP forms linearly separable decision boundaries for real-valued inputs as well as binary. The only difference in using binary inputs rather than real-valued ones is that for binary inputs the training points now lie at the corners of an n-dimensional hypercube, rather than being distributed across the whole of the input space.

3.1 Training Data Distribution and Generalisation

As was mentioned in the introduction, learning systems are only useful if their generalisation abilities are appropriate. However, this is highly dependent on both:

- the type (structure) of learning system used and
- the distribution of the training data.

Consider using an SLP to learn the 2-input logical AND problem with one missing training point. As illustrated in figure 9, only the training pair which has the input $(0, 0)$ can be removed from the training set without effectively changing the decision boundary. Removing any of the training pairs which have inputs $(0, 1)$, $(1, 0)$ and $(1, 1)$ from the training set may cause incorrect generalisation when the removed input is used to query the network. For networks which form decision boundaries, the inputs which are closest to the decision boundary are the most important to collect for the training set.

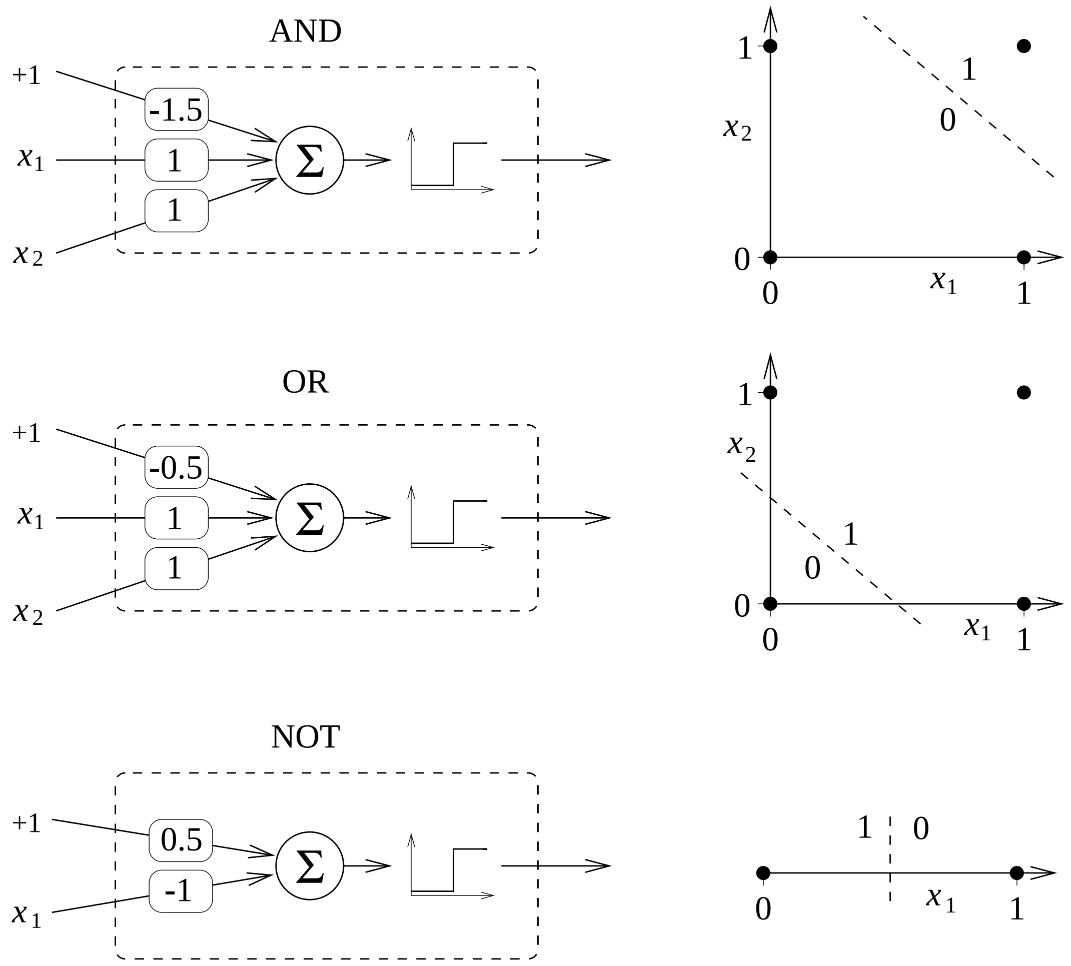


Figure 8: The linear discriminants for the 2 input logical AND, OR and a simple 1 input logical NOT.

3.2 What Can't SLPs do?

The simple answer is that if a problem is *not* linearly separable, then an SLP won't be able to solve it exactly[3]. Such functions are termed *nonlinearly* separable and a famous example of this is the 2-input eXclusive OR logical function (XOR) which has a truth table shown in figure 10.

The XOR logical function can be expressed as:

$$(\text{NOT } x_2 \text{ AND } x_1) \text{ OR } (\text{NOT } x_1 \text{ AND } x_2)$$

and it should be clear from this and figure 10 that the XOR is a nonlinearly separable problem. However, it could be represented by a 2 layer MLP as illustrated in figure 11, as each node in the hidden layer calculates a function of the form:

$$\text{NOT } x_1 \text{ AND } x_2$$

[3]This only refers to the adaptive output layer of weights, not necessarily to the original problem.

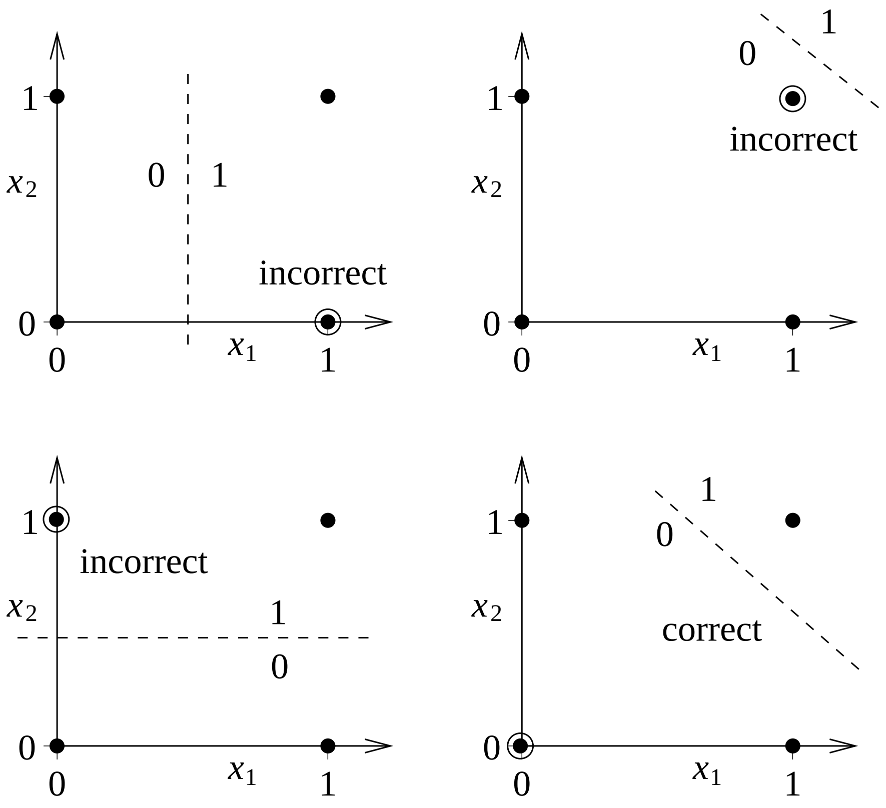
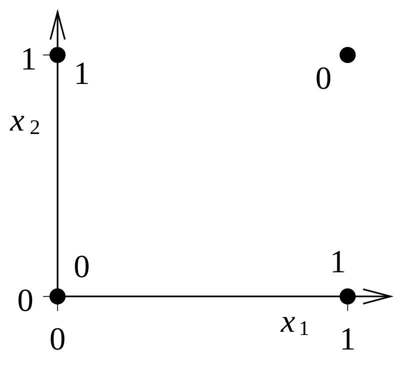


Figure 9: Generalisation to unseen data for the logical AND problem and an SLP. The circled input point in each diagram shows which input pattern was left out of the training set and used to query the network.

		x_1	
		0	1
x_2	0	0	1
	1	1	0

(a)



(b)

Figure 10: The XOR truth table and its geometrical representation.

and the output layer forms the logical OR of these two quantities. This is illustated in figure 11. The more complex representation provided by an MLP means that nonlinearly

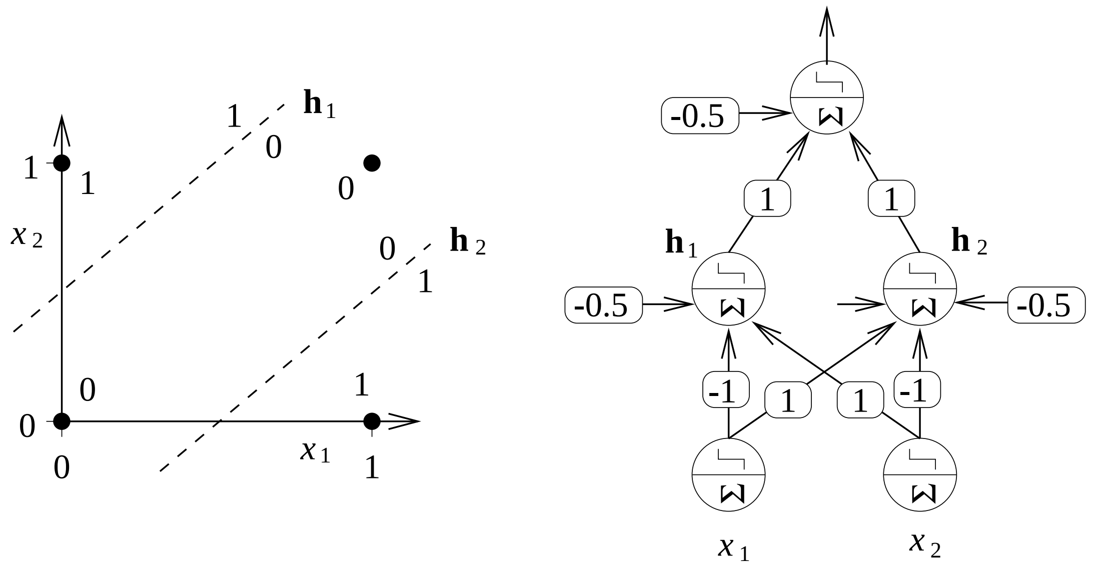


Figure 11: A 2-layer MLP solution to the 2-input XOR problem.

separable problems can be solved and it can be shown that *any* logical function can be implemented using a 2 layer MLP by expressing it in terms of its Conjunctive Normal Form, see section 6.2. It is worthwhile noting that if the hidden layer representation of an SLP performed this type of nonlinear transformation, the XOR problem is solvable by that SLP network. However, this network would not be able to learn the 2 input logical AND and OR problems, which illustrates the point that the computational properties of an SLP are determined by the *fixed* nonlinear tranformation which is independent of the data used in network's design.

It is also worthwhile noting that there are other solutions to this problem and probably the simplest is to introduce an extra input $x_3 = x_1 \times x_2$. This transforms a nonlinearly separable problem into a linearly separable one, as shown in figure 12, which can be solved using an SLP. This is a general technique for logical functions *only*, and does not apply to more general nonlinearly separable classes. When there exists n inputs, this can be expanded to 2^n terms (including the bias) for which *any* logical problem is linearly separable. This is because in order to uniquely identify the 2^n weights, the same number of independent training samples must be presented to the network. Hence the network is simply learning to operate as a digital memory and it will never be asked to generalise. These expanded inputs for a 3-input problem are shown below:

$$\begin{aligned} x_0 &= 1 \\ x_1 &= x_1 \\ x_2 &= x_2 \\ x_3 &= x_3 \\ x_4 &= x_1 \times x_2 \\ x_5 &= x_2 \times x_3 \end{aligned}$$

$$\begin{aligned} x_6 &= x_1 \times x_3 \\ x_7 &= x_1 \times x_2 \times x_3 \end{aligned}$$

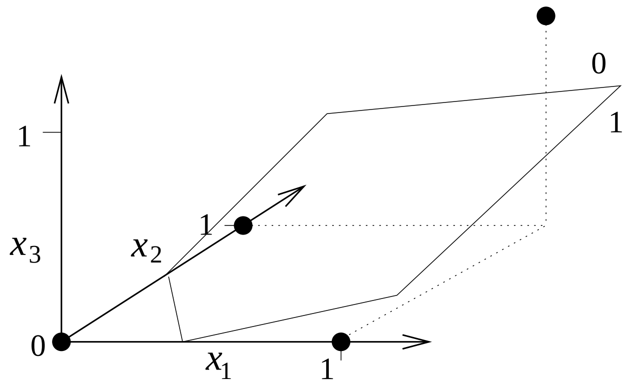


Figure 12: The SLP solution to the XOR problem which introduces a new input $x_3 = x_1 \times x_2$.

4 Convergence of the Rule

Rosenblatt was able to prove a remarkable fact about the SLP Perceptron training rule, namely that if the network is indeed capable of solving a particular problem, the rule will find a solution. We will prove this *Perceptron convergence theorem*, and do so for a single output, as all the outputs are independent and if one converges, they *all* will.

Let $\mathbf{w}$ be the weight vector for the current output node and $\mathbf{X}$ be the set of inputs used in training, with $\mathbf{x}^p$ being a specific input (the p^{th} row) as before. We separate $\mathbf{X}$ into $\mathbf{X}^+$ (the positive training instances for which the output should be 1) and $\mathbf{X}^-$ (the "negative" training instances for which the output should be 0), i.e. $\mathbf{X} = \mathbf{X}^+ \cup \mathbf{X}^-$.

Now imagine a negative set of $\mathbf{X}^-$, where the inputs (including the extra bias term) are all negated, that is $x_i^p = -x_i^p$. Then the corresponding target output of this negated set is always 1, as, due to the linear weighted sum, the value of $i_i = -i_i$ and the thresholded output values are switched. So, by considering a new training set $\mathbf{X} = \mathbf{X}^+ \cup -\mathbf{X}^-$, the target output is always 1, and the objective of the learning rule is to find a weight vector $\hat{\mathbf{w}}$ such that:

$$\hat{\mathbf{w}}^T \mathbf{x}^p > 0 \qquad\qquad \text{for every training pattern } p$$

To show this occurs let:

$$a = \max_p \|\mathbf{x}^p\|_2^2 \qquad\qquad \text{and } b > 0 \tag{13}$$

and we can select $\hat{\mathbf{w}}$ such that:

$$\hat{\mathbf{w}}^T \mathbf{x}^p > \frac{a+b}{2} > 0 \tag{14}$$

as $\hat{\mathbf{w}}$ is scale independent.

To prove that the weight vector converges to a linear discriminant, we show that the *error in the weight vector* tends to zero as training increases. This is done by showing that the error decreases by a *finite* amount for each training iteration, so after a finite number of iterations, the network will act as a linear discrimant for the data. So calculating the distance between $\widehat{\mathbf{w}}$ and $\mathbf{w}(t)$ (the size of the error in the weight vector):

$$\|\widehat{\mathbf{w}} - \mathbf{w}(t)\|_2^2 = \|\widehat{\mathbf{w}}\|_2^2 + \|\mathbf{w}(t)\|_2^2 - 2\widehat{\mathbf{w}}^T\mathbf{w}(t) \tag{15}$$

and using the update rule (assuming, without loss of generality, the output is incorrect):

$$\|\widehat{\mathbf{w}} - \mathbf{w}(t)\|_2^2 - \|\widehat{\mathbf{w}} - \mathbf{w}(t+1)\|_2^2 = \|\mathbf{w}(t)\|_2^2 - \|\mathbf{w}(t+1)\|_2^2 - 2\widehat{\mathbf{w}}^T(\mathbf{w}(t) - \mathbf{w}(t+1)) \tag{16}$$

but by squaring each side of the Perceptron training rule:

$$\|\mathbf{w}(t)\|_2^2 - \|\mathbf{w}(t+1)\|_2^2 = -2\mathbf{w}^T(t)\mathbf{x}^p - \|\mathbf{x}^p\|_2^2$$

and again using the Perceptron training rule:

$$2\widehat{\mathbf{w}}^T(\mathbf{w}(t) - \mathbf{w}(t+1)) = 2\widehat{\mathbf{w}}^T\mathbf{x}^p$$

hence, equation 16 becomes

$$\|\widehat{\mathbf{w}} - \mathbf{w}(t)\|_2^2 - \|\widehat{\mathbf{w}} - \mathbf{w}(t+1)\|_2^2 = -2\mathbf{w}^T(t)\mathbf{x}^p - \|\mathbf{x}^p\|_2^2 + 2\widehat{\mathbf{w}}^T\mathbf{x}^p \tag{17}$$

As $\mathbf{w}^T(t)\mathbf{x}^p \leq 0$ (the output is incorrect), and from equations 13 and 14 this becomes:

$$\|\widehat{\mathbf{w}} - \mathbf{w}(t)\|_2^2 - \|\widehat{\mathbf{w}} - \mathbf{w}(t+1)\|_2^2 > -a + 2\frac{a+b}{2} = b > 0 \tag{18}$$

This shows that whenever there exists a classification error, the error in the weight vector $\|\widehat{\mathbf{w}} - \mathbf{w}(t)\|_2^2$ decreases by a finite amount $> b$. Therefore, after a *finite* number of iterations $\mathbf{w}(t)$ should converge to a linear discriminant which separates the classes (but not necessarily to $\widehat{\mathbf{w}}$!).

4.1 Parameter Convergence

When an SLP is trained using a finite number of examples in the training set, if there exists one solution to the classification problem there exists an *infinite number* as illustrated in figure 13. Learning is terminated whenever the weight vector separates the classes and so does *not* necessarily produce an "optimal" separation which is the furthest from both classes.

We've already come across the nonlinearly separable XOR binary problem, and a more complex nonlinearly separable class is illustrated in figure 14. To solve problems such as these, it is necessary to allow the network to form *richer* decisions boundaries than a simple line (hyperplane).

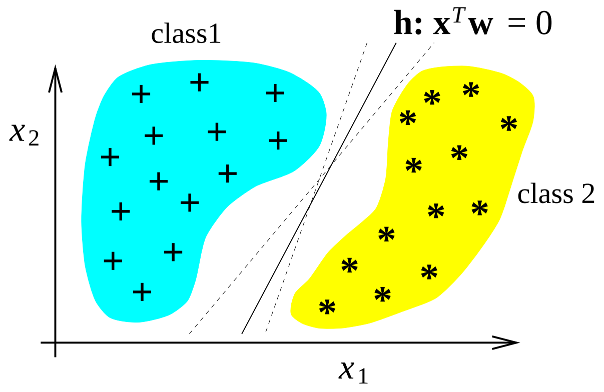


Figure 13: An infinite number of solutions exist (solid and dashed lines) to a 2-class problem when a finite number of samples is used to train an SLP .

5 Adalines and the Widrow-Hoff Rule

Following Rosenblatt's work on the Perceptron, Widrow and Hoff introduced a very similar pattern processing device (Widrow and Hoff, 1960; Widrow, 1962) called the ADAptive LInear NEuron (ADALINE). Multi-layer networks of such neurons were called MADALINES, i.e. *multiple* or *many* ADALINEs.

This device differed from Rosenblatt's Perceptron in not having the random, fixed-weight connections to the "association" units. In modern terminology, it was a single-layer of M&P neurons and so can be considered identical to an SLP, as depicted in figure 5. Widrow and Hoff's training algorithm, also, took a slightly different approach to the credit assignment problem.

In the Perceptron training rule, the weights are adjusted according to:

$$w_{ij}(t+1) = w_{ij}(t) + \eta\,\Delta w_{ij}^p \tag{19}$$

where η is the learning rate and

$$\begin{aligned} \Delta w_{ij}^p &= (d_i^p - o_i^p)\,x_j^p \qquad (\eta \text{ is the learning rate}) \\ &= \delta_i^p\,x_j^p \end{aligned} \tag{20}$$

where the "forward" error δ_i^p is evaluated after 1/0 thresholding. Note that the recommended weight change is split up into two parts:

- the signal due to the output error δ_i^p and
- the incoming input signal x_j^p.

In calculating the output error, we do not know precisely how much the "linear" weighted sum i_i^p was in error prior to thresholding by the signum function. In training the ADALINE, Widrow and Hoff evaluated the error *before* thresholding, effectively ignoring the non-linearity[4], hence

[4]Thus upholding the age old practice of *if you don't know what to do either ignore the effect or linearise*!

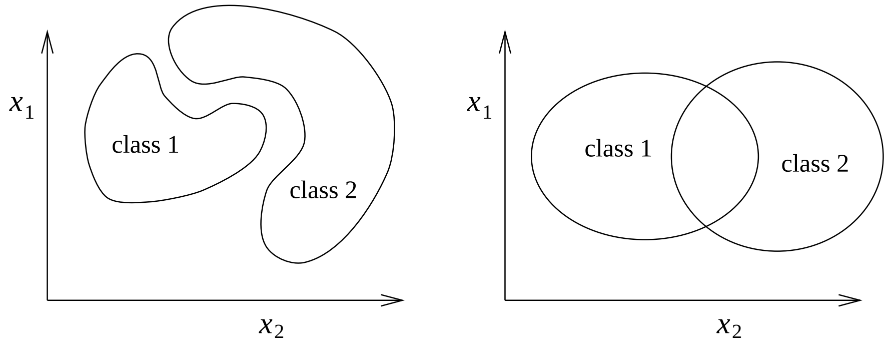


Figure 14: Nonlinearly separable classes.

the somewhat confusing "linear" in the name. That is:

$$\delta_i^p = d_i^p - i_i^p = d_i^p - \sum_{j=0}^{J} w_{ij}\, x_j^p \tag{21}$$

where the desired outputs, d_i^p, are specified *prior* to the binary thresholding transfer function. Updating according to this δ_i^p yields the *Widrow-Hoff* or *delta* rule.

How do we know what the target values before thresholding, d_i^p, ought to be? In answering this question, it is conventional to assume that the output nonlinearity produces hard-limited values of ± 1, rather than 0 and 1. If the target values before thresholding are set equal to the target values *after* thresholding, i.e. ± 1, then the ADALINE will operate correctly. Hence, although it would be sufficient for i_i^p to be "just the right side" of zero to give the right polarity output for the Perceptron network, we use targets spaced equally some way from zero. Ignoring the output non-linearity, and making the target values before thresholding identical to the target values after thresholding, is indistinguishable from using a linear transfer function *during training*, as shown in figure 15.

The ADALINE weight update rule is generally never turned off as the weights will be updated even when the output (after thresholding) would not have been in error. This produces a decision (class) boundary which "best" separates the classes as it is equally far from both classes, unlike the SLP which has an infinite number of optimal weight vectors when the classes are linearly separable.

The Widrow-Hoff delta rule is sometimes called the *Least Mean Square* (LMS) algorithm, since it can be shown (see section 5.2) that it is equivalent to an instantaneous gradient descent error minimisation procedure in the least square sense.

5.1 Mean Squared Error Performance Function

During training, the weights in the i^{th} ADALINE node are adjusted such that they try to minimise a *performance function* of the form:

$$E_i^p = \frac{1}{2}(d_i^p - o_i^p)^2 \tag{22}$$

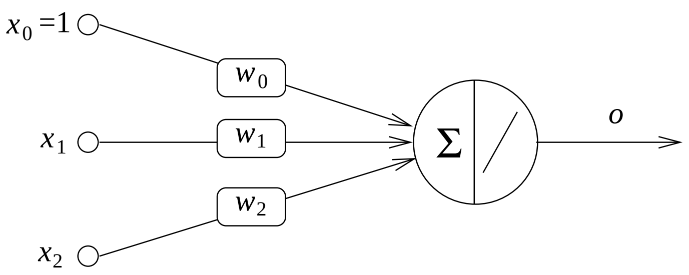


Figure 15: An ADALINE node with a linear output transfer function (during training).

which is a *non-negative* measure of how well the network can predict the desired output d_i^p. Averaging over all the patterns in the training set produces:

$$\begin{aligned} E_i &= \frac{1}{P}\sum_{p=1}^{P} E_i^p \\ &= \frac{1}{2P}\sum_{p=1}^{P}(d_i^p - o_i^p)^2 \end{aligned} \tag{23}$$

which is a measure of how well each output node is performing over the whole training set. In addition, averaging over each output node in the network produces a single performance statistic for its ability to model the complete data set:

$$\begin{aligned} E &= \frac{1}{I}\sum_{i=1}^{I} E_i \\ &= \frac{1}{2IP}\sum_{i=1}^{I}\sum_{p=1}^{P}(d_i^p - o_i^p)^2 \end{aligned} \tag{24}$$

The total network performance is a measure of the summed squared output error over each output node and each training pattern. Sometimes the IP factors are not used, and this doesn't affect the basic form of the performance function. This is because we're interested in finding the extrema of the performance function, and this is unchanged when it is multiplied by a constant factor such as IP.

When a performance function for a particular training problem is specified, the objective is to find a particular set of parameter values (weight values) which minimise/maximise the performance function. For these Mean Squared Error (MSE) performance functions, the aim is to find the weights' values which *minimise* its value.

5.1.1 Linear Networks

For an ADALINE with a linear output node transfer function, equation 23 (neglecting the $2P$ factor) can be re-written as:

$$E_i = \sum_{p=1}^{p}(d_i^p - \mathbf{w}_i^T\mathbf{x}^p)^2 \tag{25}$$

$$= \|\mathbf{d}_i\|_2^2 - 2\mathbf{d}_i^T \mathbf{X}\mathbf{w}_i + \mathbf{w}_i^T \mathbf{X}^T \mathbf{X}\mathbf{w}_i \tag{26}$$
$$= E_i^{\min} + (\widehat{\mathbf{w}}_i - \mathbf{w}_i)^T \mathbf{X}^T \mathbf{X} (\widehat{\mathbf{w}}_i - \mathbf{w}_i) \tag{27}$$

where $\mathbf{d}_i$ $(d_i^1, d_i^2, \ldots, d_i^P)$ is the vector of desired outputs for the i^{th} output node and $\mathbf{X}$ is the matrix of input vectors whose p^{th} row is $\mathbf{x}^p$. So the MSE performance function is a *non-negative, quadratic* function of the adjustable paramters $\mathbf{w}_i$ as illustrated in figure 16. The lowest point of this performance function corresponds to $\mathbf{w}_i = \widehat{\mathbf{w}}_i$ and the corresponding value of the performance function is $E_i^{\min}$. As the weight vector moves away from this optimal value, the errors and the value of the performance function both increase. This bowl-shaped performance function is a simple generalisation of a quadratic function to many inputs and because it is non-negative, it must have a global minima. Sometimes the performance function is more valley-shaped and this can be a cause of slow parameter convergence.

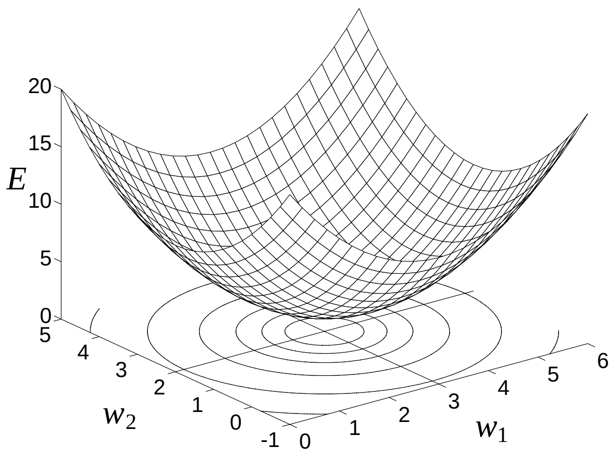


Figure 16: A quadratic MSE performance function for a linear ADALINE with an optimal weight vector $\widehat{\mathbf{w}}_i = (3, 2)$.

The shape of the performance function is determined by the matrix:

$$\mathbf{R} = \mathbf{X}^T \mathbf{X} \tag{28}$$

where $r_{ij} = \sum_{p=1}^{P} x_i^p x_j^p$ determines the degree of the correlation of the inputs in the training set. The matrix $\mathbf{R}$ determines how the errors in the weight vector affect the performance function in equation 27. Finding the eigenvalues and eigenvectors of this matrix tells us which weight changes the performance function is sensitive to, and how well-conditioned the learning problem is (given by the condition[5] of $\mathbf{R}$).

5.2 Iterative Parameter Identification Methods

Iterative parameter identification methods such as the Perceptron learning rule have the basic form:

$$w_{ij}(t+1) = w_{ij}(t) + \eta\, \Delta w_{ij} \tag{29}$$

[5]The condition of a matrix is given by the ratio of the largest eigenvalue over the smallest. It is well conditioned if this ratio is close to unity and badly conditioned if this ratio is large.

where η is the *learning rate* and Δw_{ij} is the recommended weight change for the ij^{th} weight. Different iterative parameter estimation techniques are distinguished by method and information used to calculate Δw_{ij}. The Perceptron training rule uses only one piece of training data, $\{\mathbf{x}^p, \mathbf{d}^p\}$, in order to update the weights whereas other learning rules may use all the information in the training set before adapting the weights.

The **learning rate** determines how fast the parameters adapt. Having a learning rate which is too small causes slow weight changes and hence a slow reduction in the performance function. Too large a learning rate, however, may cause *unstable* learning and hence an increase in the performance function's value. Selection of an appropriate value for the learning rate is therefore critical to the successful application of such learning rules.

An **epoch** is the number of training pairs used to estimate the weight change, and can range from a value of 1 where the weights are adapted after the presentation of each training pair to a value of P, where all the training data is presented to the network and then the weights are updated. Generally, when the epoch size is 1, the training rule is known as an *instantaneous* learning algorithm and when it is P, it is known as a *batch* learning algorithm. *Gradient descent* rules for both batch and instantaneous training will now be derived and described.

5.2.1 Batch Gradient and Steepest Descent Learning

Gradient descent is a parameter optimisation technique where the weights are updated in a direction parallel the *negative gradient* of the performance function. Batch gradient descent evaluates the error for each training sample, before updating the weights. The weight update rule can therefore be described by:

$$\Delta w_{ij} = -\frac{\partial E_i}{\partial w_{ij}} \tag{30}$$

Denoting the value of the performance function for a particular combination of weights, $\mathbf{w}$, by $E_i(\mathbf{w})$ and using a Taylor series expansion, the new value of the performance function is approximated by:

$$E_i(\mathbf{w}(t+1)) \approx E_i(\mathbf{w}(t)) - \eta \left\| \frac{\partial E_i(\mathbf{w}(t))}{\partial \mathbf{w}_i} \right\|_2^2 \tag{31}$$

for a suitably small *positive* learning rate η. Therefore, for a suitably small step size η, the performance function's output is reduced and the parameters are "closer" to their optimal values, and this is illustrated in figure 17 when there exists a single adjustable parameter. In addition, when the parameters achieve their optimal values:

$$\Delta w_{ij} \approx 0 \tag{32}$$

hence learning is switched off. Therefore the gradient descent learning rules have the desired characteristics:

- it is turned off at the extrema of the performance function and
- the weight updates reduce the value of the performance function, for a suitably small step size.

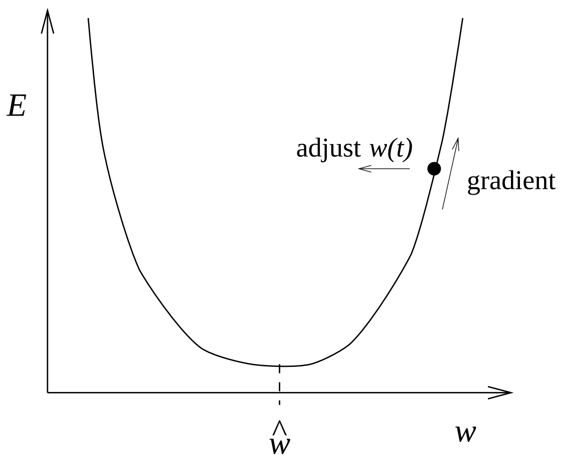


Figure 17: Gradient descent for a single adjustable parameter w.

From equation 26, it is possible to obtain an expression for the gradient:

$$\frac{\partial E_i}{\partial \mathbf{w}_i} = -2\mathbf{X}^T\mathbf{d}_i + 2(\mathbf{X}^T\mathbf{X})\mathbf{w}_i \tag{33}$$

which gives a learning rule of the form:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta\left(\mathbf{X}^T\mathbf{d}_i - (\mathbf{X}^T\mathbf{X})\mathbf{w}_i\right) \tag{34}$$

where the factor of 2 has been absorbed into the learning rate. It can be shown that this learning algorithm is equivalent to the ADALINE learning procedure, averaged over each pattern in the data set. The convergence properties of this training procedure are illustrated in figure 18 for both a well-conditioned and a badly-conditioned set of data with two adaptive weights.

The rate of convergence and the stability of this learning algorithm is related to the choice of the learning rate η. For stable learning, it should be selected such that:

$$0 < \eta < \frac{2}{\lambda_{\max}} \tag{35}$$

where $\lambda_{\max}$ is the largest eigenvalue of matrix $\mathbf{R}$[6]. However, this can cause slow learning, especially when the data is badly conditioned, as illustrated in figure 18.

In the *steepest descent* learning algorithm, the learning rate is calculated at each iteration, so that the search direction $\mathbf{s}(t)$ is chosen according to:

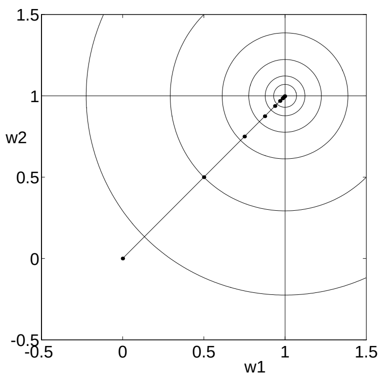
$$\mathbf{s}(t) = \mathbf{X}^T\mathbf{d}_i - (\mathbf{X}^T\mathbf{X})\mathbf{w}_i(t) \tag{36}$$

and the learning rate $\eta(t)$ is calculated to minimise the performance function in this direction:

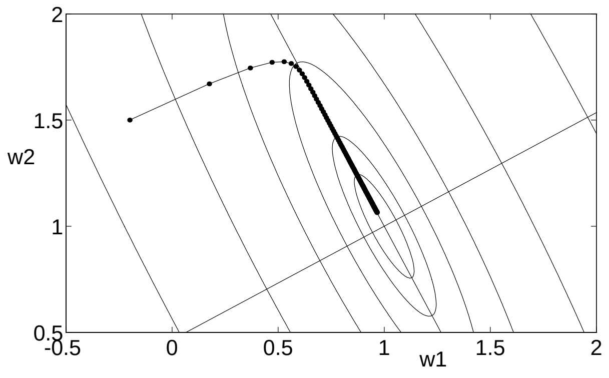
$$\eta(t) = \frac{\mathbf{s}^T(t)\,\mathbf{s}(t)}{\mathbf{s}^T(t)\,\mathbf{R}\,\mathbf{s}(t)} \tag{37}$$

where $\mathbf{R}$ is given in equation 28. The steepest descent formula can be calculated recursively, thus reducing the cost of implementing the learning rule, and its behaviour is illustrated in figure 19.

[6]Choosing the value of the learning rate is related to finding the *curvature* (second derivative) of the performance function.



(a)



(b)

Figure 18: Gradient descent learning for well-conditioned (a) and ill-conditioned (b) learning problems.

5.2.2 Instantaneous Gradient Descent and Steepest Descent

The LMS algorithm attempts to find the optimal weight vectors by performing a "drunken" *gradient descent* walk down the sides of this performance function. This is because, instead of performing gradient descent on the complete, batch performance function, it uses the instantaneous estimate (see equation 22) and updates the weights after each presentation of a training pair. Hence, the instantaneous estimate of the gradient is noisy and weight updates appear to move in sometimes random directions.

The gradient of the instantaneous performance function E_i^p is given by:

$$\begin{aligned} \frac{\partial E_i^p}{\partial w_{ij}} &= \frac{E_i^p}{\partial o_i^p}\frac{\partial o_i^p}{\partial w_{ij}} && (38) \\ &= -2\,(d_i^p - o_i^p)\,x_j^p && (39) \end{aligned}$$

which is simply the (negative) output error, δ_i^p, multiplied by the incoming input signal, x_j^p. Therefore, the instantaneous gradient descent LMS rule is given by:

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \eta\,(d_i^p - o_i^p)\,\mathbf{x}^p \qquad (40)$$

where the factor of 2 has again been absorbed into the learning rate η.

To evaluate the effect of the LMS rule on the output error, consider taking the inner product of equation 40 with $\mathbf{x}^p$:

$$\mathbf{w}_i^T(t+1)\mathbf{x}^p = \mathbf{w}_i^T(t)\mathbf{x}^p + \eta\,(d_i^p - o_i^p)\,\|\mathbf{x}^p\|_2^2 \qquad (41)$$

multiplying both sides by -1 and adding d_i^p gives:

$$\begin{aligned} d_i^p - o_i^p(t+1) &= (d_i^p - o_i^p(t)) - \eta\,(d_i^p - o_i^p(t))\,\|\mathbf{x}^p\|_2^2 && (42) \\ &= (1 - \eta\,\|\mathbf{x}^p\|_2^2)(d_i^p - o_i^p(t)) && (43) \end{aligned}$$

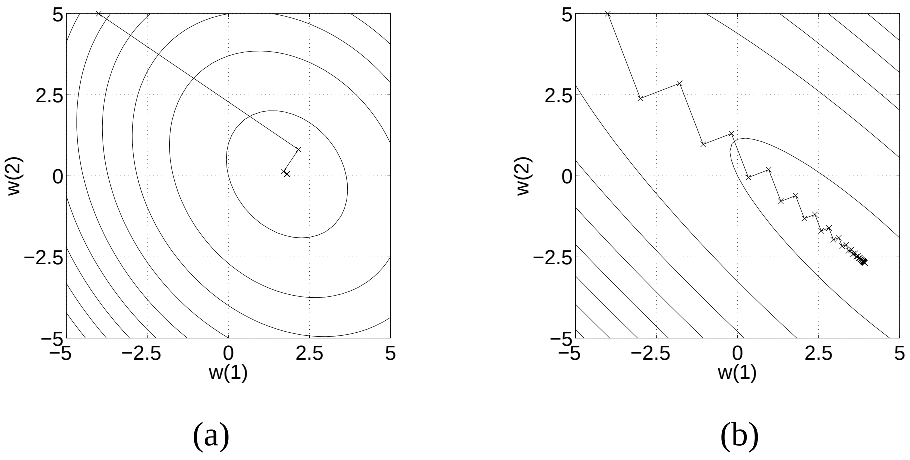


Figure 19: Steepest descent learning for well-conditioned (a) and ill-conditioned (b) learning problems.

so the output error $(d_i^p - o_i^p(t))$ is reduced by a factor $(1 - \eta \, \|\mathbf{x}^p\|_2^2)$. When the learning rate is chosen (adaptively) according to:

$$\eta = \frac{1}{\|\mathbf{x}^p\|_2^2} \tag{44}$$

the output error is reduced to *zero* for that training pattern after the application of the LMS rule. Strictly speaking, this is now an *instantaneous steepest descent*-type learning rule where the learning rate is modified for each learning iteration, and the instantaneous steepest descent rule is known as the Normalised Least Mean Square (NLMS) algorithm, described by:

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \eta \, \frac{(d_i^p - o_i^p) \, \mathbf{x}^p}{\|\mathbf{x}^p\|_2^2} \tag{45}$$

where the learning rate should lie in the range (0,2) for stable learning and the output error is reduced to zero when $\eta = 1$. When the training set just consists of one sample, the batch steepest descent learning algorithm given in equation 36 and 37, reduces to this expression.

Figure 20 shows the convergence of the weights for the NLMS rule, for both a well-conditioned and an ill-conditioned learning problem. The LMS and NLMS rules perform a kind of "drunken walk" down the performance surface and the noisy path is due to the instantaneous *estimate* of the gradient. The amount of noise is influenced by the condition of $\mathbf{R}$, so if the learning problem is badly conditioned, normal gradient and steepest descent rules will be slow and the instantaneous versions will contain a lot of noise. Hence it is important to choose the data and select a network structure which makes the learning problem as simple as possible.

6 Multi-Layer Networks

Multi-layer Perceptron networks were responsible for the decline and resurgance of interest in the field of neural computing. Perceptron networks with a hidden layer can extract nonlinear concepts from the data, thus overcoming the restriction that SLP networks can only act as

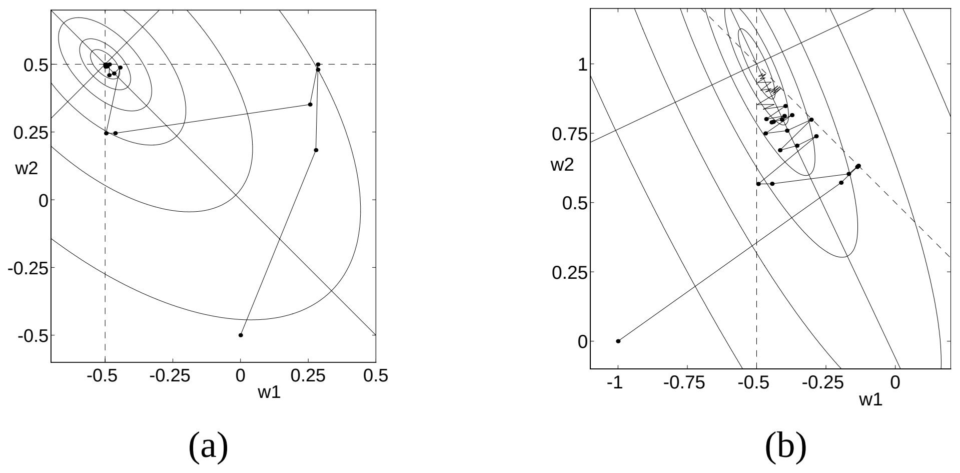


Figure 20: NLMS learning for well-conditioned (a) and ill-conditioned (b) learning problems.

linear discriminants. However, nonlinear training algorithms must now be used and the convergence proofs associated with the SLP (convergence in a finite number of iterations) can no longer by applied.

A typical MLP network with one hidden layer is shown in figure 21. Networks may have more than one hidden layer and each layer can contain a variable number of nodes. Each layer performs a *transformation* of the signals received from the previous layer and hopefully the representation is improved for the subsequent layers.

These MLP networks can be used both for function approximation (surface fitting) and classification where the output nodes approximate the *posterior* class probabilities (the probability of an input lying in a certain class). Generally, the output nodes have linear transfer functions, but they could easily be sigmoidal, although when sigmoidal output nodes are used, the target data should be scaled to lie in the unit interval. The hidden layer nodes' transfer functions should *not* be linear as two weight layers would have the same effect as a single one (the product of two matrices is another matrix), so the sigmoidal nodes in the hidden layers develop *nonlinear* representations. It is important to allow the hidden layer weights to adapt as if they are regarded as fixed, the same criticisms that were used to describe the SLP can be applied to these MLP networks. Indeed, during the Sixties it was well-known that MLP networks could implement arbitrary logical functions, but a training rule for the hidden nodes had not been found.

6.1 Node Functionality

The nodes inside an MLP are similar to those used in the SLP and ADALINE networks, except that continuous transfer functions are usually employed instead of the binary, hard limiter. Therefore, the incoming input signals are multiplied by the weights and summed to produce an activation value. This is then passed through a linear or sigmoidal-type transfer function, as described in equation 4.

In the SLP network, there was *almost* a free parameter, as the weight vector could be scaled

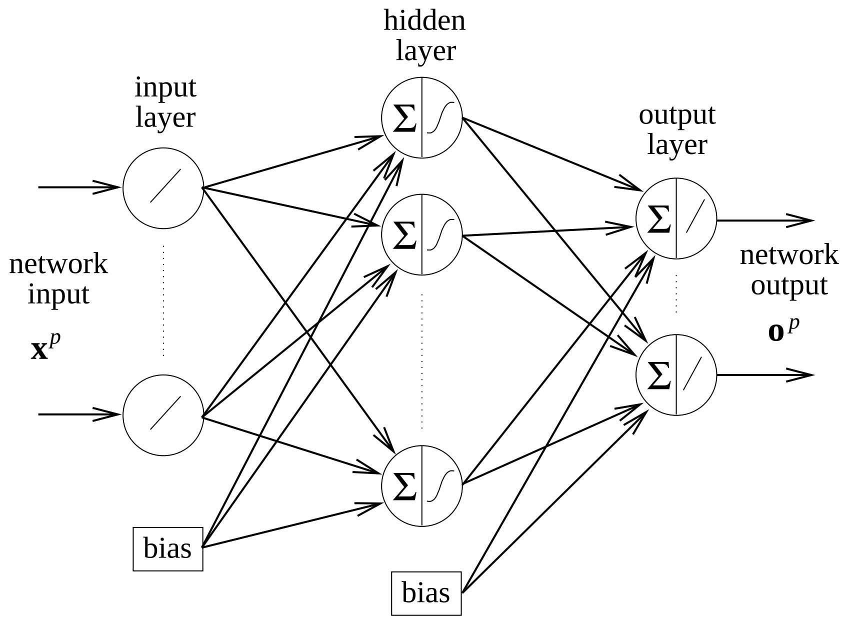


Figure 21: A typical 2-layer MLP with sigmoidal nodes in the hidden layer and linear nodes in the output layer.

by an arbitrary non-zero value and the decision line (hyperplane) was unchanged. Remember that this fact was used in the convergence proofs for the SLP training rule, as the optimal weight vector $\widehat{\mathbf{w}}$ was chosen such that $\widehat{\mathbf{w}}^T\mathbf{x}^p > (a+b)/2$. However, for continuous transfer functions, the *size* of a node's activation value is an important quantity as it determines the value of the node's output, see figure 3. This is obvious for a linear transfer function.

To interpret this quantity, we re-write the node's activation as:

$$i_i = \left(\|\mathbf{w}_i\|_2^2 \times \frac{\mathbf{w}_i}{\|\mathbf{w}_i\|_2^2} \right)^T \mathbf{x} \tag{46}$$

where $\|\mathbf{w}_i\|_2^2$ is the *size* of the weight vector and $\mathbf{w}_i/\|\mathbf{w}_i\|_2^2$ is the *unit* normal vector. The latter quantity specifies how the weight vector is oriented and the former indicates the strength of the node which can be regarded as the k multiplier in equation 4. Figure 22 shows two sigmoidal nodes with differing orientations and magnitudes.

These functions are constant along lines for which:

$$i_i = c \tag{47}$$

where c is a constant, i.e.

$$\mathbf{w}_i^T\mathbf{x} = c \tag{48}$$

Hence they are sometimes referred to as *global ridge functions.*

6.2 Construction Algorithm for Determining the Weights

From figure 7, it is clear that no single-layer device can compute the logical parity function (eXclusive OR function, or its dual, the equivalence function) since the problem is not linearly

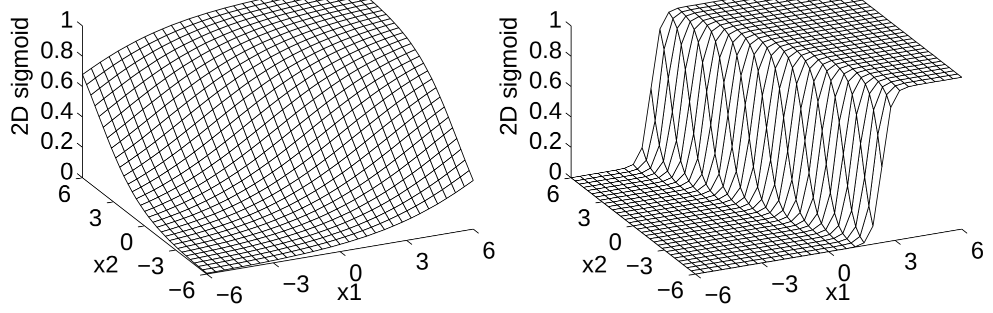


Figure 22: Two 2-dimensional sigmoidal transfer functions with differing orientations and magnitudes.

separable. That is, no single straight line can separate points (1,0) and (0,1) from (0,0) and (1,1). Although depicted for two dimensions, the same is obviously true for any number of inputs.

Computation of the parity function requires more than a single layer. As depicted in figure 11 for the 2-input case, we can use two M&P neurons to separate each of the (1,0) and the (0,1) points from the remainder but their output must then be combined in a subsequent OR layer. This construction illustrates that, for binary units at least, one hidden layer is sufficient to perform any desired (consistent) mapping.

This is achieved by allocating a hidden layer node (hard limiting transfer function) to each positive sample (desired output = 1) in the training data set. The weights are chosen so that this unit will only be turned on for exactly the same input, and all of hidden layer nodes are combined using an output OR layer. Therefore, the network will be on only for those positive examples in the training set and be turned off otherwise, and this means it can store any logical function. This is not surprising as it is equivalent to expressing any logical function in its Conjunctive Normal Form. Note that although an algorithm has been described for positive samples, a similar procedure can be derived so that the network's hidden layer nodes respond to negative (zero output) samples. In practice, the class with the *least* number of training samples would be used to construct the network, as this would ensure that the hidden layer had the smallest number of nodes.

The algorithm simply sets the weights for a node in the hidden layer (positive sample) equal to 1 if the input is switched on and -1 otherwise. The bias for this node is then set equal to (minus) the total number of inputs switched on +0.5, which ensures that its activation value is greater than zero *only* for that input pattern. The weights in the output layer are all set equal to one, and the bias associated with that output node is -0.5, and the network will produce a positive response if any of the hidden layer nodes are switched on.

6.2.1 Example

Consider a training set with 4 examples, as shown in table 1. This has 3 input nodes, and two of the training set are positive examples so the hidden layer nodes will be constructed to respond to those inputs. and the output layer will represent the logical OR.

p	$\mathbf{x}^p$	d^p
1	(0 1 1 0)	1
2	(0 1 0 1)	0
3	(1 0 1 1)	1
4	(1 0 0 1)	0

Table 1: A set of binary training data used to directly construct an MLP network.

Consider choosing the weight vector of the 1st hidden layer node to remember the first training pattern. The weight should be 1 whenever the input is on and -1 otherwise. Similarly, the bias term should be the number of inputs swiched on - 0.5. Therefore, the hidden layer weight vector $\mathbf{w}_1$ should be:

$$(w_{10}, w_{11}, w_{12}, w_{13}, w_{14}) = (-1.5, -1, 1, 1, -1)$$

Similarly, the second hidden layer node should only respond to the third input pattern so:

$$(w_{20}, w_{21}, w_{22}, w_{23}, w_{24}) = (-2.5, 1, -1, 1, 1)$$

Finally, the output node should form the logical OR of these two nodes in the hidden layer, so its weight vector is simply unity for every incoming weight and -0.5 for the bias term.

Note that this is the construction used to set the weights in the MLP solution to the XOR problem in section 3.2.

6.2.2 Comments

This approach to building multi-layer networks has three clear problems:

- Because of the localist hidden representation, it is inherently inefficient. We need as many hidden units as there are positive training instances. Of course, if the number of *negative* training instances is less, we can always effect the dual construction but, in the worst case, we will still need $P/2$ hidden nodes where P is the size of the training set.
- Again because of the localist representation, there is no generalisation to unseen inputs. In fact, the arbitrary decision for inputs not contained in the training set is to always assign the output class for which was not used in the construction procedure. Therefore, the choice about which instances (positive or negative) are used in the construction procedure determines how the network generalises to *all* unseen cases.
- We would like the MLP to be trainable, rather than having to be constructed, in a way which produces a distributed representation, so offering generalisation capability in a network of reasonable size. Yet neither the Perceptron training rule nor the Widrow-Hoff (delta) rule can be used for networks with hidden nodes. These rules require target outputs, which can be obtained from labelled training data. But such training data do not tell us what the outputs from hidden units should be.

In the next section, we consider the very important error back-propagation rule which allows us to train MLPs from example data, before considering what size network is appropriate to a given problem.

6.3 The Error Back-Propagation Algorithm

The Error Back-Propagation (EBP) algorithm is widely regarded as the main factor behind the recent revival of interest in neural computation. Its main feature was that for the first time it allowed the weight vectors associated with the hidden nodes in an MLP to be adapted, using a relatively simple optimisation procedure. It is a *supervised* learning technique which requires a data set of input-output training pairs and the EBP algorithm adjusts the MLPs weights so that the network's output is close to the desired behaviour. The learning rule requires that the network uses continuous transfer functions in each node, so rather than incorporating the thresholding nonlinear activations functions found in an SLP, it uses the smoother linear and sigmoidal functions. However, unlike the SLP learning rule, it is *not* guaranteed to converge and learning can often be very slow.

6.4 Derivation of the Error Back Propagation Algorithm

We follow here the classical development of the rule as given by Rumelhart, Hinton and Williams (1986) and Rumelhart, Hinton and McClelland (1986). This is a three-stage development:

Stage 1: formulate the learning problem as a nonlinear gradient descent procedure for minimising the MSE,

Stage 2: derive the gradient descent rule for the output layer of weights with a differentiable transfer function which is similar to the Widrow-Hoff delta rule, and

Stage 3: generalise the gradient descent implicit in the delta rule to non-linear units and to multiple layers.

In view of Stage 3, the EBP algorithm is sometimes called the *generalised delta rule.*

6.4.1 Stage 1: Performance Function

The performance function for the MLP networks is generally the MSE which was used to derive the ADALINE training algorithm, see equation 22. Like the ADALINE network, the instantaneous version is often used, although the batch version can be easily derived by simply averaging over each pattern in the training set. Also, as the network is allowed to have multiple output nodes (which are independent for the output weight layer, but coupled for the hidden layers of weights), so the MSE should be an average measure over each output node.

Therefore, the instantaneous MSE performance function is:

$$E^p = \frac{1}{2}\sum_{i=1}^{I}(d_i^p - o_i^p)^2$$

and the EBP algorithm is the instantaneous gradient descent rule applied to nonlinear MLP networks.

6.4.2 Stage 2: Output Layer

Assuming that the output nodes have a differentiable transfer function $f()$, the i^{th} node's input-output relationship is expressed as:

$$o_i^p = f(i_i^p) \tag{49}$$

where the activation value is given by:

$$i_i^p = \sum_{j=0}^{J} w_{ij}\, a_j^p \tag{50}$$

and a_j^p is the output of the j^{th} node in the hidden layer. This is equivalent to the ADALINE network in the training phase, except that the linear transfer function has been replaced by a *differentiable* one.

To derive the gradient descent rule, it is necessary to calculate:

$$\begin{aligned}
\frac{\partial E^p}{\partial w_{ij}} &= \frac{\partial E_i^p}{\partial o_i^p}\,\frac{\partial o_i^p}{\partial i_i^p}\,\frac{\partial i_i^p}{\partial w_{ij}} && (51)\\
&= -(d_i^p - o_i^p)\, f'(i_i^p)\, a_j^p && (52)\\
&= -\delta_i^p\, a_j^p && (53)
\end{aligned}$$

where $\delta_i^p = (d_i^p - o_i^p)\, f'(i_i^p)$ is the "back-propagated error" used in the weight update equations and $f'(i_i^p)$ is the derivative of the transfer function. It was also possible to replace E^p by E_i^p because, for the output layer, the weight vectors and hence performance measures are independent. This produces a learning rule for the output layer of the form:

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \eta\,(d_i^p - o_i^p)\, f'(i_i^p)\,\mathbf{a}^p \tag{54}$$

which is similar to the Widrow-Hoff LMS rule, except that the use of a nonlinear transfer function has caused the introduction of the extra factor $f'(i_i^p)$ (which is unity for the linear tranfer function used in the Widrow-Hoff rule). This relationship between a weight vector for the i^{th} output node and the performance function is illustrated in figure 23.

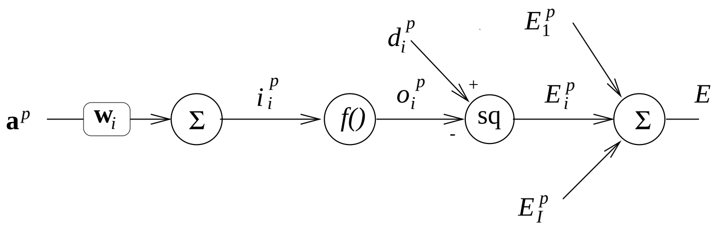


Figure 23: The mapping between an output layer weight w_{ij} and the performance function E_i^p

6.4.3 Stage 3: Hidden Layer

The training rule for the weights in the hidden layer is another gradient descent learning algorithm where:

$$\Delta w_{ij}^{l,p} = -\frac{\partial E^p}{\partial w_{ij}^l}$$

for the ij^{th} weight in the l^{th} hidden layer w_{ij}^l. The basic idea is to obtain a recursive expression for the "output errors" $\delta_i^{l,p}$ such that:

$$\delta_i^{l,p} = g(\delta^{l-1,p}) \tag{55}$$

where $g()$ is some function of the "output errors" on the next layer. Hence, by priming the "output errors" on the output layer (layer 1, $\delta_i^{1,p}$), using the expression given in equation 53, it is possible to *back-propagate* this signal to the preceeding layer, hence the name of the learning rule. This is illustrated in figure 24.

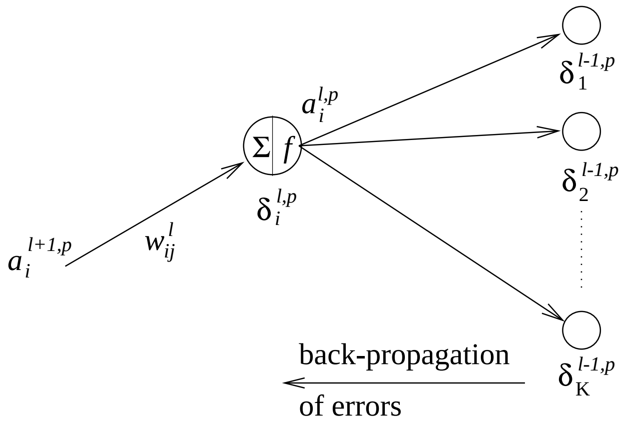


Figure 24: The mapping between two successive hidden layer "error signals" δ_j^{l-1} and δ_k^l.

Now, using the chain rule:

$$\frac{\partial E^p}{\partial w_{ij}^l} = \frac{\partial E^p}{\partial i_i^{l,p}} \frac{\partial i_i^{l,p}}{\partial w_{ij}^l} \tag{56}$$

and by definition:

$$\delta_i^{l,p} = \frac{\partial E^p}{\partial i_i^{l,p}} \tag{57}$$

and $a_j^{l+1,p}$ is the incoming input signal from the $l+1$ layer. Therefore, equation 56 can be written as:

$$\frac{\partial E^p}{\partial w_{ij}^l} = \delta_i^{l,p} a_j^{l+1,p} \tag{58}$$

Using the chain rule for $\delta_i^{l,p}$ produces

$$\begin{aligned}\delta_i^{l,p} &= \sum_k \frac{\partial E^p}{\partial i_k^{l-1,p}} \frac{\partial i_k^{l-1,p}}{\partial a_i^{l,p}} \frac{\partial a_i^{l,p}}{\partial i_i^{l,p}} \\ &= \sum_k \delta_k^{l-1,p} \, w_{ik} \, f'(i_i^{l,p})\end{aligned}$$

This gives us the required expression for $\delta_i^{l,p}$ in terms of $\delta_k^{l-1,p}$, thus the derivative can be "back-propagated" through the MLP network and any number of hidden layers can be trained in this manner.

Note as was mentioned in section 5.2.1, the batch version of this learning rule can be obtained by simply averaging the recommended weight updates for each training pattern in the data set.

6.5 Derivative of the Transfer Function

A popular transfer function for use in back-propagation networks is the *sigmoidal* or *logistic* function:

$$o_i^p = \frac{1}{1+\exp(-i_i^p)}$$

Not only is this differentiable and non-decreasing (i.e. semi-linear), it also has a particularly simple derivative.

The EBP algorithm requires that differentiable transfer functions be used in each layer, otherwise the derivative cannot be calculated. Therefore the quantity:

$$f'(i_i^p) = \frac{\partial a_i^p}{\partial i_i^p}$$

needs to be calculated for each node in each layer in the network and it can easily be shown that:

$$f'(i_i^p) = a_i^p(1-a_i^p)$$

which lies in (0,0.25], so it is possible to efficiently evaluate the transfer function's derivative. Another popular transfer function is the tanh() function which is described by:

$$\tanh(i_i^p) = \frac{1-\exp(-2i_i^p)}{1+\exp(-2i_i^p)} \tag{59}$$

as this produces an output in the range (-1,1) rather than (0,1). The derivative of this transfer function is described by:

$$f'(i_i^p) = (1-(a_i^p)^2) \tag{60}$$

which lies in $(0,1]$, so again it has a fairly simple formulation.

These two transfer functions have a significant value for the set of inputs which produce a near zero activation and conversely their value is small when the activation is large (positive or negative). This is illustrated in figure 25, for the two 2-input transfer functions shown in figure 22. Therefore, significant learning will only take place when the input vectors lie along ridges, thereby living up to their name of ridge functions.

This is easier to visualise by considering the activation to node output mapping. The normal sigmoid and the tanh() function together with their derivatives are shown in figure 26. and it can be clearly seen that the derivative of the sigmoidal transfer function is a lot less than the derivative of the tanh() function. This is an important point to notice because the the EBP learning algorithm multiplies the "output error" by the deriviative of the transfer function *at*

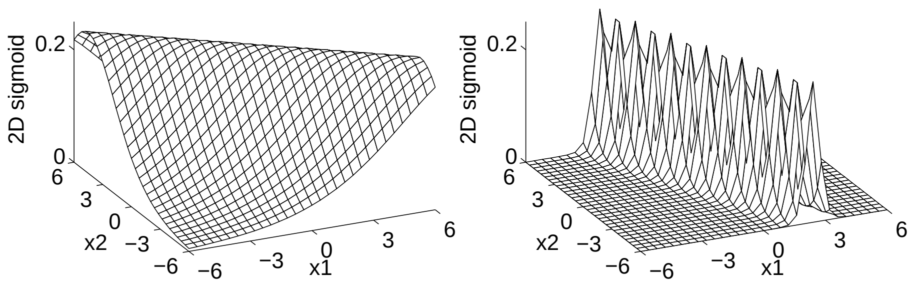


Figure 25: Derivative of the transfer functions for the 2-input sigmoidal transfer functions shown in figure 22.

each layer. Hence, when this derivative is a small value, it effectively *squashes* the training signal down to zero. This is compounded by increasing the number of hidden layers as the derivative values are multiplied together.

It is possible to choose the learning rate to partially compensate for this effect and some authors have proposed assigning a separate learning rate to each layer, thus effectively re-scaling the effect of the transfer functions' derivative on each layer.

6.6 Starting and Stopping Training

When you start training a MLP network using EBP, all the weights in the network should be initialised to be *small random* numbers. This allows nodes in the hidden layer to develop appropriate nonlinear representations, and choosing small values means that the initial value of the weights do not make it difficult for the network to learn. This is because if i_i is large (positive or negative), $f'(i_i)$ is tiny and when $i_i \approx 0$ (small weights), then $f'(i_i) \approx 0.25$.

Training is stopped whenever the value of the performance function is sufficiently small $J < \epsilon$ or when the number of iterations is unacceptably large. Remember that this learning algorithm is not guaranteed to converge, unlike the single-layer version. This involves estimating the desired accuracy of the model and choosing too small a value may cause *over-fitting*, especially when there training data contains noise.

6.7 Nonlinear Performance Functions

One of the reasons why the nonlinear gradient descent EBP algorithm may fail to converge is the possibility of getting stuck in a local minima or on a plateau area as shown in figure 27. The quadratic-shaped MSE performance function only occurs for *linear* networks, and because the MLP networks have nonlinear weights in their hidden layers, the shape of the performance function can be very complex. There may be multiple (local or global) minima and large plateau regions where the gradient is almost zero. Clearly, a gradient-based optimisation

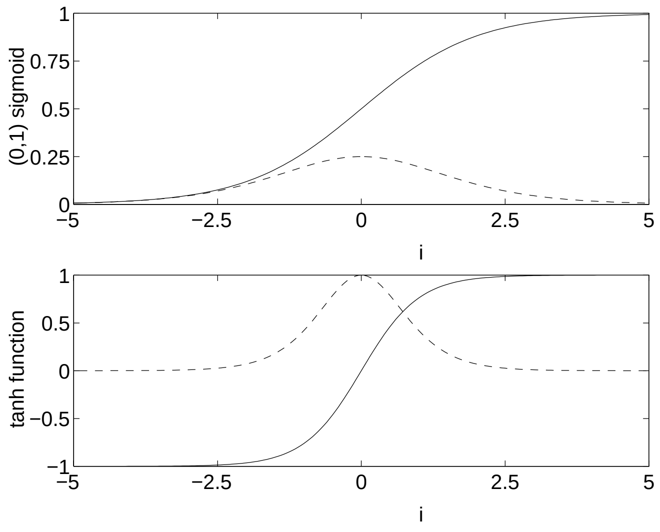


Figure 26: Derivative of the normal sigmoidal and tanh(.) transfer functions.

technique will get trapped in either of these regions as:

$$\Delta w = -\eta \frac{\partial E}{\partial w} \approx 0$$

and learning will cease prematurily.

Various modifications have been proposed to the EBP algorithm to overcome some of its well known deficiencies and these include:

- Adding a small *random* step to the weight, so that it explores new directions.
- Including a *momentum* term so that the weight keeps moving in the same direction:

  $$\Delta w(t) = -\eta \frac{\partial E}{\partial w} + \mu \Delta w(t-1) \tag{61}$$

  This may help EBP get off plateau regions, but it can cause oscillatory behaviour close to a local or global minima.
- Using second-order learning algorithms which make use of second derivation information as well as the gradient (first order) in the learning algorithm.

In practice, the basic EBP algorithm works fairly well, although sometimes either taking a random step or re-starting from a point nearby can overcome some of the problems associated with local minima.

7 Analysis of the Trained Network

The EBP gives us a means of training a layered feedforward net with hidden units, but the network size is a given. How then do we decide:

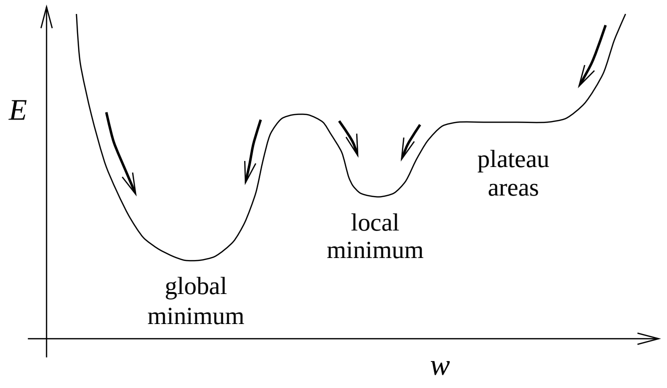


Figure 27: A performance function of a nonlinear parameter w which contains local minima and a plateau region.

- how many input units?
- how many output units?
- how many hidden layers?
- how many hidden units in each layer?

In general, the issues of input and output representation do not cause undue difficulty. It is usually adequate to let the nature of the problem dictate the input/output codings and, thereby, the numbers of units in the input and output layers. The main dimension of choice is a localist versus a distributed representation, with the final selection often based on empirical investigation. As a working principle, however, the more "meaningful", or *iconic*, the representations, the easier it is for the net to learn an appropriate internal representation (weight set). Thus, we might prefer a 'thermometer' coding of amplitude rather than, say, the more abstract binary-coded decimal representation.

The issues of number and size of hidden layers in MLPs are somewhat more difficult. We have already seen by the construction in section 6.2 that a single layer is adequate for any input/output mapping problem using binary units, although a large number of hidden units ($O(P)$ where there are P training instances) is required. However, these results do not generalise to networks with real-valued inputs and outputs. Adding extra nodes into a hidden layer increases the network's flexibility and as such it has the potential to model the data more accurately than a simple network. But this result can be misleading, as any real-world data set is simply a finite-sample of the true underlying function and modelling the data too accurately may mean that the network's generalisation abilities are poor, as illustrated in figure 28.

In section 3, the generalisation abilities of a SLP network was described in terms of its geometrical partitioning of the input space, and section 6.1 showed how the node's weight vector may be interpreted. The only real way of characterising an MLP network's generalisation capabilities is to analyse its structure and assess whether or not it does anything unexpected.

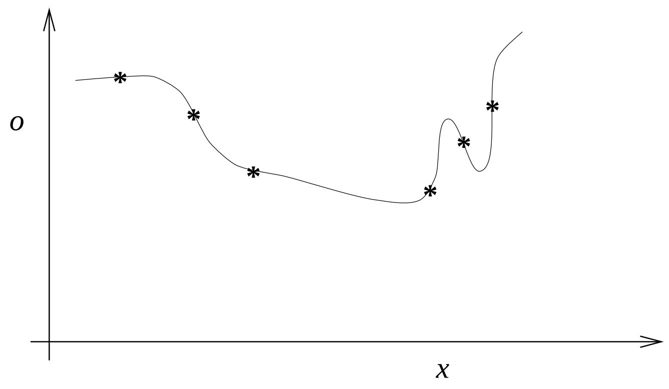


Figure 28: Overgeneralisation which occurs when the network is too flexible.

7.1 An Analysis of a 2-Layer MLP Network

Using the analysis of an MLP node given in section 6.1, it is possible to characterise how nodes are distributed and which nodes are similar in the hidden layer, and as such gain an insight into the nonlinear transformations performed by the hidden layer. For each node, the set of inputs for which:

$$i_i = 0 \tag{62}$$

cause:

$$f(i_i) = 0.5 \tag{63}$$

and these can be denoted by the "centre" of the node. The size of the transfer function's slope at this centre is given by:

$$f'(0.5) = \|\mathbf{w}_i\|_2 \tag{64}$$

as the inputs move perpendicularly away from the input centre plane. Therefore, hidden layer nodes with similar weight vectors (size and orientation) will contribute similar amounts to the network's output. By investigating the output layer weights as well, the sign of that nodes contribution to the output can be determined and the overall effect of similar nodes can be investigated. If similar nodes in the hidden layer have similar output layer weights, the network is effectively singular and it may be possible to reduce the number of nodes. If an output layer weight for any node is almost zero, this may imply that a node can be removed as well. However, to really assess the effect of adding or removing a node in the hidden layer, the network *must* be retrained and its possible increase in the MSE noted.

References

[BryHo69] Bryson, A.E. and Ho, Y-C. (1969) *Applied Optimal Control*, Blaisdell, Waltham, MA.

[Cov65] Cover, T.M. (1965) "Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition", *IEEE Transactions on Electronic Computers*, **EC-14**, 326–334.

[Cyb88] Cybenko, G. (1988) *Continuous Valued Neural Networks with Two Hidden Layers are Sufficient*, Technical Report, Department of Computer Science, Tufts University, Medford, MA.

[Cun85] le Cun, Y. (1985) "Une procédure d'apprentissage pour réseau à seuil assymétrique", *Cognitiva 85: A la Frontière de l'Intelligence Artificielle des Sciences de la Connaissance des Neurosciences*, CESTA, Paris, pp. 599–604.

[Dre62] Dreyfus, S. (1962) "The numerical solution of variational problems", *Math. Anal. Appl.*, **5**, 30–45.

[Horetal89] Hornik, K., Stinchcome, M. and White, H. (1989) "Multilayer feedforward networks are universal approximators", *Neural Networks*, **2**, 359–366.

[KroSma93] Kröse, B. and van der Smagt, P. (1993) "An Introduction to Neural Networks", Technical Report, Department of Computer Science, University of Amsterdam.

[MinPap69] Minsky, M. and Papert, S. (1969) *Perceptrons: an Introduction to Computational Geometry*, MIT Press, Cambridge, MA.

[MirCao89] Mirchandini, G. and Cao, W. (1989) "On hidden nodes in neural networks", *IEEE Transactions on Circuits and Systems*, **CAS-36**, 661–664.

[MitDur89] Mitchison, G.J. and Durbin, R.M. (1989) "Bounds on the learning capacity of some multi-layer networks", *Biological Cybernetics*, **60**, 345–356.

[Ros58] Rosenblatt, F. (1958) "The Perceptron: a probabilistic model for information storage and organization in the brain", *Psychological Review*, **65**, 386–408.

[Ros62] Rosenblatt, F. (1962) *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan, New York.

[Rumetal86a] Rumelhart, D.E., Hinton, G.E. and McClelland, J.L. (1986) "A general framework for parallel distributed processing", in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, D.E. Rumelhart and J.L. McClelland (eds.), Bradford/MIT Press, Cambridge, MA.

[Rumetal86b] Rumelhart, D.E., Hinton, G.E. and Williams. R.J. (1986) "Learning representations by back-propagating errors", *Nature*, **323**, 533–536.

[Wid62] Widrow, B. (1962) "Generalization and information storage in networks of adaline "neurons" ", *in* M.C. Jovitz, G.T. Jacobi and G. Goldstein (eds.), *Self-Organizing Systems*, Spartan, Washington, DC, pp. 96–104.

[WidHof60] Widrow, B. and Hoff, M.E.Jr. (1960) "Adaptive switching circuits", *IRE Western Electric Show and Convention Record (WESCON)*, Part 4, 96–104.

[Wer74] Werbos, P.J. (1974) *Beyond regression: New tools for prediction and analysis in the behavioral sciences*, Doctoral Dissertation, Department of Applied Mathematics, Harvard University, MA.

A Mathematical notation

This appendix contains a brief review of some of the vector and matrix concepts which are used to describe and analyse the structural and learning properties of the layered Perceptron networks.

In these notes, a bold capital letter such as $\mathbf{X}$ represents an $(n \times m)$ matrix with elements x_{ij}, $i = 1, \ldots, n$ and $j = 1, \ldots, m$. A bold lower case letter such as $\mathbf{w}$ represents a vector of length n with elements w_i, $i = 1, \ldots, n$.

A.1 Vector Norms

The *norm* of a vector is simply a measure of its *size*, and is sometimes known as a distance metric or measure. It is useful to measure the size of the error in the weight vector for different training procedures because if you can show that the *size* of the weight vector error tends to zero, then each of the individual weights tend to their optimal values. In this section, we only use the Euclidean norm which is defined by:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{I} x_i^2}$$

which is simply the square root of the sum of the square of the individual elements. In two dimensions ($I = 2$), this simply corresponds to the length of the hypotenuse of a right angled triangle where the other sides are of length x_1 and x_2. The Euclidean norm is simply a generalisation of this idea to vectors with I components. One useful expression which is used several times in these notes is:

$$\mathbf{x}^T\mathbf{x} = \|\mathbf{x}\|_2^2$$

ie., the inner product of a vector with itself is simply the Euclidean norm squared.

Other vector norms can be defined and the two most widely used are the infinity norm:

$$\|\mathbf{x}\|_\infty = \max_i |x_i|$$

and the 1-norm or city block distance:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{I} |x_i|$$

These three norms will give different values for the size of the weight vector, although the following expression always holds:

$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1$$

Strictly speaking, a vector norm is an abstract measure of the vector's size and can be any function $\|.\|$ which satisfies the following expressions:

1. $\|\mathbf{x}\| \geq 0 \qquad \forall \mathbf{x}$

2. $\|\mathbf{x}\| = 0 \qquad \text{iff } \mathbf{x} \equiv \mathbf{0}.$
3. $\|a\mathbf{x}\| = |a| \; \|\mathbf{x}\| \qquad \text{for any scalar } a.$
4. $\|\mathbf{x} + \mathbf{y}\| \leq \mathbf{x} + \mathbf{y}$

Therefore a norm measure should be non-negative (first condition) and is zero if and only if (iff) the vector is identically zero (second condition). The third condition simply says that if we multiply the vector by a factor a, the distance measure is also multiplied the corresponding amount and the triangle inequality (fourth condition) simply says that the size of the sum of two vectors is never greater than the sum of the sizes of the individual vectors.

As an exercise, you may want to verify that the three vector norms described in this section satisfy these four conditions.