Table of Contents

Chapter III - Multilayer Perceptrons

Version 2.0

This Chapter is Part of:

Neural and Adaptive Systems: Fundamentals Through Simulation© by

Jose C. Principe
Neil R. Euliano
W. Curt Lefebvre

The goal of this chapter is to provide the basic understanding of:

- Definition of neural networks
- McCulloch-Pitts PE
- Perceptron and its separation surfaces
- Training the perceptron
- Multilayer perceptron and its separation surfaces
- Backpropagation
- Ordered derivatives and computation complexity
- Dataflow implementation of backpropagation

Go to next section

Go to the Appendix

1. Artificial Neural Networks (ANNs)

There are many definitions of artificial neural networks . We will use a pragmatic definition that emphasizes the key features of the technology. ANNs are learning machines built from many different *processing elements* (PEs). Each PE receives connections from itself and/or other PEs. The interconnectivity defines the topology of the ANN. The signals flowing on the connections are scaled by *adjustable parameters called weights*, wij. The PEs sum all these contributions and produce an output that is a *nonlinear (static) function of the sum*. The PEs' outputs become either system outputs or are sent to the same or other PEs. Figure 1 shows an example of an ANN. Note that a weight is associated with every connection.

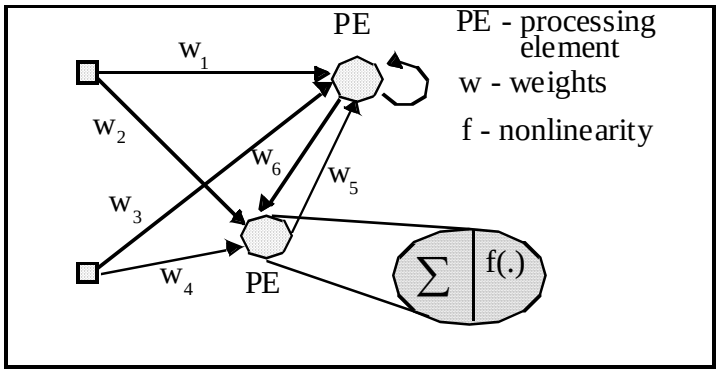


Figure 1. An artificial neural network

The ANN builds discriminant functions from its PEs. The ANN topology determines the *number and shape* of discriminant functions. The shape of the discriminant functions changes with the topology, so ANNs are considered semi-parametric classifiers. One of the central advantages of ANNs is that they are sufficiently powerful to *create arbitrary*

discriminant functions so, ANNs can achieve optimal classification.

The placement of the discriminant functions is controlled by the network weights. Following the ideas of nonparametric training, the weights are adjusted directly from the training data without any assumptions about their statistical distribution. Hence, one of the central issues in neural network design is to utilize systematic procedures (*a training algorithm*) to modify the weights such that a classification as accurate as possible is achieved. The accuracy has to be quantified by an error criterion.

There is a *style in training* an ANN (Figure 2). First, data are presented and an output is computed. An error is obtained by comparing the output with a desired response and it is utilized to modify the weights. This procedure is repeated using all the data in the training set until a convergence criterion is met. So, in ANNs (and in adaptive systems in general), the designer does not have to specify the parameters of the system. They are automatically extracted from the input data/desired response by means of the training algorithm.

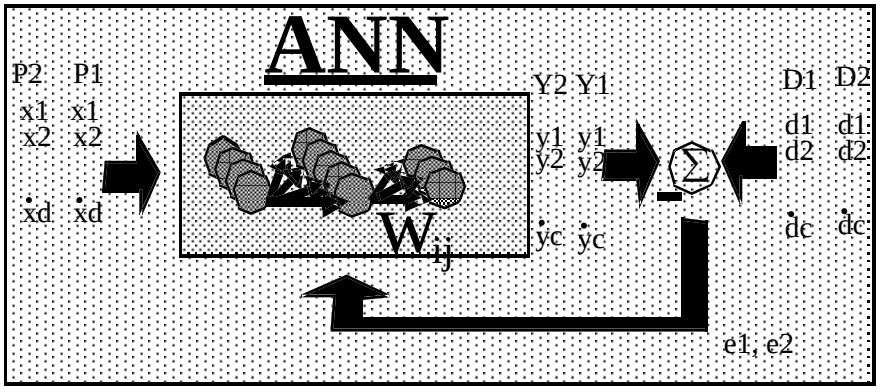


Figure 2. General principles of adaptive system's training

The two central issues in neural network design (semi-parametric classifiers) are the selection of the shape and number of the discriminant functions, and their placement in pattern space such that the classification error is minimized. We will address all these issues in this chapter in a systematic manner. The function of the PE is explained, both in

terms of discriminant function capability and learning. Once this is understood, we will start putting PEs together in feedforward neural topologies with many layers. We will discuss both the mapping capabilities and training algorithms for each of the network configurations.

Go to next section

2. Pattern recognition ability of the McCulloch-Pitts PE

The McCulloch-Pitts (M-P) processing element is simply a sum-of-products followed by a threshold nonlinearity (Figure 3). Its input-output equation is

$$y = f(net) = f\left(\sum_i w_i x + b\right)$$

Equation 1

where w_i are the weights and b is a bias term. The activation function f is a threshold function defined by

$$f(net) = \begin{cases} 1 & for \quad net \geq 0 \\ -1 & for \quad net < 0 \end{cases}$$

Equation 2

which is commonly referred as the *signum function*. Note that the M-P PE is the adaptive linear element (adaline) studied in Chapter I followed by a nonlinearity.

We will now study the pattern recognition ability of the M-P PE. The study will utilize the interpretation of a single discriminant function as given by Eq.10 of Chapter II. Note that such a system is able to separate only two classes (one class associated with the +1, the other with the -1 response). Figure 3 represents the network we are going to build in NeuroSolutions.

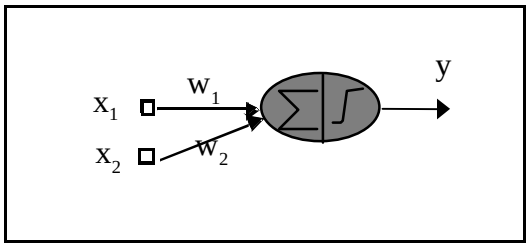


Figure 3. Two input, one output (2-1) McCulloch-Pitts PE.

NeuroSolutions 1

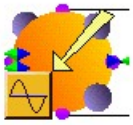
3.1 McCulloch and Pitts PE for classification

The McCulloch and Pitts PE is created by the concatenation of a Synapse and of an Axon. The synapse contains the weights w_i, and performs the sum-of-products. The Synapse Inspector shows that the element has 2 inputs and one output. The number of inputs x_i is set by the input Axon. The soma level of the Inspector shows that the element has two weights. The number of outputs is set by the component to its right (the ThresholdAxon).

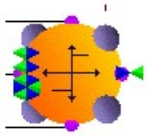
The ThresholdAxon adds its own bias b to the sum-of-products and computes a static nonlinearity. The shape of the nonlinearity is stamped in the Axon icon, which is a step for the ThresholdAxon. So this M-P PE maps 2D patterns to the values {-1, 1}. Basically the M-P PE is like the adaline we built in Chapter I, but now the BiasAxon (which is linear) is substituted by a nonlinearity. This network is very simple, but we can call upon our geometric intuition to understand the input-output map of the M-P PE.

In this example, we will use two new components, threshold axon and the function generator. The function generator is a component which is typically used for input and can create common signals such as sine waves, ramps, impulse trains, etc.

New NeuroSolutions Components



Function Generator on an Input Axon



Threshold Axon

NeuroSolutions Example

The question that we want to raise now is: what is the discriminant function created by this neural network? Using Eq.2 the output of the processing unit is

$$y = \begin{cases} -1 & if \quad \sum_{j=1,2} w_j x_j + b < 0 \\ 1 & if \quad \sum_{j=1,2} w_j x_j + b \geq 0 \end{cases}$$

Equation 3

We can recognize that the output is controlled by the value of $w_1 x_1 + w_2 x_2 + b$, which is the equation of a plane in 2D. This is the *discriminant function* g(x1, x2) utilized by the M-P PE, and we see that it is the output of the adaline. When the threshold works on this function it divides the space into two half planes, one with a positive value (+1) and another with a negative value (-1). This is exactly what we need to implement a classifier for the two-class case (see Chapter II, section 2.6). The equation for the decision surface reads

$$g(x_1, x_2) = w_1 x_1 + w_2 x_2 + b = 0 \rightarrow x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2}$$

Equation 4

which can be readily recognized as a straight line with slope

$$m = \frac{-w_1}{w_2}$$

Equation 5

passing through the point (0, -b/w2) of the plane (x_2-intercept). Or alternatively at a distance -b/|w| from the origin, where $|w| = \sqrt{w_1^2 + w_2^2}$. For this reason *b is called a bias*.

Can we visualize the response of this system to inputs? If the system was linear, linear system theory could be applied to arrive at a closed form solution for the input-output map (the transfer function). But for nonlinear systems the concept of transfer function does not apply. Eq.3 provides the answer for the M-P PE, but this is a very simple case where the output has just 2 values (-1,1). In general the output is difficult to obtain analytically, so we will resort to an exhaustive calculation of the input-output map, i.e. values (of 1) are placed in every location of the input space and the corresponding output is computed. Let us restrict our attention to a square region of the input space between -1, 1 ($x_1, x_2 \in [-1,1]$) for now.

NeuroSolutions has a probe component that will exactly compute and display this input-output map. It is called the *discriminant probe*. Its function is to fire a sequence of x1,x2 coordinates scanning the input field, compute the corresponding output, and display it as a gray scale image. Negative values are displayed as black. The discriminant probe computes Eq. 3 and displays the results in the input space. The discriminant function is a plane and its intersection with the x_1, x_2 plane is a line (the decisioin surface) that is given by Eq. 4. This is the line we see in the scatter plot between the white and the black regions, and represents the decision surface.

Before actually starting the simulation, let us raise the question: what do you expect to see? Eq.4 prescribes the dividing line between the 1 and -1 responses. Using the values of NeuroSolutions Example 2, the decision surface is a line described by the equation

$$g(x_1, x_2) = x_1 + x_2 + 0.277 = 0$$ **Equation 6**

with slope m=-1 and passing through the point x_2=0.277. vector interpretation of the separation surface . The dividing line (that is the decision surface for the two class case) passes through the point x_2=-0.277 and the slope is -1, corresponding to the angle 135° (second quadrant). The position of the decision surface allows us to imagine the location of the discriminant function (figure 4).

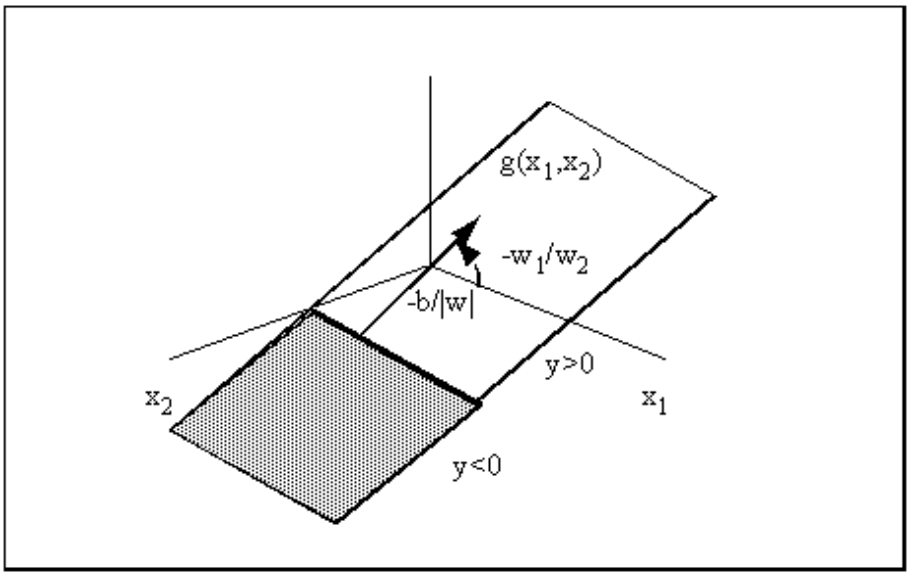


Figure 4. Linear Discriminant function in two dimensions for the two class case

Let us now observe the simulation.

NeuroSolutions 2

3.2 Discriminant probe to visualize the decision surface

This example brings the discriminant probe to the breadboard. The discriminant probe is a pair of DLLs which force data through the network and display the system response, giving us an image of the input/output map of the system. In our case we will use it to show the discriminant line (separation surface) created by the M-P PE as given by Eq.4 . One component of the discriminant probe is placed on the input axon to send the sequential data through the network and the other component is placed on the output axon to display the system response.

You are free to experiment with the M-P PE. We suggest that you modify the Synapse weight values, and the Threshold Axon bias by placing the cursor in the Matrix Editor field with the mouse and typing in the new values. For every combination, you should first guess the solution by computing the slope and y intersect according to Eq.4 , and then finding out if you are correct by running the example. To run the example you need to press the Start button on the toolbar controller.

NeuroSolutions Example

2.1. Sigmoid Nonlinearities

This simple example shows that the decision surface between the 1/-1 responses created by the M-P PE is a line in 2D space (a function that is linear in the parameters). The same conceptual picture works for higher dimensional spaces (but unfortunately we lose our intuition...), where the straight line becomes a hyperplane in a space of dimension one less than the input space dimension.

Notice also how crisp the decision surface is, since a hard threshold acts upon the output of the discriminant function. However, other nonlinearities can be utilized in conjunction with the M-P PE. Let us now smooth out the threshold yielding a sigmoid shape for the nonlinearity. The most common nonlinearities are the logistic function and the hyperbolic tangent (tanh) functions of Figure 5.

$$hyperbolic \qquad f = \tanh(\alpha net)$$

$$logist \qquad f = \frac{1}{1+\exp(-\alpha net)}$$

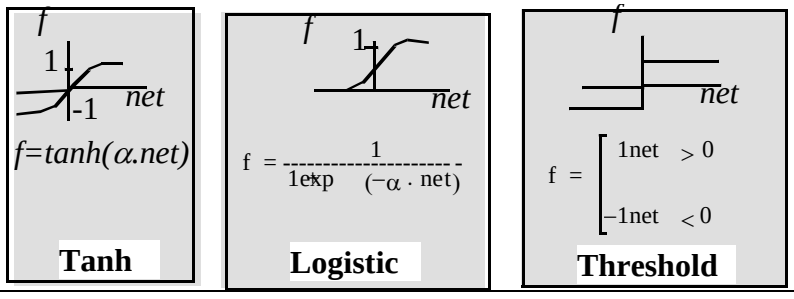


Figure 5. Common nonlinearities in neurocomputing

α is a slope parameter, and normally is set at 1. The major difference between the two sigmoid nonlinearities is the range of their output values. The logistic function produces values between [0,1], while the hyperbolic tangent produces values between [-1,1]. An alternate interpretation of this PE substitution is to think that the discriminant function has been generalized to

$$g(x) = f\left(\sum_{i=1} w_i x_i + b\right)$$

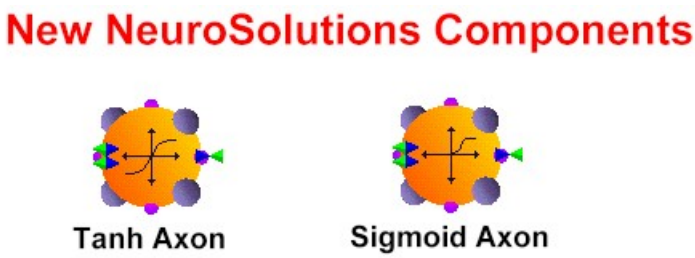
Equation 7

that is sometimes called a ridge function. The ridge function is no longer an hyperplane, since it saturates at 0 (or **–**1) and +1. However the intersection of ridge functions can still be approximated in most of the input space by the intersections of their arguments. In fact, the argument of the ridge function is still linear in the input variables and as long as the function f is monotonically increasing and steep, the *decision surface is still approximately linear*. separation surfaces of the sigmoid PEs So the previous interpretation for threshold PEs still hold approximately for sigmoid PEs.

NeuroSolutions 3

3.3 Behavior of the sigmoid PEs

This breadboard substitutes the Threshold Axon with a Tanh Axon and then a Sigmoid Axon. The difference between the Threshold Axon and the Tanh Axon is that there is a smooth transition between the values of -1 and 1. We can visualize the PE nonlinearity in the component's corresponding Inspector.



The net effect of this modification in the PE input-output function is that the crisp separation between the two regions (positive and negative values of g(.)) is substituted by a smooth transition region between the values of -1 and +1. Hence the name ridge function. However, the orientation of the separation surface is still defined by the ratio of w_1 and w_2, and its vertical placement is still controlled by the bias of the axon. A new feature of this nonlinearity is that the absolute values of the weights control the width of the gray region. By increasing the values of the weights while leaving the ratio of the weights constant, we can make the separation surface become crisper—eventually approximating the one of the ThresholdAxon.

The logistic nonlinearity is similar to the tanh, however the range is between 0 and 1.The tanh is an antisymmetric function (y intercept is zero), while the logistic is not (y intercept is 0.5). The Tanh Axon and Logisitic Axon are normally interchangeable, with the final selection being determined by the desired range of the output (either [-1,1] or [0,1]).

NeuroSolutions Example

The output of the logistic function varies from zero to one. It is interesting to note that under some conditions, the logistic function allows a very powerful interpretation of the output of the PE as *a posteriori probabilities for Gaussian distributed input classes*. probabilistic interpretation of sigmoid outputs . The tanh is closely related to the logistic function by a linear transformation in the input and output spaces, so neural networks that use either of these can be made equivalent by changing weights and biases.

The other big advantage of the TanhAxon (and also of the Sigmoid Axon) is that the nonlinearity is smooth, which means that the derivative of the map exists. This point is going to be very important later for adaptation. We conclude that both the TanhAxon or the SigmoidAxon can substitute the ThresholdAxon with some practical advantages.

We will refer to the combination of the Synapse and the TanhAxon (or the Sigmoid Axon)

as the *modified McCulloch-Pitts PE* because they all respond to the full input space in basically the same functional form (sum of product followed by a nonlinearity). Let us now observe the impact of different nonlinearities in the separation surface.

2.2. Classification implies the control of the discriminant function location

Remember that the ratio of the weights controls the slope (orientation) of the separation surface, and the PE bias controls the x_2 intersect (Figure 4). So, how can the M-P PE system be used to distinguish two classes of patterns?

The discriminant function placement should be controlled such that the system outputs the value 1 for one of the classes, and -1 (or 0) for the other. In order to accomplish this, the *discriminant function must be moved around in the input space, such that a minimum number of errors occurs*.

As external observers, we can do this easily by looking at the data clusters in 2D and placing the separation line between them. But in higher order spaces we can not visualize the data clusters so one needs to follow some type of step by step procedure. As a historical note, Rosenblatt proposed the following procedure to change the weights of the M-P PE:

Get an example of class I and examine the output. If the output is correct (let us say 1) do nothing. If the response is -1, tweak the weights and bias until the response becomes 1. Now go to another example and repeat the procedure, until all the patterns are correctly classified. This procedure is basically the perceptron learning algorithm. This procedure can be automated by the machine itself, without any outside help, if we *provide some feedback to the machine* on how it is doing. The feedback comes in the form of the definition of *an error criterion or cost function* that must be minimized. For each training pattern we can define an error (e_P) between the desired response (d_P) and the actual output (y_P). Note that when the error is zero, the machine output is equal to the desired response.

There are many possible definitions of the error (we will treat this subject in Chapter IV), but commonly in neurocomputing one uses the *error variance (or power),* i.e. the sum of the square difference between the desired response and the actual output. We already found this criterion in Chapter I and called it the mean square error (MSE). For ease of explanation we copy it below,

$$J = \frac{1}{2N}\sum_p \varepsilon_p{}^2 = \frac{1}{2N}\sum_p \left(d_p - y_p\right)^2$$

Equation 8

where p is the pattern index. The goal of the classifier is to minimize this cost function by changing its free parameters. *This search for the weights to meet a desired response or internal constraint is the essence of any connectionist computation*. We already found this methodology in linear regression (Chapter I), and we find it here again. Let us minimize the MSE using NeuroSolutions.

NeuroSolutions 4

3.4 Classification as the control of the decision surface

The input file placed on the input Axon links NeuroSolutions to the computer file system. We created a file with 8 points according to the following Table. The third column represents the class membership. Since our network is built with a logisitc function, these values can also be interpreted as the desired network response.

X1	X2	Class
-0.50	.0.35	0
-0.75	0.85	0
-0.60	0.65	0
-0.50	0.75	0
0.50	0.00	1
-0.30	-0.20	1
0.20	0.10	1
0.10	-0.10	1

We attached the L2Criterion (Mean Squared Error) at the output of the LogisiticAxon. This PE computes the square difference between the system output and the desired response for each input pattern. The output file works as the

desired response, and in this case is built with the values used in the column class of the above table. Note that we have a MatrixViewer on top of the L2Criterion that provides a numerical indication of the power of the error (MSE), i.e. the difference between the machine output and the output file value. We will fire all the eight input samples through the network and display the average error over the entire data set.

One other component deserves to be mentioned. The ScatterPlot probe is placed at the output of the input Axon and that will show the x, y locations of the input data. Since we are interested in displaying all the training patterns, the data buffer size is set to 8. The purpose of this example is to relate the position of the decision surface and the MSE. So experiment with several weights to obtain perfect classification of all the patterns.

As you adjust the network weights, note that the sign of the response may be correct, but the error is still not zero. This is the problem of using a saturating nonlinearity. We have to use very large weights to obtain a response close to 1 and 0. This example shows that acceptable solutions for classification do not require the error to be exactly zero.

Let us modify the data. In the directory ~ NSBook/chapter3/examples/2.5 McPitts3 there is another file called mcpitts_data1.asc. Go to the input file component and with the Inspector open, select the level of file input. Click the remove button to deselect the present file and click on the add button. This will put you in the win95 open file panel. Select the file and NeuroSolutions will ask if it is an ASCII column training file (click the close button to finalize the selection). Another panel pops up allowing you to select which columns are used in the experiment. Since the file has 3 columns, 2 for inputs and one for the desired response, and we are selecting the input data, we have to skip the 3rd column. Select the 3rd row, click the skip button, and close the panel. We just completed the selection of the new file. You

have to do the same thing for the desired signal file, but now you should skip the first two columns.

NeuroSolutions Example

The central problem to be solved in the road to machine-based classifiers is how to automate this process such that the machine can *independently* do these weight changes, without the need for hidden agents or external observers.

2.3. The first learning algorithm for a nonlinear machine

The M-P PE weights can be trained using a very simple rule proposed by Rosenblatt and called the perceptron learning algorithm. The perceptron learning algorithm enunciated above can be put into the following equation

$$w\ (n+1) = w\ (n) + \eta(d(n) - y(n))x\ (n)$$

Equation 9

where η is the stepsize, *y* the M-P PE output and *d* is the desired response.

It is important at this stage to compare this equation with the LMS algorithm we used to train the adaline in the first example of Chapter II. Note that the functional form is the same, i.e. the old weights are incrementally modified proportionally to the product of the error and the input, but there is a significant difference. We can not say that this corresponds to gradient descent since the system has a discontinuous nonlinearity. *In the perceptron learning algorithm, y(n) is the output of the nonlinear system*. So the algorithm is minimizing directly the difference between the response of the M-P PE and the desired response, instead of minimizing the difference between the adaline output and the desired response.

This subtle modification has tremendous impact in the performance of the system. For one thing, *the M-P PE learns only when its output is wrong*. In fact, when *y(n) = d(n)* the weights remain the same. The net effect is that the final values of the weights are no longer equal to the linear regression result, because the nonlinearity is brought into the weight update rule.

Another way of phrasing this is to say that the weight update became much more selective, effectively gated by the system performance. Notice that the LMS algorithm is selective to the error to a certain degree because the error is the difference between the desired response and the system output. So larger errors have more effect on the weight update then small errors, but all patterns affect the final weights – implementing a "linear gate". In the perceptron the net effect is that the placement of the discriminant function is no longer controlled linearly by all the input samples as in the adaline, but only by the ones that are important to place the discriminant function in a way that minimizes the output error explicitly. We are now one step closer to actually train nonlinear systems for classification.

NeuroSolutions 5

3.5 Perceptron learning rule

In this example, we will use the perceptron learning algorithm to classify the two previous data sets – the two class problem from the previous example and the healthy/sick patient data. With the perceptron learning algorithm, the system will learn automatically from the data without the need for our weight selection as we did in this Chapter up to now.

The two examples are very different because one (the two class problem with 8 patterns) is linearly separable while the other is not. The perceptron learning algorithm trains only when the system response is incorrect, therefore, the weights will converge if all the inputs are classified correctly. Another side effect of training only on the errors is that if the problem is not linearly separable, the training will never stop. This can cause abrupt changes in the weights of the system near the end of the training which may affect the classification performance. Since the perceptron learning algorithm does not search for the best answer, only a satisfactory answer, the network may not perform well on data that was not included in its training set. Note that there are many final weight values that produce an error of zero, i.e. that exactly solve the linearly separable problem.

The network is now learning by itself so we will again have to add the backprop and gradient descent layers that we used at the end of Chapter 1. At the end of this chapter, we will finally explain the details of these two additional layers. We have also to control the learning rate or stepsize to make sure that the system will converge to the optimum. You should experiment with other learning rates and observe the learning curve as our thermometer for learning.

NeuroSolutions Example

Let us assume that the patterns are linearly separable, i.e. there is a linear discriminant function that produces zero classification error. The solution of the perceptron learning rule is a weight vector **w*** such that

$$\begin{cases} \sum_i x_i(n) w_i^* > 0 \quad for \qquad d(n) = 1 \\ \sum_i x_i(n) w_i^* < 0 \quad for \qquad d(n) = -1 \end{cases}$$

Equation 10

where n is an index that runs over the training set data. This equation can be written in simpler form as $d_i(\sum_i w_i^* x_i(n)) > 0$. The knowledge of the M-P PE tells us that for each input pattern (each n) there is a weight vector that produces the partitioning given by the desired response. The optimal weight vector is the vector that simultaneously meets all of these desired partitions. The solution in 2D is a line of equation $\mathbf{x}^T \mathbf{w}^* = 0$ (i.e. the optimal weight vector **w*** has to be orthogonal to every data vector **x**). The weight update equation moves the weights directly towards this solution when it corrects **w**(n) by +/-η**x**(n) depending upon the sign of the error. perceptron learning algorithm This learning rule takes a finite number of steps to reach the optimal solution for linearly separable patterns . There are two major problems with this solution: First, as soon as the last sample is correctly classified the discriminant function stalls, producing many possible solutions that may not generalize well. Second, this learning rule converges only if the classes are linearly separable. Otherwise the solution will oscillate.

2.4 The Delta Rule

We have seen in Chapter I that an individual error was defined as the difference between the output measurement and the system output. From this sample by sample error, an error criterion was defined (the sum of squares) over the ensemble of samples. The minimum of this cost was found by following the opposite direction of the performance surface gradient estimated at each point. A systematic step by step procedure based on gradient descent (the LMS rule) was able to modify the weights such that the minimum of the performance surface was reached. The algorithm (you should have memorized this formula by now) is beautifully simple. It adds to the present weight a quantity proportional to the product of the error and activation available at the PE (just two multiplications per weight), i.e.

$$w(n+1) = w(n) + \eta \varepsilon_p(n) x_p(n)$$

Equation 11

We are going to re-examine the LMS algorithm using a different concept that is central to learning in neural networks. This concept is an old result from calculus that is called the *chain rule*. Basically the chain rule tells how to compute the partial derivative of a variable with respect to another when a functional form links the two. Let us assume that *y = f(x)*, and the goal is to compute $\delta y/\delta x$, the sensitivity of y with respect to x. As long as f is differentiable, then

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial f}\frac{\partial f}{\partial x}$$

Equation 12

Note that the value of Eq. 15 computes how much a change in *x* is reflected in *y*, i.e. how sensitive *y* is to a change in *x*. When we work with sensitivities the calculations *progress from the dependent variable to the independent variable*. Keep this in mind.

One can show that the LMS rule is equivalent to the chain rule in the computation of the sensitivity of the cost function *J* with respect to the unknowns. derivation of LMS with chain rule Our goal is to extend the LMS concept to the M-P PE, which is a nonlinear system. How can we compute the *sensitivity* through a nonlinearity?

First let us examine the problem. Figure 6 details the modified M-P PE, where we show multiple weights connected to its input. Note that the output of the M-P PE is a nonlinear function of the weights (Eq.1).

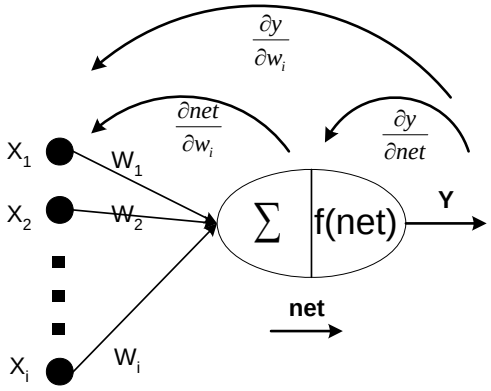


Figure 6**.** Illustration of the sensitivity computation through a nonlinear PE

But we can still compute the partial derivative of the output PE with respect to its weights using the chain rule Eq.12 , by first computing the partial derivative of the output with respect to the intermediate signal *net*, and then compute the partial derivative of *net* with respect to the weight wi, i.e.

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial net}\frac{\partial}{\partial w_i} net = f'(net)x_i$$

Equation 13

where f′(.) is the partial derivative of the static nonlinearity. derivation of sensitivity through nonlinearity

This is another application of the famous chain rule, but *now the chain rule is applied to the topology*. As long as the PE nonlinearity is smooth (differentiable) we can compute how much a change in the weight Δw_i affects the output y. Or looking from the point of view of the sensitivity, how sensitive the output is (Δy) to a change in a particular weight Δw_i. Note that we compute this output sensitivity by a product of partial derivatives

through intermediate points in the topology. For the nonlinear PE there is only one intermediate point, *net*, but we really do not care how many of these intermediate points there are. The chain rule can be applied as many times as necessary.

In practice we have an error at the output (the difference between the desired response and the actual output) and we want to adjust all the PE weights such that the error is minimized in a statistical sense. The obvious idea is to *distribute the adjustments according to the sensitivity of the output to each weight*. Why is this obvious? If one wants to minimize the output error, one should change more the weights that affect the output value the most, which is measured by the sensitivity. This is what the gradient descent does, hence the LMS rule.

So to modify the weight, we actually *propagate back* the output error to intermediate points in the topology, and *scale it along the way as prescribed by Eq.13* according to the elemental transfer functions that we find, as shown in Figure 6. This methodology is very powerful, because we *do not need to know explicitly the error at intermediate places*, such as *net*. The chain rule *derives automatically the error* for us. This observation is going to be crucial to adapt more complicated topologies, and will result in the backpropagation algorithm.

Let us now complete the formulas to adapt the M-P PE weights. Note that now we have two indices, the index i for the weight, and the index p for the pattern, and n for the iteration number. The mean square error is rewritten

$$J = \frac{1}{2N}\sum_{p=1}^{N}\left(d_p - y_p\right)^2$$

$$y_p = f\left(\sum_i w_i x_{ip}\right)$$

Equation 14

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial y_p}\frac{\partial y_p}{\partial net_p}\frac{\partial}{\partial w_i}net_p = -\left(d_p - y_p\right)f'\left(net_p\right)x_{ip} = -\varepsilon_p f'\left(net_p\right)x_{ip}$$

Equation 15

By applying the chain rule twice, one for the output and another for the net, we get

The application of the gradient descent gives again (compare with Eq.11)

$$w_i(n+1) = w_i(n) + \eta \varepsilon_p(n) x_{ip}(n) f'\left(net_p(n)\right)$$

Equation 16

This rule is called the *delta rule* and is a direct extension of the LMS rule to nonlinear systems with smooth nonlinearities. Note that the derivative of the nonlinearity is computed at the operating point $net_p(n)$ for the corresponding input pattern. Note also that *the delta rule is local to the pattern and to the weight*, i.e. it only requires knowledge of that specific pattern, the PE error and its input. optimizing linear and nonlinear systems

Since the smooth nonlinearities discussed so far are saturating, i.e. they approach exponentially the values of -1 (or 0) and 1, the multiplication by the derivative will reduce appreciably the error in most of the operating range since the derivative is bell shaped around netP=0. error attenuation

The derivative of the logistic function and the tanh are respectively

$$f'_{\log istic}\left(net_i\right) = x_i\left(1 - x_i\right)$$

$$f'_{\tanh}\left(net_i\right) = 0.5\left(1 - x_i^2\right)$$

Equation 17

Our goal is to visualize the movement of the discriminant function during the adjustment of the PE weights using the delta rule. Let us go back to NeuroSolutions and automate the placement and display of the discriminant surface for this simple example.

NeuroSolutions 6

3.6 Delta rule to adapt the MC-P PE

We will now use the delta rule to train the breadboard from Example 5. We will use the same two class data set with 8 points and again have the additional layers of learning components. When we run the network, we will be able to observe the discriminant function move. The output mean square error (MSE) decreases close to zero when the discriminant function is placed between the two classes of points.

The output of the network towards the end of learning mimics the desired response. Notice that at the beginning of the run the discriminant line changes its orientation quickly and then at the end there is a period of fine tuning. These are the two basic phases of learning, the discovery phase (what is the direction of the minimum?), and the convergence phase (fine tune to get to the minimum). In our case, the convergence phase corresponds to increasing both the weight and bias (while keeping the ratio constant) in order to get a sharper cutoff between the two classes.

A small exercise to test your knowledge is to guess the network weights upon visualization of the discriminant function. You should at least be able to correctly guess the signs of the weights and approximately their ratio. Note that the final MSE is much smaller than the value obtained when we entered by hand the parameters for the M-P PE. Also note that there is no analytic solution for the optimal weights like we had in the linear case (the least square method). So iterative solutions are very appealing and practical when dealing with nonlinear systems.

It is instructive to change the learning rates and see how the solution behaves. You should be able to do this by now, even if the breadboard is not prepared for it (go to the Step icon, right click the mouse button to open the Inspector, and then enter other values of stepsize). Try very large stepsizes (300) for both parameters and see what happens. In this case the data is linearly separable and it is very difficult to make the system weights diverge. The nonlinearity is always keeping the output between 0 and 1 no matter what the value of the weights are, so divergence of a single PE nonlinear system is not apparent from the value of the output.

So an interesting challenge is to find out a set of learning rates that will prevent the system from finding the correct discriminant line.

NeuroSolutions Example

2.5. Implications of the PE nonlinearity

Several key aspects changed in the performance surface with the introduction of the nonlinearity. The nice parabolic performance surface of the least square problem is lost. Why is that? Note that the performance surface describes how the error changes with the weights. But the performance depends on the topology of the network through the output error. So, when nonlinear processing elements are utilized to solve a given problem the relationship **"**performance - weights**"** becomes nonlinear and there is no guarantee of a single minimum. The performance surface may have *several minima*. The minimum that produces the smallest error in the search space is called the global minimum. The others are called local minima. Alternatively, we say that the performance surface is nonconvex. This impacts the search scheme, because gradient descent uses local information to search the performance surface. In the immediate neighborhood, local minima are indistinguishable from the global minimum. So the gradient search algorithm may be caught in these sub-optimal performance points **"**thinking**"** it has reached the global minimum (Figure 7).

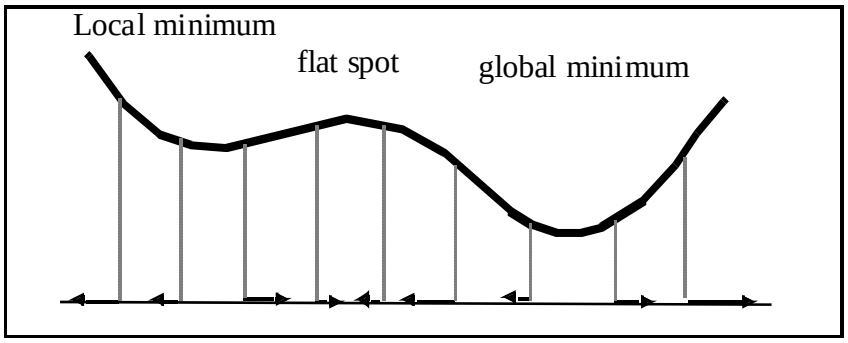


Figure 7**.** Nonconvex performance surface, with gradients depicted.

Other conditions where the gradient is basically zero are called saddle points (or *flat spots*) and are also more frequent than in the linear PE case. Since the weight update is

ultimately produced by the gradient, when the gradient becomes very small the weights do not change much and the adaptation may “stall”. Sub-optimal performance may then result, since the designer may think that the best performance has been reached.

Fortunately, the noisy estimate produced by the LMS rule, which we called a nuisance before (rattling - see Chapter I), becomes an advantage. In fact the noisy gradient increases the chance of escaping both local minima and flat spots. It is obvious that the control of the adaptation algorithm becomes much more delicate in non convex performance surfaces. If the gradient search becomes less robust a fair question is: Why nonlinear PEs?

Let us go to NeuroSolutions to verify this behavior.

NeuroSolutions 7

3.7 Comparing a linear and nonlinear PE for classification

Let us use the same problem as in the previous example, and simply change the PE from a LogisticAxon to a BiasAxon, which simply adds the contributions of the weights plus a bias. Running the simulations, we see that in fact the discrimination probe shows a separation line passing between the classes of points as in the nonlinear case. The minimum error is 0.011276, and can not be decreased further since the output must be a linear combination of the inputs, i.e. all the possible outputs must exist in a plane. This should be compared with the nonlinear solution that is able to decrease the error further by nonlinearly operating on the inputs (a zero error is possible). This effectively corresponds to “bending” the regression plane to fit better the 0 and 1 responses in the two half planes.

It is also interesting to verify that the linear system is much more sensitive to the learning rates than its nonlinear counterpart. Explore the breadboard by changing the learning rates, and observe the different speeds that the regression plane moves to its final position.

NeuroSolutions Example

Go to Next Section

3. The Perceptron

Rosenblatt's perceptron is a pattern recognition machine that was invented in the 50's for optical character recognition. The perceptron has multiple inputs fully connected to an output layer with multiple McCulloch and Pitts PEs (Figure 8). Each input xj is multiplied by an adjustable constant wij (the weight) before being fed to the jth processing element in the output layer, yielding

$$y_j = f\left(net_j\right) = f\left(\sum_i w_{ij} x_i + b_j\right)$$

Equation 18

where bj is the bias for each PE. The number of outputs is normally determined by the number of classes in the data. These processing elements (PEs) add the individual scaled contributions *and respond to the entire input space*.

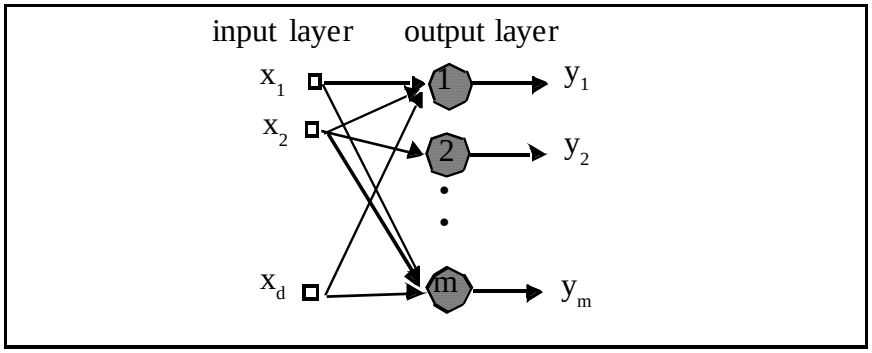


Figure 8**.** The perceptron with d inputs and m outputs (d-m)

After studying the function of each M-P PE, we are ready to understand the pattern recognition power of the perceptron. The M-P PE is restricted to classify only two classes. In general the problem is the classification of one of m classes. In order to have this power the topology has to be modified to include a layer of m M-P PEs, so that each one

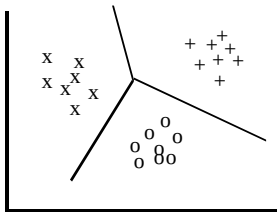
of the PEs is able to create its own linear discriminant function in a D dimensional space (Figure 9). The advantage of having multiple PEs versus the single M-P PE is the ability to tune each PE to respond maximally to a region in the input space.

One of Rosenblatt's theoretical achievements was the demonstration that the perceptron could be trained to recognize linearly separable patterns *in a finite number of steps*. This showed that these adaptive devices not only produced useful pattern classification by tweaking parameters. There are *systematic algorithms to change the weights that converge in a finite number of steps*. These algorithms are called learning rules. The perceptron also had a remarkable property: it was *able to generalize* . Hence, one can say that the perceptron started the field of learning theory. The recent interest in large margin classifiers shows that the perceptron and its algorithm is still at the center stage to design practical classifiers.

3.1 Decision boundaries of the perceptron

An m output perceptron can divide the pattern space into m distinct regions. Suppose that the ith and jth regions share a common boundary. The decision surface is a segment of a linear surface of equation $g_i(x) - g_j(x) = 0$. There are *m(m-1)/2* such equations and so the decision surfaces of a perceptron are segments of at most the same number of hyperplanes. The hyperplanes that effectively define the decision boundary must be contiguous (the others are called redundant). The decision regions of the perceptron are *always convex regions* because we require during training that one and only one of the outputs to be positive. The PE that responds maximally to an input pattern means that the input is inside the region represented by the PE. Hence each PE identifies patterns that belong to a class. The perceptron is therefore a physical implementation of the linear pattern recognition machine presented in Figure 12 of Chapter II.

This example shows the discriminant function and the decision boundary created by the perceptron for a three class problem in 2-D space. The data that created the example is shown in the following Fiugre. What we want to stress is the convex shape of each decision region. This is a characteristic of a single layer network. Notice that there are areas in the space that there is no class assigned to them.



NeuroSolutions Example

3.2. Delta rule applied to the perceptron

What changed in the delta rule when we went from a single PE to the perceptron? Not much, except that now there are several (m) outputs so the cost must be computed not only as a sum over the training set but also as a summation of each output PE. So the cost J becomes

$$J = \frac{1}{2N}\sum_{p=1}^{N}\sum_{i=1}^{m}\varepsilon_{pi}^{2}$$

Equation 19

where p is the index over the patterns and i over the output PEs. Rewriting Eq.15 to adapt the j weight of the ith PE as

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_{ip}}\frac{\partial y_{ip}}{\partial net_{ip}}\frac{\partial}{\partial w_{ij}}net_{ip} = -\left(d_{ip} - y_{ip}\right)f'\left(net_{ip}\right)x_{jp} = -\varepsilon_{ip}f'\left(net_{ip}\right)x_{jp}$$

Equation 20

and the update rule for the weights would be the same as Eq.16 . Note that the gradient of the cost with respect to the weight w_{ij} is computed by multiplying the partial of the cost

with respect to the PE state $\left(\frac{\partial J}{\partial y_{ip}}\right)$ scaled by the derivative of the nonlinearity of the PE and the input activation. Let us define the local error δi for the ith PE as

$$\delta_{ip} = \frac{\partial J}{\partial y_{ip}} f'\left(net_{ip}\right)$$

Equation 21

We can then conclude that the gradient of the performance surface with respect to weight wij is computed by

$$\frac{\partial J}{\partial w_{ij}} = \delta_{ip} x_{jp}$$

Equation 22

which are local quantities available at the weight, i.e. the activation x_{jp} that reaches the weight wij, from the input and the local error δ_{ip} propagated from the cost. So, the amazing thing about this algorithm is that it is *local to the weight*, i.e. only the local error δi and the local activation xj are needed to update a particular weight. This means that it is immaterial how many PEs the net has, and how complex their interconnection is. The training algorithm can concentrate in each PE, and work only with the local error and local activation.

NeuroSolutions 9

3.9 The perceptron for character recognition

This NeuroSolutions breadboard implements Rosenblatt's perceptron with two minor modifications: Instead of using threshold PEs it implements the classifier with tanh nonlinearities. The reason is to utilize the delta rule to train the machine, instead of the perceptron learning algorithm developed by Rosenblatt.

The task will be character recognition. We created the 10 digits in 8 pixel by 8 pixel black and white images. We have placed an image viewer on the input axon so you can view the images. The output layer is made up of 10 PEs, one for each of the digits. Since we are going to use the delta rule, we placed an L2Criterion at the output and included an output file with the desired response, which is a value of one for the corresponding digit and zero for the others. We will place a BarGraph

probe over the output file to show the net response for each input pattern.

Training will be done using on-line learning, i.e. the weights are modified after the presentation of each pattern. Learning rates are appropriately set for the task (more about this later). We will place a MegaScope at the cost access of the L2Criterion to display the change of the MSE with the iterations, i.e. our already known learning curve.

What do you expect to see? In the beginning the weights are random, so any combination of outputs can appear at the output, which produces a large error. But after a few iterations the weights are modified to track the desired response. In the BarChart, the largest response will scroll from top to bottom meaning that only a large output is present, and that it matches the digits (presented in sequence 0-9). So, when the network is trained the output will follow the desired response.

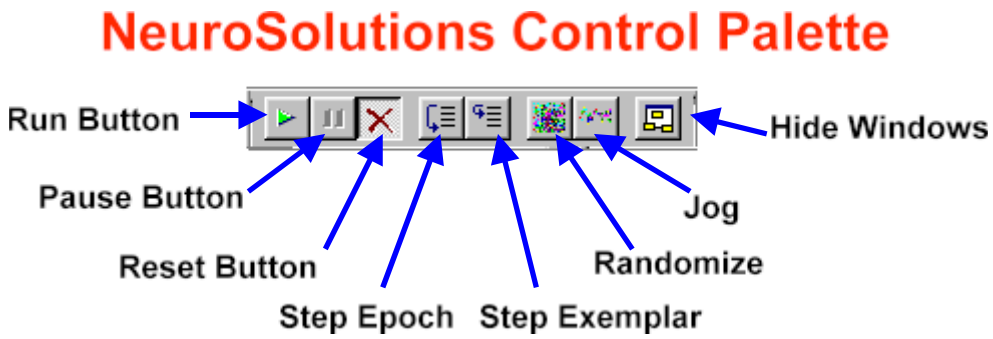
It is remarkable that 640 weights (the Synapse Inspector displays the number of weights) are trained so quickly. In practice, however, the problem of digit recognition is much more difficult because the characters differ from person to person, they appear concatenated, and can appear misaligned.

One interesting aspect is to find out how robust the obtained solution is. This can be measured by altering the input with the weights fixed, and finding the network response. The network does not classify the noisy inputs very well. The inputs are very different from the clean noiseless images used for training. An interesting question is: can we improve the robustness of the classification if we include noise during the training? This can be easily checked with the set up. The initial condition will be the weights obtained without noise. We will train the network more with the new noisy data and see if it can improve its performance.

We will see that the error will decrease further, albeit in an erratic way. This means that the network has learned to cope with small amounts of noise to a certain extent. The outputs should look cleaner, and we can even increase the noise

variance and still obtain reasonable results. This experiment shows how adding small amount of noise to clean data sets may improve the performance when noise is present. However, the method requires a tight control of the experimentation to produce good results.

During this example, we will ask you to single step (step exemplar) through the training to watch the input and output of the system for each data point. This is done by clicking the "step exemplar" button on the NeuroSolutions Control Palette. At the end of a run, the network may not allow you to single step the network since you have completed the experiment. You can fix this situation by increasing the number of training epochs in the Controller inspector panel.



Remember that this is a "live" breadboard so you can experiment with the breadboards at will. We recommend that you change the learning rates, the noise source variance, use other probes such as the Hinton probe to display the weight values, etc.

NeuroSolutions Example

3.3. Large Margin Perceptrons

We have seen above that the perceptron algorithm is very efficient, but not very effective because the movement of the discriminant function stalls as soon as the last sample is classified without error. This normally leaves the separation surface very close to the last sample misclassified. Obviously that this solves the problem in the training set, but we also feel that it may not produce the best possible classification for data in the test set.

This is the reason we would like to modify the perceptron training such that the location of the decision surface is placed in the valley between classes and at equal distances from the class boundaries. For this we have to introduce and define the concept of margin.

Suppose we have a set of data and labels $S = \{(x_1, d_1)..., (x_n, d_N)\}$, with d = {-1,+1}, and we have a linear discriminant function defined by (w,b). We define the *margin* of the hyperplane to the sample set S as

$$\gamma = \min_{x \in X} \left| \langle x, w \rangle + b \right| > 0$$

Equation 23

where <.> means the inner product of x and w. We can show that the margin is related to the inverse of the L2 norm of the hyperplane's weight vector w, i.e. $\gamma = 1/\|w\|^2$ (Vapnik).

We define the *optimal hyperplane* as the hyperplane that maximizes the margin between the two classes (Figure 9). As can be seen in this Figure from all the possible hyperplanes that separate the data, the optimal one is halfway between the samples that are closest to boundary between the classes.

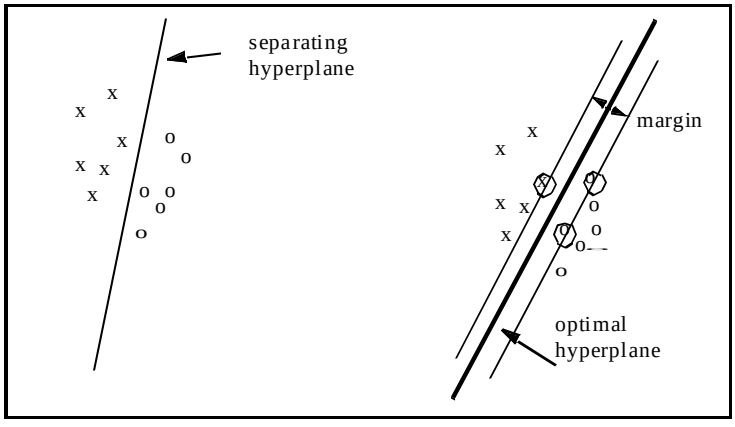


Figure 9. Hyperplane with largest margin.

Vapnik showed that the optimal hyperplane provides the smallest bound on the VC dimension, which is one of the best things we can do to guarantee low error rate in the test set. So the issue is how to find this optimal hyperplane. We see from Figure 9 that

we first have to find the points that are closest to the boundary (called the support vectors), and then place the discriminant function midway.

3.4. The Adatron Algorithm

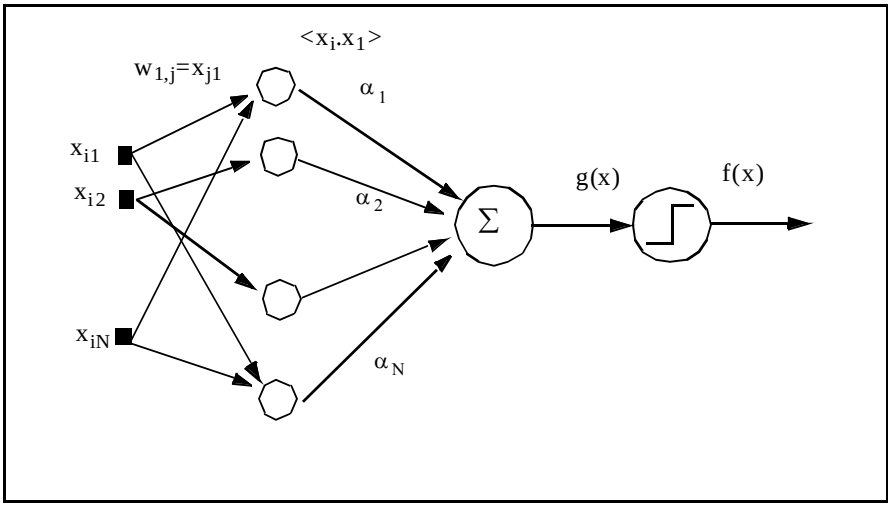
Here we will give a very simple algorithm known as the Adatron algorithm to find the parameters of the discriminant function which possesses the largest margin. This algorithm is sequential and is guaranteed to find the optimal solution with an exponentially fast rate of convergence.

In order to explain the procedure, we have to write the discriminant function of the perceptron in terms of the *data dependent representation*, i.e.

$$f(x) = \mathrm{sgn}(g(x)) \quad where \quad g(x) = \langle x, w \rangle + b = \sum_{i=0}^{N} \alpha_i \langle x, x_i \rangle + b$$

Equation 24

where <.> is the inner product, N is the number of samples, αi are a set of multipliers one for each sample, and we consider the input space augmented by one dimension with a constant value of 1 to provide the bias. Let us see how to construct a topology that creates this data dependent representation. We just have to read the equation: Notice that one can create the inner product by creating a system with N linear PEs where the weights from the input layer are exactly the samples from the training set. The α are then weights connecting the hidden layer PEs to the output. This system creates an output that is the same as the perceptron. Note also that once the training data is given the first layer weights are immediately fixed. We even recommend that the weights be stored multiplied by the corresponding desired response (see below the algorithm).



In this representation, the perceptron learning algorithm of Eq. 9 updates the αi instead of the weights when there is an error (i.e. when $sign(g(x_i)) \neq d_i$) and it becomes

$$\alpha(n+1) = \alpha(n) + \eta d(n)$$
$$b(n+1) = b(n) + \eta d(n)$$

where *d(n)* is the desired response of the perceptron at iteration *n* and η is the stepsize. In on-line learning we assume that we start from the first pattern and move on in the training set until convergence, so we substitute the ith index by *n*. The Adatron algorithm chooses the alphas such that the following quadratic form is optimized

$$J(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j d_i d_j \langle x_i, x_j \rangle$$

$$subject \quad to \quad \sum_{i=1}^{N} \alpha_i d_i = 0 \qquad \alpha_i \geq 0, \forall i \in \{1,...N\}$$

Equation 25

This in general is a difficult optimization problem that has nevertheless a simple solution provided all the inner-products among the input data are calculate beforehand (as specified by the Adatron algorithm). The algorithm is as follows:

Define $g(x_i) = d_i(\sum_{j=1}^{N} d_j \alpha_j \langle x_i, x_j \rangle + b)$ *and* $M = \min_i g(x_i)$ and choose a common starting multiplier (e.g. αi=0.1), learning rate η, and a small threshold (e.g., t =

0.01). Note that we can compute g(xi) locally at each PE if dj is available at the input layer (from the data file), or equivalently, if the weights to the hidden layer are stored multiplied by the corresponding desired response.

Then, while M>t, choose a pattern xi, and calculate an update $\Delta\alpha_i = \eta(1 - g(x_i))$ and perform the update

$$\begin{cases} \alpha(n+1) = \alpha(n) + \Delta\alpha(n), \quad b(n+1) = b(n) + d(n)\Delta\alpha(n) & \alpha(n) + \Delta\alpha(n) > 0 \\ \alpha(n+1) = \alpha(n) \quad , \quad b(n+1) = b(n) & \alpha(n) + \Delta\alpha(n) \le 0 \end{cases}$$

Notice that for each input and corresponding desired response we can compute $\Delta\alpha$i locally. So the Adatron algorithm adheres with the local implementation constraint of neurocomputation.

The Adatron algorithm is applied to a perceptron (threshold nonlinearity) with a single output, i.e. it is only able to discriminate between two classes. If the problem has multiple classes, it must be solved as a sequence of two class decisions. The algorithm resembles the perceptron learning algorithm given above, except in the updates. Let us see how the Adatron works for the problem we solved in section 2.3.

It is very useful to compare the Adatron algorithm with the delta rule described above, to contrast their differences. In the delta rule, the positioning of the boundary is primarily controlled by the samples that produce outputs that differ from the desired values of 1 or -1. These samples tend to exist in the boundary between the classes (Figure 10). So the MSE is controlled by the samples that are close to the boundary between classes. However due to the fact that *J* (Eq. 19) is a continuous function of the error, all the samples contribute somewhat to *J*. Therefore, the MSE is a function of the full data distribution and the location of the boundary will respond to the shape of the data clusters.

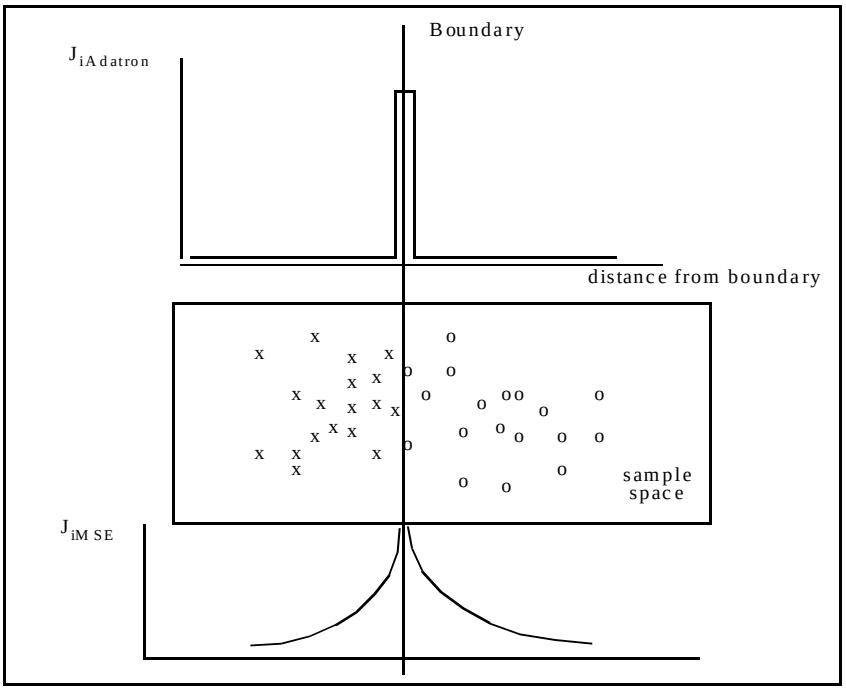


Figure 10. Differences in the distribution across samples in the Adatron and delta rule

In the Adatron algorithm a very different behavior happens. During the adaptation, most of the αs go to zero, and the location of the boundary is solely determined from a few samples close to the boundary which are called the *support vectors*. So the adaptation algorithm is insensitive to the overall shape of the data clusters and concentrates on the local neighborhood of the boundary to set the location of the hyperplane. Depending on the data distributions this may provide different boundaries. In particular it has been shown by Vapnik that the large margin classifier generalizes better.

3.5. Limitations of the perceptron

The prototype problem that is not linearly separable and thus cannot be solved by perceptrons is the exclusive-or function (XOR). The exclusive-or truth table is presented in Figure 11.

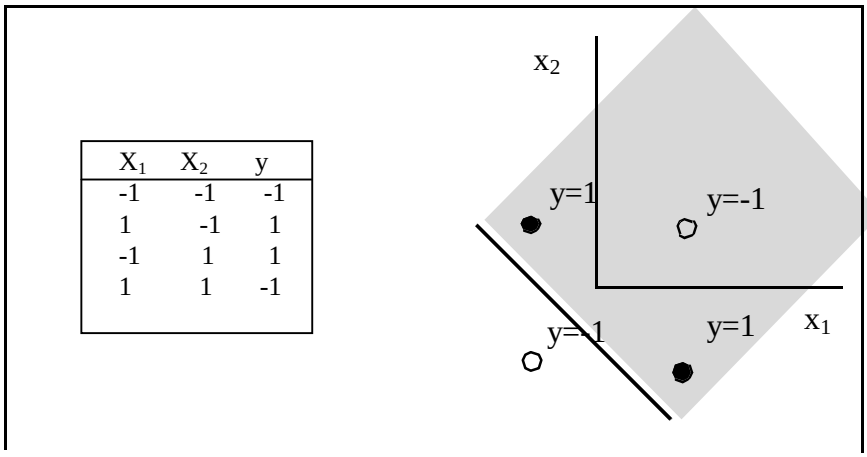


Figure 11. The XOR problem in pattern space

No matter where we place the half plane that includes the "1" responses it will always include one of the "0" response. So this is an example of *non-linearly separable patterns*. The parity function extends the XOR to higher dimensions. NeuroSolutions can be used to build this example.

NeuroSolutions 10

3.10 Perceptron and the XOR problem

This example will show what happens when the perceptron tries to solve the XOR problem. We will start with a modified M-P PE, built with the tanh nonlinearity. The network has two inputs and one output. Training will be done with the delta rule in batch mode. The Discriminant probe shows the separation surface, the MegaScope displays the learning curve to observe how learning progresses. There are two important points about the XOR problem and the M-P PE. First, there is no way for the M-P PE to solve the problem. The XOR is not linearly separable. Second, there are many different local minima in the performance surface. The least interesting of which is the one where all the weights quickly approach zero and the network output is thus always zero. This will produce a discriminant plot which is all grey. The other more interesting minima are when the discriminant line separates one of the four points from the other 3 – this is the best the network can

do, getting 3 out of four correct. Notice that both of these minima produce the same mean squared error. Why is this so when one network correctly classifies 3 of four points and the other produces no output at all?

With batch mode training, the weights are updated after an entire epoch. This produces a smoother weight track, but in this example, the smooth weight track tends to very often lead directly to the all zeros (uninteresting) solution. If we change the training to on-line you will find other solutions. Why? Because the on-line training is much noisier than the batch mode training, thus it has a tendency to bounce out of local minima. Imagine doing gradient descent on the performance surface in Figure 7, a smooth track will lead to a single minima whereas a noisy track will bounce around and end up in various locations.

This example clearly shows that this problem has multiple minima and depending upon the update rule some of the solutions are preferred. But none is able to correctly classify all the patterns. Notice that the inclusion of nonlinearity, even in a one layer network produced performance functions that are non-convex.

NeuroSolutions Example

Minsky showed that the perceptron was not a general purpose processing device because the possible decision regions that the machine can create are convex, formed through the intersection of hyperplanes (an extension of a plane to more than two dimensions). A lot of problems in practice do not fit this description. The XOR function just described is a simple case.

Go to next section

4. One hidden layer Multilayer Perceptrons

Multilayer perceptrons (MLPs) extend the perceptron with hidden layers, i.e. layers of processing elements that are not connected directly to the external world. multilayer perceptrons

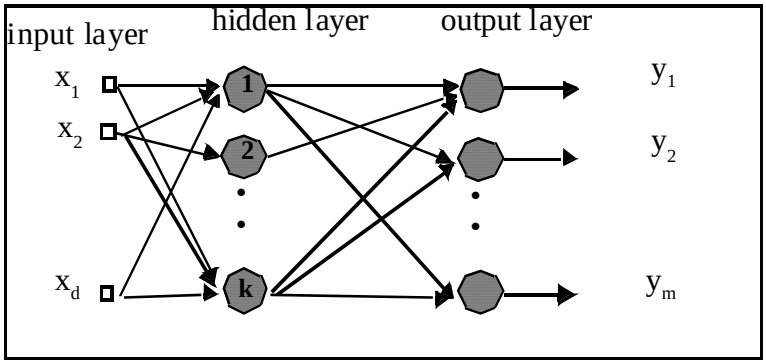


Figure 12. A multilayer perceptron with one hidden layer (d-k-m)

Figure 12 shows a one hidden layer MLP with d inputs, k hidden PEs and m outputs (MLP(d-k-m)). Normally the PEs in MLPs are nonlinear sigmoid PEs. Let us analyze the extra processing power that a layer of nonlinear PEs achieves in terms of discriminant functions.

We will extend the perceptron given in Figure 8 by cascading one extra processing element. The hidden layer will have two processing elements as shown in Figure 13. We will start our study of the MLP with threshold processing elements (to facilitate the explanation we will assume that the outputs are between 0,1). The goal here is to find the discriminant functions produced by one hidden layer MLPs.

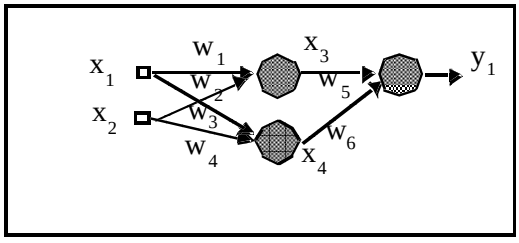


Figure 13. A one hidden layer (2-2-1) perceptron

Conceptually the one hidden layer MLP is a cascade of perceptrons. It is straight forward

to see using this interpretation that the two hidden layer PEs create two linear discriminant functions in the (x_1, x_2) space. Let us label the output of each hidden processing element as x_3 and x_4. Each of these variables will be positive above a straight line with a slope given by the ratio of the respective local weights.

In the space (x_3, x_4), the output PE is also a perceptron. It will also construct a linear discriminant function, i.e. it will have a positive response above a straight line with slope given by $-w_6/w_5$. The problem is that we are interested in finding the overall positive response in the input space (x_1,x_2). This is a straight forward (but messy!) problem in the composition of functions, since we know the (nonlinear) parametric relation between x_3 and x_1,x_2, and between x_4 and x_1,x_2. It is instructive to write the overall input-output map as

$$y = f\left(w_5 x_3 + w_6 x_4 + b_3\right) = f\left\{w_5\left[f\left(w_1 x_1 + w_2 x_2 + b_1\right)\right]\right.$$
$$\left. + w_6\left[f\left(w_3 x_1 + w_4 x_2 + b_2\right)\right] + b_3\right\} = f\left\{g_1 + g_2 + b_3\right\}$$

Equation 26

4.1 Discriminant functions of the MLP

The multilayer perceptron constructs input-output maps that *are a nested composition of nonlinearities*, i.e. they are of the form

$$y = f\left(\sum f\left(\sum\left(\ \bullet\ \right)\right)\right)$$

Equation 27

where the number of function compositions is given by the number of network layers. The resulting map is very flexible and powerful, but it is also hard to analyze. Our goal now is to find out what type of discriminant function can be created with the map of Eq.27).

Let us assume that the output layer weights are set to one. Each expression inside the brackets creates one linear discriminant function, yielding after the nonlinearity a function with a positive value across a half-plane (a step function). The location of the transition in the input space is controlled by the discriminant function. So the expression inside the curly brackets is the addition of two step functions, g_1 and g_2, with a bias term b_3. In the

region on the input space where both functions g_1 and g_2 are positive the value of y will be the largest. The output *y* will have an intermediate value in a subspace where either one of the *g* functions is positive (but not both); and finally there is an area in the input space where *y* is equal to the bias b_3 because each one of the functions g_1 and g_2 is zero.

The shapes of these areas are controlled by the placement of the original discriminant functions (which in turn are controlled by the values of w_1,w_2,w_3,w_4). Notice also that the bias value b_3 is added to the result of the hidden layer partition. So its value will dictate if only the peak value of *y* is positive, or if the peak and one of the plateaus are positive, if all are positive or if all are negative. Hence the *role of the bias at the output layer is substantially different* from the simple control of the *y* intersect as the input PEs' biases. The *bias reveals different details of the composition of functions*, effectively changing the overall assignment of values to the partition created by the hidden layer. The output weights w_5 and w_6 enhance the flexibility (can give different weights to the output of each hidden PE) and change further the sign of the hidden PE activations.

The interplay among all these parameters becomes quite involved, but we have the feeling that the discriminant function of the one hidden layer perceptron is much more flexible than that of the perceptron. Let us go to NeuroSolutions and create a breadboard to help us get familiar with the discriminant function of the one layer MLP. Remember that the discriminant probe provides a way to visualize the shape of the overall discriminant function by gray coding the input space with the value of the output.

NeuroSolutions 11

3.11 One hidden layer MLP in 2-D space

This breadboard implements the one-hidden-layer perceptron shown in Figure 12, with two TresholdAxons in the hidden layer but a TanhAxon at the output. We have the discriminant probe attached to the output of the network, thus it will show the combination of the discriminant lines created by the hidden layer PEs. We will

see how the MLP separates the input space into multiple areas with different values (depending on the values of the weights) – this is often called a tesselation . Notice that since the tessellation is made up additively, a change in sign of one of the weights changes all the regions, not only the ones that belong to the PE which had the weights modified. This tight coupling between the PEs is what makes the MLP so powerful, because to obtain a given tessellation, *all the PEs have to contribute to it*. In other words the training procedure for the MLP is not greedy . Another interesting aspect of the MLP is that *different weight combinations can lead to the same tessellation*.

NeuroSolutions Example

Figure 14 shows a summary of the tessellation obtained with the two hidden layer MLP. The conclusion is that by adding an extra layer to the perceptron we have qualitatively changed the shape of its discriminant function. The decision regions are no longer restricted to be convex , because the network has a more powerful composition mechanism.

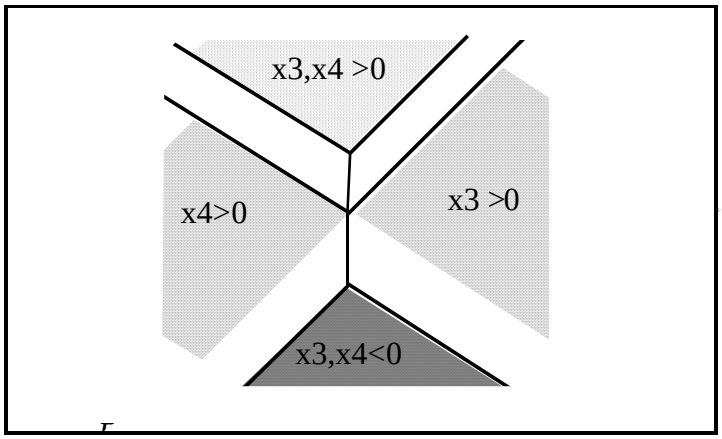


Figure 14. Tessellation of a one hidden layer MLP (2,1) in 2D space.

There are several important conclusions to be drawn from this example.

- First, the maximum number of distinct regions of the input space is controlled by the number of hidden PEs (2k for k>>d). An alternative statement is: each hidden PE creates a linear discriminant function.
- Second, the PEs on the top layer have the ability to combine some of the regions created by the hidden PEs either by a multiplicative effect or by an additive effect. This creates decision regions that are no longer convex.
- Third, there is more than one weight combination that achieves a particular arrangement of decision regions.

Can we solve the XOR problem with the MLP (2,2,1)? The answer is YES if the discriminant functions are modified according to the goal of creating a slanted (45 degree) "bright" strip that passes through the origin (see Figure 9). Let us do it first by hand.

NeuroSolutions 12

3.12 Solving the XOR problem by hand with the one hidden layer MLP

The Table for the XOR is presented in Figure 11. Note that to solve this problem we have to define a "bright" region that includes two of the outputs only. This can be accomplished if we place two discriminant functions at 45 degrees which are parallel to each other and are positive towards the center. So we need only two discriminant functions, i.e. two hidden PEs. When you have finished the demonstration, try to find two other solutions that will implement a solution to the XOR problem. Remember that the assignment of 1 or -1 to a given class is arbitrary, which gives one more solution if we reverse the assignments. The other solution is obtained with lines at 135 degrees.

The important aspect of this problem is to observe how the single layer MLP globally constructs the solution. Changing a single weight changes the overall decision surface.

NeuroSolutions Example

So we conclude that classification with the MLP is accomplished by adequately controlling the position of the discriminant functions according to the input data and the

desired response. This is the same principle utilized for linear regression and for the perceptron. The machine will automatically discover the position of the discriminant that *correctly classifies the training data*.

4.2. Mapping capabilities of the one hidden layer MLP

An important consequence of the mapping capabilities of the one hidden layer MLP is that it can construct a bump in the input space, i.e. a single limited extent region of large values, surrounded by a region of low values. Notice that the decision region of the low values is no longer convex, so the bump can not be implemented by a perceptron. The simplest bump is triangular and it is obtained with 3 hidden PEs.

3.13 Creating a "bump" with the one hidden layer MLP

In this example we will create a "bump" in a two dimensional space using a one hidden layer perceptron. The goal is to create a bright triangular region around the origin of the input space. In this breadboard we will need three tanh PEs in the hidden layer, since a triangular region requires the combination of 3 linear discriminant functions. When you have successfully solved the problem, experiment with moving the discriminant plot by hand. This experience will come in handy in the future.

NeuroSolutions Example

This shows that the one hidden layer perceptron is able to create a limited extent (local) bump in the input space. This feature is going to be very important for function approximation in general, and for classification in particular. But one hidden layer MLPs can also create other types of nonconvex regions, and even disjoint regions. mapping capabilities of the 1 hidden layer MLP .

Another important conclusion is that if the user knows what is the desired placement of the discriminant functions, then it is possible to synthesize the solution directly. The geometric picture that we developed also applies to higher dimensional spaces but we

loose the ability to visualize the solutions.

4.3. Training the one hidden layer MLP

As a historical note, the perceptron and the multilayer perceptron are trained with *error correction learning,* which means that the desired response for the system must be known. In pattern recognition this is normally the case, since we have our input data labeled, i.e. we know which data belong to which class. We already know how to train the perceptron (Eq.20). If one wants to utilize the delta rule, we must know how to compute the error at each PE, which requires the availability of a desired response for each network PE. An explicit error is not available in the hidden layer PEs of the MLP. This is known as the *credit assignment problem* and represented the stumbling block that ended the first connectionist era around 1970.

The resurgence of interest in neural networks can be traced to the discovery of a method to train MLPs, which is best known as backpropagation. *or generalized delta rule*. The method was re-invented many times (inventors of backprop) and it was known in control theory since the late sixties, but the connectionist version is much more efficient and can be considered a contribution of connectionism to the theory of gradient descent learning.

We already covered one of the fundamental concepts required to extend the delta rule to MLPs when we discussed and applied the chain rule to the M-P PE. Remember that we showed that the chain rule is a systematic procedure to propagate sensitivities across an undetermined number of internal (hidden) points of a topology. As long as we have an analytic expression that relates the variables along the forward path, we can compute sensitivities to the same intermediate points in the path. If we accept that the output error should be scaled by the network weights (the second ingredient of backprop), then we can propagate the output error to any point along the path, and update the weights with the scaled error. *In other words, we are effectively substituting the availability of the desired signal at the intermediate points in the path by the propagated and scaled output*

error. This worked for the M-P PE, and works in general. How can this help us train the MLP?

Let us examine Figure 15 that depicts a piece of a one hidden layer MLP. The goal is to adapt the weights that are connected to the hidden ith PE. Notice that we do not have a desired response at this point since the PE is not connected to the external world (i.e., it is hidden). In order to adapt its weights we are going to apply the following methodology:

- we are going to substitute an explicit desired response at the ith PE by an error.
- this error is going to be the propagated and scaled output error.
- this sensitivity is going to be automatically computed by the chain rule.

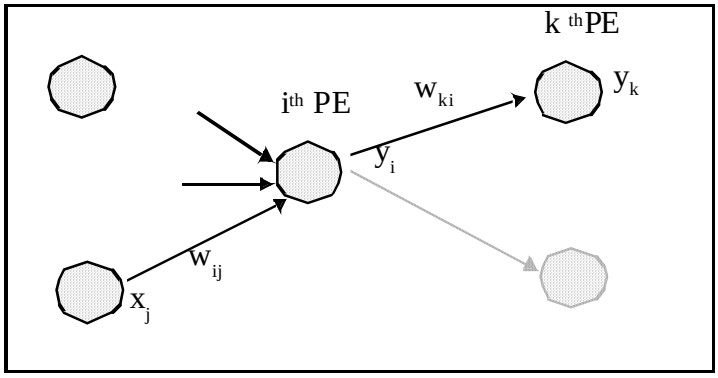


Figure 15. Detail of an hidden layer network

backprop derivation

The weight update using backpropagation is

$$w_{ij}(n+1) = w_{ij}(n) + \eta f'\left(net_i(n)\right)\left(\sum_k e_k(n) f'\left(net_k(n)\right) w_{ki}(n)\right) x_j(n)$$

Equation 28

Let us re-interpret Eq. 28 using the definition of the local error (Eq.21). The summation in Eq. 28 is a sum of local errors δ_k at each network output PE, scaled by the weights connecting the output PEs to the ith PE. Thus, the term in parenthesis in Eq. 28 effectively computes the total error reaching the ith PE from the output layer (which can be thought of as the i'th PE's contribution to the output error). When we pass it through

the ith PE nonlinearity we have its local error which can be written as

$$\delta_i(n) = f'\left(net_i(n)\right)\sum_k \delta_k w_{ki}(n)$$

So there is an unifying link in all the gradient descent algorithms presented so far. ALL the weights in gradient descent learning are updated by multiplying the local error ($\delta i(n)$) by the local activation (xj(n)) according to Widrow's estimation of the instantaneous gradient first shown in the LMS rule

$$\Delta w_{ij}(n) = \eta \delta_i(n) x_j(n)$$ **Equation 29**

What differs is the calculation of the local error, depending upon if the PE is linear or nonlinear and if the weight is attached to an output PE or a hidden layer PE.

Case I: If the PE is linear and at the output, f′(.) is a constant and there is no scaling of the output error that arrives at the ith PE. This is the case found in the LMS rule (Eq.11).

Case II: If the PE is nonlinear and it is at the output, then the delta rule (Eq.21) used to train the perceptron is exactly Eq. 29 with δi substituted by

$$\delta_i(n) = \varepsilon_i(n) f'\left(net_i(n)\right)$$ **Equation 30**

where εi is the error associated with the ith output. We can update the output weights of the MLP with the delta rule since we know the desired response at the output.

Case III: If the PE is nonlinear and is hidden, then the local error is computed by summing all the contributions of the local errors in the output layer, scaled by the corresponding weights, which yields

$$\delta_i(n) = f'\left(net_i(n)\right)\sum_k \delta_k w_{ki}(n)$$ **Equation 31**

So, structurally, the weight update equations do not change since learning is still using gradient descent. Everyone should remember Eqs. 29, 30 and 31 to apply gradient descent learning to MLPs. Let us apply the backpropagation algorithm to train the one hidden layer MLP how to solve the XOR problem. multilayer linear networks

NeuroSolutions 14

3.14 Solving the XOR with backpropagation

In this example we will use the same breadboard from the previous example, set the number of hidden PEs to 2, and add the learning layers so that we can implement the backpropagation algorithm to train the network to solve the XOR problem. When you finish the demonstration, try changing the step size (learning rate) by selecting the gradient descent component, opening the inspector, and typing new values. Notice how this changes the dynamics of learning, exemplified by the learning curve. Another thing that you might notice is that the learning curve has sometimes plateaux where the error is basically the same. This corresponds to almost flat regions where the gradient is very small. One can imagine that the performance surface is no longer the simple "bowl" we found in linear regression. Sometimes instead of flat regions one gets local minima and the search gets stuck there because a local minima and the global minimum are indistinguishable with the local gradient. This is one of the added difficulties of working with nonlinear systems.

One other thing to observe is that although the classification is successful, i.e. the MSE is very small, the weights of the system have very different values from run to run. This is due to the fact that the system is started with random initial conditions and there are many possible solutions to solve the XOR. So during adaptation the search takes the system's state along different paths and one of the solutions (we do not know which) is reached. But from the point of classification anyone of them is OK.

NeuroSolutions Example

Notice that there are many different solutions to the XOR problem. When the network weights are seeded randomly different solutions can be found. However, the most important conclusion is that the network can automatically find the placement of the discriminant functions that are so difficult for us to create by hand.

The other example that we will run at this point is the **"**bump**"**. The goal is to show that backpropagation is able to train a one hidden layer MLP to discover a triangular shape decision region. In order to train such an MLP, one needs to construct a training set that will tell the network where its response must be 1 and where it should be zero. Here we selected 20 points in the plane. Ten points are organized in a triangle and the desired response for these samples is 1. The remaining 10 samples are placed around the first cluster, and the desired network response is 0.

NeuroSolutions 15

3.15 Solving the bump with backpropagation

In this example we will use the same MLP that we used previously to construct the bump by hand (3 hidden layer PEs) and add the learning dynamics. We have placed a scatter plot at the net input to help you build the correspondence between the samples and the discriminant function found through training. In particular, pay attention to the relationship between the discriminant function being created, and the evolution of the learning curve. Depending on the initial conditions, the bright area can appear in the center, or more frequently will appear in one corner and find its way towards the center, until if finally creates a central bright spot. Notice that the learning curves are different for each case. Sometimes you will find an initial decrease in the error followed by a long period where the error basically is constant until finally the final solution appears. You should link the learning curve behavior to the positioning of the discriminant function.

Some other times the discriminant functions form a wedge and stay there for ever…. This is a local minimum. It is important to figure out why this behavior occurs. It is obvious that this solution misclassifies one sample (see the scatter plot). And it is also obvious that there is one extra discriminant function that does not appear in the plot. The reason is that it has parameters very close to one of the other discriminants (i.e. they are superimposed on each other). This

happens when the initial conditions or the gradient descent path produce weight updates that create hidden layer weights that are very similar. In such cases the error produced by misclassifying a single sample is not enough to pull the hidden weights apart, and the system rests in this solution. When this happens open the matrix viewers and observe the weight values.

NeuroSolutions Example

One of the central issues in neurocomputing is to appropriately set the number of hidden PEs. There are two extreme cases: either the network has too many hidden PEs to do the job, or it has too few. Each case is important in its own right, because setting the number of PEs correctly is more of an art than a science at our present state of knowledge. Let us start with an overdimensioned hidden layer.

NeuroSolutions 16

3.16 Effect on the number of hidden PEs

In this example we will experiment with the number of hidden layer PEs and how it affects the output and learning dynamics of the network. It is obvious that if the problem can be solved with three hidden PEs it can also be solved with 6. What happens to the positioning of the other 3 PEs discriminant functions?

The chances of getting the correct classification in the training set increase, and the system normally requires less iterations to reach the solution. However, the computational burden is increased, which slows down the training. But the big problem is that the redundant PEs may have detrimental effects on the test set performance (data that the system never saw before) because they may memorize the training data. This may create overfitting or spurious areas that produce in-class responses in regions of the input space that do not contain any training data (and therefore are "don't care" regions given the training data). Putting it another way the machine may not perform very well in data it was not trained with. This does not happen in this case due to the symmetry of the training data, but can

happen in practice.

You should use a MatrixViewer to observe the values of the weights, from the output layer to the input. You will find that some of the output weights are small, so they are not important for the mapping. This is a benign case. The problem is if some of the weights are large and make the system respond to regions of the input space void of training samples. This is when the system makes mistakes in the test data. The input layer weights appear also duplicated (i.e. several superimposed discriminant surfaces). This is pure waste, but does not affect the performance in the test set much.

NeuroSolutions Example

The other important condition to study is when the network does not have enough hidden PEs to solve the problem correctly. We saw this when we tried to solve the XOR problem with the perceptron. The machine does not know if the problem is linearly separable or not, so it will try to do its best, classifying most of the samples correctly.

The discriminant function first finds placements that correctly classify the *majority of samples*, and slowly moves to classify the areas with lesser samples. The reason for this behavior is rooted in the form of the cost function (a sum over all the training patterns) that is guiding the automatic placement of the discriminant function. If the learning machine does not have enough degrees of freedom, the error will stabilize at a high value, and the weights and bias will basically stay unchanged. Sometimes, oscillations can occur when the learning rates are large. The oscillations may have high amplitude and correspond to sudden change in weights between disjoint sets of values.

NeuroSolutions 17

3.17 Fewer hidden PEs than required

Let us now train the MLP to create a bump, but in this case let us reduce the number of hidden PEs to 2. This MLP can not solve the problem exactly since at least 3 PEs are required. So what will the training do? The machine does not know

if it has enough degrees of freedom or not. It blindly changes the weights to decrease the MSE. So the solutions found will always classify correctly the majority of samples (unless in pathological cases of symmetry where the error can be minimized by other means).

For some initial conditions we may observe an oscillation, where the error abruptly goes up and then down. During these periods the weights change rapidly, and the discriminant function oscillates between two positions. This means that the weights are driven to values that makes the discriminant suddenly increase the number of errors. So the weights have to move back. One nice way to observe this effect is to place a MegaScope on top of the weights and see the weights change with iteration.

NeuroSolutions Example

Go to next section

5. MLPs with two hidden layers

5.1. Discriminant functions of the two hidden layer MLP

What are the discriminant functions of two hidden layer perceptrons? Such a network has three levels of function composition (Eq.27), i.e.

$$y = f\left(\sum f\left(\sum f\left(\sum \left(\bullet\right)\right)\right)\right)$$

So we can once again study the problem by finding the decision regions created by the one hidden layer MLP (the previous topic), and their composition created by the added output perceptron. This becomes a lot more complex, but it is important to understand which are the basic capabilities of the overall discriminant functions.

From the previous discussion, the one hidden layer MLP can create local "bumps" in the input space (output is positive or negative). Another layer with several PEs can be

thought of as combining bumps in disjoint regions of the space. This is a very important property, because in the theory of function approximation (we will touch on this subject in Chapter V), there are well established theorems (Cybenko , Hornik, Stinchcombe and White) which state that a linear combination of localized bumps can approximate any reasonable function. Therefore an *MLP with two hidden layers is also a universal approximator*, i.e. it can realize any input-output map, just like the one-hidden layer MLP.

A word of caution is in order at this point. These theorems are *existence theorems* (the number of PEs is unconstrained), so they do not address the engineering question of **"**how many PEs and layers do I need to solve my problem?**"**. This is still an open question for which experimentation is necessary. But it is extremely important to know that theoretically an MLP is a universal approximator, and that one or two hidden layers are all it takes to reach this arbitrary mapping capability.

One should associate the number of PEs in the first hidden layer with the number of linear discriminant functions in the input space. One will need in general 2N hidden PEs in the first hidden layer (N is the number of dimensions in the input space) and a modified M-P PE in the second hidden layer to form a single **"**bump**"**. The number of PEs in the second hidden layer creates the number of **"**bumps**"** in the input space that are needed for the approximation. The output layer simply combines these bumps to produce the desired input-output map. So this is a constructive reasoning that two hidden layer MLPs can approximate any function.

This view is valuable to understand the function of MLPs for classification and to appreciate the difficulty of selecting an appropriate topology. If we have some *a priori* knowledge of our data clusters we can judiciously set the size of the network. Unfortunately, since pattern recognition problems are normally high dimensional and the cluster knowledge is scarce, this method does not go very far....

3.18 Creating a Halloween mask by hand with a two hidden layer MLP

In this example we will set the weights of a two layer MLP by hand to create a halloween mask in the input space as illustrated in the Figure 16. This mask is a simple example of a general input output function. It can be thought of as a classification problem where one of the classes is scattered (multimodal) in the input space (the location of the bright regions), and the other class is in the dark region.

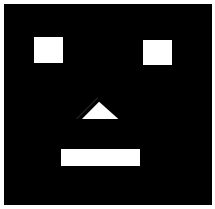


Figure 16**.** Example of input output mapping

We will design the MLP by hand since we know the placement of the discriminants. Observe that we have 4 distinct regions in the input space. Therefore we need 4 PEs in the second hidden layer, one to build each feature in the input space. Looking at the figure we see that we need 13 discriminant functions: 3 for the nose, 4 for one eye, 2 for the other (we use 2 of the first), and 4 for the mouth. Figure 17 shows the topology of the MLP.

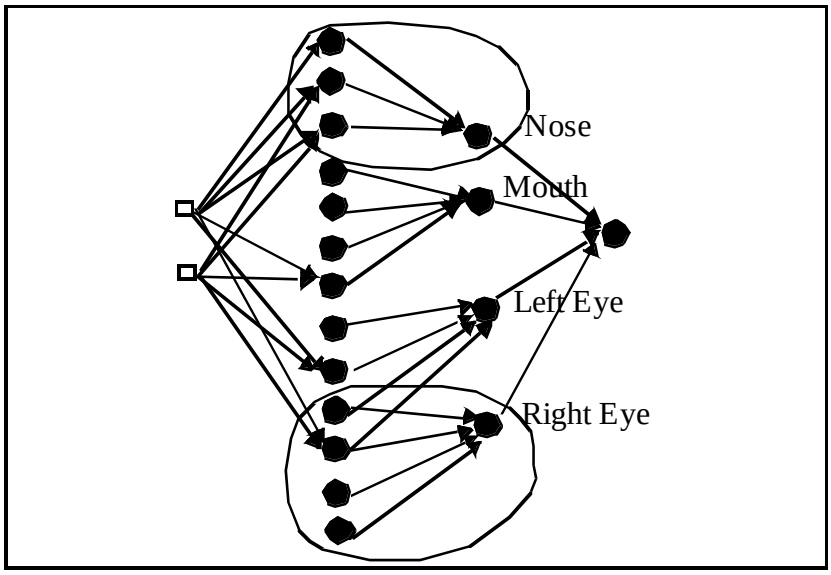


Figure 17. MLP topology (2,13,4,1)

This is a long example, but you should get a good feel for how each layer's weight and bias affect the final discriminant function or tessellation. After creating the mask, change the weights and see if you can predict how it will affect the output.

NeuroSolutions Example

This example illustrates that *when we know* the decision regions we can design the discriminant functions directly. Unfortunately this is very rarely the case for two main reasons: One, the structure of our data sets may be unknown; but even if known we are not able to visualize them in higher dimensional spaces so that we can specify the placements of the discriminant functions. So we need to resort to training for the adjustment of the weights.

5.2. Training 2 hidden layer MLPs with backpropagation

The beauty of the training approach using the backpropagation algorithm is that it is a *systematic, step-by-step procedure* that can be applied independent of the topology of the network and the input dimensionality. Backpropagation (BP) may be more time consuming to position correctly the discriminants of the two hidden layer MLP, but this is a matter of difficulty not of ability to achieve proper training.

The application of BP to the two hidden layer MLP follows the approach outlined for the one hidden layer MLP and will be simply summarized here. First, an input pattern is presented to the network and propagated forward as activation until an output is computed. The injected error is defined as the difference between the desired response and the output. The injected error is then used to compute a local error at every PE in the topology by starting at the output layer and going backwards layer by layer until the input. At this point an activation and a local error is available to every weight in the network. According to gradient descent we just need to change the weights proportionally to the product of these two quantities (Eq.38).

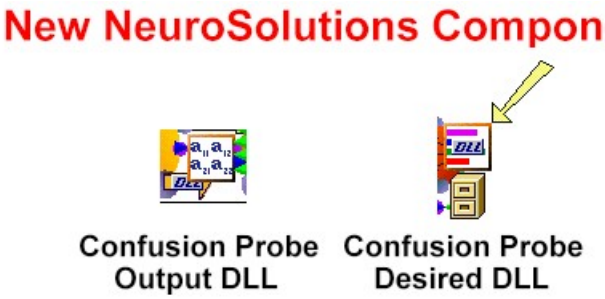
We can expect a very similar behavior between the training of the two types of MLPs, i.e.

there are solutions that take a long time to reach, if at all, due to the existence of local minima and saddle points.

NeuroSolutions 19

3.19 Comparison of the perceptron and MLPs in real data

In this example we will summarize the network topologies we have discussed so far by trying to solve the male/female classification problem using a single perceptron, one hidden layer MLP, and two hidden layer MLP. The male/female classification problem was discussed in Section 2 and involves the trying to determine if a subject is male or female based upon their height and weight. Please refer to Figures 5 and 8 of Chapter II. As you can tell, there is no way to correctly classify all the data points. Remember that the optimal discriminant is quadratic. How will this affect the best choice of network topologies?



We have added a new probe in this example, the Confusion Probe. The Confusion Probe generates a Confusion matrix which is a simple methodology to display the classification results of a network. The confusion matrix is defined by having the desired classifications on one axis and the predicted classifications on the other. Thus, for each exemplar, a one is added to the cell entry defined by (desired classification, predicted classification). Since you want the predicted classifications to be the same as the desired classifications, the ideal situation is to have all the exemplars end up on the diagonals of the matrix. A few example confusion matrices are shown below:

		Predicted Classification	
		Male	Female
Desired Classification	Male	50	0
	Female	0	50

Example 1

		Predicted Classification	
		Male	Female
Desired Classification	Male	45	5
	Female	9	41

Example 2

In example 1 we have perfect classification. Every male subject was classified by the network as male and every female subject was classified as female. There were no males classified as females or vice versa. In example 2 we have imperfect classification. We have 9 females classified incorrectly by the network as males and 5 males classified as females.

One of the interesting things about this example is that the error sometimes increases during adaptation. This clearly shows that the weight tracks are not moving exactly along the gradient descent direction. This may happen because we are using a local estimate of the gradient that may be noisy. The second thing to note is the relationship between the number of classification errors and the MSE. They are related in a coarse manner, i.e. in the beginning of training both are high and they decrease. However, the MSE may be decreasing and the number of misclassifications can be stable or even increase. This is due to the fact that MSE is sensitive to the differences between the desired responses and the actual output, while the number of mistakes is a digital quantity that only looks at the largest output (here the largest output is assigned to the class).

NeuroSolutions Example

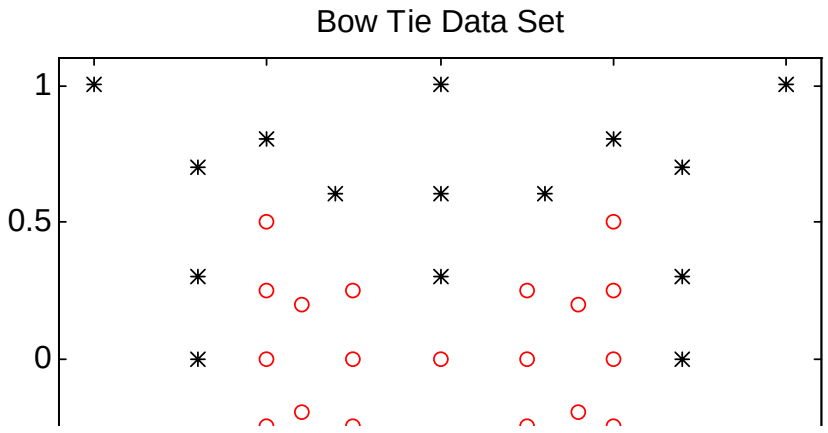
In the previous example the one hidden layer MLP and the 2 hidden layer performed at the same level of accuracy. But sometimes there are problems for which the 2 hidden

layer MLP has an advantage. As we mentioned, such machines are able to generate arbitrary decision regions which can be disjoint and non-convex. In many problems the performance is the same and the only difference is the speed of convergence (two hidden layer is slower to converge). But the fact is that the two hidden layer machines can form all of the discriminant functions of the one hidden layer MLP and many others, so they are more versatile, although we know that asymptotically a very large one hidden layer MLP would approximate the same performance. As a rule of thumb, we should always start our experiments with the simpler topology, since the two hidden layer MLP trains normally slower than the one hidden layer.

NeuroSolutions 20

3.20 Solving the bowtie with MLPs

This example solves a problem that a one hidden layer MLP can only approximate, while it can be exactly solved by a two hidden layer machine. The class distribution in 2-D space looks like a bow-tie, so we will call it the bow-tie problem. We created the two class problem with the distribution shown in the figure below.



Let us experiment with the three topologies for this case (the perceptron, the one hidden and the two hidden layer perceptrons). Modify the number of hidden layer PEs and watch the confusion matrix. You will see that the 2 hidden layer MLP gives a lower error for this problem, but if you increase the number of hidden layer PEs the one hidden layer MLP the performance differential disappears.

NeuroSolutions Exaumple

Go to next section

6. Training static networks with the backpropagation procedure

We saw that a two hidden layer MLP can be trained with the backpropagation algorithm derived initally for the one hidden layer MLP. The backpropagation algorithm can be applied without any modification to the two hidden layer MLP, or for that matter to ANY feedforward topology. This is due to the fact that the *formula Eq.31 applies locally to each hidden PE*, no matter where it is placed in the topology. The only fact to remember is that we have to start computing the local errors from the output of the network towards the input.

There is an *intrinsic flow* in the backpropagation algorithm. First, one data sample is sent through the network to find an output and compute an error (Figure 18). Then we start by calculating the injected error at the output layer and reflect it to the input of the output PE (the δk in Eq.28)). Then the errors in the previous layer can be computed by Eq. (31), and so on until we reach the input layer. Once all the local errors are found, Eq.19 is used to compute the output weight updates and Eq.28 is used to compute the hidden layer weight updates.

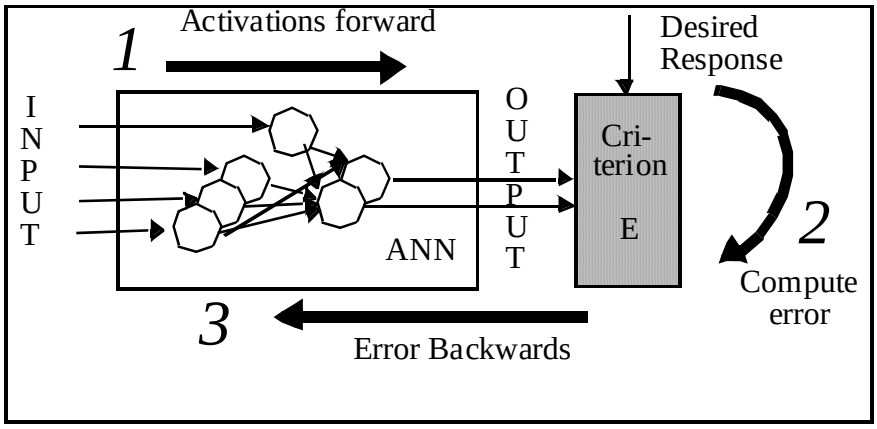


Figure 18**.** Chaining of operations in the backpropagation algorithm

Although these general rules for computing the gradients seem pretty reasonable from our presentation, we have mostly focused on the local computations of the gradients, and we do not have a clear idea of the possible limitations of the technique, if any. In order to answer this question a more principled approach to study gradient calculations in distributed architectures seems necessary.

6.1. Gradient computation and ordered networks

We will study briefly the gradient computations in *ordered networks*. An ordered network is a network where the state variables can be computed one at a time in a specified order. A MLP is such a network. Normally each PE (and weight) is numbered starting from the input (left) to the output (right) as in Figure 19

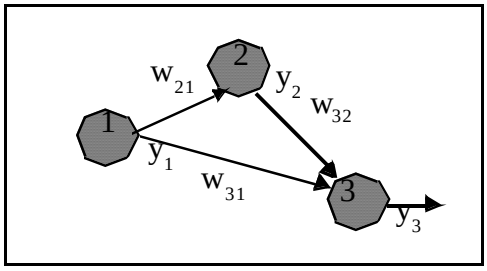


Figure 19**.** Simple 3 PE network

Suppose that we want to compute the partial derivative of $\frac{\partial y_3}{\partial y_1}$. In ordered networks there are two contributions to this derivative: an *explicit* or direct dependence, and an *implicit* or indirect dependence through the network. This total partial is called an ordered derivative . The direct dependence will be denoted with a superscript d. In the network of Figure 19, we have

$$\text{explicit} \quad \frac{\partial^d y_3}{\partial y_1} = w_{31} \qquad \text{implicit} \quad \frac{\partial y_3}{\partial y_1} = w_{32}\frac{\partial y_2}{\partial y_1} = w_{32}w_{21}$$

$$\text{order derivative} = w_{31} + w_{32}w_{21}$$

Equation 32

The output of a high numbered PE (ith) in an ordered network with N PEs requires the computation of the output of ALL lower numbered PEs (jth), i.e.

$$y_i = f\left(\sum_{i>j} w_{ij} y_j\right) + x_i$$

Equation 33

where f(.) is the static nonlinearity, and xi is the PE input (if the PE is an input PE). Note that the summation index enforces i>j, which implies a feedforward topology. This is exactly what happens in the MLP. The intrinsic dependence in the computations can be captured in an ordered list L

$$L = \left[\left\{w_{ij}\right\}, y_1, \ldots y_N\right]$$

Equation 34

where wij are the weights and yi the PE activations (the state variables). What this means is that variable yi only depends on the variables j that are *located to its left in the ordered list,* i.e. i>j. Since the activations y are a function of the weights, these have to appear first in the list.

Let us define a performance function J(y1,...yN) for this network. Werbos proved that in ordered networks the ordered partial derivatives of J with respect to the states y can be

computed by

$$\frac{\partial J}{\partial y_i} = \frac{\partial^d J}{\partial y_i} + \sum_{i<j} \frac{\partial J}{\partial y_j} \frac{\partial^d y_j}{\partial y_i}$$

Equation 35

This expression states that the ordered derivatives can be composed from the explicit influence (first term) and the implicit effect through the topology (the sum). Note that the gradient computation must be ordered from high indices to low indices, *i.e. in the reverse order of the ordered list*. This is a clean mathematical proof that the computation of the gradients must be done in the way we described, i.e. from the output layers to the input. Hence the name backpropagation for the procedure.

According to the chain rule the derivative of J with respect to the weights is

$$\frac{\partial J}{\partial w_{ij}} = \sum_k \frac{\partial J}{\partial y_k} \frac{\partial^d y_k}{\partial w_{ij}}$$

Equation 36

so by substituting Eq. 35 we can compute it easily. Note the similarity with backpropagation: Eq. 35 is computing the backpropagated error, while Eq. 36 composes it with the local sensitivity of the state with respect to the weight and yields the local contribution for the gradient. rederivation of backprop with ordered derivatives What we have gained with this analysis is insight on the requirements to apply backpropagation, and on the characteristics of the method. We will address two aspects: the computational complexity of the method, and efficient implementations.

6.2. Computation complexity

The backpropagation algorithm is a contribution of neural network theory to gradient descent learning. In order to fully appreciate this point we have to ask the question, how were gradients computed traditionally? The fields of control theory and digital signal processing have addressed the same problem long ago.

They used what is called the *direct differentiation method* to compute gradients. The equations are very easy for the MLP. Suppose that we want to minimize the cost given by

Eq.22 with the forward structure of Eq.33 with N PEs. Applying the chain rule to Eq. 22 we get

$$\frac{\partial J}{\partial w_{ij}} = -\sum_k \varepsilon_k \frac{\partial y_k}{\partial w_{ij}}$$ **Equation 37**

Let us define the gradient variable

$$\alpha^k{}_{ij} = \frac{\partial y_k}{\partial w_{ij}} \qquad i, j, k = 1,..., N$$ **Equation 38**

where the superscript k refers to the k^{th} state variable. The gradient variable can be computed by differentiating the state equation Eq.33 to yield

$$\alpha^k{}_{ij} = \frac{\partial}{\partial net_k} f\left(net_k\right) \frac{\partial}{\partial w_{ij}} net_k = f'\left(net_k\right)\left[\delta_{ik} y_j\right]$$ **Equation 39**

where f′ denotes the derivative and δik is the kronecker delta that is equal to one only when k=I (zero otherwise). This expression applies to feedforward topologies such as the MLP.

This method is straight forward to implement but it is computationally demanding. It basically says that one needs to compute the sensitivity α of every state to every weight (Eq. 38). Since in an N PE fully connected MLP we have M>N weights, this gives MN quantities. Eq. 39 also shows that for each gradient variable we need a constant number of multiplications. So, the end result is a computational complexity proportional to MN that we will denote by O(MN). The storage for the algorithm is lead by the storage of the gradient variables which is O(MN).

Now let us analyze the backpropagation procedure. The backpropagation procedure is $O(N^2)$, since for a N PE net we have to compute N errors δ, and for each one needs N multiplications (Eq.31). This is the same complexity of the forward path (Eq.33). *So the asymptotic complexity of the backpropagation algorithm is* $O(N^2)$. This should be compared with O(MN) for the direct method. Whenever the network has more weights

than PEs (which is normally the case with MLPs) the direct procedure is more expensive computationally. In terms of storage, *the backpropagation algorithm is also more efficient than the direct computation.*

The savings come from the *use of the topology* to compute the weight updates. However, this also brings a shortcoming that connectionism has not fully coped with, i.e. *the need to re-derive the learning equations* for each new topology. We will show how this step can be avoided in the simulations if the principles embodied in the ordered derivatives are fully exploited.

6.3. Data flow algorithm for backpropagation

The local nature of the backpropagation algorithm has very deep and important implications for simulation in digital computers. Unfortunately it is not widely used in neurocomputing. So, we will discuss here a local implementation that only requires knowledge of the topology and of the PE input-output map.

For convenience, let us rewrite the activation equation and the error equations again

forward equation $$y_i = f\left(\sum_{i>j} w_{ij} y_j\right) + x_i$$

backward equation $$e_i = -\varepsilon_i + \sum_{j>i} w_{ji}\delta_j$$

Analyzing closely these equations, one can conclude that the expression that computes the local error in the internal PEs is tightly coupled with the original network topology. In the forward pass the network works with the input data *x* and produces activations *y*. The backward equation works with the injected error ε and produces errors *e. Furthermore note that the equations become the same if wij is substituted by wji*. This indicates that once the topology is known both the forward and backward equations can be automatically computed. Figure 20 presents this similarity in more detail. In the top part we show the original neural network around the ith PE, while in the bottom part of the figure we are implementing a network that realizes the backward equation. The network

of Figure 20b is called the *dual (or transpose network)* of the network of Figure 20a.

At the ith PE, the flow of activations (xi) in the original neural topology is from left to right, while in the topology that computes the error (δi) it is from right to left, i.e. inputs become outputs and outputs, inputs. Note also that the summing junctions in Figure 20a become splitting nodes in Figure 20b, and splitting nodes become summing junctions. The weights keep the same values. The incoming error in the dual network is multiplied by $f'(net_i)$.

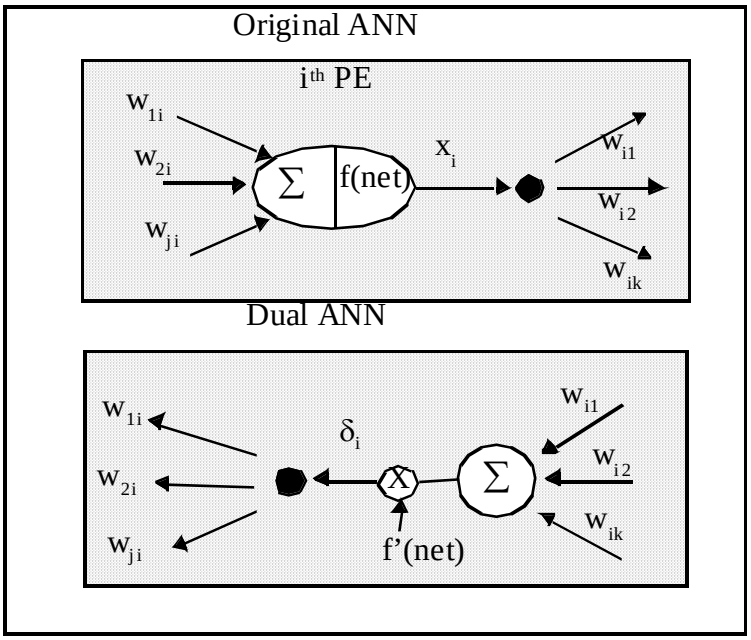


Figure 20**.** a) and b). Original and dual networks

The conclusion of this observation, is that one can imagine that the output error εi created by the difference between the network output and the desired response flows into the dual topology. So, collecting the local error at each PE in the dual network is equivalent to deriving and computing the messy Eq.31 . Note also that in the dual topology, the value of *f'(.)* is computed at *net*i, the activation level of the corresponding PE, i.e. *it corresponds to a linearization of the forward network at the operating point*. This leads to

the following alternate description of backpropagation (Figure 21).

The procedure starts by inputting data to the neural network and obtaining the local activation at every PE (step 1). Then the network output is computed and compared to the desired response to obtain the output error, using the appropriate error criterion (step 2). This error is injected through the dual network and a local error is obtained at each PE (step 3). Note that the error in the dual network is scaled at each node by the derivative of the nonlinearity at the operating point (given by the activation in the original ANN). Now that we have the local error and the local activation we can apply again Eq.29 to compute each and all weight updates (step 4).

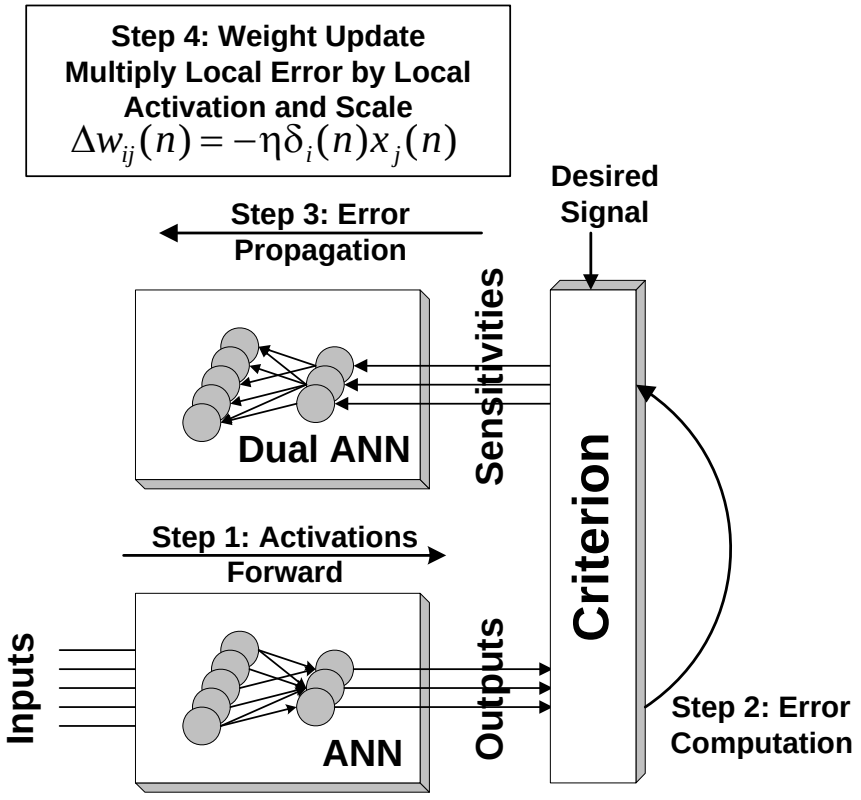


Figure 21. The algorithm for backpropagation

The beauty of this arrangement that we call the dataflow algorithm for backpropagation is

two fold: first, the *dataflow has been separated from the local computations* which brings flexibility for the simulations. Later chapters will demonstrate that this gradient descent methodology is also valid for training recurrent networks and networks through time, i.e. the data flow algorithm is a general implementation of gradient descent learning.

Second, the *local error is available as a signal in the dual topology*, which means that we do not need to write equations to compute the local error, the biggest problem when simulating arbitrary neural networks with backpropagation. Only the topology of the network needs to be specified by the user. The flow of errors through the dual network topology is doing the backpropagation computations for us, effortlessly. Implementing the backpropagation algorithm with the dual is much more versatile than coding directly the previous equations, since the dual network can be programmed very simply from the user's specified topology. NeuroSolutions uses the data flow algorithm presented in Figure 21. Next we will specify the operations required locally to implement the dataflow algorithm.

6.4. Specification of the local operations

We saw how the backpropagation algorithm can be implemented as a data flow machine. Analyzing Figure 21 we can also see that the *PE only enters in the choice of the local computation* (*f(.)* and its derivative in the dual). What this means in practice is that the *flow of signals through the network (i.e. the activations and the errors) can be decoupled from the operations of the PEs*. This provides a very powerful way to simulate neural networks, because different PEs can be interchanged at will in the topology simply by specifying new local operations that each building block must execute. So, one does not need to derive learning equations for each topology. Only two items need to be specified:

- the network topology (preferably in a graphical form) by numbering PEs in a left to right manner. A computer program does this easily and produces a graph of interconnections.
- the PE and the dual PE input output relations, which we call the *local maps* .

To be part of the dataflow machine the ith PE must be programmed to do basically two things when it is activated (fired) by the dataflow algorithm:

- propagate an activation forward (forward equation)

$$x_i = f\left(\sum_j w_{ij} x_j\right)$$ **Equation 40**

- propagate an error backward (backward equation)

$$\delta_i = f'(net_i)\sum_k w_{ik}\delta_k$$ **Equation 41**

Both Eq 40 and 41 are required because the ith PE is part of a larger network. So it not only needs xi, δi to update its weights (with the gradient descent rule), but also to pass xi forward to continue the chain, and also pass the δi backwards such that the dual PEs in the preceding layer can perform adaptation of their own parameters. Figure 22 shows the mechanics of the method.

Finally, the weight updates are computed from xi, δi with the gradient search method of choice, as specified in Chapter IV. Until now we only discussed the straight gradient descent, algorithm so this simply means $\Delta w_{ij} = x_j\delta_i$. But in general the update is a function *h*(.) of the activation and error

$$\Delta w_{ij} = h\left(x_j, \delta_i\right)$$ **Equation 42**

Different PE types (logistic, tanh, linear, quasi-linear, etc) simply require different *local maps* for the PE and its dual. This means that PEs can be effectively organized into families of components using an object-oriented programming approach as exploited in NeuroSolutions. Moreover, each PE can be associated with an icon that leads to a very effective graphical user interface to construct neural networks. Note that the forward PE and its dual share the same weight values wij.

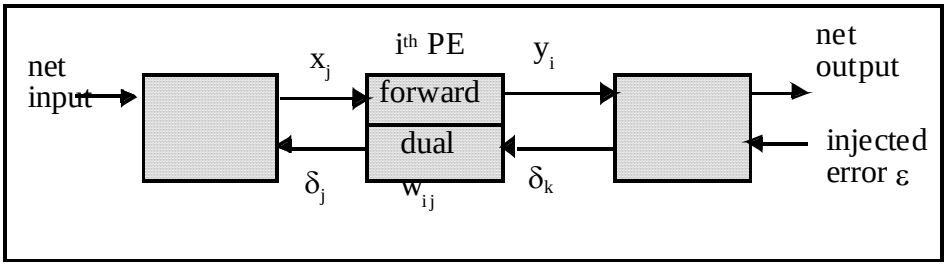
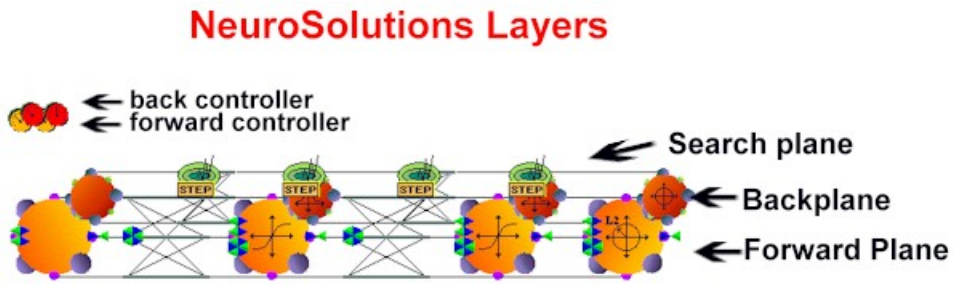


Figure 22. Communications among PEs in the topology

NeuroSolutions 21

3.21 Dataflow implementation of backpropagation

This explanation needs to be compared with the iconic representation of the breadboard that implements the MLP. Note that the breadboard has three layers: the large bubbles' layer (the forward network also called the forward plane) that is computing the forward activation (Eq.37); the smaller bubbles' layer (the dual network also called the backplane) that is computing local errors (Eq.41); and the gradient descent components (the search plane) that use the local activations and local sensitivities to compute the weight updates (Eq.42).



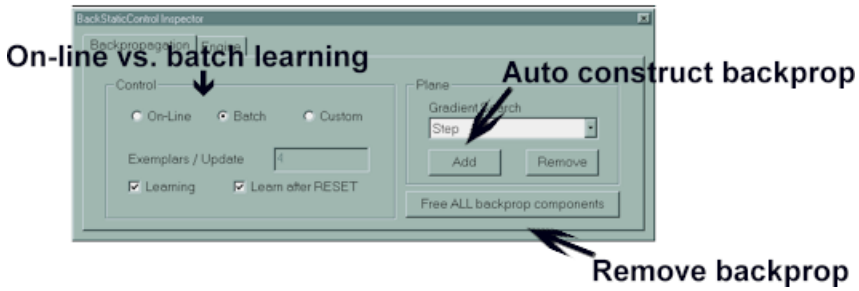
The data flow is implemented by the controllers that sit normally at the top left of the network. To implement backpropagation there are two controllers: the forward controller (that sends input data forward through the network), and the back controller (that sends sensitivities through the dual network). The alternation between these controllers implements the data flow necessary to update the weights using backpropagation. These controllers are crucial to decide what type

of backpropagation is implemented (static, fixed point learning or backpropagation through time) as we will see later.

The data flow still needs to be further specified (remember that backpropagation implicitly imposes a dataflow). One can send one sample at a time through the network, compute the corresponding local error and perform the adaptation. This is called the *on-line or pattern mode learning*. When the exemplars per epoch is set to 1, on-line is implemented. An alternative is to fire all the samples of the training set. Store locally all the PE activations. Compute the network outputs for every input. Compare and compute the errors at the network output for each input desired response pair, and send in sequence all the output errors through the dual network. Now we can compute the weight update for every sample, sum them up and only then change the weights. This is called the *batch learning mode*, where the weights are updated always with the information contained in the full training set. The static controller Inspector allows this choice if one specifies the number of training patterns in the exemplars/epoch field.

In order to demonstrate how easy it is to construct the dual network, the Backcontroller has a switch to automatically construct the backpropagation plane. If the "remove" button is pressed the backpropagation plane is gone. The network now only has the forward dynamics, so it can not learn. This is the way the network should be used for testing. Freeing the backplane is preferable to setting the learning rates to zero because it is more efficient (no sensitivities are ever calculated).

NeuroSolutions Backcontrol Inspector



Now if the "allocate" button is pressed, the backplane and the gradient plane are automatically constructed. The network will be able to learn again. Notice that with the arrangement of the simulations in planes, *the user only has to worry with the construction of the forward plane,* i.e. the network topology. Once this is done, NeuroSolutions has the information to automatically construct the learning dynamics. The biggest advantage of this arrangement, is that there is no constraint on the topology that the user can build. NeuroSolutions will always be able to compute the dual network, and train the weights with backpropagation. No equations are ever explicitly written.

NeuroSolutions Example

Go to Next section

7. Training embedded adaptive systems

Backpropagation can be used to compute gradients in ordered networks. We applied it here to MLPs. However, the algorithm can be applied to any system built from modules that are differentiable (not necessarily adaptive) and which can be ordered. This is indeed a very large class of systems which include macro-economical, life science, and engineering models. The beauty of the procedure is that we can propagate sensitivities up-and down the modules with the goal of finding optimal coefficients that meet a given external criterion.

This is to say that we *can mix adaptive and fixed parameters sub-systems* and seek with backpropagation an overall optimal operating point. This is particularly important when

we have a priori knowledge about the problem and we want to design sub-systems that include that knowledge but have other subsystems that are adaptive. Even if only some of the modules are adaptive, we can propagate sensitivities through the ones with fixed parameters, to optimally train the overall system. We will use this property later in control applications.

Here we would like to treat the case of Figure 23 where an adaptive sub-module of a system needs to be adapted given a desired external response. Notice that the adaptive module is internal, i.e. it is not in direct contact with neither the input nor the desired signal. Can we optimally set the parameters of this system? The answer is affirmative provided the sub-module 2 is differentiable and the overall network is ordered.

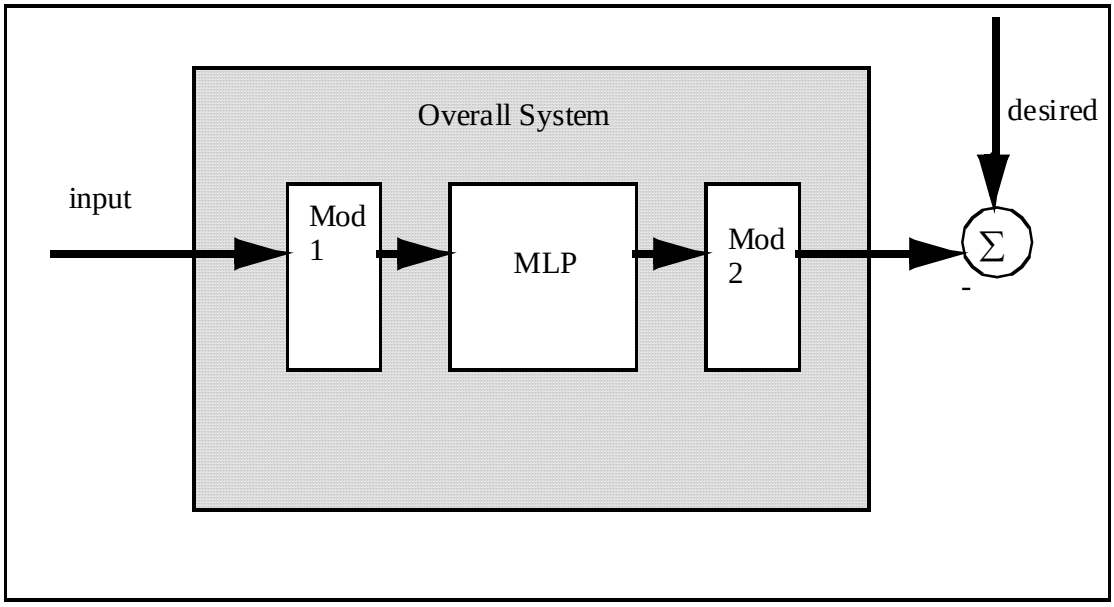


Figure 23. Adapting an embedded adaptive system.

Using our known tools of the chain rule and ordered derivatives, we can see that this adaptation problem can be easily solved taking into consideration that

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_{\text{mod } 2}} \frac{\partial y_{\text{mod } 2}}{\partial y_{MLP}} \frac{\partial y_{MLP}}{\partial w_{ij}}$$

$\leftarrow\!\!\rightarrow_{\varepsilon}$ $\leftarrow\!\!\rightarrow_{BP}$

Equation 43

The first term is the injected error, while the last is the term that backpropagation computes for us. So if we know the functional form of the input-output relation for module 2, we can calculate the effect of passing the injected error through module 2. We still have a little problem in computing the weight update using gradient descent since we have to know the input to the MLP, but this is no big problem if we known the input-output relation of module 1. Of course if the modules are themselves nonlinear they may complicate the training because they attenuate errors and activations, but this does not affect the applicability of the method.

This analysis shows that we can *adapt easily the parameters of an embedded adaptive* system in larger systems built from fixed coefficients and differentiable input-output relationships. *We can design the subsystem to optimize the overall system performance.* This is a very important aspect of backpropagation that is still today largely unexplored.

NeuroSolutions 22

3.22 Training embedded neural networks with backpropagation

In this example we are going to adapt an embedded adaptive systems (MLP) in a larger system built from a frontend fuzzy module and with an output that goes through a nonlinear subsystem before it is compared to the desired response.

We made up the example. Suppose that you are in a fair and you are testing your strength in one of those machines where you pound a lever with a sledgehammer and project a pellet up a column in the hope that it will ring a bell up in the column. We would like to train a system to predict the chances of a contestant ringing the bell, given their height and weight. But since we do not have a scale nor a measuring device, we simply would like to use the quantifiers of "light", "medium", "heavy" for the weight, and "short", "medium", "tall" for the height. We also think that the machine is totally unfair due to the disparity in strength of the general population. A much "fairer" system is one in which the bell height is positioned according to the past performance of people with the same height and

weight. This will be codified as a nonlinear function in the following way: first the output of the MLP is cubed. If this value is below 0.3 this becomes the output of our system. However, if the value is above 0.3 we take the square root and add 0.16. The desired response is obtained my modifying mathematically the height and weight data directly. These operations are shown in the diagram. Can we still train the MLP?

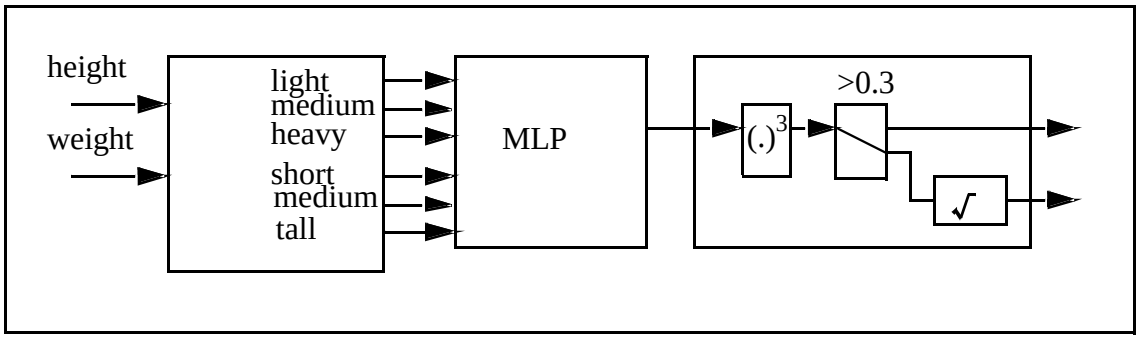


Figure 24. System block diagram

We will create the linguistic variables "light", "medium" etc. from the height and weight using fuzzifiers. The fuzzy module is built from sigmoid nonlinearities. The advantage of a fuzzy layer is that it includes in a very effective way the a priori knowledge that we may have about the task. Fuzzy sets are described by a membership function (MF), so the first problem is to devise a way to create membership functions in NeuroSolutions. This is not difficult because we can very easily create a MF by a cascade of one sigmoid layer with a linear layer with preset parameters (which may later be fine tuned through adaptation). Let us look at Figure 24.

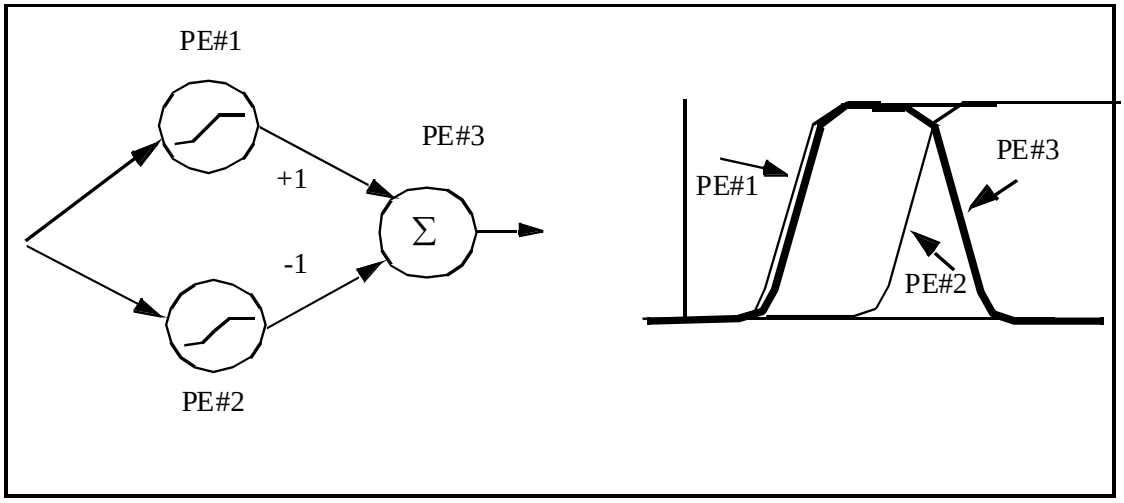


Figure 25. Creation of a fuzzy layer with sigmoids

Pairs of sigmoids combined with +1 and -1 will create a "Gaussian" like membership function (Fig. 24). The maximum is located at the average value of the bias. Several of these paired PEs with different biases will create a span of the input space with the linguistic variables. The knowledge from the tasks will set the bias and the slope of each sigmoid such that the required membership functions are constructed. Notice that there are no adaptable parameters in this layer. Open the MatrixEditors in the breadboard to see how we have created this layer. As usual you are free to modify any parameter.

The next submodule will be our one hidden layer MLP which will be followed by a second module which adjusts the height of the bell. Module 2 is here a function such as

$$y = \begin{cases} z^3 & z < 0.3 \\ \sqrt[2]{z^3} + 0.16 & z \geq 0.3 \end{cases}$$

where z is the output of the MLP. This function is coded in the DLL at the output of the MLP. Notice that the dual component has to implement the dual of the function. NeuroSolutions and its dataflow implementation of backpropagation is capable of training the MLP with the activations and sensitivities being passed

through the fuzzy layer and the output function. Run the system and verify that the MLP can still optimize the overall system.

NeuroSolutions Example

Feedforward neural networks are an example of ordered networks. But notice that the list of dependencies in Eq.34 is static in the sense that it does not depend upon time. When time dependencies are brought into the picture as will be done in later chapters during the study of recurrent networks, the ordered list of dependencies must be revisited to apply backpropagation. Another requirement to apply this methodology is to choose smooth nonlinearities for the PEs as was mentioned in several occasions. One aspect that should be stressed again is that backpropagation is much more computationally efficient procedure than direct differentiation to compute gradients in feedforward networks.

Go to next section

8. MLPs as optimal classifiers

Before concluding this chapter, let us go back to the framework of statistical pattern recognition, and ask the question: can the MLP in fact implement *optimal classifiers*? An optimal classifier must have the potential to create arbitrary discriminant functions that separate data clusters according to the *a posteriori* probability. Since we know that the MLP has known universal mapping capabilities, we can suspect that the MLP meets this requirement. An optimal classifier using the Bayes framework should produces outputs that are the a posteriori probability of the class given the data.

Does the MLP produce outputs that can be interpreted as probabilities? The answer to this question is yes if the training is done under certain conditions (Bishop). Moreover, we can also show that the *MLP is directly producing estimates of a posteriori probabilities*, unlike any of the classical methods of pattern recognition. Remember that in statistical pattern recognition we needed to use Bayes theorem to practically evaluate the *a*

posteriori probabilities. *With the MLP we obtain their estimates directly as outputs*, provided the training and the topology are appropriately specified. This is a departure from the well established statistical reasoning that applies Bayes rule to estimate a posteriori probabilities. Let us briefly cover the basic concepts of this theory.

Assume the learning machine is being trained to minimize the MSE.

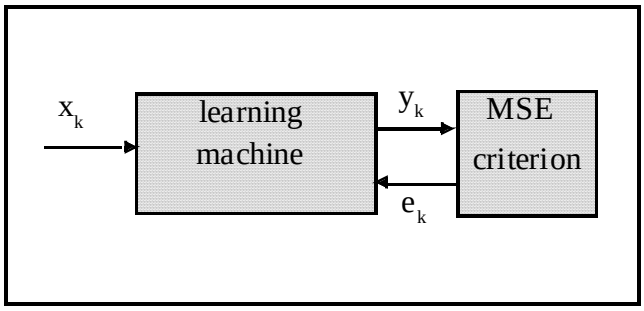


Figure 26**.** A learning machine whose outputs are estimates of *a posteriori* probabilities

We have to assume that the MLP has sufficient number of PEs to produce the required map from input space to targets. We also have to assume that the training data is sufficient, and that the training does indeed take the learning system to the global minimum. The final requirement is that the outputs are between 0 and 1 and that they all sum to one for every input pattern (so that each output can represent the probability that the input is in the specified class). In order to guarantee that the outputs sum to one, we can not utilize the logistic function PE. We must utilize a new output axon with the *softmax* activation function

$$y_k = \frac{\exp\left(net_k\right)}{\sum \exp\left(net_j\right)}$$

Equation 44

The softmax function is similar to the tanh and logisitc functions except that the outputs are scaled by the total of the activations in the output layer (so that the sum of the outputs sums to one). For the two class case the single output PE can be a logistic function since the probability requirements are still met.

Notice that we did not specify that the learning machine must be a MLP. The MLP is just an example of a viable and efficient implementation that produces this result since it is an universal mapper. The important aspect for this behavior is the minimization of the MSE.

The output yk of such a MLP can be shown to be an estimator for the average conditional probability of the target data given the inputs, i.e.

$$y_k(x, w^*) = \sum_i t_{i,k}\, p(t_{i,k}|x)$$

Equation 45

where w* is the optimal weight value and we are using *t* for the desired response Derivation of the conditional average . For a classification problem where the desired response is 1 and 0 and we assume c outputs (one output per class), it is easy to show that we have

$$y_k(x) = P(c_k|x)$$

Equation 46

What this equation says is that the output of the MLP is providing the *a posteriori* probability of the class given the data. If we recall from the pattern recognition section, the *a posteriori* probabilities minimize the classification error and are therefore the best one can hope for to build optimal classifiers. So one can conclude that if the learning machine is powerful enough and has been trained appropriately, its outputs (with the restrictions mentioned above) can be interpreted as *a posteriori* probabilities of the classes given the data. We now have a methodology to estimate *a posteriori* probabilities directly from the data, unlike when we use statistical pattern recognition, where Bayes rule is conventionally applied.

This is a very important result because it allows us to work *with the numerical outputs* of the network as *a posteriori* probabilities for a variety of applications in both pattern recognition and signal processing. Some of these applications are:

- Apply rejection thresholds for decision making.
- Implement minimum risk decisions in detection and diagnostic.
- Estimate observation probabilities in a variety of applications (Hidden Markov Models,

statistical signal processing).

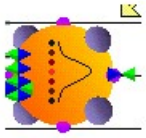
We will address these aspects further in later chapters.

NeuroSolutions 23

3.23 Softmax and a posteriori probabilities

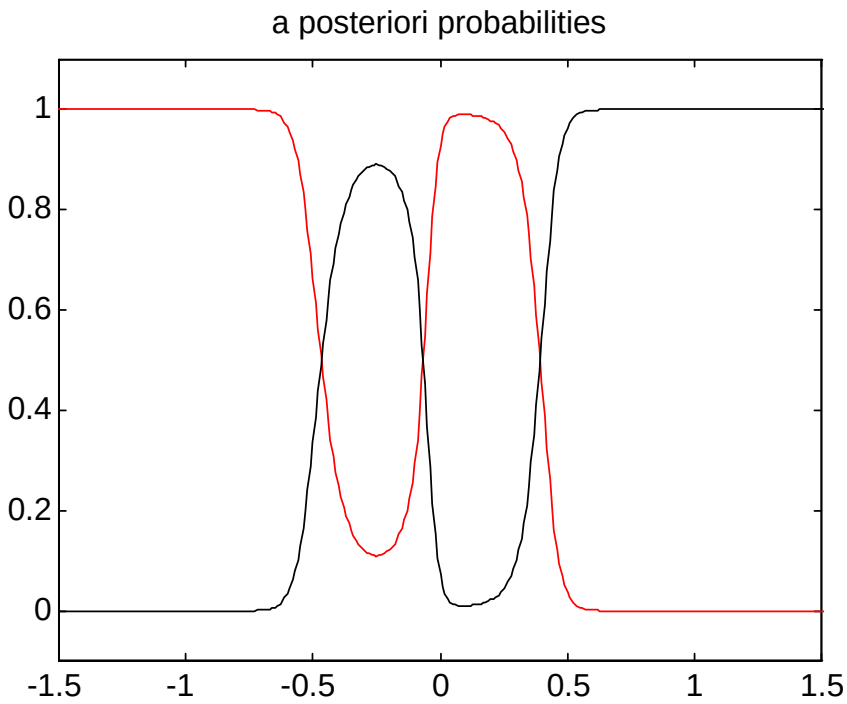
This example uses the probability considerations above to show how a MLP can be used to estimate the *a posteriori* probabilities for the healthy/sick example discussed previously. We will train a one-hidden layer MLP with two soft-max output axons. The soft-max output axons are similar to sigmoid axons except they are normalized so that the sum of the outputs is always one. The soft-max allows the outputs of the MLP to be considered as *a posteriori* estimates of the probability that the input exemplar belongs to each class. For example, if the output from PE 1 is 0.90 and the output from PE 2 is 0.10, then the probability that the subject is healthy is 90% and the probability that the subject is sick is 10%.

New NeuroSolutions Components



Softmax Axon

In order to check how well this really works we have to have quantitative data regarding the class distributions. So we created a two class problem where each class is composed by two Gaussians. The a posteriori probability is shown in the figure below.



Red is the a posteriori probability of one class while black is the a posteriori probability of the other class. We trained our MLP with a sofmax output in data created by each class. Run the network and observe how similar the discriminant functions of the MLP come to this plot. This is a more specify example of the important optimally property of MLPs. Change the number of PEs to test what the net does when it does not have enough degrees of freedom. Let it train for a long time and compare the final separation surface with the picture above. Overtraining will make the separation surface go to +-1, and will destroy the optimally in terms of statistical interpretation.

NeuroSolutions Example

Go to next section

9. Conclusions

In this chapter we covered one of the most important applications of neural networks -

pattern recognition. ANNs are semi-parametric classifiers since the discriminant functions are functions that belong to a given class. But we do not know a priori what is going to be the actual discriminants that will be employed by the ANN.

In this chapter we studied multilayer perceptrons (MLPs) which are feedforward topologies. The topology is what defines the functions that can be used for discriminants. The perceptron can only construct linear discriminant, but the two hidden layer MLP is an universal approximator, i.e. it can construct arbitrarily complex input-output mappings. MLPs are very efficient approximators in high dimensional spaces, so they can perform better than other classifiers.

MLPs were trained with a gradient descent procedure called backpropagation. Backpropagation is a very powerful and computational efficient algorithm. It is in fact a contribution of the field of neural networks for optimization theory. We have shown how the algorithm is derived using the chain rule, and we also covered the ordered derivative method that is more principled. In fact it tells us a lot about the types of topologies that can be trained with backpropagation. It also lead us to the data flow implementation which is the best possible way to implement backpropagation with a computer algorithm. Its great advantage is that it can be applied to arbitrary topologies as long as the computer can construct the dual (or transpose) network. This is trivial to do in ordered networks. Much of the power of NeuroSolutions is based on this innovation (which was first coded in 1991).

This chapter concentrated on the MLP principles, but little was said how to apply them to real world data. This is the purpose of the next chapter.

NeuroSolutions Examples

3.1 McCulloch and Pitts PE for classification

3.2 Discriminant probe to visualize the decision surface

3.3 Behavior of the sigmoid PEs

3.4 Classification as the control of the decision surface

3.5 Perceptron learning rule

3.6 Delta rule to adapt the MC-P PE

3.7 Comparing a linear and nonlinear PE for classification

3.8 Decision boundaries of the perceptron

3.9 The perceptron for character recognition

3.10 Perceptron and the XOR problem

3.11 One hidden layer MLP in 2-D space

3.12 Solving the XOR problem by hand with the one hidden layer MLP

3.13 Creating a **"**bump**"** with the one hidden layer MLP

3.14 Solving the XOR with backpropagation

3.15 Solving the bump with backpropagation

3.16 Effect on the number of hidden PEs

3.17 Fewer hidden PEs than required

3.18 Creating a Halloween mask by hand with a two hidden layer MLP

3.19 Comparison of the perceptron and MLPs in real data

3.20 Solving the bowtie with MLPs

3.21 Dataflow implementation of backpropagation

3.22 Training embedded neural networks with backpropagation

3.23 Softmax and a posteriori probabilities

Concept Maps for Chapter III

Chapter III


Multilayer Perceptrons 1
Training Embeddded Systems 7
Nonlinear Models Chapter V, X, XI
Discriminant function 2.2
Single PE Networks 2
Nonlinearities 2.1
MLPs as Optimal Classifiers 8
Single layer Networks (Perceptron) 3
Nonconvex Performance 2.5
Training
Adatron algorithm
Perceptron rule 2.3
Delta rule 3.1
Limitation of Perceptrons 3.2
Multilayer Networks
1 Hidden Layer 4
2 Hidden Layers 5
Universal Mappings 4.2
Discriminant Functions 4.1
Backpropagation Training 6
Discriminant Functions 5.1
Algorithm 4.3
Ordered Networks 6.1
Dataflow implementation 6.3
Training MLPs in Practice Chapter IV
Local Operators 6.4

Go to Next Chapter

Go to Table of Contents

separation surfaces of the sigmoid PEs

Thinking of the separation surfaces built by ANNs made of sigmoidal PEs as intersection of hyperplanes is sometimes a crude approximation. Sigmoidal PEs create ridge functions which are functions limited between 0 and 1 (or -1 and 1). However, the weights and the bias still control the location and orientation of the ridge. Around the ridge region, the function is approximately linear. So when the separation surface is built from the intersections of ridge functions very complex curves may result, which may be far from the piecewise linear decision surfaces of the perceptron built from threshold PEs. Since the network can control the separation surface by the size of the weights, if the problem requires curved separation surfaces instead of piecewise linear, the ANN can built them.

Return to Text

probabilistic interpretation of sigmoid outputs

According to Bayes rule, the posteriori probability can be written as eq2 . Note that the denominator can be written as $P(x) = p(x|C_1)P(C_1) + p(x|C_2)P(C_2)$ where C_1 and C_2 are the two classes C_1 and C_2. Now if the conditionals are Gaussians of equal variance it is not difficult to show that

$$P(C_1|x) = \frac{1}{1+\exp(-a)}$$

where

$$a = \ln \frac{p(x|C_1)P(C_1)}{p(x|C_2)P(C_2)}$$

Note that this is exactly the definition of the activation function of the sigmoid PE.

Return to text

vector interpretation of the separation surface

A vector interpretation of this result is very useful. Let us consider (1, 1) as the end point of a vector *v* drawn from the origin. The points of coordinates (x1, x2) in the equation can also be interpreted as the end points of another vector *g* (drawn from the origin) that exists on the line. In order to satisfy Eq.4 (assume that b=0 for simplicity) *v* and *g* have to be perpendicular since their dot product is zero. Hence, the linear decision surface $g(x_1,x_2)$ has to be perpendicular to the vector *v* which is called the normal.

Remember that the PE weights are the coefficients of the discriminant function. Therefore, the weights of the PE indicate in the input space the normal direction to the separation surface.

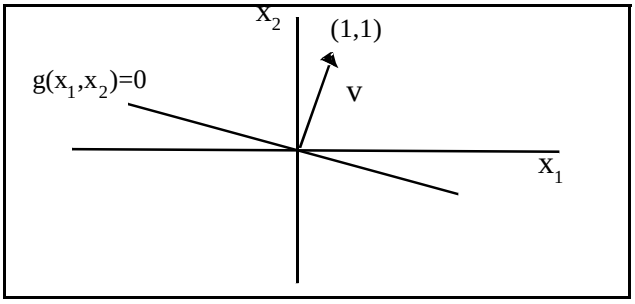


FIGURE 8. Decision surface and its normal

Return to text

perceptron learning algorithm

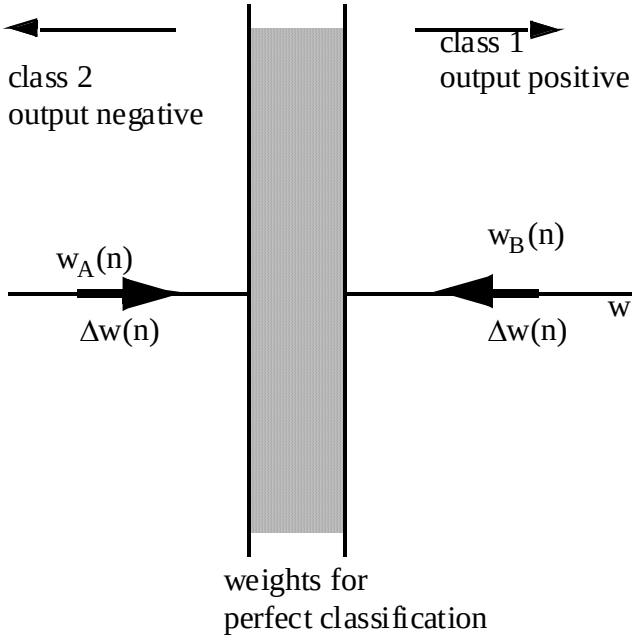
Let us study in more detail the perceptron learning algorithm. Notice that Eq.9 corrects the weights only under the condition that *y* is different from *d*. There are two cases to consider: either d=1 (class 1) and y=-1 in which case

$$w(n+1) = w(n) + 2\eta x(n)$$

or d=-1 (class 2) and y=1 in which case

$$w(n+1) = w(n) - 2\eta x(n)$$

Now let us see how these cases arise in practice. Let us consider the 1D case, i.e. just one weight as in the figure. In the first case (case A) $x_A(n)$ belongs to class 1 but the output is -1, so the present weight $w_A(n)$ must be in the left part of the figure (otherwise the output would be positive and would be no correction). In this case the update will add a value to the weight, moving it in the correct direction.



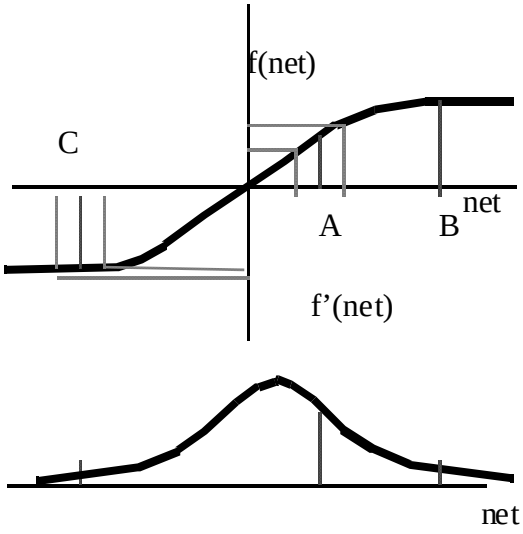
In the second case (case B), the input pattern $x_B(n)$ belongs to class 2, so if the perceptron is outputing a positive value, it is because the weight $w_B(n)$ is in the right part of the figure. In this case the weight is decreased according to the equation. So in both

cases, we see that the weight is moved to the shaded region that produces the correct classification. One can show mathematically that for linearly separable patterns this algorithm converges in a finite number of steps. Haykin

Return to text

error attenuation

Let us look at the Figure to understand what is happening. The top part of the figure shows the tanh nonlinearity and the bottom part its derivative. When the input variable *net* is in the linear region (point A), the derivative is close to 1 so the sensitivity Eq.13 is close to xi, the equivalent sensitivity of the linear PE. However, when the nonlinearity is operating close to saturation (case B or C), the derivative of the tanh at those points is close to zero. This corresponds effectively to an attenuation of the sensitivity when compared to the linear PE. What this means is that *y* became much less sensitive to a change in xi when the operating point is B or C.



So we see that nonlinear PEs have a dual role of saturating the activations for large values of the input and attenuating the sensitivities. These two factors bring stability to the learning process because learning becomes much less dependent upon outliers, i.e. points that deviate a lot from the mean. Moreover, the nonlinearity allows each PE in the

network to specialize in a portion of the input space (the weights associated with a given PE learn more when that PE is in the linear region).

Return to text

optimizing linear and nonlinear systems

You may know that our mathematical knowledge is very limited in computing solutions for nonlinear differential equations, so it may seem hopeless to attack the problem of optimizing a nonlinear system. Although it is true that we loose the ability to analytically solve for the minimum, one can still use iterative procedures to find it. This is one of the advantages of optimization techniques.

For the linear problem treated in Chapter I, the analytical solution can be applied and will always produce the best possible results. However, the same method can not be used for the M-P PE. Optimization (in the form of the gradient descent procedure) can be applied to problems that are beyond analytical solutions. As we are going to see, we can apply gradient descent as long as the function is smooth, which include many nonlinearities.

Return to text

derivation of LMS with the chain rule

Let us then re-derive the LMS algorithm for a linear PE using the chain rule. We want to compute the partial of the cost J with respect to w and equate it to zero to find the minimum. Now, *J is a function of the weights* through the network output yP, i.e.

$$J = \frac{1}{2}\sum_p \left(d_p - y_p\right)^2$$

and $$y_p = wx_p$$

so using the chain rule Eq.12 we get

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial y_p}\frac{\partial y_p}{\partial w} = -\left(d_p - y_p\right)x_p = -\varepsilon_p x_p$$

Using the gradient descent idea to update the weights as we did before,

$$\Delta w = -\eta\frac{\partial J}{\partial w} = \eta\varepsilon_P x_P$$

which is exactly the same result as obtained in the LMS algorithm.

Interpreting this equation with respect to the sensitivity concept, we see that the gradient measures the sensitivity. So LMS is updating the weights proportional to how much they affect the performance, i.e. proportional to their sensitivity. This makes perfect sense, since if the goal is to decrease the error, we should be modifying more the weights that impact the error the most.

Return to Text

derivation of sensitivity through nonlinearity

Let us write

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial net}\frac{\partial net}{\partial w_i}$$

and note that from Eq.1

$$\frac{\partial y}{\partial net} = f'(net)$$

and

$$\frac{\partial net}{\partial w_i} = \frac{\partial}{\partial w_i}(\sum_i w_i x_i) = 0+\ldots+\frac{\partial w_i x_i}{\partial w_i}+0+\ldots = x_i$$

so finally we have the result in the text Eq.13

Return to text

Why nonlinear PEs?

The answer to this question is related to better performance and new computing power. Nonlinear systems may provide for instance a better fit to data than linear regression. In terms of the performance surface, this means that the global minimum achieved by the nonlinear network is lower than the minimum of the linear network. More importantly, they may effect new types of computation such as classification, which can not be done well with linear systems. So there is a real need for nonlinear processing, in spite of the added difficulty of working with nonlinear structures.

It is instructive to use the last example and train a linear system and a nonlinear system with the same data to understand the difference between them. The M-P PE separates the two data clusters. What do you think is going to happen if we use a linear system to provide a desired response of ones and zeros?

Let us think in terms of least squares. If a set of points in 2D space is given and a desired response of ones and zeros is provided, what the linear system will do is regression between the input and the desired response. It will provide the best regression plane (this is a 2D input) between the 0 and 1 responses given the values of the input samples.

Return to Text

mapping capabilities of the 1 hidden layer MLP

The 1 hidden layer MLP with sigmoid PEs is an universal mapper, i.e. it can approximate arbitrary well any continuous decision region, provided the number of hidden layer PEs is large enough.

There has been many proofs of this statement by Cybenko , Gallant and White , Hornik et

al, many of them based on the Stone Weierstrass theorem . These proofs are difficult to follow and will be omitted here. They only tell about the mapping capabilities (existence proofs), they do not say how to get a MLP with those characteristics (constructive proofs), so they are useful to describe the power of the technology.

An interesting thing is that the activation function does not seem to be that important for the mapping capabilities, since the proofs use several of them (even discontinuous nonlinearities). The key aspect is the form of the approximation function which is an *embedding of functions*, i.e. the results of one function become the argument for the next, i.e. $y = f(f(.))$

Return to text

backpropagation derivation

In this derivation we will drop the dependence on the pattern for clarity. First assume that the kth PE is the only output PE in the net. With the machinery of the chain rule, let us write the gradient of the cost with respect to the weight as the product of the output error propagated to the PE (the gradient with respect to the PE state) times the sensitivity of the PE output with respect to the weight, i.e.

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial net_i}\frac{\partial}{\partial w_{ij}} net_i = -\delta_i x_j$$

Equation 47

where we substituted the definition of the local error given by Eq.21 . Notice that the sensitivity of the cost with respect to the weights is decomposed into the sensitivity of the cost with respect to the state yi times the sensitivity of the state with respect to the local weights. The state sensitivity $\delta J/\delta y$ can be computed by the chain rule, which yields from the figure

$$\frac{\partial J}{\partial y_i} = \frac{\partial J}{\partial y_k}\frac{\partial y_k}{\partial y_i} = \frac{\partial J}{\partial y_k}\frac{\partial y_k}{\partial net_k}\frac{\partial net_k}{\partial y_i}$$

Equation 48

Remember that the index k denotes a single PE on top (i.e. in the layer closer to the

output) of the ith PE. Now let us substitute these partial derivatives with network quantities.

$$\frac{\partial J}{\partial w_{ij}} = -e_k\, f'\left(net_k\right) w_{ki}\, f'\left(net_i\right) x_j$$

Equation 49

This expression computes the gradient of the cost with respect to the weight wij. We assumed a single output PE in this derivation. For completeness the case of multiple output PEs is treated next.

The idea is basically the same, the only difference is that now there are many output PEs (denoted by k) connected to the ith PE, each contributing additively to the gradient of the cost with respect to the PE state, i.e.

$$\frac{\partial J}{\partial y_i} = \left(\sum_k \frac{\partial J}{\partial y_k} \frac{\partial y_k}{\partial net_k} \frac{\partial}{\partial y_i} net_k \right) = \sum_k e_k\, f'\left(net_k\right) w_{ki}$$

Equation 50

Substituting back Eq. 47 in the original equation Eq.44 we get finally

$$\frac{\partial J}{\partial w_{ij}} = -x_j\, f'\left(net_i\right) \sum_k e_k\, f'\left(net_k\right) w_{ki}$$

Equation 51

According to the rules of gradient descent learning, one just changes the weights proportional to the negative of Eq. 48

$$w_{ij}(n+1) = w_{ij}(n) + \eta f'\left(net_i(n)\right) \left(\sum_k e_k(n)\, f'\left(net_k(n)\right) w_{ki}(n) \right) x_j(n)$$

Equation 52

The algorithm just presented is the *backpropagation algorithm*. The expression that we arrived at seems pretty daunting, with a summation and lots of indices. But in fact can be easily interpreted.

Computer algorithm

Let us assume a sample by sample presentation of data, i.e. {x(n), d(n)}. We also assume that the network weights have been initalized with small random values such that

the PEs work in their linear region.

Step 1: Present an input –desired response pair {x1,d1}

Step 2: Compute the outputs of every PE starting from the input layer (l=1) up to the output layer (l=L). We can formally write this step as

$$y_i^l(n) = f(net_i^l(n)) \qquad \text{and} \qquad net_i^l(n) = \sum_{j=1}^{p} w_{ij}^l y_j^{l-1}(n)$$

where f(.) is the nonlinearity, the superscript means layer *l=1,…L*, n is the iteration number, and wij is the weight that links th ith PE to jth PE. In the first layer $y_j^0 = x_j$. If the PEs have biases, $y_0^l = -1$. If the PE is the top layer (output PE) make $y_j^L = y_j$.

Step 3: Compute the injected error as $e_i(n) = d_i(n) - y_i(n)$

Step 4: Compute the local errors starting from the top layer until the first layer. In the top layer the error is (delta rule)

$$\delta_i^L(n) = e_i(n) f'(net_i^L(n))$$

In all the other layers

$$\delta_i^l(n) = f'(net_i^l(n)) \sum_k \delta_k^{l+1}(n) w_{ki}^{l+1}(n)$$

Once we have these local errors and the activations of step 2, every weight in the network can be updated according to Eq. 49. Note also that the weights to compute the errors and the activations are the **"**old weights**"**.

Step 5: Repeat this procedure for every input pattern, and for the number of iterations required for convergence. For best results the patterns should be randomized from presentation to presentation.

This summarizes the computer algorithm to implement backproapgation.

Return to Text

multilayer linear networks

Why did we not discuss multilayer networks with linear PEs? It turns out that from a point of view of input-output mappings, a multilayer network with linear PEs is equivalent to a no hidden layer network. So there is little interest of studying such networks. However, if we want to train one with gradient descent we will need to use the backpropagation algorithm.

Return to text

rederivation of backprop with ordered derivatives

With this theory we can re-derive the backpropagation procedure for any ordered topology in half a page. We will assume a network (with smooth nonlinearities) given by Eq.30 . No dependence on the iteration will be included for simplicity. The direct effect in Eq. 30 is computed as

$$\frac{\partial^d y_j}{\partial y_i} = f'\left(net_i\right) w_{ji} \qquad \frac{\partial^d J}{\partial y_i} = -\varepsilon_i$$

Equation 53

where εi is the external injected error. So we can re-write them as

$$\frac{\partial J}{\partial y_i} = -\varepsilon_i + \sum_{i<j} \frac{\partial J}{\partial y_j} f'\left(net_i\right) w_{ji}$$

Equation 54

Note that the partial of J with respect to the state is equal to the injected error εi for an output PE, and zero for the other cases. To compute the gradients with respect to the weights, we get again

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial^d J}{\partial w_{ij}} + \sum_k \frac{\partial J}{\partial y_k} \frac{\partial^d y_k}{\partial w_{ij}} = 0 + \frac{\partial J}{\partial y_i} f'\left(net_i\right) y_j$$

Equation 55

Note that the sum extended to all PEs reduces to one term k=i because we are computing the local contribution to the weight wij. With the assignments that we used

before,

$$-\varepsilon_i = \frac{\partial J}{\partial y_i} \quad \text{and} \quad \delta_i = \frac{\partial J}{\partial net_i} = e_i f'(net_i)$$ **Equation 56**

we can finally write

$$e_i = -\varepsilon_i + \sum_{j>i} e_j f'(net_i) w_{ji} = -\varepsilon_i + \sum_{j>i} \delta_j w_{ji}$$
$$\delta_i = e_i f'(net_i)$$ **Equation 57**

So the sensitivity of the cost can be written

$$\frac{\partial J}{\partial w_{ij}} = \delta_i y_j$$ **Equation 58**

We can recognize these two equations as the equations necessary to implement the backpropagation procedure (Eq.29 Eq.31) that we derived PE by PE in the text. But here they were derived in a much more compact form. The weight update using the gradient descent procedure is

$$w_{ij}(n+1) = w_{ij}(n) + \eta \frac{\partial J(n)}{\partial w_{ij}(n)}$$

Return to Text

artificial neural networks

ANNs for short, are adaptive, most often nonlinear distributed systems.

topology

defines the way the PEs are connected together

feedforward

are topologies with no recurrent connections

sigmoid

any smooth nonlinear function that is monotonically increasing and has an S shape.

F. Rosenblatt

invented the perceptron in 1962 considered by many as the first learning machine

sensitivity

is the partial derivative of a function with respect to one of its independent variables. It measures how much a small change of the variable will affect the functional value.

global minimum

is the minimum that achieves the smallest MSE.

nonconvex

a performance surface is nonconvex when it displays more than one minimum.

saddle point

is a point of zero curvature along at least one direction.

linearly separable patterns

patterns that can be perfectly classified by linear machines.

generalize

a machine generalizes when it produces the correct output for inputs that belong to the same class but that were not used for training.

local error

is the product of the error that reaches the PE times the derivative of the nonlinearity at the operating point. It is really the “effective error” that is used to correct the weights.

Minsky

Marvin Minsky is one of the fathers of artificial intelligence who is credited with the down fall of neural networks in the late 60’s. This may be exaggerated. In a very influential book called Perceptrons he poised the theory of perceptrons for predicate calculus where he showed their limitation versus the Turing machine.

multilayer perceptrons

are feedforward neural networks with one or more hidden layers, i.e. layers with nonlinear PEs that are not directly connected to the outside world.

bump

this is a loose terminology that indicates well the nature of the region being created. A bump is a function that has large values over a limited space extent, i.e. it is basically a local function. A multidimensional Gaussian is an example of a “bump”. Important theorems about function approximation with local functions exist.

backpropagation

is a training algorithm for multilayer perceptrons that extends the delta rule to hidden layer networks. It uses the methodology of gradient descent learning and solves the credit assignment problem by using the weight values in the topology.

inventors of backpropagation

Many researchers worked in the computation of sensitivity across a network. Werbos with the ordered derivative was probably the first. In the connectionist arena, the paper that really attracted general attention was the paper by Rumelhart, Hinton and Williams. Le Cun in France presented the same method at basically the same time.

ordered derivative

is the partial sensitivity in an ordered network, which is made up of the explicit and implicit (through the network) dependencies.

local maps

are the definitions of the PE function ad of its dual which specify how the activations are modified when they go through the topology, as well as the errors when they flow across the dual topology.

dataflow

the dataflow machine is the chaining of algorithmic operations that are necessary to implement the neural network function as well as its training. Since a neural network is distributed, data is sent into its input and transformed when it goes through the machine. Likewise for the errors that are used during training.

topology

is the particular interconnection of PEs in the neural network.

a posteriori probability

is the probability of an event after some measurements are made.

likelihood

is the probability density function of each event.

probability density function

intuitively, is the function that specifies the probability of a given event in an experiment. It is the limit of the histogram for arbitrary large number of trials. See the Appendix for a definition.

eq2

$$P(c_i|x) = \frac{p(x|c_i)P(c_i)}{P(x)}$$

adaline

stands for ADAptive LInear Element, and is the processing element proposed by Widrow that implements a weighted sum of inputs.

Eq.1

x belongs to c_i if $P(c_i|x) > P(c_j|x)$ for all j $\neq$ i

Eq.6

x***k*** belongs to c***i*** if g***i***(x***k***)>g***j***(x***k***) for all j≠i

Eq.8

$$g_i(x) = -1/2(x-\mu_i)^T \Sigma_i^{-1}(x-\mu_i) - d/2\log(2\pi) - 1/2\log|\Sigma_i| + \log P(c_i)$$

Eq.10

$$g_x = w_1x_1 + w_2x_2 + \ldots + w_Dx_D + b = \sum_{i=1} w_ix_i + b$$

convex

a surface is convex when any point in a line joining two points in the surface belongs to the surface.

Eq.9

$$g_{new}(x) = g_{healthy}(x) - g_{sick}(x)$$

Eq.12

$$f(net) = \begin{cases} 1 & for \quad net > 0 \\ -1 & for \quad net < 0 \end{cases}$$

Eq.13

$$y = \begin{cases} -1 & if \quad \sum_{j=1,2} w_j x_j + b < 0 \\ 1 & if \quad \sum_{j=1,2} w_j x_j + b > 0 \end{cases}$$

Eq.14

$$g(x_1, x_2) = w_1 x_1 + w_2 x_2 + b = 0$$

LMS

$$w(n+1) = w(n) + \eta x(n) \varepsilon(n)$$

Eq.7

Eq.21

$$w(n+1) = w(n) + \eta \varepsilon_p(n) x_p(n)$$

Eq.20

$$\begin{cases} \sum_i x_i(p) w_i^* > 0 \quad for \quad d_p = 1 \\ \sum_i x_i(p) w_i^* < 0 \quad for \quad d_p = -1 \end{cases}$$

Eq.11

$$y = f(net) = f\left(\sum_i w_i x + b\right)$$

Eq.23

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial net}\frac{\partial}{\partial w_i} net = f'(net)x_i$$

Eq.33

$$y = f\left(w_5 x_3 + w_6 x_4 + b_3\right) = f\left\{w_5\left[f\left(w_1 x_1 + w_2 x_2 + b_1\right)\right]\right.$$
$$\left. + w_6\left[f\left(w_3 x_1 + w_4 x_2 + b_2\right)\right] + b_3\right\} = f\left\{g_1 + g_2 + b_3\right\}$$

Eq.30

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_{ip}}\frac{\partial y_{ip}}{\partial net_{ip}}\frac{\partial}{\partial w_{ij}} net_{ip} = -\left(d_{ip} - y_{ip}\right) f'\left(net_{ip}\right) x_{jp} = -\varepsilon_{ip} f'\left(net_{ip}\right) x_{jp}$$

Eq.31

$$\delta_{ip} = \frac{\partial J}{\partial y_{ip}} f'\left(net_{ip}\right)$$

Eq.38

$$\delta_i(n) = f'\left(net_i(n)\right)\sum_k \delta_k w_{ki}(n)$$

Eq.26

$$w_i(n+1) = w_i(n) + \eta\varepsilon_p(n)x_{ip}(n)f'\left(net_p(n)\right)$$

Eq.36

$$\Delta w_{ij}(n) = \eta\delta_i(n)x_j(n)$$

Werbos

Paul Werbos proposed this method in his Ph.D. dissertation entitled **"**Beyond regression: new tools for prediction and analysis in the behavioral sciences**"**, Harvard, 1974.

Eq.29

$$J = \frac{1}{2N}\sum_p\sum_i \varepsilon_{pi}^2$$

Eq.41

$$L = \left[\left\{w_{ij}\right\}, y_1, \ldots y_N\right]$$

Eq.40

$$y_i = f\left(\sum_{i>j} w_{ij}y_j\right) + x_i$$

Eq.46

$$y_i = f\left(\sum_i w_{ij} x_j\right)$$

Eq.47

$$\delta_j = w_{ij}\delta_i = w_{ij} f'(net_i)\sum_k \delta_k$$

Eq.48

$$\Delta w_{ij} = s(x_j, \delta_i)$$

Widrow

Widrow also utilized the adaline for classification by including a nonlinearity after the linear PE. See Widrow and Hoff, **"**Adaptive switching circuits**"**, IRE WESCON Convention Record, 96-104, 1960.

Eq.22

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial f}\frac{\partial f}{\partial x}$$

Eq.a

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial net_i}\frac{\partial}{\partial w_{ij}} net_i = -\delta_i x_j$$

Eq.34

$$y = f\left(\sum f\left(\sum(\ \bullet\ )\right)\right)$$

Eq.19

$$w\ (n+1) = w\ (n) + \eta(d(n) - y(n))x\ (n)$$

McCulloch and Pitts

were a very influential duo of researchers (Warren McCulloch was a neurobiologist and Walter Pitts an engineer) who proposed in the 40's a model of the neuron as a system for logic computation. See Brain Theory Vol 1, Ed.. Shaw and Palm, World Scientific, 1988.

perceptron

is a multiple input multiple output pattern recognition machine made from one layer of McCulloch - Pitts PEs. See Rosenblatt F., Principles of NeuroDynamics, Spartan Books, 1962.

Eq.25

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial y_p}\frac{\partial y_p}{\partial net_p}\frac{\partial}{\partial w_i}net_p = -\left(d_p - y_p\right)f'\left(net_p\right)x_{ip} = -\varepsilon_p f'\left(net_p\right)x_{ip}$$

greedy

a greedy method in this setting is one that uses resources proportional to the size of the input.

tesselation

is a division of the space in convex regions that totally fill the space.

Eq.35

$$w_{ij}(n+1) = w_{ij}(n) + \eta f'\left(net_i(n)\right)\left(\sum_k e_k(n) f'\left(net_k(n)\right) w_{ki}(n)\right) x_j(n)$$

Eq.4

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad \sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x-\mu)^2$$

ordered list

is the list that summarizes the dependencies of the network topology. Each variable in the order list only depends on the variables to its left in the list. The weights appear first in the list.

Cybenko

Approximation by superposition of a sigmoid function, Mathematics for Control, Signals, and Systems, 2(4):303-314, 1989.

Gallant

On learning derivatives of an unknown function with MLPs, Neural Networks 5 (1), 129-138, 1992.

Hornik, Stinchcombe and White

MLPs are universal approximators, Neural Networks, 2:359-366, 1989.

Haykin

Neural Networks: a comprehensive foundation. MacMillan, 1994. pp108.

Bishop

Neural Networks for Pattern Recognition, Oxford, 1995.

connectionist

is normally taken as a synonym of neural networks. The name comes from the fact that neural networks are highly distributed systems where the computation is done in the links (connections among PEs)

state variables

A state variable represents an internal quantity in the system that changes during operation. It may either be the output of the PE or the weight. Here we will be using it to represent the output of the PE.

Derivation of the conditional average

This result can be demonstrated (see Bishop for a full treatment) if we write the MSE for the case of large number of patterns as an integral

$$J = \frac{1}{2}\sum_k \iint [y_k(x,w) - t_k]^2 \, p(t_k, x) dt_k dx$$

For clarity we denote the desired response by *t*. Note that the index k sums over the targets, and the sum over the data exemplars was transformed in the integral, which has to be written as a function of the joint probability of the desired response and the input. This joint probability can be factored in the product of the input pdf p(x) and the conditional of the target data given the input p(tk|x).

The square can be written

$$\left(y_k(x,w)-t_k\right)^2=\left(y_k(x,w)-<<t_k\mid x>>+<<t_k\mid x>>-t_k\right)^2$$

where <<tk|x>> is the conditional average given by $<<t\mid x>>=\int t_k\,p(t_k\mid x)dt_k$. We can write further

$$\left(y_k(x,w)-t_k\right)^2=\left(y_k(x,w)-<t_k\mid x>\right)^2+2(y_k(x,w)-<t_k\mid x>)(<t_k\mid x>-t_k)+(<d_k\mid x>-t_k)^2$$

Now if we substitute back into the MSE equation we obtain

$$J=\frac{1}{2}\sum_k\int\left(y_k(x,w)-<<t_k\mid x>>\right)^2p(x)dx+\frac{1}{2}\sum_k\int(<<t_k^2\mid x>>-<<t_k\mid x>>^2)\,p(x)dx$$

The second term of this expression is independent of the network, so will not change during training. The minimum of the first term is obtained when the weights produce

$$y_k(x,w^*)=<<t_k\mid x>>$$

since the integrand is always positive. This is the result presented in the text.

Return to Text

Vladimir Vapnik

The nature of statistical Learning theory, Springer Verlag, 1995.

pp 128

Adatron

Anlauf J., and Biehl M., The adatron: an adaptive perceptron algorithm, Europhysics Letters, 10(7), 687-692, 1989.

Index