Neural Network Interest Group

Título/*Title*:
Empirical Study of MLP in Classification

Autor(es)/*Author(s)*:
Luís M. Silva

Relatório Técnico/*Technical Report* No. 2 /2004

FEUP/INEB, Rua Dr. Roberto Frias, 4200-465, Porto, Portugal

Título/*Title*:

Empirical Study of MLP in Classification

Autor(es)/*Author(s)*:

Luís M. Silva

Relatório Técnico/*Technical Report* No. 2 /2004

Publicado por/*Published by*: NNIG. http://paginas.fe.up.pt/~nnig/

Contents

Chapter 1

The perceptron

1.1 Introduction

The perceptron is the most simple type of ANN and is based in a single unit or neuron. The working operation of the perceptron is based on the model proposed by McCulloch and Pitts: it takes a vector of real-valued inputs, produces a linear combination, followed by a binary decision. More precisely, with inputs $x_1, \ldots, x_d$ we have the output $o(x_1, \ldots, x_d)$

$$o(x_1, \ldots, x_d) = \left\{ \begin{array}{lll} 1 & if & w_0 + \sum_{i=1}^{d} w_i x_i > 0 \\ -1 & if & w_0 + \sum_{i=1}^{d} w_i x_i \leq 0 \end{array} \right. \tag{1.1}$$

where each $w_i \in \mathbb{R},\ i = 1, 2 \ldots, d$, determines the contribution of input x_i to the output of the perceptron and $w_0 \in \mathbb{R}$ is designated as *bias*. These constants are designated as *weights*[1]. If we take $x_0 = 1$ and define the extended input vector

$$\mathbf{x} = [1, x_1, \ldots, x_d]^T$$

and the weight vector

$$\mathbf{w} = [w_0, w_1, \ldots, w_d]^T$$

the perceptron output can be written as

$$o(x_1, \ldots, x_d) = \varphi(s)$$

where $s = \mathbf{w}^T \mathbf{x}$ and φ is the Heaviside function

$$\varphi(s) = \left\{ \begin{array}{lll} 1 & if & s > 0 \\ -1 & if & s \leq 0 \end{array} \right.$$

[1]For now on w_0 will be designated as weight also unless it is explicitly said.

The function φ, responsible for the binary decision, is designated the *activation function* of the perceptron. As we will see later, it's usual (and preferable) in more complex problems to use a monotonically increasing function to make a continuous and differentiable transition between the saturated parts (where $\varphi(s) = 1$ or $\varphi(s) = -1$). Also, it is usual to use a mapping into the interval $[0, 1]$.

The perceptron can be represented graphically as an oriented graph like in Figure

FIGURA

1.2 Representational ability of perceptrons

Considering a perceptron with Heaviside's activation function with output given by (1.1), it is easily seen that it defines a hyperplane decision surface in the d-dimensional space of examples (input data) with equation $\mathbf{w}^T\mathbf{x} = 0$. This decision surface defines a binary classification rule for a two class problem where an example belongs to one class if it lies in one side of the hyperplane (for example, it outputs 1 for examples of class 1) and the other class if it lies in the other side (outputs -1 for examples of class 2). Of course, a zero misclassification error objective is only achieved with linearly separable sets of examples, i.e, those where the two classes are completely separated by a hyperplane. For example, the perceptron can represent many boolean functions like AND (see Figure 1.1(a)) or OR (assuming 1 is true and -1 is false) and their negations. However, boolean functions like XOR cannot be represented by a perceptron because they do not correspond to a set of linearly separable examples (see Figure 1.1(b))

Nevertheless, the ability to represent the simplier boolean functions is important because every boolean function can be represented by some network of perceptrons only two levels deep (multilayer perceptron with a single hidden layer) (Mitchell,Bishop).

1.3 Perceptron learning

1.3.1 Percetron training rule

The idea beyind the perceptron learning is to find an optimal weight vector $\mathbf{w}$ that causes the network to output the correct class for each training example. A simple algorithm used to adjust the perceptron weights is the *perceptron training rule* that revises the weight w_i according to

$$\begin{aligned} w_i &= w_i + \Delta w_i \\ \Delta w_i &= \eta(t-o)x_i \end{aligned} \tag{1.2}$$

Here t is the target or true response (class) of the actual training example, o is the output of the perceptron (1 or -1) and η is a positive constant called learning rate. The latter controls the rate of change of the weights at each step. Note that the weight adjustments are only made when $t \neq o$ and in this case we have an adjustment proportional to the corresponding input. This simple procedure is proven to converge (Haykin,Bishop) in a finite number of iterations, provided the training examples are linearly separated.

1.3.2 Gradient descent: the delta rule

The perceptron training rule is not assured to converge in the case where the two classes are not linearly separable. To overcome this problem, another training rule called *delta rule* is used to achieve the best fit approximation to the input-output mapping of the training examples. So, although no hyperplane exists that completely separates the two classes (with 100% correct classification), the delta rule converges to a hyperplane that minimizes some measure of training error (different from misclassification error[2]). The derivation of the delta rule can be seen using the simple linear unit (where the activation function is not applied to the linear combination) for which the output o is given by

$$o(x_1, \ldots, x_d) = \mathbf{w}^T\mathbf{x}$$

The measure of training error used is half the sum of the mean square error between the desired target and the output of the linear unit for each training example

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(t_n - o_n)^2 \tag{1.3}$$

where N is the total number of training examples, t_n is the desired target for nth example and o_n the corresponding output of the linear unit. $E(\mathbf{w})$ defines an error surface in the $m+1$-dimensional real space where m is the number of weights (including bias). The idea is to find the weight vector $\mathbf{w}$ that minimizes $E(\mathbf{w})$. Starting with an initial arbitrary weight vector,

[2]Indeed, it is not guaranteed that we are minimizing the misclassification error.

the delta rule uses the gradient descent search to adjust the weights in the direction that produces the steepest descent along the error surface (towards the minimum). We know from calculus that the gradient vector of a vector-valued real function f specifies the direction of increase in f. Hence, the gradient vector of $E(\mathbf{w})$

$$\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right] \tag{1.4}$$

points to the direction of increasing error, so $-\nabla E(\mathbf{w})$ gives the direction of steepest decrease in E. Therefore, the delta rule becomes

$$\begin{aligned} \mathbf{w} &= \mathbf{w} + \Delta\mathbf{w} \\ \Delta\mathbf{w} &= -\eta \nabla E(\mathbf{w}) \end{aligned} \tag{1.5}$$

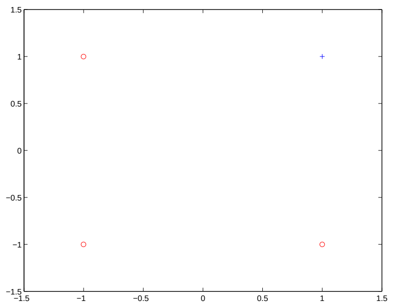
or, more specifically

$$\begin{aligned} \Delta w_i &= -\eta \sum_{n=1}^{N} (t_n - o_n)(-x_{in}) \\ &= \eta \sum_{n=1}^{N} (t_n - o_n)\, x_{in} \end{aligned} \tag{1.6}$$

where x_{in} is the input of training example n associated with w_i. This procedure is proven to converge at least to a local minimum of the error surface, whether the training examples are linearly separable or not, provided that η is small. The learning rate η controls the amount of change in each weight. If η is too large, the algorithm may overpass the minimum and convergence may not be achieved; if η is too small, convergence may be too slow. Figure 1.2 can give intuiton to what can happen depending on the learning rate. We can see that for a large η (dashdot red line), the learning process speed is very high, so high that it can pass over the minimum many times. The convergence (this is different from learning speed) is not assured in this case. Otherwise, if η is small the learning speed is smaller but it is more probable for the algorithm to reach the minimum (dashed black line).

1.3.3 Stochastic version of gradient descent

Gradient descent search is a very important learning rule because it serves as basis for the Back-propagation algorithm used to train multilayer perceptrons. However, there are several practical problems in it's application:


(a)

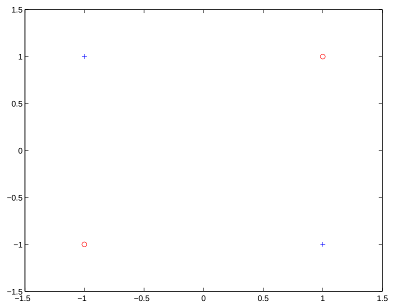

(b)

Figure 1.1: Graphical representation of patterns from a) AND and b) XOR

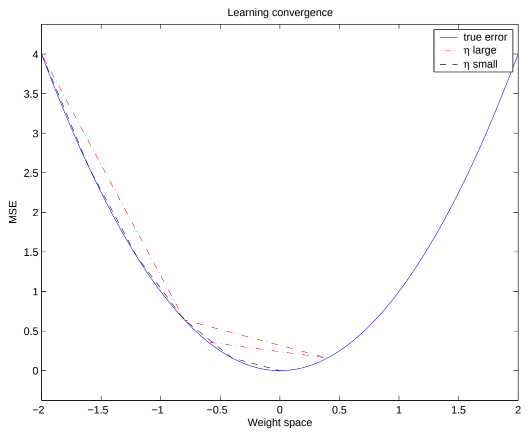


Figure 1.2: Different behaviours of algorithm's convergence, depending on the value of η.

if the error surface has many local minima, it is not assured convergence to the global minimum and the convergence for a local minimum can be slow. There is a modification of the gradient descent rule intended to avoid those problems. Instead of making weight updates after a pass of all training examples by the linear unit (see equation (1.6)), the *stochastic gradient descent rule* updates each weight after a pass of each training example. This corresponds simply to consider a new error function for each example

$$E_n(\mathbf{w}) = \frac{1}{2}\left(t_n - o_n\right)^2 \tag{1.7}$$

and changing the update formula to

$$w_i = w_i + \eta\left(t_n - o_n\right) x_{in} \tag{1.8}$$

where t_n and o_n are the target and unit output for example n and x_{in} is the input associated with w_i. This procedure gives a reasonable approximation to the true gradient descent search and if η is sufficiently small this approximation can be arbitrarily close (Mitchell).

As we will see later, this is designated as the sequential mode of training (each example at a time) whereas the former (true gradient descent) is designated as the batch mode (update weights after obtaining the sum of the error of all training examples).

Notice that, the derivation of the gradient descent rule (delta rule) was based on the linear unit, mostly because the perceptron has a non-differentiable activation function, needed in equation (1.5). The delta rule is used in the derivation of the Back-propagation algorithm, but there, the activation function of each unit is differentiable.

1.4 Some experiments

This section is dedicated to some experiments made with the perceptron. Each experiment tries to compare the number of iterations needed for convergence varying the initial weigth vector ($[0, 0, 0]$, $[-0.1, 0.1, -0.1]$ and $[-2, 3, -7]$) and learning rate η (0.1, 0.5 and 1). Experiments for boolean functions and two Gaussian populations are presented.

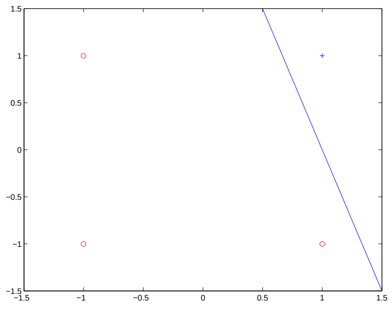
1.4.1 Boolean functions: AND and OR

The first experiment was applied to the boolean functions AND and OR. The results are given in Table 1.1. As we can see these are two simple

AND	$\eta = 0.1$	$\eta = 0.5$	$\eta = 1$	OR	$\eta = 0.1$	$\eta = 0.5$	$\eta = 1$
$[0, 0, 0]$	2	2	2		2	2	2
$[-0.1, 0.1, -0.1]$	2	2	2		1	1	1
$[-2, 3, -7]$	19	5	2		21	6	3

Table 1.1: Number of iterations needed for the perceptron to converge varying the initial weight vector and η for the case of boolean AND and OR.

examples (the two classes are well separated) of use of the perceptron. We see that the use of small initial weight values implies a reduction in number of iterations needed to convergence. Using large initial weight values like $[-2, 3, -7]$ caused a great increase in number of iterations needed, except when the learning rate η was large too. Of course, if we start far from the optimum value, a large η can rapidly take the process to convergence, as we can see from the last line of Table 1.1. Figure 1.3 show an example of a decision line obtained for these examples.


(a)

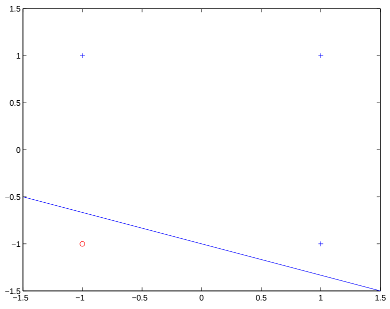

(b)

Figure 1.3: Graphical representation of patterns from a) AND and b) OR with a decision line corresponding to $\eta = 0.1$ and initial weight vector $[-0.1, 0.1, -0.1]$

1.4.2 Two Gaussian populations

The next experiments applied the perceptron to two bivariate Gaussian populations (Gaussian1 and Gaussian2) to obtain a decision line. Figure 1.4 shows graphical representations of the patterns for each population including a final solution to the problem.

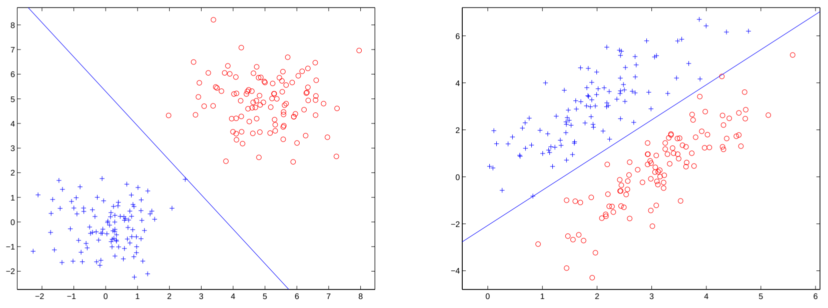


Figure 1.4: Graphical representation of patterns from Gaussian populations a) Gaussian1 and b) Gaussian2 with a decision line corresponding to $\eta = 0.1$ and initial weight vector corresponding to the minimum number of iterations (see Tables 1.2 and 1.3)

The experiments were conducted as before and results are given in Tables 1.2 and 1.3. Below the number of iterations we present the final weight values in the form $[w_0, w_1, w_2]$. Table 1.2 presents the results for population

	$\eta = 0.1$	$\eta = 0.5$	$\eta = 1$
$[0, 0, 0]$	48	48	48
	$[88.6, -13, -31.8]$	$[443, -65.2, -159]$	$[886, -130.4, -318.1]$
$[-0.1, 0.1, -0.1]$	16	16	16
	$[33.9, -8.9, -6.4]$	$[169.9, -45.1, -31.6]$	$[339.9, -90.3, -63]$
$[-2, 3, -7]$	31	47	47
	$[55.8, -9.8, -17.5]$	$[418, -68.1, -139.9]$	$[836, -136.8, -282.6]$

Table 1.2: Number of iterations needed for the perceptron to converge (below is presented the final weight vector) varying the initial weight vector and η for the case of Gaussian1.

Gaussian1. We can see that η doesn't affect the results except for the case of larger initial weight values. The best results are obtained with small initial values, different from zero, were only 16 iterations were needed. Results from population Gaussian2 are similar to the previous except that there isn't a great difference between the different conditions. The number of iterations are close, but again, with small initial weight values we need less passes of the algorithm. It is obvious (and expected) that in Gaussian2 more iterations would be needed, because the separability of the two classes is less evident and more difficult to obtain.

	$\eta = 0.1$	$\eta = 0.5$	$\eta = 1$
$[0, 0, 0]$	63	63	63
	$[32.2, -23.3, 15.6]$	$[161, -116.5, 77.9]$	$[322, -233, 155.7]$
$[-0.1, 0.1, -0.1]$	69	69	69
	$[34.1, -25.1, 16]$	$[170.9, -125.9, 80.6]$	$[341.9, -252, 161.3]$
$[-2, 3, -7]$	71	76	76
	$[33.6, -24.6, 15.8]$	$[181, -134.6, 85.1]$	$[368, -273, 171.5]$

Table 1.3: Number of iterations needed for the perceptron to converge (below is presented the final weight vector) varying the initial weight vector and η for the case of Gaussian2.

There is an interesting thing that the previous Tables show. The final weight vector for $\eta = 0.5$ is a multiple of the final weight vector for $\eta = 0.1$ (indeed, we have the proportion $0.5/0.1 = 5$). This can be explained by the fact that the adjustments made to the weight vector are simply to sum or subtract the inputs multiplied by η (see equation 1.8), and so, η is only a re-scaling factor. This happens in almost of the cases, except for the combination $(\eta = 0.1, [-2, 3, -7])$ which is very strange. We can also see that with this combination, the number of iterations differ from the other.

Chapter 2

Multilayer Perceptron

2.1 Introduction

Multilayer perceptrons (MLP) are natural extensions of the single perceptron to network architectures with more than one layer of units or neurons. This networks consist of an input layer constituted by a set of sensory units or source nodes (input variables), one or more hidden layers (with one or more neurons) and an output layer with one or more neurons. The network is used in a simple manner: each pattern is propagated in a forward direction on a layer-by-layer basis.

MLP's are trained with the *back-propagation learning algorithm*. This algorithm consists in two passes through the network. The first, in a forward direction, propagates a pattern from the input to the output layer, producing the actual response of the network, that is usually different from the desired response (target) for that pattern. This produces an error signal that is then propagated backward through the network and the network weights are adjusted in accordance with an error-correction rule (based on the error signal obtained), in a way that the actual response become closer to the desired response. This is the second pass.

Figure ref shows the architectural graph of a MLP with one hidden layer and an output layer. It is presented the model of a fully connected network where each neuron is connected to all the neurons in the previous layer (or input nodes in case of the first hidden layer). The left to right arrow direction shows the way every pattern is propagated through to obtain the network's actual response.

In the forward pass, each neuron in the network behaves like the single perceptron. The only difference is the smooth nonlinear activation function applied to the linear combination. A well known and used form of nonlinearity is the sigmoidal nonlinearity which the logistic function is an example

$$\varphi(s) = \frac{1}{1+e^{-\alpha s}} \qquad \alpha > 0 \tag{2.1}$$

Another example is the hiperbolic tangent function

$$\varphi(s) = \alpha \tanh(\beta s) = \frac{e^{\beta s} - e^{-\beta s}}{e^{\beta s} + e^{-\beta s}} \qquad \alpha, \beta > 0 \tag{2.2}$$

Each hidden neuron or output neuron is designed to perform two computations

1. Apply the nonlinear activation function to the linear combination obtained from the inputs to that neuron and weights connected to it. This is the forward pass.

2. Compute an estimate of the gradient vector (in terms of local gradients of the error surface with respect to the weights connected to the inputs of that neuron) used to make weight adjustment. This is needed for the backward pass.

2.2 Back-propagation algorithm

Let us consider $\mathbf{x}_n = (x_{1n}, \ldots, x_{dn})$ as the nth d-dimensional training example with target vector $\mathbf{t}_n = (t_{1n}, \ldots, t_{mn})$, N the total number of training patterns and $\mathbf{o}_n = (o_{1n}, \ldots, o_{cn})$ is the corresponding output of the network for $\mathbf{x}_n$. Of course, $c = m$ is the network's number of output neurons. The back-propagation (BP) algorithm used for training MLP's is based in an error-correction learning rule. This learning rule is based in an estimate of the Mean Square Error (MSE) given by

$$E_{av} = \frac{1}{N}\sum_{n=1}^{N} E_n \tag{2.3}$$

where

$$E_n = \frac{1}{2}\sum_{j=1}^{c} e_{jn}^2 \tag{2.4}$$

and $e_{jn} = (t_{jn} - o_{jn})$, the error signal. For now on, the subscript n will be dropped except for E_n. The BP algorithm applies a correction Δw_{ji} to the corresponding weight based on the delta rule[1]

$$\Delta w_{ji} = -\eta \frac{\partial E_n}{\partial w_{ji}} \tag{2.5}$$

where η is the learning rate of the BP algorithm and w_{ji} is the weight connecting neuron j to neuron i in the previous layer. The minus sign in (2.5) is used to force the search on the weight space to be in gradient descent (looking for lower values of E_n). Using the chain rule

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial e_j}\frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}}$$

and some analytic manipulations we can see that

$$\frac{\partial E_n}{\partial w_{ji}} = -e_j \varphi_j^{'}(v_j) y_i$$

Taking

$$\delta_j = e_j \varphi_j^{'}(v_j)$$

it is easily seen that

$$\Delta w_{ji} = \eta \delta_j y_i \tag{2.6}$$

where δ_j is the *local gradient* at output neuron j. This is the way to compute the changes for the weights connected to each output neuron. For weights w_{ji} when j is an hidden neuron we have the problem of computing e_j because there is no direct measure of error at that point. This problem is solved by back-propagating through the network the error obtained in the output layer. The local gradient for an hidden neuron j can be written as

$$\begin{aligned} \delta_j &= -\frac{\partial E_n}{\partial y_j}\frac{\partial y_j}{\partial v_j} \\ &= -\frac{\partial E_n}{\partial y_j}\varphi_j^{'}(v_j) \end{aligned}$$

Using again the chain rule of calculus and some straightforward calculations, we can derive the expression of the local gradient of an hidden neuron j

$$\delta_j = \varphi_j^{'}(v_j) \sum_k \delta_k w_{kj}$$

[1] Recall from the previous chapter.

where δ_k is the local gradient of neuron k in the next layer and w_{kj} is the connecting weight between neuron j and neuron k.

In the derivation of the expressions for the local gradients δ_j we saw that they depend on $\varphi_j^{'}(.)$, the first derivative of the activation function $\varphi_j(.)$. Here we will consider that every unit has the same activation function, so $\varphi_j(.) \equiv \varphi(.)\ \forall j$. The activation functions considered (logistic and hyperbolic tangent) are both continuosly differentiable. For the logistic function we have

$$\varphi^{'}(v_j) = \frac{\alpha e^{-\alpha v_j}}{\left(1 + e^{-\alpha v_j}\right)^2}$$

As $y_j = \varphi(v_j)$ we may write

$$\varphi^{'}(v_j) = \alpha y_j \left(1 - y_j\right)$$

hence, for an output neuron j (where we can write $y_j = o_j$)

$$\delta_j = \alpha \left(t_j - o_j\right) o_j \left(1 - o_j\right)$$

In the case of an hidden neuron we have

$$\delta_j = \alpha y_j \left(1 - y_j\right) \sum_k \delta_k w_{kj}$$

For the hyperbolic tangent function we have

$$\varphi^{'}(v_j) = \frac{\beta}{\alpha} \left(\alpha - y_j\right) \left(\alpha + y_j\right)$$

and for a neuron j in the output layer

$$\delta_j = \frac{\beta}{\alpha} \left(t_j - o_j\right) \left(\alpha - o_j\right) \left(\alpha + o_j\right)$$

and for an hidden neuron

$$\delta_j = \frac{\beta}{\alpha} \left(\alpha - y_j\right) \left(\alpha + y_j\right) \sum_k \delta_k w_{kj}$$

As we can see, it is possible to compute the local gradients without knowing an explicit expression for the activation function.

2.3 Some issues about BP learning

2.3.1 Representational ability of MLP's

As we have seen in the previous chapter, MLP's with just one hidden layer are capable of representing every boolean function (Mitchell). The power of feedforward MLP's is in two other results. The first one states that every bounded continuous function can be approximated with arbitrarily small error by a network with two layers of sigmoid units (Mitchell, Bishop). This is a very important result because in classification we have in general continuous decision boundaries and, hence, we only need to restrict ourselves to networks with only one hidden layer. Of course, the number of hidden neurons depend on the complexity of the decision boundary. The more general result states that any function can be approximated to arbitrary accuracy by a network with three layers.

2.3.2 Convergence and local minima

The back-propagation algorithm is a powerfull algorithm capable of implement an approximation to gradient descent search through the space of possible network weights. However, because the error surface may contain several local minima, backpropagation is only assured to converge to a local minimum of E. Fortunately, in practice, having a minimum with respect to some weight doesn't mean that it is also for the other weights and so, gradient descent can proceed. In the next sections we discuss some cautions that can avoid local minima.

2.3.3 Initial weight values

Initialization is an important issue of MLP training. A suitable choice for the initial conditions can be of extreme importance, leading to a good final solution and also to an improvement in the learning speed. Usually, the initial weight values are taken randomly from a certain distribution with a prespecified mean and standard deviation. Taking small initial weight values causes BP to operate on a very flat area around the origin which is a saddle point (Haykin, Bishop). Otherwise, large initial values could leave the neurons into saturation. We should take for initial weights (including bias), values from a zero-mean uniform (Haykin) or sphericall Gaussian (Bishop) distribution with variance chosen to make the standard deviaton of v_j lie at the transition between linear and saturated parts of the activation function. This amounts to choose $\sigma_w = m^{-1/2}$, where σ_w is the standard deviaton of the weight distribution and m is the number of connections of a neuron.

In practical cases it is usually used an heuristic method to generate the initial weight values: random generation from an uniform distribution in some interval of the type $[-a, a]$ (usually $a = 0.1$).

2.3.4 Learning Rate

The learning rate parameter η controls the rate of convergence (or divergence, in some cases) of the BP algorithm. A smaller η causes smaller weight changes as we can see from (2.5). This causes the trajectory in the weight space smoother and closer to the one computed by the method of steepest descent. Of course in this case we have a slower rate of learning. On the contrary, if η is large, weight changes will be greater and the speed of learning will be increased. The problem is that if η is too large the network may become unstable and convergence isn't guaranteed as we have seen in the previous chapter (see Figure 1.2). Several methods to choose appropriately the value of η are discussed in the literature, including versions of BP with adaptive learning rate. The latter reduces η at each iteration (epoch) to ensure convergence. Others use different learning rates for different weights. Empiricaly it is known that η should be chosen small and in the interval $[0, 1]$. A typical choice is $\eta = 0.1$ (Haykin).

It is possible to increase the learning rate without having convergence problems. This is done by modifying the delta rule to a generalized form by adding a momentum term

$$\Delta w_{ji}^{n} = \alpha \Delta w_{ji}^{n-1} + \eta \delta_{jn} y_{in} \tag{2.7}$$

where $\alpha > 0$ is the momentum constant. This is not studied in this report.

2.3.5 Training Mode

There are also some issues concerning the network's training mode. The first one is randomization of the training set. This procedure allows a random presentation of the patterns to the network, preventing some cycling behaviours (like presenting all patterns of class 1, then all patterns of class 2, etc.) that tend to speed down the algorithm or biasing the weights estimates. This randomization tends to make the search in weight space stochastic over the learning cycles (Haykin).

Second one is the way training itself is done. We have two methods

1. Sequential mode.
   In this procedure (also referred as on-line, pattern or stochastic mode) weight updating is performed after each pattern is presented to the network: a pattern is chosen randomly and is forward propagated through the network; an error signal is obtained and back propagated to perform the weight updating. Then, another pattern is presented and so on untill one epoch is completed. Note that the above derivation of BP was based in the sequential mode of training.

2. Batch mode.
   In this procedure, weight updating is done after one epoch of training examples is presented to the network. This requires minor changes in the weight update expressions. Here

$$E_{av} = \frac{1}{2N} \sum_{n=1}^{N} \sum_{j=1}^{c} e_{jn}^2 \tag{2.8}$$

   and the adjustment applied to w_{ji} becomes (dropping n)

$$\Delta w_{ji} = -\frac{\eta}{N} \sum_{n=1}^{N} e_{jn} \frac{\partial e_{jn}}{\partial w_{ji}} \tag{2.9}$$

   where $\frac{\partial e_{jn}}{\partial w_{ji}}$ can be computed as before.

The question that arises at this moment is: which method to choose?
In this report the sequential mode is preferred. The reasons are described below.
Advantages sequential mode (Haykin,Bishop)

- requires less storage for each weight
- possibility of escaping from local minima because of the stochastic search in weight space
- prevents problems with training set redundancy (if it is the case)
- popular and simple to implement algorithm

Advantages batch mode (Haykin)

- easier to establish theoretical conditions for convergence
- provides an accurate estimate of the gradient vector
- easier to parallelize

2.3.6 Activation function

As we saw earlier, MLP training with back-propagation uses nonlinear sigmoidal activation functions. Two examples were given: the logistic function and hyperbolic tangent function. Which of them is the best? By (Haykin), it is preferable an antisymmetric activation function (i.e, odd function) because, in general, the learning process is faster. This is the case of the hyperbolic tangent function

$$\varphi(v) = \alpha \tanh(\beta v)$$

Suitable values for constants α and β are (LeCun 1989,1993)

$$\begin{aligned} \alpha &= 1.7159 \\ \beta &= 2/3 \end{aligned}$$

The reason for choosing an antisymmetric function is that the output of each neuron is permitted to assume both positive and negative values in intervals of the type $[-a, a]$. In the case of the logistic function, the output is restricted to $[0, 1]$, introducing a source of systematic bias for those neurons beyond the first hidden layer (Haykin).

On the other hand, sigmoid logistic activation functions provide at network's output an approximation to *posterior* probabilities of each class as we will see in a next section. This provides a classification rule based on the Bayes rule.

2.3.7 Target encoding

The target value of each pattern must be within the range of the activation function (otherwise we could have a large MSE value). It is also recommended (Haykin) an offset by some amount ϵ of the form

$$t_j = 1 - \epsilon \text{ or } t_j = 0 + \epsilon$$

for the case of the logistic function. This prevents saturation of neurons and can speed up the learning process.

2.3.8 Pre-processing of input data

Also important in a practical point of view is the normalization of the training set (Le Cun, 1993) suggests some normalization procedures applied in this order

1. mean removal of each input variable
2. the input variables should be uncorrelated (using PCA for example)
3. covariance equalization of the decorrelated variables (allowing an approximately equal learning speed of each weight)

A typical procedure for normalizing the input data is to standardize each input variable

$$\tilde{x}_i = \frac{x_i - \bar{x}_i}{s_i}$$

where $\tilde{x}_i$ is the pre-processed variable, $\bar{x}_i$ and s_i is the mean and standard deviation respectively of input variable x_i.

2.4 Multi-class classification problem

For a M class ($M \geq 3$) classification task, the network built needs, in theory, a total of M output neurons to represent all possible classification decisions. Using binary target values (for each output neuron with logistic activation function), each class is represented by a vector of the form

$$\begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \tag{2.10}$$

where the only 1 appears at position k. In this way, the above vector is associated with a pattern from class k ($k = 1, 2, \ldots, M$). We may, thus, associate the output of a MLP trained with BP to an asymptotic approximation of the underlying *a posteriori* class probabilities. Indeed, by the work of (White, 1989a; Richard and Lippmann, 1991) the conditional expectation of the desired response vector, given pattern $\mathbf{x}$, equals the *a posteriori* class probability (see Haykin for more details).

So, if the training size is large enough and BP doesn't converge to a local minimum, we can use the MLP output as an approximation of the *a posteriori* class probabilities. This gives hints on a decision rule for this general problem based in the usual Bayes rule

$$\text{pattern } \mathbf{x} \text{ belongs to class } k \text{ if } o_k(\mathbf{x}) > o_j(\mathbf{x}) \; \forall j \neq k$$

where $o_j(.)$ is the jth output of the network.

In practice, if we use the offset strategy suggested in previous sections the outputs are no longer exactly the *a posteriori* class probabilities but a mapping to the closed interval $[\epsilon, 1-\epsilon]$ in a way that $P\left(class\ k|\mathbf{x}\right) = 0$ is mapped to an output ϵ and $P\left(class\ k|\mathbf{x}\right) = 1$ onto $1-\epsilon$.

2.5 Generalization

One of the objectives of classification tasks is to build a classifier not only for the training set (used to build the classifier) but also to unseen patterns (drawn from the same population of the training set) of a test set. This is known as *generalization*. The neural network learning process may be viewed as a curve-fitting problem, where the aim is to approximate the input-output mapping (in a mean square sense) of the underlying training distribution. One of the problems encountered is the possible overfitting of the training data due to an excessive training. This causes the network to learn too much the training patterns (and usually, the natural noise that exists is also modeled) loosing ability to predict new unseen patterns with different (but similar) input signals. The final product is a non smooth input-output mapping extremely adapted to the training set. Thus, the idea is to search a model that is a tradeoff between smoothness (which implies better generalization capability) and approximation of the input-output mapping according to a certain criterion (in this case MSE).

This poses the problem: when do we stop the training procedure (i.e, number of iterations or epochs)? A standard tool usually used to overcome this issue is to consider a validation subset of patterns that are not used to train the network. The procedure can be as follows: at each epoch, after weight adjustments, evaluate the validation subset with the actual model to compute MSE for test patterns. We will probably get something like Figure 2.1. Then we should choose the number of epochs that made the MSE of validation set minimum.

2.6 Estimating misclassification error: cross-validation

The final task of a classification problem is to access an estimate of the misclassification error of the chosen model. Recall that the objective is to classify a pattern in one of M classes. To reach to a final model we used an error measure that is differentiable: the mean square error. Note that this measure is different from misclassification. Indeed, a pattern can be correctly classified, but can contribute to MSE.

In order to obtain an accurate estimate of misclassification error, we use a standard statistical tool known as *cross-validation*. The total set of available patterns is divided in k subsets. Then we use subset 1 for test set and the model is built with the other $k-1$. The misclassification error (according to the Bayes rule) is obtained for test set. Then, subset 2 is used as test set and the model built with $k-1$ and so on k times. At the end, we use the mean of the k errors obtained as an estimate of the misclassification error.

The question of which value should we choose for k then arises. Using $k = N$ where N is the number of training patterns we get a special case of cross-validation known as *leave-one-out*. In this case, cross-validation is approximately unbiased for the true prediction error, but can have high variance because the N training sets are very similar. Also, it is more computational expensive because we have to build N models. If k is small we can overestimate the true prediction error. In practice, it is usually used $k = 10$ as a compromise between bias and variance.

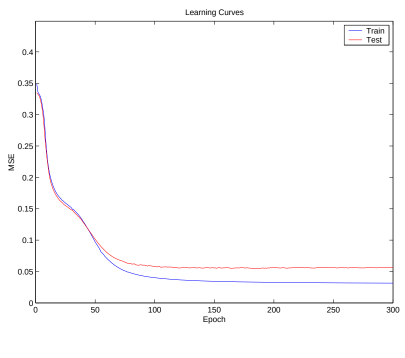


Figure 2.1: Training and validation errors used to choose the number of training epochs.