Chapter 3

Simple Supervised Learning

3.1 Objectives

After this Chapter, you should

1. understand simple supervised learning.
2. be able to implement a perceptron.
3. be able to implement an Adaline.
4. know the difference between the perceptron learning rule and the LMS rule
5. know simple activation functions commonly used and their properties .
6. know the role of the bias in simple supervised learning.
7. be able to explain linear separability geometrically.
8. understand error descent procedures .

3.2 Introduction

We will consider a simple two layer[1] feedforward ANN in this chapter. We shall be interested in supervised learning (learning with a teacher) in which the network is trained on examples whose target output is known. We therefore must have a training set for which we already know 'the answer' to our questions to the network.

3.3 The perceptron

The simple perceptron was first investigated by Rosenblatt [Rosenblatt,1962]. We will define the activation passing of the perceptron by $NET_i = \sum_{j=1}^{N} w_{ij} x_j$, where
NET_i is the activation of the i^{th} output neuron,
x_j is the firing (or output) of the j^{th} input neuron
and w_{ij} is the weight from the j^{th} input neuron to the i^{th} output neuron.

[1] We will use the convention that this refers to the number of layers of neurons; some authors refer to the number of layers of weights, in which case the networks of this chapter are 1 layer networks

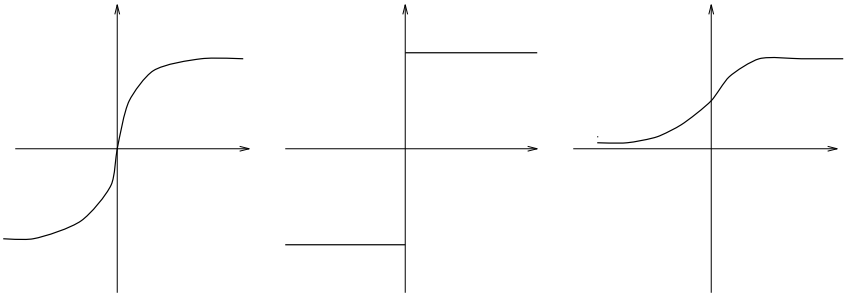

Figure 3.1: The tanh(), Heaviside, and logistic functions

We calculate the output of the i^{th} output neuron by

$$y_i = f(NET_i) = f(\sum_{j=1}^{N} w_{ij} x_j) \tag{3.1}$$

where typically f() is a non-linear function such as a threshold function, a sigmoid or a semi-linear function (see below). We will also write this as

$$y_i = f(NET_i) = f(\mathbf{w}_i.\mathbf{x}) \tag{3.2}$$

where $\mathbf{w}_i = [w_{i0}, w_{i1}, ..., w_{iN}]^T$ is the vector of weights into the i^{th} output neuron and $\mathbf{x} = [x_0, x_1, x_2, ..., x_N]^T$ is the vector of input activations.

We will often omit any mention of a **threshold** in the networks we use since we can, if we need a threshold, select a particular input (often x_0) to be on (i.e. equal to 1) all of the time and choose weights $w_{i0} = \theta_i$, to give threshold θ_i since

$$y_i = f(NET_i) = f(\sum_{j=0}^{N} w_{ij} x_j) = f(\sum_{j=1}^{N} w_{ij} x_j + \theta_i) \tag{3.3}$$

The functions $f()$ are known as **activation functions**. Typical activation functions are shown in Figure 3.1

The Heaviside function, or step function, or sgn function is defined by

$$\begin{aligned} f(x) &= 1, \text{ if } && x > 0 \\ f(x) &= -1, \text{ if } && x < 0 \end{aligned}$$

The other two functions are continuous functions which asymptote at large absolute values of x. The logistic function is defined by

$$f(x) = \frac{1}{1 + e^{-ax}} \tag{3.4}$$

i.e. $f \to 0$ when x is very negative and $f \to 1$ when x is large and positive. The tanh() function similarly asymptotes at -1 and 1. A half-way stage between the step function and the sigmoid functions is the semi-linear function defined by

$$\begin{aligned} f(x) &= 0, && \text{if } x < a \\ f(x) &= 1, && \text{if } x > b \\ f(x) &= \tfrac{(x-a)}{(b-a)}, && \text{for } a < x < b. \end{aligned}$$

We will be using supervised learning with this type of ANN and so when we present the k^{th} input pattern we must also present the network with the k^{th} target pattern. We then attempt to ensure that $y_i^k = D_i^k$ for all output neurons. i.e. the learning process must use the fact that on presentation with the k^{th} input vector we desire to get the k^{th} output vector equal to the k^{th} target vector.

It is possible that the input and output patterns are the same in which case we say that we are performing **autoassociation**; otherwise we have **hereroassociation**. Typically, for this type of feedforward network, we shall be using different patterns at inputs and outputs and are thus performing heteroassociation.

3.4 The perceptron learning rule

In their simplest form, perceptrons consist of binary units; consider a perceptron with N input neurons and a single output neuron. Then the perceptron must learn the mapping $T : \{-1, 1\}^N \longrightarrow \{-1, 1\}$ based on samples of input vectors, $\mathbf{x}$. An example is the mapping $T : \{-1, 1\}^3 \longrightarrow \{-1, 1\}$ defined by

$$\begin{aligned} T : (-1, 1, 1) &\longrightarrow 1 \\ T : (1, 1, -1) &\longrightarrow 1 \\ T : (1, -1, -1) &\longrightarrow -1 \end{aligned}$$

In this mapping we have only defined the mapping on some of the possible inputs - the others are don't care values. However, in order to be deemed to have learned the mapping, the perceptron must get the above three values correct.

The output neuron of the simple perceptron is a linear threshold unit which takes the value 1 or -1 according to the rule

$$\begin{aligned} y &= f(\textstyle\sum_{j=1}^N w_j x_j + \theta) = 1, \quad \text{if } \displaystyle\sum_{j=1}^N w_j x_j + \theta > 0 \\ y &= f(\textstyle\sum_{j=1}^N w_j x_j + \theta) = -1, \quad \text{if } \displaystyle\sum_{j=1}^N w_j x_j + \theta < 0 \end{aligned}$$

i.e. f() is the simple step function. The generalisation to an M-output perceptron is

$$\begin{aligned} y_i &= f(\textstyle\sum_{j=1}^N w_{ij} x_j + \theta_i) = 1, \quad \text{if } \displaystyle\sum_{j=1}^N w_{ij} x_j + \theta_i > 0 \\ y_i &= f(\textstyle\sum_{j=1}^N w_{ij} x_j + \theta_i) = -1, \quad \text{if } \displaystyle\sum_{j=1}^N w_{ij} x_j + \theta_i < 0 \end{aligned}$$

Rosenblatt showed in 1959 that if it was possible for a mapping T to exist, then the perceptron learning algorithm could be guaranteed to converge to it. A proof of the theorem is given in Appendix A.

The algorithm can be described by:

1. begin with the network in a randomised state: the weights between all neurons are set to small random values (between -1 and 1).
2. select an input vector, $\mathbf{x}^k$, from the set of training examples
3. propagate the activation forward through the weights in the network to calculate the output y.
4. if $y^k = D^k$, (i.e. the network's output is correct) return to step 2.

Nut	type A - 1	type A - 2	type A - 3	type B - 1	type B - 2	type B - 3
Length (cm)	2.2	1.5	0.6	2.3	1.3	0.3
Weight (g)	1.4	1.0	0.5	2.0	1.5	1.0

Table 3.1: The lengths and weights of six instances of two types of nuts

5. else change the weights according to $\Delta w_j = \eta x_j^k (D^k - y^k)$ where η is a small positive number known as the **learning rate**. Return to step 2.

Thus we are adjusting the weights in a direction intended to make the output, y^k, more like the target value, D^k, the next time an input like $\mathbf{x}^k$ is given to the network.

We must emphasise the importance of this rule is that it is guaranteed to converge to the answer in a finite time if the answer exists.

3.5 An example problem solvable by a perceptron

Consider a simple classification problem. We wish to train a perceptron to differentiate between two different types of nuts. We have 3 examples of each type of nut and can measure 2 characteristics of each nut: its length and its weight. Then our data is shown in tabular form in Table 3.1. Then we will train a perceptron with the input vectors , $\mathbf{x}$, equal to
(1, 2.2, 1.4) with associated training output 1 (equal to class A)
(1, 1.5, 1.0) with associated training output 1
(1, 0.6, 0.5) with associated training output 1
(1, 2.3, 2.0) with associated training output -1 (equal to class B)
(1, 1.3,1.5) with associated training output -1
(1, 0.3,1.0) with associated training output -1
Note that the initial 1 in each case corresponds to the bias term. Its trainable weight will converge to the bias for the output neuron. So we have a network with three inputs and a single output which will tell us with a 1 if the nut is A or a -1 if the nut is a B. Therefore our network has 3 weights.

Now we set our initial weights to small random values. Select randomly one of the patterns e.g. the third. Feed the activation forward through the weights and perform a thresholding at the output. If the output neuron is firing 1, do nothing. If, however, the output neuron is firing -1, it is wrongly classifying the nut and its weights must be changed. Change each of the three weights according to the perceptron learning rule.

The weights in this simple problem are guaranteed to converge to something like $w_0 = 0.5, w_1 = 2, w_2 = 3$. A diagrammatic representation of the solution is shown in Figure 3.2.

Now it is clear that it is in fact trivial to differentiate between these two classes on the basis of these six instances. However the perceptron has been shown to be capable of solving more difficult problems though we must in any real problem ensure that the data on which we are training the network is representative of the data as a whole. Only then can we state with confidence that we have a perceptron capable of distinguishing the classes.

Statisticians would say that we have created a **decision surface**, or **discrimination line**, between the two classes. Discriminant functions for the classes would be $g_A(x, y)$ and $g_B(x, y)$ where we would state that a point (x,y) belonged to class A if and only if $g_A(x, y) > g_B(x, y)$; otherwise, (x,y) belongs to class B. The line (or plane) which differentiates the classes is $g_A(x, y) - g_B(x, y) = 0$ which is the line shown in the Figure.

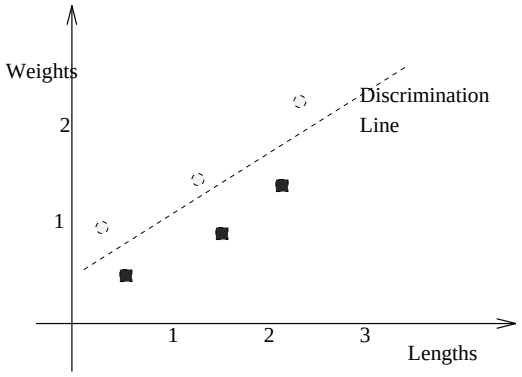

Figure 3.2: The two classes of nut and a discrimination line which can differentiate between the classes

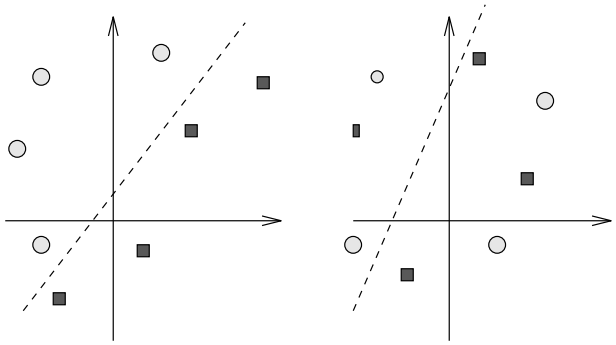

Figure 3.3: In the left part of the diagram we show that we can use a single line to categorise all rectangles as belonging to the region below and to the right of the line while all ovals lie above and to the left of the line ; in the right diagram, the line can no longer discriminate between the classes.

3.5.1 And now the bad news

Now a crucial part of the above description is the '*if it was possible*' part. Minsky and Papert showed that there exist many functions which cannot be solved by this simple mapping. So, following Hertz et al, we can show this diagrammatically in Figure 3.3.

Notice that in the right half of the diagram no line can be found which will allow us to discriminate between all rectangles (representing one class) and all ovals (representing the other class). We say that the classes in the left diagram are **linearly separable** while those in the right diagram are **linearly inseparable**.

The simple example which Minsky and Papert used was the XOR pattern which is readily shown not to be linearly separable (see Figure 3.4.

The simple perceptron cannot differentiate between the XOR set of patterns (you will test this in Laboratory 2). Minsky and Papert showed that this problem is the simplest example of a set of problems which may be characterised as belonging to the parity problem (The XOR truth table above shows even parity).

3.6 The Delta Rule

Another important early network was the Adaline (ADAptive LINear Element). The Adaline calculates its output as $y = \sum_{j=1}^{N} w_j x_j + \theta$, with the same notation as before. You will immediately note the difference

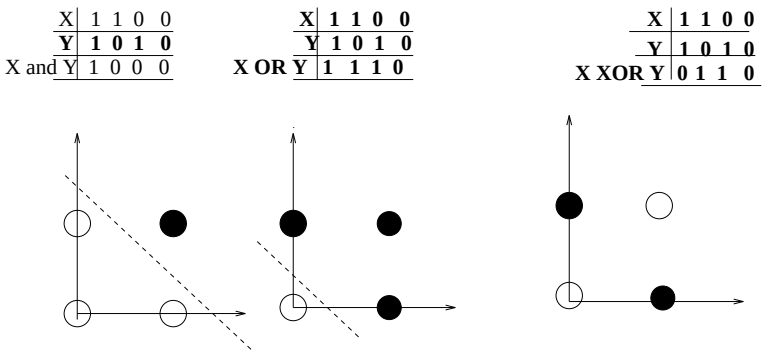

Figure 3.4: A diagrammatic representation of AND, OR and XOR with the corresponding truth tables where 0 is false and 1 is true

between this network and the perceptron is the lack of thresholding. The interest in the network was partly due to the fact that it is easily implementable as a set of resistors and switches.

3.6.1 The learning rule: error descent

For a particular input pattern, $\mathbf{x}^k$, we have an output y^k and target D^k. Then the sum squared error from using the Adaline on all training patterns is given by

$$E = \sum_{k=1}^{K} E^k = \frac{1}{2} \sum_{k=1}^{K} (D^k - y^k)^2 \tag{3.5}$$

where the fraction is included due to inspired hindsight. Now, if our Adaline is to be as accurate as possible, we wish to minimise the squared error. To minimise the error, we can find the gradient of the error with respect to the weights and move the weights in the opposite direction. If the gradient is positive, the error would be increased by changing the weights in a positive direction and therefore we change the weights in a negative direction. If the gradient is negative, in order to decrease the error we must change the weights in a positive direction. This is shown diagrammatically in Figure 3.5. Formally $\Delta^k w_j = -\gamma \frac{\partial E^k}{\partial w_j}$.

We say that we are searching for the Least Mean Square error and so the rule is called the LMS or Delta rule or Widrow-Hoff rule. Now, for an Adaline with a single output, y,

$$\frac{\partial E^k}{\partial w_j} = \frac{\partial E^k}{\partial y^k} . \frac{\partial y^k}{\partial w_j} \tag{3.6}$$

and because of the linearity of the Adaline units, $\frac{\partial y^k}{\partial w_j} = x_j^k$. Also, $\frac{\partial E^k}{\partial y^k} = -(D^k - y^k)$, and so $\Delta^k w_j = \gamma(D^k - y^k).x_j^k$. Notice the similarity between this rule and the perceptron learning rule; however, this rule has far greater applicability in that it can be used for both continuous and binary neurons. This has proved to be a most powerful rule and is at the core of almost all current supervised learning methods. But it should be emphasised that the conditions for guaranteed convergence which we used in proving the perceptron learning theorem do not now pertain. Therefore there is nothing to prevent learning in principle never converging.

However, since it is so central we will consider 2 variations of the basic rule in the sections below.

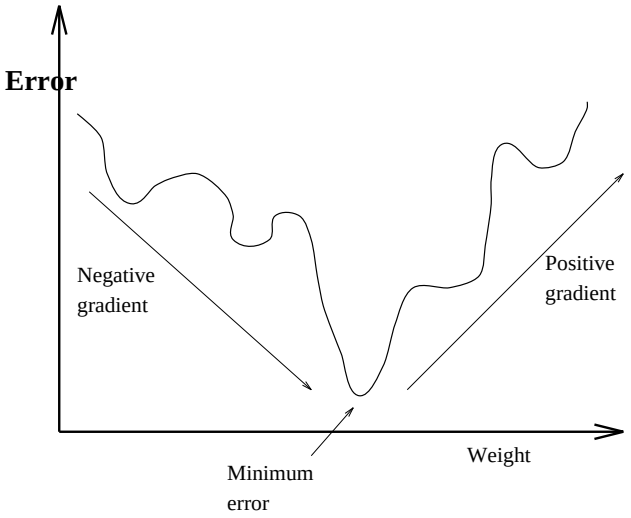

Figure 3.5: A schematic diagram showing error descent. In the negative gradient section, we wish to increase the weight; in the positive gradient section, we wish to decrease the weight

3.6.2 Non-linear Neurons

The extension to non-linear neurons is straightforward. The firing of an output neuron is given by $y = f(\sum_{j=1}^{N} w_j x_j)$, where f() is some non-linear function of the neuron's activation. Then we have

$$E = \sum_{k=1}^{K} E^k = \frac{1}{2}\sum_{k=1}^{K}(D^k - y^k)^2 = \frac{1}{2}\sum_{k=1}^{K}(D^k - f(\sum_{j=1}^{N} w_j x_j^k))^2 \tag{3.7}$$

and so

$$\begin{aligned}\frac{\partial E}{\partial w_j} &= \frac{\partial E}{\partial y}.\frac{\partial y}{\partial NET}\frac{\partial NET}{\partial w_j} \\ \frac{\partial E}{\partial w_j} &= -\sum_{k=1}^{K}(D^k - f(\sum_{j=1}^{N} w_j x_j^k)).f'(\sum_{j=1}^{N} w_j x_j^k).x_j^k\end{aligned}$$

So using the error descent rule, $\Delta w_j = -\gamma\frac{\partial E}{\partial w_j}$, we get the weight update rule

$$\begin{aligned}\Delta w_j &= \gamma\delta x_j \\ \text{where } \delta &= \sum_{k=1}^{K}(y^k - f(\sum_{j=1}^{N} w_j x_j^k)).f'(\sum_{j=1}^{N} w_j x_j^k)\end{aligned}$$

The sole difference between this rule and that presented in the last section is the f' term. Its effect is to increase the rate of change of the weights in regions of the weight space where f' is large i.e. where a small change in the activation makes a large change in the value of the function. You can see from the diagrams of the activation functions given earlier that such regions tend to be in the centre of the function's range rather than at its extreme values.

This learning rule is a **batch** learning rule i.e. the whole set of training patterns are presented to the network and the total error (over all patterns) is calculated and only then is there any weight update. A

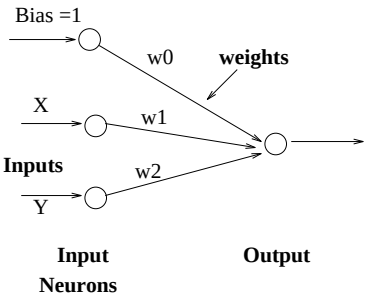

Figure 3.6: The network with a single layer of weights which we will use to attempt to solve the XOR problem

X	Y	X XOR Y	NETWORK OUTPUT	SQUARED ERROR
0	0	-1	0	1
1	0	1	0	1
0	1	1	0	1
1	1	-1	0	1

Table 3.2: A network giving 4 "don't know" answers

more usual form is to have **on-line** learning where the weights are updated after the presentation of each pattern in turn. This issue will be met again in the next chapter. The online version of the learning rule is

$$\begin{aligned} \Delta^k w_j &= \gamma \delta^k x_j^k \\ \text{where } \delta^k &= (D^k - f(\sum_{j=1}^{N} w_j x_j^k)).f'(\sum_{j=1^N} w_j x_j^k) \end{aligned}$$

Typical activation functions are the logistic function and the tanh() function both of which are differentiable. Further, their derivatives can be easily calculated:

- if f(x) = tanh(bx), then f'(a) = b(1 - f(a)*f(a)); i.e. the derivative of the function at a point a is an easily calculable function of its value at a.
- similarly, if f(x) = 1/(1+exp(-bx)) then f'(a)=bf(a)(1 - f(a)).

As noted earlier, these functions have the further important property that they asymptote for very large values.

3.6.3 XOR revisited

Consider the basic network shown in Figure 3.6. X and Y represent the (binary) inputs while x_0 is set always to 1 and will be the constant bias term.

In the case of linear (Adaline) neurons, the best which this network can do is to set $w_0 = w_1 = w_2 = 0$; in this case, the least squared error is 4 (see Table 3.2).

i.e. the total error, E=4. The network is basically giving 4 'don't know' answers.

Now if we repeat the experiment with a tanh() nonlinearity being calculated at the output neuron, we will find that another four states become possible. An example is shown in Table 3.3; to produce these results, the network converged to $w_2 = w_1 \to \infty$ and $w_0 \to -\infty$. This could be thought to be a slightly superior answer since the network is giving 3 correct answers (though 1 wrong answer).

X	Y	X XOR Y	NETWORK OUTPUT	SQUARED ERROR
0	0	-1	-1	0
1	0	1	1	0
0	1	1	1	0
1	1	-1	1	4

Table 3.3: A network giving 3 correct answers and a single wrong answer. Note that the total squared error is the same as before.

This solution has the same sum squared error as the previous one and may be thought of as a **local minimum** which the introduction of a non-linearity makes possible as a point of convergence.

3.6.4 Nuts Revisited

We use the same data (Table 3.1) that we used before i.e. we have the same inputs and targets. So we will train a non-linear net with the input vectors , **x**, equal to
$(1, 2.2, 1.4)^T$ with associated target output 1 (equal to class A)
$(1, 1.5, 1.0)^T$ with target output 1
$(1, 0.6, 0.5)^T$ with target output 1
$(1, 2.3, 2.0)^T$ with associated target output -1 (equal to class B)
$(1, 1.3, 1.5)^T$ with target output -1
$(1, 0.3, 1.0)^T$ with target output -1

Each cycle will have 3 stages:

1. Select randomly an input pattern and associated target pattern
2. Feed the input pattern forward through the current weights
3. Change the weights

The cycles will be repeated till the actual outputs are within 0.1 of the target outputs for all patterns.

Consider the simple network in Figure 3.6. We will use the tanh() function as an activation function since it asymptotes at 1 and -1.

Feedforward:

$$\begin{aligned} y &= f(NET) = f(\sum_{j=1}^{N} w_j x_j) \\ &= tanh(NET) = tanh(\sum_{j=1}^{N} w_j x_j) \end{aligned}$$

Learning:

$$\begin{aligned} \Delta w_j &= \eta(D - f(\sum_{j=1}^{N} w_j x_j)).f'(\sum_{j=1}^{N} w_j x_j).x_j \\ &= \eta(D - y).f'(\mathbf{w}.\mathbf{x}).x_j \\ &= \eta(D - y).(1 - y * y).x_j \end{aligned}$$

when f() is the tanh() function.

Randomly initialise weights: let $w_0 = 0.7$, $w_1 = 0.5$, $w_2 = 0.5$. Let the learning rate be 1.
Randomly select a pattern: pattern 3.
Feedforward:

$$\begin{aligned} Act &= w_0 * 1 + w_1 * 0.6 + w_2 * 0.5 \\ &= 0.7 + 0.3 + 0.25 = 1.25 \end{aligned}$$

Now y = tanh(act) = tanh(1.25) ≈ 0.8
Change weights:

$$\begin{aligned} \Delta w_0 &= 1 * (1 - 0.8) * (1 - 0.8 * 0.8) * 1 \approx 0.06 \\ w_0 &\rightarrow 0.76 \\ \Delta w_1 &= 1 * (1 - 0.8) * (1 - 0.8 * 0.8) * 0.6 \approx 0.04 \\ w_1 &\rightarrow 0.54 \\ \Delta w_2 &= 1 * (1 - 0.8) * (1 - 0.8 * 0.8) * 0.5 \approx 0.03 \\ w_2 &\rightarrow 0.53 \end{aligned}$$

Randomly select a pattern: pattern 5.
Feedforward:

$$\begin{aligned} Act &= w_0 * 1 + w_1 * 1.3 + w_2 * 1.5 \\ &= 0.76 + 0.54 * 1.3 + 0.53 * 1.5 \approx 2.257 \end{aligned}$$

Change weights:

$$\begin{aligned} \Delta w_0 &= 1 * (-1 - 0.98) * (1 - 0.98 * 0.98) * 1 \approx -0.1 \\ w_0 &\rightarrow 0.76 - 0.1 = 0.66 \\ \Delta w_1 &= 1 * (-1 - 0.98) * (1 - 0.98 * 0.98) * 1.3 \approx -0.13 \\ w_1 &\rightarrow 0.54 - 0.13 = 0.41 \\ \Delta w_2 &= 1 * (-1 - 0.98) * (1 - 0.98 * 0.98) * 1.5 \approx -0.15 \\ w_1 &\rightarrow 0.53 - 0.15 = 0.381 \end{aligned}$$

Stop: when for all patterns the error is less than 0.1. Sometimes a cumulative rule is used e.g. when the average of the last 5 cycles was less than a particular value.

Note: The rate of change of the weights depends on the error at the output (the first term) and the magnitude of the input(the last term) *but also* crucially on the derivative of the activation function - when the tanh() function approaches 1 or -1, the rate of change of the weights slows down.

It is essential to note that here our test for goodness of the current solution occurs outside the learning rule. In the perceptron, we test if we have the correct output and do not change the weights if we have. With the LMS rule, we change the weights every time since we have a floating point output which will never (with probability 1) be exactly equal to the output. The test criterion is a criterion for stopping the simulation not a criterion as to whether we should change the weights.

3.6.5 Stochastic Neurons

It is known that in biological neural networks, the firing of a particular neuron is not always deterministic: there seems to be a probabilistic element to their firing. We can easily model such effects by introducing

stochastic neurons whose probability of firing at a particular time depends on their net activation at that time. e.g.

$$P(y_i^k = \pm 1) = \frac{1}{1 + \exp(\mp 2\beta NET_i^k)} \tag{3.8}$$

where β is a parameter determining the slope of the probability function. This leads to an expected firing rate of $\langle (y_i^k) \rangle = tanh(\beta \sum_{j=1}^{N} w_{ij} x_j)$ which can be used in the weight update rule $\Delta^k w_j = \gamma \delta^k x_j^k$, by setting $\delta^k = (D^k - \langle y_i^k \rangle)$ where the angled brackets indicate an average value over all input patterns.

3.7 Multiple Discrimination

So far we have only discussed networks in which there has been a single output neuron. Such a network can only differentiate between two classes. However by creating a network with more than one output neuron, we can create multiple decision surfaces for a single network. But note that such a machine is only possible if the individual classes are pairwise separable. Therefore to differentiate between N classes, there must exist N linear discriminant functions, $g_i(\mathbf{x}, \mathbf{y})$, for i=1,...,N. Then we will believe that a particular (**x,y**) is an instance of class i if $g_i(\mathbf{x}, \mathbf{y}) > g_j(\mathbf{x}, \mathbf{y}), \forall j \neq i$. Such a categorisation can be achieved with the simple binary perceptron, if we allocate a single output neuron to each class. Then the output of neuron i will be 1 if and only if $g_i(\mathbf{x}, \mathbf{y}) > 0$ which will be the case for only inputs from class i. All other neurons will be -1 at this time i.e. $g_j(\mathbf{x}, \mathbf{y}) < 0$ for all other neurons. This representation of the classes is a local representation since a single neuron is firing to show the classification to a single class. This can be contrasted with a distributed representation in which the class membership is shown by a pattern of firing over the set of output neurons.

3.8 Exercises

1. For the four patterns of the AND rule, use a simple perceptron (pencil and paper) learning rule with $\eta = 0.5$ to calculate the weights of the perceptron after presentation of the 4 patterns in the order second, fourth, third and first (see Figure 3.4). Let the initial values of the weights be w_0, the bias, = 0.3, $w_1 = -0.2$, $w_2 = 0.1$. (Objective 1, 2).

2. Repeat Question 1 with a LMS rule using an activation function of your choice. (Objectives 1,2, 3, 4, 5, 6, 8).

3. (Jagota, 1995) Consider all boolean functions $f : \{0,1\}^2 \rightarrow \{-1,1\}$. Which of them are linearly separable? For each of the linearly separable ones hand-construct a simple perceptron with two input units and one output unit with a hard threshold activation function. Use any real valued threshold θ for the output neuron. (Objective 7).

4. The XOR problem is the simplest of a set of problems known as the parity problems. e.g. for 6 dimensional inputs, $(100011)^T$ has odd parity. Therefore output -1.
   $(110011)^T$ has even parity. Therefore output 1.

   Draw a 3 dimensional unit cube and explain why the perceptron cannot solve the 3-D parity problem. (Objective 7).

5. (Jagota, 1995) Consider a simple perceptron with N input units and M output units, each with a hard threshold activation function (whose range is $\{-1, +1\}$) with threshold $\theta_i = 0$. Restrict each of the weights to have the values -1 or +1. How many functions from $\{-1,1\}^N$ to $\{-1,1\}^M$ are represented in this family of networks? Prove your answer. (Objective 2).