Table of Contents

Chapter IV - Designing and Training MLPs

Version 3.0

This Chapter is Part of:

Neural and Adaptive Systems: Fundamentals Through Simulation© by

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

In this Chapter, we will address the more practical aspects of using MLPs, which include:

- Search strategies to find the minimum
- Alternate cost functions
- Control of generalization (topologies)

After presenting these aspects from a practical point of view, real world problems will be solved with the MLP topology.

Go to next section

2. Controlling Learning in Practice

Learning (or adaptation) is a crucial step in neural network technology. Learning is the procedure to extract the required information from the input data (with the help of the desired response in the supervised case). If learning is incomplete the weight values will not be near their optimal values and performance will suffer. As we have seen in Chapter I and III, the good news is that there are systematic procedures (learning algorithms) to search the performance surface. The bad news is that the search has to be controlled heuristically.

The user directly affects the search through:

- the selection of the initial weights
- the learning rates
- the search algorithms, and
- the stop criterion

One has to understand the issues affecting each one of these topics to effectively train neural networks. One should also remember that the ultimate performance is also dependent upon the amount and quality of the data set utilized to train the system.

A large portion of this chapter is devoted to extend the basic gradient descent learning developed in Chapter III, so we will concentrate on the aspects that can be improved. But it is good to remember up-front that straight gradient descent learning and its different implementations (LMS, delta rule, backpropagation) are one of the most widely utilized methods to train adaptive systems because they are an excellent compromise of simplicity, efficiency and power. So while the tone of the chapter may seem negative towards gradient descent learning, this is just derived from the exposition goal and the reader should balance the impression with the amazing power of the technique displayed already in Chapters I and III. algorithm locality and distributed systems

Before we develop a methodology to appropriately set the learning parameters, let's see how we can visualize what is happening inside the neural network during training and

describe some of the features of learning.

2.1 Visualizing Learning in a Neural Network

We will use the breadboard for the XOR, initially implemented in NeuroSolutions with tanh nonlinearities. The point is that learning is much richer than what can be imaged from the learning curve (the thermometer of learning as we call it in Chapter I). All the internal parameters of the network are being changed simultaneously according to the activation flowing in the network, the errors flowing in the dual network, and the particular search algorithm utilized to update the weights.

Since the set up of the learning parameters is problem dependent, the user has to make *decisions that are particular to the problem being solved*. The only way to make appropriate judgments when a theory is lacking is to understand, through experimentation, the principles at work. Hence, it is very instructive to visualize the behavior of the network parameters during learning, and we can do this effortlessly with NeuroSolutions.

NeuroSolutions 1

4.1. Visualization of Learning

In this example we will use the XOR network from Chapter 2 and place scopes on the weights and backpropagated errors. By viewing the errors, weights, decision surface and learning curve we will get a much better feel for what is going on in the network. Compare the evolution of the weights with the backpropagated errors. Also, compare the location of the decision surface with the actual value of the weights. Finally compare all of this activity with the learning curve, the external variable that we normally observe. Do this several times. Try to understand the relationships among the different pieces. Ultimately, everything is dependent upon the input data and the errors.

Notice that the time evolution of the weights differs every time we run the network, but the final MSE is almost the same from run to run. Every run will also produce a

different set of weights. Learning in a neural network is a very rich process and that the learning curve can only give a glimpse of these activities. Nonetheless it is a valuable tool for gauging the progress in the network.

NeuroSolutions Example

2.2. Network weights and minimum MSE

It is important to understand why the adaptation of the same topology with the same training data produces so many different sets of final weights. There are three basic reasons for this fact.

- First, there are many *symmetries in the input-output mapping* created by the MLP. Thus, two networks which produce the exact same results may have different weights. For instance, as we discussed in Chapter III, the position of the discriminant function is determined by the ratio of the weights, not their values. Also, changing the sign of the output weight of a PE will compensate for input weights with a reversed sign.
- Secondly, there is no guarantee in general that the problem has a single solution. In particular, when non-minimum topologies are utilized, the redundancies may create many possible solutions. Remember that the minimization of the output error is an external constraint. Nothing is said about the uniqueness of the weight values to provide a given output error. In fact, from the point of view of the problem formulation as long as the output error is minimized, any solution is as good as any other. system identification versus modeling
- Thirdly, the final weights are obtained in an *iterated* fashion, from a *random initial* condition. Even when we stop the adaptation at a fixed iteration number in two different training runs over the same data, the random initial weights will create different weight tracks during adaptation. Therefore, the final weight values will most likely be different.

The size of the topology will often magnify these differences and produce very different final weights from run to run. Additionally, if the topology is not minimal, there will be redundant discriminant functions, and as such there are many possible solutions for mapping the training data. Each one, however, may perform quite differently on data the network has not seen yet (test set). This aspect will be addressed later.

This analysis points out one important methodological issue. *Learning is a stochastic process* that depends not only on the learning parameters but also on the initial conditions. So, if one wants to compare network convergence times (i.e. how much faster one update rule is with respect to another) or final MSE error after a number of iterations, *it is pointless to run the network only once*. One needs to run each network several times,

with random initial conditions and pick the best or use some other strategy (such as committees).

When the goal is to compare different training algorithms, it is common practice to average out the results, i.e. to present the "mean" learning curve across the different trials. This means that learning curves should be presented also with "error bars" or at least with a percentage of the number of times the minimum was reached.

NeuroSolutions 2

4.2. Learning as a stochastic process (XOR)

Remember that adaptation is a stochastic process - depending on the initial conditions and other factors, the path that the network will take down the performance surface will be very different. There are many possible endpoints (local minimum, global minimum, saddle points, etc.) to the adaptation process and even more trajectories to get there. It is important to remember that if we are to compare one learning algorithm to another, you must average the comparison criteria over multiple runs. For example, the learning curve should always be presented as an average of the individual learning curves over many runs. In this example we show the many possible trajectories and endpoints for the XOR problem. We use a custom DLL to compute the average learning curve.

NeuroSolutions Example

2.3. Control of the step size during learning

We have already encountered the problem of step size selection when we studied the linear regression and the MLP adaptation. In the linear case we can summarize the discussion by saying that the learning rate is a trade-off between speed of adaptation and accuracy in the final weight values. In nonlinear topologies such as the MLP, we have the same basic phenomenon but the problem is compounded by the nonconvex nature of the performance surface, as we discussed in Chapter III.

It is important to realize that for quadratic performance surfaces there are ways of

selecting optimally at each iteration the stepsize through a line search. However, normally we use a trial and error approach due to the computational complexity of determining the best stepsize at each iteration (line search methods). The determination of the best stepsize for the MLP does not have an analytic solution anymore, so this approach is even less interesting.

Usually, the solution of practical classification problems requires large values for some weights, because the PEs have to saturate to approach the desired response of 0 (-1) and +1. The only way that the weights can grow is through cumulative changes during the training process. If the learning rates are small, it will take a long time to obtain weights that provide small errors. On the other hand, if the learning rates are too high, then instabilities will result. As we saw in Chapter 1, even for convergent learning dynamics, when high learning rates are applied the final values are not very accurate since the solution will "rattle" around the global minimum.

In a nonlinear network (e.g. MLP), the stepsize selection is even more important. The new situation is the existence of local minima and saddle points that may stall learning. We will discuss ways to manage this problem with more powerful search procedures later in the chapter.

The goal for the stepsize selection is to use a large learning rate in the beginning of training to decrease the time spent in the search phase of learning, and then decrease the learning rate to obtain good accuracy for the final weight values in the tunning phase. This is sometimes called learning rate scheduling or annealing . This simple idea can be implemented with a variable step size controlled by

$$\eta(n) = \frac{\eta_0}{1 + \frac{n}{n_0}}$$

Equation 1

where η_o is the initial step size, and n_0 is an iteration count. Note that for $n<<n_o$, the step size is practically equal to η_o, while when $n>>n_o$ it approaches zero geometrically. The values of η_o and n_o need to be experimentally found. Alternatively, one can schedule the step size linearly, or logarithmically as we did in Chapter I.

If the initial value of η_o is set too high, learning may diverge. The selection of n_o is tricky because it depends a lot on the performance surface. If n_o is too small, the search phase may be too short and learning can stall. If n_o is too large, then we spend too much time in the search phase, rattling around near the global minimum before we fine tune our solution with lower learning rates.

In non-convex surfaces, the annealing schedule has the added advantage of enabling the search to escape from local minima when they are encountered early in the search. In fact, with a large learning rate, the search will bounce out of local minima and when the learning rate decreases, the global minimum can be reached with accuracy. The problem is that we do not know *a priori* what is the best schedule rate, so the selection of the learning constants in Eq. 1 is problem dependent. The following example illustrates how the learning rate affects performance, and how to schedule learning rates during adaptation.

NeuroSolutions 3

4.3. Learning rate scheduling

In this example we will show how to anneal the learning rate (change the step size over time during the simulation). We start with the XOR problem and add scheduling components to the gradient descent layer. There are three available scheduling components in NeuroSolutions, the linear, exponential, and logarithmic schedulers. Each one varies the parameter over time in a slightly different manner.

Make sure you use the randomize button and play with the simulation. It is important that you change the learning parameters to make the network learn as fast as it can this problem. You should also notice that from time to time the learning will get stuck at MSE of 0.5, and will take a long time to get out of this mode. The weights will remain practically constant, and the error will not decrease. This may be due to either a region of very low gradient (flat spot) or a local minimum. Here it is a flat spot.

NeuroSolutions Example

2.4. Setting the learning rates across the network PEs

The neurocomputing literature (Haykin) suggests that the *goal for robust and fast convergence is to have all the adaptive network parameters learn at the same rate*. This is easy to accomplish in linear networks (same step size for all weights), but it is not so easy for MLPs, since in nonlinear systems the error is attenuated by the derivative of the PE nonlinearity evaluated at the operating point (see backpropagation in Chapter III). It is therefore essential to understand how the error flows inside the network to properly set the learning rates. *The rule of thumb is to increase the learning rate from the output layer to the input layer by a factor of 2-5 from layer to layer*. In the following example we will observe the squelching effect of the nonlinearity from layer to layer.

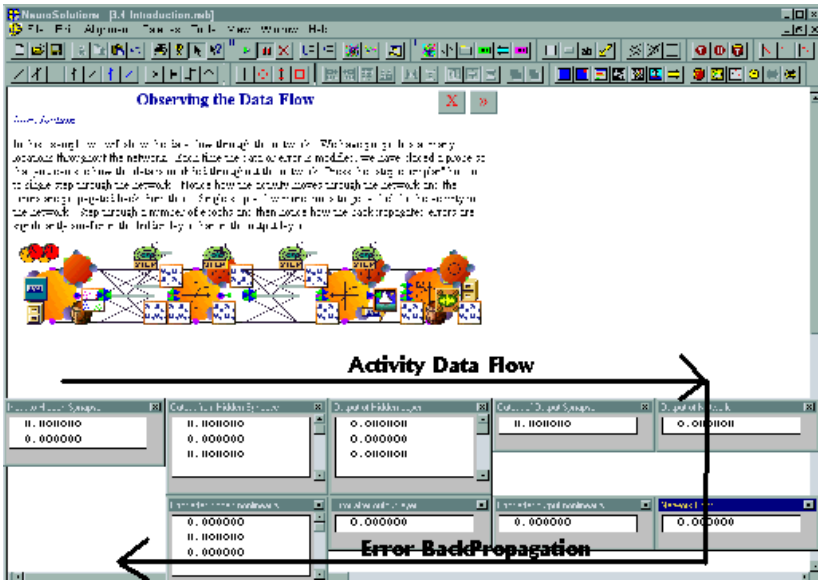
NeuroSolutions 4

4.4. Flow of errors across MLP layers

In this example we will again delve into the inner workings of the neural network. The problem to be solved is the star problem, a two class problem with a set of 4 samples per class which are placed on the vertices of two stars rotated by 45 degrees (and one smaller).

We will place matrix viewers at nearly every access point in the MLP and single step through the training. By doing this we gain many insights. We can watch the data flow through the network and understand exactly how each component of NeuroSolutions fits into the big picture. It is OK to gloss over the details of exactly

how the network operates for a while, but eventually it is important to understand the details. Notice that the data flows forward through the network and than the error flows backwards from the criterion back to the input layer. You should study these outputs in relation to the equations we derived for the MLP in chapter 2. The figure below shows how the data flows through the arrangement of viewers in the NeuroSolutions Example:



An important insight you should get is that the magnitude of the errors flowing back through the network shrinks through each layer. This is due to the multiplication by the derivative of the saturating nonlinearity that is a bell shaped curve around zero. This means that large magnitude (negative or positive) errors are multiplied by values close to zero, so they are attenuated. This means that if we set the learning rates constant across the net, the network learns faster in the layers closer to the output than deeper in the net (closer to the input). Hence to equalize training in each layer we should increase the learning rates of the components closer to the input. If we run the network with equal learning rates, and with unequal learning rates (double the step size of the gradient search over the first synapse) there will be a marked difference.

NeuroSolutions Example

2.5 Nonlinear PEs as a source of internal competition

The MLP ability to learn to discriminate patterns in the input space is linked to the attenuation of the error through the nonlinearity coupled with the saturation of the PE. The PE nonlinearity works as an internal competition mechanism which allows different PEs to specialize in different areas of the input space. In fact, recall that the weight update is a product of the local activity, the error, and the derivative of the nonlinearity *f'(.),* which is a bell shaped curve for sigmoid nonlinearities (Figure 1). So the question is the following: given several PEs in the hidden layer with different operating points (different values of *net*) are they updated equally by the presentation of a given pattern?

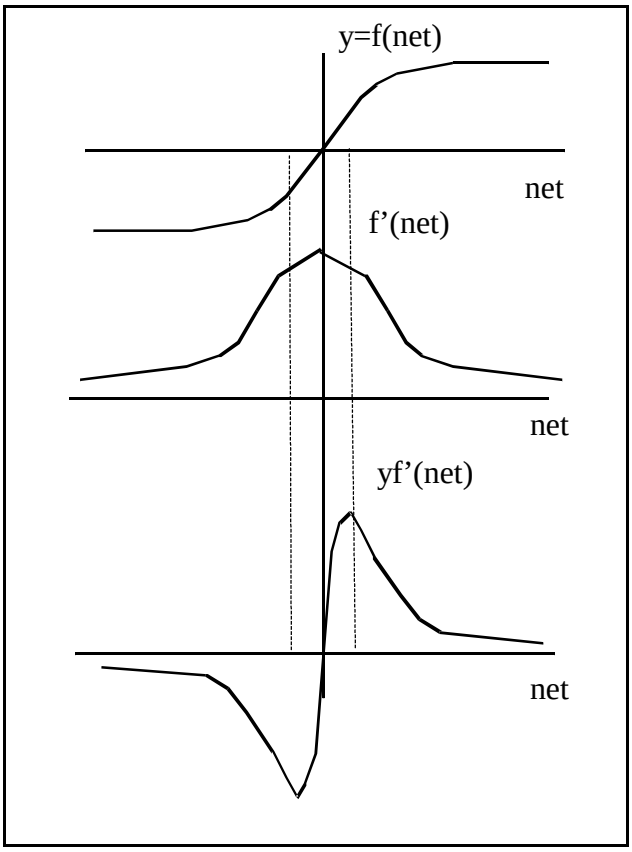


Figure 1. The derivative of the nonlinearity

The weights connected to PEs that for a particular input are operating in the first or third tier of their linear regions will be adjusted the most (assuming that they receive a constant error from the top layer). Effectively, these are the operating points with the

highest product activation x $f'_{\max}(net)$ (Figure 1). So during learning, different PE will tune to different areas of the input space. A ballpark analysis shows that sigmoid PEs are most sensitive to samples that make net ~ +/- 0.707. For normalized inputs these values correspond to approximately 45 degrees from their weight vectors.

If one of the PEs is saturated (*net* is very large in absolute value), the weights connected to it will be multiplied by a very small value of *f'(net*), so the weights will not change much. On the other hand if the PE is operating near *net=0* its output will also be small, so the weights leaving the PE will likewise have a small increment. This diverse rate of updating of the different weights due to the nonlinearity is a *source of internal competition* that tends to assign some PEs to one pattern cluster and other PEs to a different pattern cluster. Since the PEs in a MLP create the discriminant functions, this is the source of the power of the nonlinear systems for classification, and what differentiates them from linear systems. If the PEs were linear (or nonlinear but non-saturating), there would not be any internal competition *and the weights associated with each PE would tend to have the same value* (remember what happens in the adaline). PEs would never specialize and the network would not be able to respond sharply with high values for some patterns and low values to other patterns.

However, the attenuation of the error across a layer also imposes some constraints on the learning process. Here we will discuss:

- the choice of the number of layers
- the weight initialization.

For the sake of training efficiency, we should not create topologies with many layers, since the layers closer to the input will train very slowly, if at all. One should *always start to solve a problem with shallow networks* (i.e. a perceptron) for quicker training. If the perceptron does not provide a reasonable error, then try the one hidden layer MLP, and finally the two hidden layer MLPs should be tried. Since two hidden layer MLPs are universal mappers, more than two nonlinear hidden layers are rarely recommended.

Another variable that the user can set before starting a training run is the initial weight values. The initial weights affect the learning performance of the network, because an initialization far away from the final weights will increase the training time, but also because we would like that all the PEs in the network would learn at the same speed.

good initial weight values

In order to break possible symmetries that could stall learning (i.e. the degenerate solution where all weights are zero in the XOR problem), or saturate the PEs, it is common practice to start the network weights with *random initial conditions*. The random initial condition is implemented with a random number generator that provides a random value.

As we discussed earlier, a PE which is in its linear region learns faster than one that is in the saturated region. *For better training performance the goal is to have each PE learn at approximately the same rate*. If we set the variance of the random initial conditions based on the fan-in of the PE (i.e. the number of inputs it receives), each PE will be in its linear region, so all will learn at the "same" rate. For the tanh nonlinearity, a rule of thumb is to set the variance of the random number generator that assigns the initial weights at

$$\left(\frac{-2.4}{I}, \frac{2.4}{I}\right) \qquad \textbf{Equation 2}$$

where I is the fan-in of the PE.

NeuroSolutions 5

4.5. Effect of initial conditions on adaptation

In this example we will show how the initial conditions dramatically affect the adaptation of the weights in the XOR problem. When we set the initial conditions to the values discussed above, we get significantly better performance and better learning curves.

NeuroSolutions Example

Go to next section

3. Other Search Procedures

The popularity of gradient descent is more based on its *simplicity* (can be computed locally with two multiplications and one addition per weight) than on its search power. There are many other search procedures more powerful than backpropagation. We already discussed in Chapter I Newton's method which is a second order method because it uses the information on the curvature to adapt the weights. However Newton's method is much more costly to implement in terms of number of operations and nonlocality of information, so it has been used little in neurocomputing. Although more powerful, Newton's method is still a local search method, and so may be caught in local minima, and diverge sometimes due to the difficult neural network performance landscapes. Simulated Annealing or genetic algorithms (GA) are global search procedures, i.e. they can avoid local minima. The issue is that they are more costly to implement in a distributed system as a neural network, either because they are inherently slow or they require nonlocal quantities. Global Optimization is beyond the scope of this book, and the interested reader is directed to (Horst)

Here we will cover improvements to the basic gradient descent learning

$$\Delta w_{ij}(n) = -\eta \nabla J(w_{ij})$$

which is generally called a first order method . Recall that the LMS algorithm, delta rule and backpropgation use weight updates that are all particular implementations of this basic concept. They use a sample (noisy) estimate of the gradient that essentially multiplies the local error by the local activation

$$\Delta w_{ij}(n) = \eta \delta_i(n) x_j(n) \qquad \textbf{Equation 3}$$

We will improve Eq. 3 to cope with the added difficulty of searching nonconvex performance and surfaces that may have flat regions. It is obvious that a gradient-based algorithm will be trapped in any local minima since the search is based on local gradient information only (see Chapter III). In terms of local curvature, a local minimum and the

global minimum are identical, so the gradient descent will be trapped in any local concavity of the performance surface. The gradient descent method will move very slowly, if at all (called *stalling*), when the search traverses a flat region of the performance surface because the weights are modified proportional to the gradient. If the gradient is small the weight updates will be small (for a constant step size), so many iterations are needed to overcome the flat spot (or saddle point). This situation is easily confused with the end of adaptation when the search reaches the global minimum of the performance surface.

Not everything is bad in gradient descent. Fortunately, the gradient estimate in backpropagation is noisy, so adaptation has a chance to escape shallow local minima and to pass through flat spots using momentum learning. Conversely, noisy gradients make the adaptation slower (the weights are not always moving towards the minimum error). These arguments point to an appropriate control of the step sizes or learning rates and to seek improved search methodologies. In this section, we will introduce a few methods that help alleviate these problems of gradient descent search (local minimum and saddle points).

3.1. Momentum Learning

Momentum learning is an improvement to the straight gradient descent search in the sense that a memory term (the past increment to the weight) is utilized *to speed up and stabilize convergence*. In momentum learning the equation to update the weights becomes

$$w_{ij}(n+1) = w_{ij}(n) + \eta\delta_i(n)x_j(n) + \alpha\left(w_{ij}(n) - w_j(n-1)\right) \qquad \textbf{Equation 4}$$

where α is the momentum constant. Normally α should be set between 0.5 and 0.9. The reason this is called momentum learning is due to the form of the last term that resembles the momentum in mechanics. Note that the weights are changed proportionally to how much they were updated in the last iteration. So if the search is

going down the hill and finds a flat region, the weights will still be changed not because of the gradient (which is practically zero in a flat spot) but due to the rate of change in the weights. Likewise in a narrow valley, where the gradient tends to ping-pong between hillsides, the momentum stabilizes the search because it tends to make the weights follow a smooth path. The figure below summarizes the advantage of momentum learning. Imagine a ball (weight vector position) rolling down a hill (performance surface). If the ball reaches a small flat part of the hill it will continue past this local minimum because of its momentum. A ball without momentum, however, will get stuck in this valley. *Momentum learning is a robust method to speed up learning and we recommend it as the default search rule.*

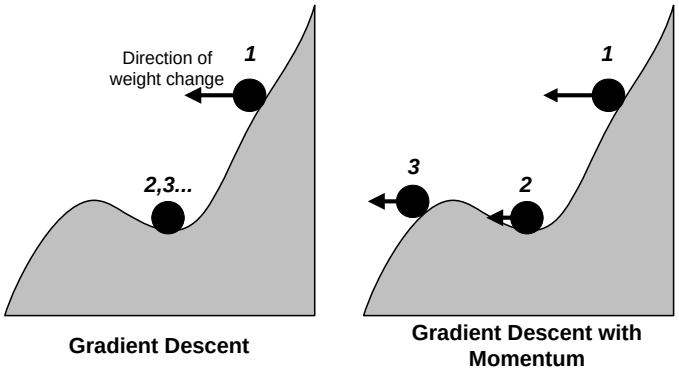


Figure 2. Why momentum learning helps

NeuroSolutions 6

4.6. Momentum Learning

This example compares the speed of adaptation of the straight gradient search procedure with momentum learning. Since there are many local minima in this problem, the momentum learning works much better since it helps us "roll" through the flat parts of the performance surface. The problem is already our well known STAR problem. Notice that with momentum learning the speed of adaptation is much higher for the same learning rate.

NeuroSolutions Example

3.2. Adaptive Stepsizes

Simplicity is the only reason to utilize the same stepsize for each weight and during the entire training phase. In fact, we know from Chapter I that the stepsize should be determined according to the eigenvalue for that particular direction. The problem is that eigenvalues are not normally known, so it is impractical to set them by hand. However, we have the feeling that observing the behavior of the error and/or weight tracks should allow a better control of the training. In fact, when the learning curve is flat, the step size should be increased to speed up learning. On the other hand, when the learning curve oscillates up and down the step size should be decreased. In the extreme, the error can go steadily up, showing that the learning is unstable. At this point the network should be reset. We can automate this reasoning in an algorithm by adapting the step sizes independently for each weight and through the training phase.

The idea is very simple: when consecutive weight updates produce the same error sign, the learning rate is too slow. Conversely, when the error sign is toggling from iteration to iteration, the step size is too large. These simple principles can be put into a rule called the *adaptive step size method* that adapts *each* step size continuously during training. If the following rules are applied to a single stepsize then there is no benefit for adaptation speed. Let us denote the learning rate for the weight w_{ij} as η_{ij}. The update to each step size is

$$\Delta\eta_{ij}(n+1) = \begin{cases} k & if \quad S_{ij}(n-1)D_{ij}(n) > 0 \\ -b\eta_{ij}(n) & if \quad S_{ij}(n-1)D_{ij}(n) < 0 \\ 0 & otherwise \end{cases}$$

Equation 5

where the products of S_{ij} and D_{ij} are measuring simply the sign of the gradient over the iterations. The first case refers to the case of slow convergence, so the step size is arithmetically increased at each iteration (increased by a constant, which is a slow process). The second case refer to the case that the present stepsize is too large, so the algorithm decreases the stepsize proportionally to its current value, which is a geometric decrease (very fast). The reason for these two different regimes comes from the fact that

one wants to avoid divergence at any cost (the search algorithm looses the information where it is in the performance surface and has to start over).

Now let us analyze carefully the conditions to increase or decrease the stepsize. D_{ij} (n) is the partial derivative of the cost with respect to weight w_{ij} (i.e. the gradient), and S_{ij} (n) is a running average of the current and past partial derivatives given by

$$S_{ij}(n) = (1-\gamma)D_{ij}(n-1) + \gamma S_{ij}(n-1) \qquad \textbf{Equation 6}$$

where γ is a number between 0, 1 (the exponential time constant). The product of S_{ij} and D_{ij} is checking if the present gradient has the same sign as the previous gradients (reflected in the value of S_{ij}).

NeuroSolutions 7

4.7 Adaptive Stepsizes

Let us solve the STAR problem now with the adaptive stepsize (delta bar delta) search to see the speedup achieved. The adaptive stepsize algorithm gives us the flexibility to have high learning rates when we are in flat parts of the performance surface and low learning rates when the adaptation begins to rattle or get unstable. The NeuroSolutions version of the delta bar delta algorithm includes momentum, so this algorithm should work better than any of the methods we have discussed so far.

NeuroSolutions Example

There are many other alternate algorithms to adapt the stepsize, such as Fahlman 's quickprop, and Almeida 's adaptive stepsize Almeida's adaptive stepsize . You should experiment with them in NeuroSolutions since they are implemented there. The big problem is that all these algorithms increase the number of free parameters that the user has to heuristically set. So they can be fine-tuned for each application only after extensive experimentation.

Another alternative is to inject random noise at either the input, weights, and desired

response. The motivation is to "shake" the weights during adaptation in order to minimize the probability of having the search caught in local minima. This idea is reminiscent of the operating principle of simulated annealing which uses a scheduling of noise to reach the global minimum of the performance surface. The process is also very simple to implement when applied to the input or the desired response. In fact, if we add zero mean white Gaussian noise to the desired response, we obtain

$$d_w(n) = d(n) + n_w(n)$$

which is then transmitted to the injected error *e(n)* and then to the weight update through backpropagation. The advantage of applying the noise to the desired response is that we have a single noise source, and since the dual network is linear as we saw in Chapter III, it is still an additive zero-mean perturbation to the weight update. But remember that the noise variance should be scheduled to zero such that the optimum solution is obtained. Unfortunately, there is no principled approach to set the noise variance, nor to schedule it. The injection of noise at the input is no longer a linear contribution to the weight update, but it has been shown to produce better generalization (see Bishop) which is highly desirable.

NeuroSolutions 8

4.8. Adaptation with noise in the desired signal

This example adds the noise component to the desired signal to show how we can get better performance. The noise helps the algorithm to bounce out of local minima. We will also anneal the noise variance so that we have a lot of noise at the beginning of training to help us move out of local minima quickly but a little noise at the end of training to reduce rattle and increase precision.

NeuroSolutions Example

Notice that until now all the modifications to the gradient descent rule of Eq. 3 utilize the same basic information, namely the activation to the PE and the error to its dual which characterize first order search methods. In NeuroSolutions, the dataflow architecture for backpropagation and the layered nature of its implementation provide a straightforward

selection of any of these methods. In fact, *only the weight update rule (the search component) needs to be modified* which can be accomplished by simply changing the search component icon on the breadboard. This enhances even further the value of the data flow concept for simulation of neural networks.

3.3. Advanced Search Methods

Numeric optimization techniques are a vast and mature field. We will only provide here a synopsis of the classes of search methods that are important for neural network researchers. The interested reader is referred to Luenberger , and Fletcher for a full coverage of the topic.

The problem of search with local information can be formulated as an approximation to the functional form of the cost function *J* (**w**) at the operating point $\mathbf{w_0}$. This immediately points to the Taylor series expansion of *J* around $\mathbf{w_0}$

$$J(\mathbf{w}) = J_0 + (\mathbf{w} - \mathbf{w_0})\nabla\mathbf{J}(\mathbf{w}_0) + 1/2(\mathbf{w} - \mathbf{w}_0)\mathbf{H}(\mathbf{w} - \mathbf{w}_0)+\ldots$$

where ∇J is our already familiar gradient, and **H** is the Hessian matrix, i.e. the matrix of second derivatives with entries

$$H_{ij} = \frac{\partial^2 J(w)}{\partial w_i \partial w_j}\bigg|_{w=w_0}$$

evaluated at the operating point. We can immediately see that the Hessian can NOT be computed locally since it uses information from two different weights. If we differentiate *J* with respect to the weights, we get

$$\nabla J(\mathbf{w}) = \nabla J(\mathbf{w}_0) + \mathbf{H}(\mathbf{w} - \mathbf{w}_0)+\ldots \qquad \textbf{Equation 7}$$

so we can see that in order to compute the full gradient at **w** we need all the higher terms of the derivatives of *J*. This is impossible to do. Since the performance surface tends to be bowl shaped (quadratic) near the minimum we normally are only interested in the first and second terms of the expansion.

If the expansion of Eq. 7 is restricted to the first term we obtain the *gradient search methods* (hence their name - first order methods), where the gradient is estimated with its value at **w**0.

If we expand to use the second order term, we obtain *Newton's method* (hence the name second order method). It is interesting to note that if we solve the equation $\nabla J = 0$ we immediately get

$$\mathbf{w} = \mathbf{w}_0 - \mathbf{H}^{-1}\nabla\mathbf{J}(\mathbf{w}_0) \qquad \textbf{Equation 8}$$

which is exactly the equation for the Newton's method presented in Chapter I. Newton's method has the nice property of quadratic termination (it is guaranteed to find the exact minimum in a finite number of steps for quadratic performance surfaces). For most quadratic performance surfaces it can converge in one iteration.

The real difficulty is the memory and the computationally cost (and precision) to estimate the Hessian. Neural networks can have a thousand of weights, which means that the Hessian will have a million entries….This is the reason why methods of approximating the Hessian have been heavily investigated. There are two basic class of approximations:

- line search methods
- pseudo-Newton methods.

The information in the first type is restricted to the gradient together with line searches along certain directions, while the second seeks approximations to the Hessian matrix.

Line search methods

The basic idea of line search is to start with the gradient descent direction and search for the minimum along the line, i.e.

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \lambda(n)\mathbf{s}(n) \qquad \textit{where} \quad \lambda(n) = \min_{\lambda} \mathbf{J}(\boldsymbol{w}(n) + \lambda \boldsymbol{s}(n))$$

Steepest descent is itself a line search method where the direction **s** is the gradient

direction $\boldsymbol{s} = -\nabla \boldsymbol{J}(\mathbf{w}_0)$. Batch learning with the LMS, delta rule or backpropagation basically implement steepest descent. The problem with the gradient direction is that it is sensitive to the excentricity of the performance surface (caused by the eigenvalue spread), so following the gradient is not the quickest path to the minimum. We analyzed this aspect in Chapter I. Alternatively, one can compute the optimal stepsize at each point which corresponds to a line search as we saw above. But we can prove that successive directions have to be perpendicular to each other, i.e. the path to the minimum is intrinsically a zig-zag path (Luenberger).

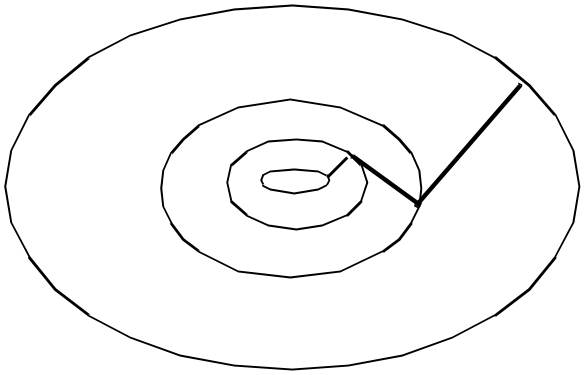


Figure 3. Path to the minimum with line minimization

We can improve this procedure if we weight the previous direction to the minimum with the new direction, i.e. cutting across the zig-zag. The formulation becomes

$$\boldsymbol{s}^{new} = -\nabla \boldsymbol{J}^{new} + \alpha \boldsymbol{s}^{old}$$

where α is a parameter that compromises between the two directions. This is called the *conjugate gradient method*. For quadratic performance surfaces, the conjugate gradient algorithm preserves quadratic termination and can reach the minimum in n steps where n is the dimension of the weight space. As we will see below, the interesting thing is that we do not need to compute second derivatives (Hessian), and in fact the algorithm is compatible with backpropagation.

Notice that the momentum learning implements this idea to a certain extent, since we can view the difference between the previous weights as the estimate of the old direction. But

α is fixed throughout instead of being estimated at each step, and there is no search for the best stepsize.

Pseudo Newton methods

In pseudo Newton methods, the idea is to come up with computationally simple and reasonable approximations to the Hessian. The simplest is just to forget about the cross terms in the Hessian matrix, i.e. use only the diagonal terms. This is equivalent to performing Newton's algorithm separately for each weight, which transforms Eq. 8 into

$$\Delta \boldsymbol{w}_i(n) = \frac{-\nabla \boldsymbol{J}(n)}{\frac{\partial^2 \boldsymbol{J}(n)}{\partial \boldsymbol{w}_i^2}}$$

Normally we replace this rule by

$$\Delta \boldsymbol{w}_i(n) = \frac{-\nabla \boldsymbol{J}(n)}{\left|\frac{\partial^2 \boldsymbol{J}(n)}{\partial \boldsymbol{w}_i^2}\right| + \beta}$$

Equation 9

where β is a small constant and avoids the problem of negative curvature and a zero denominator. Notice that Eq. 9 is in fact very similar to the normalized LMS we presented in Chapter I, since for the linear network we can estimate the diagonal entries of the Hessian by the power (or the trace) of the input. '

This is a very crude approximation of the Hessian. More accurate approximations which still are less computationally expensive than the full procedure (which is $O(N^3)$, with N being the number of weights) are : the Levenberg-Marquadt (LM), the Davidson-Fletcher-Powell (DFP) and the Broyden-Fletcher-Goldfarb-Shanno (BFGS). See Luenberger . LM is the most interesting for neural networks since it is formulated as a sum of quadratic terms just like the cost functions in neural networks.

3.3.1 Conjugate gradient method

A set of vectors {**s**j} is conjugate with respect to a positive definite matrix (e.g. the Hessian) if $\boldsymbol{s}_j^T \boldsymbol{H} \boldsymbol{s}_i = 0 \qquad j \neq i$. What this expression says is that the rotation by ***H*** of

the vector ***sj*** must by orthogonal to ***s***i. In Rn there are an infinite number of conjugate vector sets. It is easy to show that the eigenvectors of the Hessian form a conjugate set and can then be used to search the performance surface. The problem is that one needs to know the Hessian, which is not practical. However, there is a way to find a conjugate set of vectors that does not require the knowledge of the Hessian. The idea is to express the conditions for a conjugate vector set as a function of differences in consecutive gradient directions as

$$(\nabla \boldsymbol{J}(i) - \nabla \boldsymbol{J}(i-1))^T \boldsymbol{s}(j) = 0 \qquad i \neq j$$

For this expression to be true, the minimum of the gradient of *J(i)* in the direction ***s***(j) is needed. So the algorithm works as follows:

Start with the gradient descent direction, i.e. $\boldsymbol{s}(0) = -\nabla \boldsymbol{J}(0)$. Search the minimum along this direction. Then construct a vector ***s***(j) which is orthogonal to the set of vectors $\{\nabla J(0), \nabla J(1), ..., \nabla J(j-1)\}$ which can be accomplished by

$$\boldsymbol{s}(j) = -\nabla \boldsymbol{J}(j) + \alpha \boldsymbol{s}(j-1)$$

There are basically three well known ways to find α, namely the Fletcher-Reeves, the Polak-Ribiere, or the Hestenes-Steifel formulas which are equivalent for quadratic performance surfaces, and are given by

$$\alpha_j = \frac{\nabla J^T(j) \nabla J(j)}{\nabla J^T(j-1) \nabla J(j-1)} \qquad \alpha_j = \frac{(\nabla J(j) - \nabla J(j-1))^T \nabla J(j)}{\nabla J^T(j-1) \nabla J(j-1)} \qquad \alpha_j = \frac{(\nabla J(j) - \nabla J(j-1))^T \nabla J(j)}{\nabla J^T(j-1) s(j-1)}$$

In quadratic performance surfaces, repeat the procedure *n* times, where *n* is the size of the search space. The minimization along the line can be accomplished for quadratic performance surfaces as Eq. 29 .

The problem is that for non-quadratic performance surfaces such the ones found in neurocomputing, quadratic termination is not guaranteed and the line search does not have an analytic solution.

The lack of quadratic termination can be overcome by executing the algorithm for *n* iterations and then reset it to the current gradient direction again. The problem of the line search is more difficult to solve. There are two basic approaches: search or the scaled conjugate method (Shepherd). The first involves multiple cost function evaluations and estimations to find the minimum which complicates the mechanics of the algorithm (an unknown number of samples are needed, and we have to go back and forth in the estimations). The scaled conjugate is more appropriate for neural network implementations. It uses Eq. 24 and avoids the problem of non-quadratic surfaces by massaging the Hessian such to guarantee positive definiteness, which is accomplished by H+λI, where I is the identity matrix. Eq. 24 becomes

$$\mu_j = \frac{-\nabla \boldsymbol{J}_j^T \boldsymbol{s}_j}{\boldsymbol{s}_j^T \boldsymbol{H}_j \boldsymbol{s}_j + \lambda \left\| \boldsymbol{s}_j \right\|^2}$$

Equation 10

At first, we may think that this method is more computationally expensive than search due to the Hessian matrix. But in fact this is not the case, since there are fast methods to estimate the product of a vector by the Hessian (the product only has n components). We can use the perturbation method to estimate the product LeCun, Simard, Pearlmutter

$$\boldsymbol{s}^T \nabla(\nabla \boldsymbol{J}) = \frac{\nabla \boldsymbol{J}(\boldsymbol{w} + \varepsilon \boldsymbol{s}) - \nabla \boldsymbol{J}(\boldsymbol{w})}{\varepsilon} + O(\varepsilon)$$

or use an analytic approach due to Pearlmutter . Both methods are compatible with backpropagation as we will see below.

We still need to address how to set λ, which is not difficult, but involves trial and error. The idea is the following: if from one step to the next the error increases is because we are in an area of *J* that is far from quadratic, so the Hessian is not positive definite. In such cases, we should back-off and increase λ until the error decreases. Notice in fact that for large λ (the denominator becomes approximately $\lambda \left\| s \right\|^2$) we are just using gradient descent which is known to be convergent to the minimum (albeit slow). When

the error decreases, then λ should be again decreased to fully utilize the potential of exploiting the local quadratic information of the performance surface. NeuroSolutions implements conjugate gradients with re-scaling.

3.3.2. Levenberg-Marquadt Quasi-Newton Method

The Levenberg-Marquadt algorithm uses the Gauss-Newton method to approximate the Hessian. Let us assume that the performance function is a sum of individual components

$$J(\boldsymbol{w}) = \sum_{i=1}^{N} e_i^2(\boldsymbol{w}) = \boldsymbol{e}^T(\boldsymbol{w})\boldsymbol{e}(\boldsymbol{w})$$

where N is the number of samples in the training set, and ***e***i are the instantaneous errors. Then it is easy to show that the gradient is

$$\nabla \boldsymbol{J}(\boldsymbol{w}) = 2\boldsymbol{J}^T(\boldsymbol{w})\boldsymbol{e}(\boldsymbol{w}) \qquad \textbf{Equation 11}$$

where ***J*** is the Jacobian matrix given by

$$J(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial e_1(\boldsymbol{w})}{\partial w_1} & \cdots & \frac{\partial e_1(\boldsymbol{w})}{\partial w_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial e_N(\boldsymbol{w})}{\partial w_1} & \cdots & \frac{\partial e_N(\boldsymbol{w})}{\partial w_n} \end{bmatrix}$$

The Hessian is easily obtained from Eq. 11 as

$$\nabla^2 \boldsymbol{J}(\boldsymbol{w}) = 2\boldsymbol{J}^T(\boldsymbol{w})\boldsymbol{J}(\boldsymbol{w}) + 2\boldsymbol{S}(\boldsymbol{w}) \qquad \textbf{\textit{where}} \qquad \boldsymbol{S}(\boldsymbol{w}) = \sum_{\boldsymbol{i}=1}^{\boldsymbol{N}} \boldsymbol{e}(\boldsymbol{w})\nabla^2 \boldsymbol{e}_{\boldsymbol{i}}(\boldsymbol{w})$$

Assuming that ***S(w)*** is small, the Hessian can be approximated by

$$\nabla^2 \boldsymbol{J}(\boldsymbol{w}) \cong 2\boldsymbol{J}^T(\boldsymbol{w})\boldsymbol{J}(\boldsymbol{w})$$

So Eq. 8 can be written

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) - [\boldsymbol{J}^T(\boldsymbol{w}(n))\boldsymbol{J}(\boldsymbol{w}(n))]^{-1}\boldsymbol{J}^T(\boldsymbol{w}(n))\boldsymbol{e}(\boldsymbol{w}(n))$$

This weight update does not require second order derivatives. The approximation introduced, may result in difficulties in inverting ***H***. But as we saw above for the conjugate gradient, we can add a small value (λ) to the diagonal of ***J*** to make sure that the matrix is full rank. This provides the Levenberg-Marquardt algorithm

$$\Delta \boldsymbol{w}(n) = -[\boldsymbol{J}^T(\boldsymbol{w}(n))\boldsymbol{J}(\boldsymbol{w}(n)) + \lambda(n)\boldsymbol{I}]^{-1}\boldsymbol{J}^T(\boldsymbol{w}(n))\boldsymbol{e}(\boldsymbol{w}(n)) \quad \textbf{Equation 12}$$

This is a really interesting formula: Notice that if λ is increased such that the first term of the inverse is negligeable, then the weight update is basically gradient descent with a stepsize (1/ λ (n)). On the other hand if λ is zero than the information about the curvature is utilized in the search.

We mentioned this fact because we have to practically set λ during the adaptation. The goal is to have λ as small as possible to guarantee inversion. But this depends upon the data. So we should start with a small value of λ (λ =0.01), and see if the next weight vector produces a smaller error. If it does, continue. If the error for this new position is higher than before, we have to backtrack to the old weight vector, increase λ and try again. At each try λ should be increased by a nominal amount (5 to 10 is normally recommended). Notice that if we continue doing this, we will default to gradient descent which is known to be convergent with small stepsizes. When the error for the new evaluation of the weights produces a smaller error, then start decreasing λ by the same factor.

This algorithm is particularly well suited for training neural networks with the MSE. Moreover, it interfaces well with the backpropagation formalism. Note however, that here we need to keep separated all the errors since we need to use the Jacobian matrix. In conventional backprop the errors from all the outputs get added up at each PE, and in batch we even add all these errors across the training set. However, here each partial of the error must remain accessible during training of a batch, which causes huge storage requirements. Nevertheless, the backpropagation algorithm can still propagate sensitivities from the output to each node to evaluate each entry in the Jacobian. As a rule, backprop must be applied repeatedly and independently to each output to avoid the addition of errors, i.e. the injected error vector for a P output system becomes

$$e^1 = \begin{bmatrix} e_1 \\ 0 \\ \dots \\ 0 \end{bmatrix} \longrightarrow e^2 = \begin{bmatrix} 0 \\ e_2 \\ \dots \\ 0 \end{bmatrix} \longrightarrow e^P = \begin{bmatrix} 0 \\ 0 \\ \dots \\ e_P \end{bmatrix}$$

So each input-desired response pair creates P errors. Each of these backprop sweeps for a given sample will create a row of the Jacobian (the number of columns is given by the number of weights in the network). Each new sample will repeat the process. So one can see that the Jacobian matrix gets very large very quickly. The other difficulty is that the algorithm is no longer local to the weight. Nevertheless the Levenberg-Marquardt algorithm has been shown much faster than backpropagation in a variety of applications. As a side note, the conjugate gradient of Eq. 10 also requires the use of the Jacobian matrix, so it uses the basic same procedure.

Go to next section

4. Stop Criteria

We have addressed many aspects of gradient descent learning, but we have not yet discussed how and when to stop the training of the neural network. Obviously training should be stopped when the learning machine has learned the task. The problem, however, is that there are no direct indicators that measure this.

4.1. Stopping based on training set error

One of the simplest ways to stop the training phase is to limit the number of iterations to a pre-determined value, as we have done so far. But the only appeal of this criterion is simplicity. It does not use any information or feedback from the system before or during training. When the number of iterations are capped at a predefined value, there is no guarantee that the learning machine has found coefficients that are close to the optimal values.

This suggests an analysis of the output MSE to stop the training. One can choose an acceptable error level for the problem and *threshold the MSE*. Choosing the MSE

threshold value, however, is tricky because the MSE is just an indirect variable in classification. Moreover, one has no guarantee that the learning system will achieve the pre-selected MSE value, so the training may never end.

Still another alternative is to let the system train until the decrease of the MSE from one iteration to the next is below a given value (*threshold the incremental MSE*). The idea is to let the system train until a point of "diminishing returns", i.e. until it basically can not extract more information from the training data. Unfortunately, this method has the problem of prematurely stopping the training in flat regions of the performance surface.

NeuroSolutions 9

4.9. Stopping based on MSE value

In this example we use a new component, the transmitter, to stop training. There are two types of transmitters, the threshold transmitter and the delta transmitter. The threshold transmitter transmits a message (in this case to the controller to stop training) when the value of the error gets below a certain point. The delta transmitter stops training when the difference between two successive errors is below a certain value. When you look at the properties of the delta transmitter you will notice that there is a smoothing algorithm applied to the error – this can help reduce the probability of stopping because of outliers. Notice that both of these stop criterion have their problems – we will discuss better methods in the next section.

NeuroSolutions Example

4.2. Stopping criterion based on generalization

The previous stop criteria did not address the *problem of generalization* , i.e. how well the learning system performs in data that does not belong to the training set. Old ideas in data modeling and recent developments in learning theory (Vapnik) clearly indicate that after a critical point, an MLP trained with backpropagation will continue to do better in the training set but the test set performance will begin to deteriorate. This phenomenon is called *overtraining*.

One method to stop the training is at the *point of maximum generalization* (given the present data and topology). This method is called *early stopping,* or *stopping with crossvalidation*. It has been experimentally verified that the training error always decreases (for a sufficient large net) when the number of iterations is increased. If we plot the error in a set of data that the network was not trained with (the validation set) we find that the error starts to decrease with the number of iterations, but then starts to increase again (Figure 4*). So training should be stopped at the point of the smallest error in the validation set*.

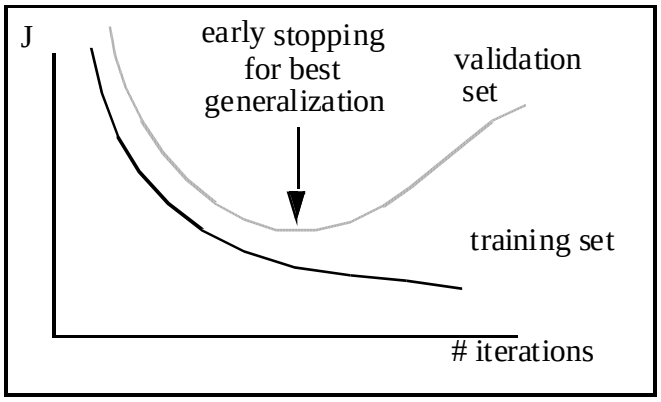


Figure 4 . Cross validation or early stopping

To implement this method, the training set should be divided in two sets, the *training* and the *crossvalidation* sets. The crossvalidation set is normally taken as 10% of the total training samples. Every so often (i.e., 50 -100 iteration), the learning machine performance with the present weights is tested against the validation set. Training should be stopped when the error in the crossvalidation set starts to increase. This point is the point of maximum generalization.

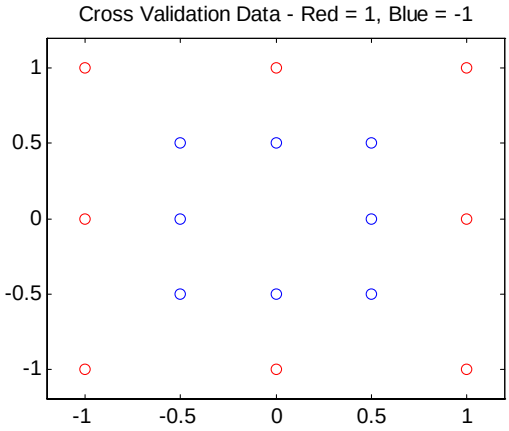
The problem with this methodology is that crossvalidation decreases the size of the training set. Since neurocomputing tends to suffer from a lack of data to begin with, crossvalidation makes this situation even worse. In this case, however, the benefits (accurate stopping point) typically outweigh the costs (less data). *This method is the recommended stopping criterion for real world applications mainly when comparisons of neural topologies are being conducted*. NeuroSolutions implements this method as we

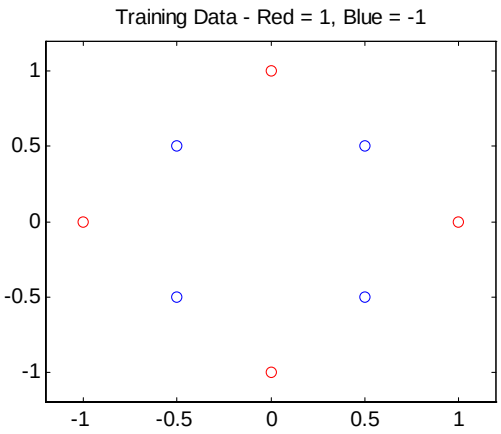
will exemplify next.

NeuroSolutions 10

4.10 Stopping with Crossvalidation

We will again solve the STAR problem but this time using Cross Validation as the stop criteria. We have created an additional set of points which we will use as our Cross Validation (or test) set. The figure below shows the training set and test set:





In order to implement the crossvalidation in NeuroSolutions we have to bring two input and desired files. In the file component we have to specify one pair will be the training/desired files while the other will be the crossvalidation files. Likewise in the controller we have to specify that we are using crossvalidation.

This example will clearly show that overtraining can greatly reduce the generalization ability of the network. You will also see how important it is to have a sufficient amount of training data that covers the input space as much as possible.

Any points in the input space which are not in the training set are considered to be "don't cares" by the network. This can result in odd shaped discriminant plots which have very good mean square errors but poor performance with new data.

NeuroSolutions Example

Go to next section

5. How good are MLPs as learning machines?

5.1. Training set size

The size of the training set directly influences the performance of any classifier trained non-parametrically (e.g. as neural networks). This class of learning machines requires a lot of data for appropriate training, because there are no a priori assumptions about the data. It is important to know how the requirement in training set size scales as a function of the size of the network for a given precision in the mapping.

The number of training patterns (*N*) required to classify test examples with an error of δ is approximately given by

$$N > \frac{W}{\delta}$$

Equation 13

where W is the number of weights in the network. This equation shows that the number of required training patterns increases linearly with the number of free parameters of the MLP, which is excellent compared to other classification methods. A rule of thumb states that N~10W, that is, the *training set size should be 10 times larger than the number of network weights to accurately classify test data with 90% accuracy.*

In this rule of thumb it is assumed that the *training set data is representative* of all the conditions encountered in the test set. The main focus when creating the training set should be to collect data that covers the full known operating conditions of the problem we want to model. If the training set does not contain data from some areas of patterns

space, the machine classification in those areas will be based on generalization. This may or may not correspond to the true classification boundary (your desired output), so one should always choose samples for the training set that cover as much of the input space as possible.

We should remark that in Eq.13 the practical limiting quantity is the number of training patterns. So, most of the time one needs to compromise the size of the network to achieve appropriate training for the learning machine. A reasonable approach to reduce the number of weights in the network is to sparsely connect the input layer to the first hidden layer (which normally contain the largest number of weights). This will help achieve the requirement of Eq. 13. Another possibility is to use feature extraction (i.e. a preprocessor) that decreases the input space dimensionality, thus reducing the number of weights in your network.

NeuroSolutions 11

4.11 Sparse connectivity in the input layer

Here we will show that with the arbitrary synapse generalization can be improved with respect to the fully connected case. We will use the STAR data set that we have been training with 4 hidden PEs. But instead of 8 weights in the first Synapse now we will restrict the number to 4, randomly connected. Run the network and show that the solutions found are normally more reasonable as the ones using 8 weights.

One way to show this is to use the cross data as you did before,

NeuroSolutions Example

5.2. Scalability

Another very important point in learning machines is to address how well their properties scale when the size of the problem increases. The literature is full of examples of systems that perform very well in small problems, but are unable to extend the same performance to larger problems. One very important proof advanced by Barron through the analysis of the MSE for several size problems, states that the (one hidden layer) *MLP*

error is independent of the size of the input space and scales as the inverse of the number of hidden PEs (O(1/N)). This is much better than polynomial approximators where the error is a function of D, the dimension of the space, i.e. when the input space dimension increases to keep the same error one has to exponentially increase the number of parameters ($O(1/\sqrt[D]{N^2})$). So this means that MLPs are particularly well suited to deal with large input dimensional problems. This may explain the good performance of MLPs in large classification problems.

5.3. Trainability

The training time of an MLP using backpropagation was experimentally verified to increase exponentially with the size of the problem, i.e. although the required number of patterns increases only linearly with the number of weights, the training of larger networks seems to scale exponentially to their size*. This indicates that there are problems that can not be solved practically with MLPs trained with backpropagation*. However, we can use modular network architectures or sparse connections to counteract this law, as well as training rules that do not converge linearly with the number of iterations as the gradient descent. In practice, this exponential scaling of training times with network size *provides another argument to start first with small networks and increase their size if the results are not satisfactory*. Moreover, this property emphasizes the importance of training methods that extract information more efficiently from the data than gradient descent. Now you may understand a little better the importance of the discussion of the conjugate gradient and the quasi-Newton methods presented in section 3.3.

5.4. Did the network learn the task?

In the application of neural networks to real world problems, it is very important to have a criterion for accepting the solution. Only then can we successfully act to overcome any potential difficulties.

The learning curve is a precious indicator for observing the progression of learning, but

the *MSE either in the training or test sets is only an indirect measure of classification performance*. The MSE is dependent upon the normalization and characteristics of the input data and desired response. One should normalize the total error by the variance of the desired response to have an idea of how much of the desired variance was captured by the neural model. This is reminiscent of the correlation coefficient for linear regression, but there is no precise relation between classification accuracy and MSE.

The performance of a classifier is measured in terms of classification error as we saw in Chapter II. The accuracy of the classifier is one minus the classification error. Therefore, a much better approach is to construct the confusion matrix to count exactly the number of misclassifications. The confusion matrix is a table where the true classification is compared with the output of the classifier (see Table I). Let us assume that the true classification per class is the column and the classifier is the row. The classification of each sample (specified by a row) is added to the column of the true classification. A perfect classification provides a confusion matrix that has only the diagonal populated. All the other entries are zero.

Confusion Matrix

true/ machine	class 1	class 2	total machine
class 1	# entries (or percentage)		
class 2			
total true			total samples

The confusion matrix also enables easy visualization of where the classifier has difficulties. In general some of the classes will be more separable than others, so the confusion matrix immediately pin points which classes produce miscalssifications (off diagonal entries that have large values). In summary, the confusion matrix is an excellent way of quantifying the accuracy of a classifier. As we discussed in Chapter II, the test of the classifier should be performed in the test set, so the confusion matrix should be

constructed in the test set.

NeuroSolutions 12

4.12 Confusion matrix for classification performance

In this example we will show how for classification, the confusion matrix gives us a much better picture of the performance of the network than the Mean Squared Error. The confusion matrix tells us exactly what we want to know – how well the network is classifying the data. The mean squared error only tells us the average difference between the network output and the desired output. Since classification is an "all or nothing" type of problem, it doesn't always matter how close you are to the desired +/- 1 as long as you are beyond the classification threshold.

NeuroSolutions Example

Once we find that learning is not successful (a large classification error assessed by the confusion matrix), the next step is to find out why the network did not learn correctly. Poor performance may have many different explanations:

- the network may not have the discrimination power (number of layers) to correctly classify the data (just remember the perceptron and the XOR);
- the network may not have enough PEs (remember the case of the bump with two hidden PEs); or
- learning may be stuck in a local minimum or flat spot.
- we may not have collect enough data to represent the problem well.
- the problem may be intrinsically difficult with the features (measurements) that we are using, so we may also want to transform/filter the inputs or add new inputs to simplify the classification problem.

When the learning curve stabilizes after many iterations at an error level or classification error that is not acceptable, it is time to rethink the network topology (more hidden PEs or more hidden layers, or a different topology altogether) or the training procedure (other more sophisticated gradient search techniques).

Unfortunately there is no general rule to test any of these issues, so the designer of classifiers must use his insight and knowledge of the problem to improve the machine

classification. And adopt a "manual supervision" of the learning instead of setting initially the learning parameters and dim the monitor until the next morning. We have shown how NeuroSolutions probes are particularly useful to check what is going on during learning.

5.5. Some hints on how to improve training

We will present below a set of heuristics that will help decrease the training times and produce in general better performance.

- Normalize your data to the range of the network activations.
- Use the tanh nonlinearity instead of the logistic function.
- Use a sofmax PE at the output layer.
- Normalize the desired signal to be just below the output nonlinearity "rail" values (i.e. if you use the tanh, use desired signals of +/- 0.9 instead of +/- 1).
- Add a constant value of 0.05 in the derivative of the nonlinearity (errors will always flow through the dual).
- Set the step size higher in the layers closer to the input.
- Shuffle the training set from epoch to epoch in on-line learning.
- Initialize the net weights in the linear region of the nonlinearity (choose the standard deviation of the random noise source as Eq. 2).
- Use more sophisticated learning methods (delta bar delta, add noise, conjugate gradients).
- Always have more training patterns than weights. You can expect the test set performance of your MLP to be limited by the relation $N>W/\varepsilon$, where N is the number of training patterns, W the number of weights and ε the performance error. You should train until the mean square error is less than $\varepsilon/2$.
- Use cross validation to stop training.
- Always run the network several times to gauge performance.
- Use a committee of networks to improve classification.

Go to next section

6. Error Criterion

6.1. Lp norms

In supervised learning, the difference between the desired response and the actual learning system output is utilized to move the system state such that a minimum error of the performance surface is achieved. The issue is how to define the performance, also called the *error criterion* or the *cost*. In normal operation the learning machine will provide an output for each input pattern. So the total cost will be computed as a sum of individual costs, $J_{n,k}$, obtained from each input pattern presentation, i.e.

$$J = \sum_{k} \sum_{n} J_{n,k}$$

Equation 14

where k is an index over the system outputs, n is an index over the input patterns, $J_{n,k}$ is the individual cost defined as $J_{n,k}=f(d_{n,k}-y_{n,k})=f(\varepsilon_{n,k})$. The only issue is then how to compute the individual cost as a function of $\varepsilon_{n,k}$, which will be called the *instantaneous error* $\varepsilon_{n,k}$.

The mean square error (MSE) criterion defines the individual cost as the square of the instantaneous error between the desired response and the system output, i.e $J_{n,k}=(d_{n,k}-y_{n,k})^2$. The error power (MSE) has a meaning in itself, and has three other major appeals:

- it leads to a linear optimization problem in linear feedforward networks, which accepts an analytical solution;
- it provides a probabilistic interpretation for the output of the learning machine as we discussed in Chapter 2;
- the criterion is easy to implement since it is the instantaneous error that is injected into the dual system (no additional computations are needed).

Is there a need for other error criteria? Let's look at Figure 5 and understand what the mean square error criterion does.

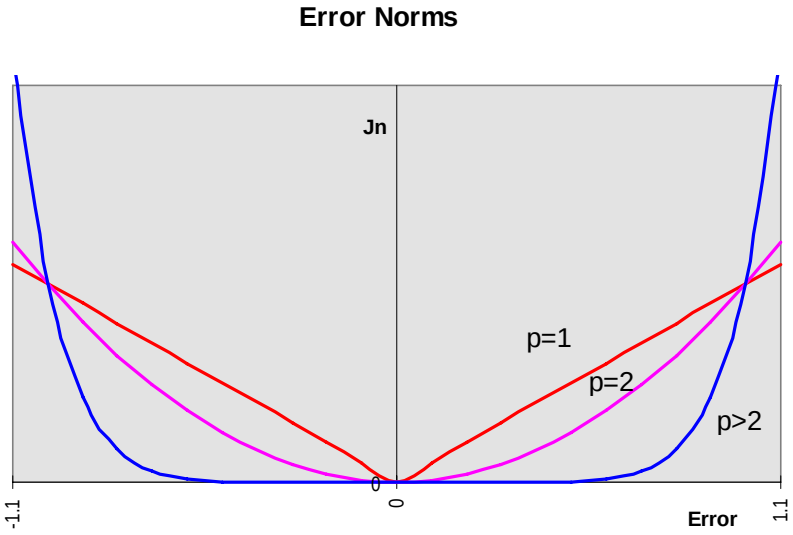


Figure 5 Error norms

In Figure 4 we present several cost functions derived from different powers *p* of the instantaneous error. With the MSE, the instantaneous cost is the square of the magnitude of the instantaneous error (*p=2*). This means that when the learning machine minimizes the error power, *it weights the large errors more (quadratically)*. If you recall the weight update performed by gradient descent, $\Delta w_{ij} = \eta \delta_j x_i$, you will realize that the weight values will be updated proportionally to the size of the error, so the weights are very sensitive to the larger errors. This is reasonable if the data is clean without many large deviations, but in practice the data sets may have outliers. So, outliers may have an inordinate effect on the optimal parameter values of the learning machine. Learning machines with saturating nonlinearities control this aspect better than linear PE machines, but they still are more sensitive to large errors than small errors. Since the values of the weights set the orientation and position of the discriminant function one can deduce that outliers will **"**bias**"** the position of the discriminant function.

NeuroSolutions 13

4.13 Regression performance as a function of the norm

In this example we use the linear regressor and the data from chapter 1 to show

how the choice of error criterion affects the performance of the linear regression. We have modified the data set so that one point is an outlier, it is much higher than the others. As you will see in the example, the norms which weight large errors more heavily will skew their regression line much closer to the outlier (giving a less accurate estimate of the rest of the data). Remember not to try to compare the errors as displayed in neurosolutions – the values of the error reported are dependent on the error criterion and thus can't be compared.

NeuroSolutions Example

This argument shows that if we want to modify how the instantaneous error influences the weights we can define the instantaneous cost more generally as

$$J_{n,k} = \left| d_{n,k} - y_{n,k} \right|^p \qquad \textbf{Equation 15}$$

where *p* is an integer, which is normally called the p norm of the instantaneous error $\varepsilon_{n,k}$. When *p=2* we obtain the L_2 norm that leads to the MSE criterion. When *p=1* one obtains the L_1 norm that is called also the Manhattan metric. Notice that the L_1 norm weights the differences proportional to their magnitude, so it is far less sensitive to outliers than the L_2 norm. For this reason it is called a more robust norm . In general the L_p norm for p>2 weights large deviations even more. Different norms will provide different solutions to a learning problem, because the weights are being modified with information that depends on the choice of the norm, so the positioning of the discriminant functions is affected by the norm (for the same training data set).

For positive finite integers p, the derivative of the norm can be computed quite easily, as

$$\frac{\partial J_{n,k}}{\partial y_{n,k}} = \left| d_{n,k} - y_{n,k} \right|^{p-1} sign(d_{n,k} - y_{n,k}) \qquad \textbf{Equation 16}$$

but L_p norms do not cover all the cases of interest. In L_p norms the instantaneous cost $J_{n,k}$ increases for errors larger than 1 always at the same rate or faster than the instantaneous error, which may not be our goal. There are cases of practical relevance

that do not have an analytic solution such as the L_∞ norm (all errors are zero except the largest). Another possible criterion is to simply use the sign of the deviation (p=0).

Minskowski measures

NeuroSolutions 14

4.14 Classification performance as a function of the norm

In this example we will show how the choice of error criterion affects the location of the discriminant function. We will use the height and weight data (trying to classify male/female subjects based on their height and weight) using different norms. You will see that the discriminant function ends up in different places depending on the norms since each norm weights the errors differently.

NeuroSolutions Example

6.2. Constructing the error directly

In backpropagation learning, the partial derivative of the cost with respect to the weight is given by

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_k}\frac{\partial y_k}{\partial w_{ij}}$$

Equation 17

The first partial is the instantaneous error that is injected in the dual network. Given a cost function one can differentiate it to compute the instantaneous error of Eq.16 and then use backpropagation to compute the sensitivity with respect to the individual weights w_{ij} (the second term in Eq. 16).

An alternate practical approach is to *separate the propagation* of the instantaneous error (done by the backpropagation algorithm) from its *generation*. Instead of thinking of an error produced by the derivative of a cost function (which requires differentiable cost functions) a function of the error, $g(\varepsilon_{k,n})$, can be directly injected in the dual system, i.e.

$$\frac{\partial J}{\partial w_{ij}} = g\left(\varepsilon_k\right)\frac{\partial y_k}{\partial w_{ij}}$$

Equation 18

The function g(.) is derived to meet our needs. For instance *arctanh* ($\varepsilon_{k,n}$) implements a reasonable approximation of the L_∞ norm, since it weights large errors exponentially. Another example is simply to take the sign of the instantaneous error and inject +/- 1, but many more examples with unexplored properties exist.

6.3. Information theoretical error measures

In information theory, one can measure the divergence (divergence is a water down definition of distance) between two probability distributions *q(x)* and *p(x)* by the *Kullback-Leibler* (K-L) *information criterion or cross entropy* *cross entropy criterion*

$$L = \sum_r P_r \ln(\frac{P_r}{Q_r})$$ **Equation 19**

Since the learning system output is approximating the desired response in a statistical sense, it is reasonable to utilize the K-L criterion as our cost. In this case P becomes the target density constructed by +1/-1, and Q the learning system output density. This is particularly appropriate for classification problems where the L_2 assumption is weak because the distribution of the targets is far from Gaussian. In classification we work with a discrete training set, so the integral is substituted by a summation. For c classes, the cost function becomes

$$J = \sum_n \sum_k d_{n,k} \ln\left(\frac{y_{n,k}}{d_{n,k}}\right)$$ **Equation 20**

where n is the index over the input patterns and k over the classes. Since this criterion works on probabilities, the output PEs should be a *softmax* PE which exponentiates the sum of the inputs. One can easily show (Hertz) that the instantaneous error backpropagated through the network (the partial of *J* with respect to *y*) with the softmax is

$$\frac{\partial J}{\partial net_k} = y_k - d_k$$ **Equation 21**

This is an interesting result, since it says that the cross entropy criterion can be

implemented by the MSE criterion (which also specifies the injection of the error as Eq. 21). However, the network utilizes an output PEs that implement the softmax function. This can be easily accommodated if we associate the softmax PE with a linear PE in the backplane as its dual (Figure 6).

This result reduces in the two-class problem (single output PE) to the use of a logistic output PE. By not subjecting the error to the attenuation produced by the derivative of the output PE nonlinearity, the network converges faster. It is also possible to show that the cross entropy is similar to the L_1 norm, which means that this criterion weights small errors more heavily when compared to the L_2 norm.

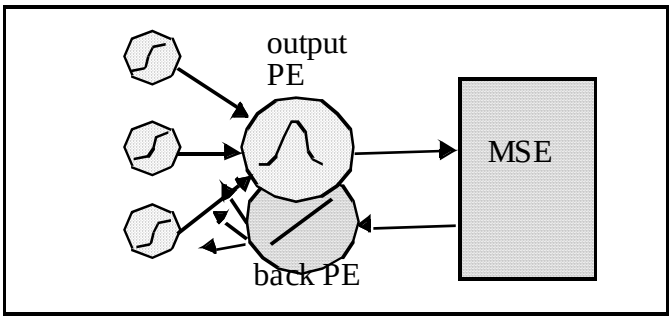


Figure 6. Implementation of the cross-entropy criterion

The other interesting thing about training the MLP with cross entropy is that the interpretation of the output as the a posteriori probability of the class given the data, also hold in this case (Bishop).

NeuroSolutions 15

4.15 Cross-entropy training

In this example we will show you how easy it is to implement the cross entropy criterion for the two class case in NeuroSolutions. All we need to do is change the back-PE of the logistic nonlinearity at the network output to a linear back-PE. This PE changes the backpropagated error such that the network will learn using entropy. Notice that the learning is faster, as we would expect since the error is not attenuated by the derivative of the nonlinearity.

NeuroSolutions Example

Go to next section

7. Network Size and Generalization

The coupling between the number of required discriminant functions to solve a problem and the number of PEs was heuristically established in Chapter III, and the relation between the number of weights and training patterns was just discussed in section 5.1. From these facts one could think that the larger the learning machine the better its performance (provided we have enough data to train it). The point about scalability shows however that larger machines may never learn well. But this is not the most pressing issue. All these arguments pertain to the training data. The fundamental question in any practical application is: how does the learning machine perform in data that it was not trained with, i.e. the test set data? This is the problem of generalization .

As we discussed earlier*, MLPs trained with backpropagation do not control their generalization ability*, which can be considered a shortcoming of the technology. Using a cross validation set to stop the training allows us to maximize generalization for a given network size. However it does not provide a mechanism for establishing the best network topology for generalization. The issue is the following: do networks of growing size preserve generalization?

If we reflect on how the network performs its function, we immediately see that the size of the machine (sometimes called the model complexity) is related to performance: too few degrees of freedom (weights) affect the network ability to achieve a good fitting to the target function. However, if the network is too large, then it will not generalize well, because the fitting is too specific to the training set data (memorization). An intermediate size network is our best choice. Therefore, *for good performance, methods of controlling the network complexity become indispensable in the MLP design methodology*.

The problem of network size can be stated in a simplified manner using Occam's razor argument as follows:

Any learning machine should be sufficiently large to solve the problem, but not larger.

The issue is to know what is large enough. Structural learning theory (V-C dimension) gives a theoretical answer to generalization but it is difficult to apply to the MLP. Alternate theories give partial answers that elucidate the principles at work, and will be covered in Chapter IV. early stopping and model complexity

There are two basic approaches that deal with the learning machine size (Hertz, Krogh, Palmer). Either we start with a small machine and increase its size (*growing method*); or we start with a large machine and decrease its size by pruning unimportant components (*pruning method*). Pruning is the only method we will address here.

Pruning reduces the size of the learning machine, by eliminating either weights or PEs. The basic issue of pruning is to find a good criterion to determine which parameters should be removed without significantly affecting the overall performance of the network. We will describe two basic methods: eliminate weights based on their values, or compute the importance of the weight in the mapping.

7.1. Weight elimination

The idea is to create a driving force that will attempt to decrease to zero all the weights during adaptation. If the input-output map requires some large weights, learning will keep bumping up the important weights, but the ones that are not important will be driven to zero. This idea is called weight decay. Weight decay can be implemented very simply by adding an extra term into the weight adaptation as shown next for the gradient descent rule

$$w_{ij}(n+1) = w_{ij}(n)(1-\lambda) + \eta\delta_i x_j \qquad \textbf{Equation 22}$$

where δ is the local error, x the local activation, η the learning rate, and λ the weight decay constant. Weights that are smaller than a certain value can be eliminated, reducing the overall number of degrees of freedom of the network. Weight decay should not be applied to the biases of the network, just to the weights.

Alternatively one can only use the sign of the weight to change its value. i.e.

$$w_{ij}(n+1) = w_{ij}(n) + \eta\delta_i x_j + \lambda\,\mathrm{sgn}\left(w_{ij}\right) \qquad \textbf{Equation 23}$$

where sgn(.) is the signum function. The problem with Eq. 22 is that it favors many small weights instead of a large one, producing normally model bias (Bishop). One way to counteract this problem is to create a weight decay term that is smaller for larger weights, i.e.

$$w_{ij}(n+1) = w_{ij}(n)\left(1 - \frac{\lambda}{\left(1+w^2{}_{ij}\right)^2}\right) + \eta\delta_i x_j \qquad \textbf{Equation 24}$$

The weight elimination method is very easy to implement because the weights are being updated and decayed during adaptation. The issue is how to select a good weight decay constant λ such that convergence is achieved, and unnecessary weights do go to zero.

The weight decay equations can also be applied in lieu of the early stopping as a method to constrain the complexity of the model. It has been shown that for quadratic performance surfaces, weight decay is equivalent to early stopping. This will be further described in Chapter V.

NeuroSolutions 16

4.16. Weight-decay

In this example we will use the weight decay algorithm from

$$w_{ij}(n+1) = w_{ij}(n)(1-\lambda) + \eta\delta_i x_j$$ **Equation 22. In general, we do not know how many PEs will be required to solve a given problem. However, based on generalization considerations, we know that we want as small a network as possible. If we have too many nodes we can obtain low errors during training but our generalization will be poor since we may have overtrained the network or obtained a solution too specific to the training data. Weight decay subtracts a small portion from each weight during each update. Since the weights which are**

important to the solution will be constantly updated while the others are not, the important weights will move to the correct location and the other weights will move towards zero. This is effectively limiting the number of PEs to only those which are required. A very good result.

NeuroSolutions Example

7.2. Optimal brain damage

The previous method is solely based on weight magnitude, which is a crude measure of the importance of the weight in the input-output map. A better measure to compute weight saliency is to find the effect on the cost of setting a weight to zero (LeCun). It has been shown that the Hessian H with elements

$$H_{ij} = \frac{\partial J}{\partial w_i \partial w_j}$$

Equation 25

contains the required information. The problem with this computation is that it is non-local (requires information from pairs of weights). A local approximation (sometimes poor) to the Hessian can be computed taking into consideration only the diagonal terms, which leads to the following calculation of the *saliency* s_i for each weight w_i,

$$s_i = H_{ii} w^2{}_i / 2$$

Equation 26

To apply this method, a large network should be trained in the normal way and saliencies computed for each weight. Then the weights are ordered in terms of saliency and a percentage with the smaller saliences discarded. The network needs to be retrained using the previous values of the weights as the initial condition. The process can be repeated. We have covered the basic principles of topology selection and training of the MLP. We are ready now to apply MLPs to a real world problem.

7.3. Committee of networks

We saw that learning was a stochastic process, which means we should run the same network several times to make sure we get a good and stable training. We just saw that

even the topology of a neural network is a difficult decision because we have to take into consideration the generalization ability of the machine that is dependent upon its size.

A way of improving the performance of neural network classifiers is to use several networks of different sizes and characteristics to solve the same problem. Suppose we train C different networks in the same data. One temptation is to utilize the network that produced the best possible error in the training set. This strategy is not very good because it first wastes the training of all the other runs, and second because the best performer in the training set does not guarantee that this would extrapolate to the test set. A much better strategy is to use ALL the trained networks, i.e. making decisions with a committee of networks. Let us analyze what happens if we simply add their outputs, i.e.

$$y_{com} = \frac{1}{C}\sum_{i=1} y_i$$

Equation 27

If we assume that the errors from each network are zero mean and *uncorrelated* with each other, we can show (Perrone) that the error of the committee is

$$J_{com} = \frac{1}{C} J_i$$

Equation 28

which is a large reduction. This is optimistic in general since the errors among networks are not independent. The advantage comes exactly from a reduction to the variance of the error due to the averaging produced by the addition of each individual outputs. When we use networks to be used in committees we should use a network larger than the usual since we would like to improve the bias, since the variance will be reduced by the averaging produced by the committee.

Weight factors (proportional to how good each network is) can be used instead of the fixed weighting for even better results. This is easy to do using the backpropagation formalism. The goal is to train just the weights from each network to the output adder (or softmax). We will demonstrate committees in NeuroSolutions

NeuroSolutions 17

4.17 Committees

The idea of the committees is an interesting way out from the approach of putting "all the eggs in one basket", i.e. designing an unique neural network which we tweak for optimal performance. One can in principle use very different topologies (in fact the more distinct the better, since their performance will tend to be less correlated) in a more or less ad-hoc fashion to improve classification results. Of course the price paid is a large increase in the computational cost of the simulation. But these days with the increasing power of PCs and workstations this in fact just a little price to pay for increased performance. When taken to the limit this method of design sidetracks some of the difficulties we found in optimally selecting the topology, setting the number of PEs, the stopping criterion, and the training parameters. We rely on statistics to provide the improvement of all sub-optimal solutions, which is comparable to the best possible. This is only possible because the big difficulty in neural network performance is the variance of the estimation. So when used in committees the topologies should be larger than when they used stand alone (we would like to have small bias, because the committee improves the variance through averaging).

Here we will implement the training on-line of all the individual networks. But notice that each network has its own criterion to guarantee that each network is trained indepdendently from the others. We will use two MLPs (one hidden layer and 2 hidden layer) and a RBF network (it is another class of networks - more on this latter in Chapter V) to solve the male/female classification using the height and weight variables. If you recall we had about 5 errors when we solved this problem with the MLP in Chapter III. So this is the number to beat, but the real proof is to test the performnace in a test set.

We will start by fixing the vote of each network to 1/3. Notice that the performance is basically the same as a single network. Now we will also train the vote of each network. The performance improves dramatically from before, since

the number of mistakes decreases to 2 misclassifications. As you can see this is a powerful way to "throw" technology at the problem.

You should create a test set to validate the improvement of the committee with respect to a single network. Notice also that this method lends itself very nicely to parallelization since each network can be run in a separate machine and the results combined.

NeuroSolutions Example

Alternatively, we can simply train all the networks on-line, i.e. concurrently, as if they were a larger modular network made up of MLPs and the like. Effectively this is similar to train a much larger network and set some of its weights to zero, i.e. add some a priori structure. This configuration looks very similar to the committee, but notice that it is trained together, so the networks interact during training. Normally each will specialize in a given portion of the pattern space. The big advantage comes when the networks have different discriminant functions. But notice that this training is collaborative, not competitive (as we will exemplify later in Chapter VII). Modular networks can be easily implemented in NeuroSolutions, so you should modify the previous breadboard to create a modular network and compare results.

Go to next section

8. Project: Application of the MLP to real world data

In this section we are going to use MLPs to solve several real world classification problems. There are several URL sites that store very nice data for classification purposes. Please see the WEB sites in Chapter I, and also the following:

http://markov.stats.ox.ac.uk/pub/PRNN

http://128.2.209.79/afs/cs/project/ai-repository/ai/areas/neural/bench/0.html

http://www.scs.unr.edu/~cbmr/research/data.html

http://neural-server.aston.ac.uk/NN/databases.html

We will start with the crab data taken from the book by Campbell and Mahon , 1974. The goal is to classify rock crabs from two species as male or female utilizing anatomic measurements of front lip, rear width, length, width and depth. The data set is composed of 200 specimens, 50 males and 50 females from the two species.

We will then move to a more demanding data set - the Iris data set (source Fisher - to show the importance of the topology of the classifier in performance. The goal is to classify three types of Iris plants (Setosa, Versicolour, and Virginica) based on measurements of sepal length, sepal width, petal length, and petal width (all in cm). There are 50 samples of each class for a total of 150 samples. As you can see this problem formulation is very similar to the previous, however the distribution of the data clusters in the input space is more complex. The difficulty of the task is not knwon a priori so a step by step approach to define the optimal classifier is normally required.

NeuroSolutions 18

4.18 Crab data classification

The first thing to do is to examine the data sets. The data is composed of 5 parameters per crab, and one tag for the species for a total of 6 inputs. The desired result is a classification of male or female, so we can either use one or two outputs. In the first case we would code male as 1 and female as 0 (or vice-versa). In the second case each gender will have its own output. In this case the desired file is composed of two columns with pairs of values (0, 1) depending upon the class membership.

The next step is to normalize and divided the data. Normalization of the data allow us to get experience with stepsizes and use systematic weight initializations. Since the data is all positive, we will normalize it between [0,1]. We will divide the data in training and test sets (and validation set). The training will be used to arrive at

optimal weights. The test data is used to gauge the performance of the classifier, and the validation set to help us stop the training at the point of best generalization. A good rule of thumb is to use 2/3 (66%) of the data for training and 1/3 (33%) for testing.

The next important definition is the topology. The idea is always to start small, so we should star with a perceptron to see if the machine can separate the data. Since the data is normalized between [0,1] let us use the logistic function as the nonlinearity.

The next important decision is how to stop the training. Here we can use first the number of iterations (I=200) to get a feel for how difficult is the problem, and then when the topology is finally chosen switch to crossvalidation.

We still have to decide on the search method and the criterion. The search as we said should be momentum and the criterion L2.

We are ready to run the network on the crab data. Do it several times and watch the learning curve. If the network converges to the same small error in basically the same number of epochs than the training is successful and we can worry on how to stop the training with crossvalidation. If not we should select a different learning rate, search procedure, and/or modify the topology.

The error is around 0.1 so it means that almost all the data was classified correctly in the training set. We also see that the training is repeatable, since the learning curves are very similar to each other. Let us go to the test set to see the performance. As we mentioned in the test, the best way to test the classifier is not by MSE but with the confusion matrix. We see that we get 3 errors in the test set which is reasonable. But can we improve the performance?

In order to answer this question we have to change the topology and try a one hidden layer network with 5 hidden PEs. Let us train the network and see what is the performance. As you can see the error is much smaller. Let us test the system.

The confusion matrix shows that the classification is perfect. In this case it is questionable if the data is in fact linearly separable or not, since we do not have enough data points. The errors could be due to imprecise measurements (noise in the data).

We can now make sure that we fine tune the system. We can first train it with the crossvalidation and see up to where we should train. The crossvalidation for this problem/topology does not help since there is no observable increase in the learning curve.

An alternative to cross validation is to use the weight decay on the weights. We start with the same network but we use the idea of driving to zero all the weights. Training the network we can see that the network size was slightly reduced but our initial guess was pretty good.

NeuroSolutions Example

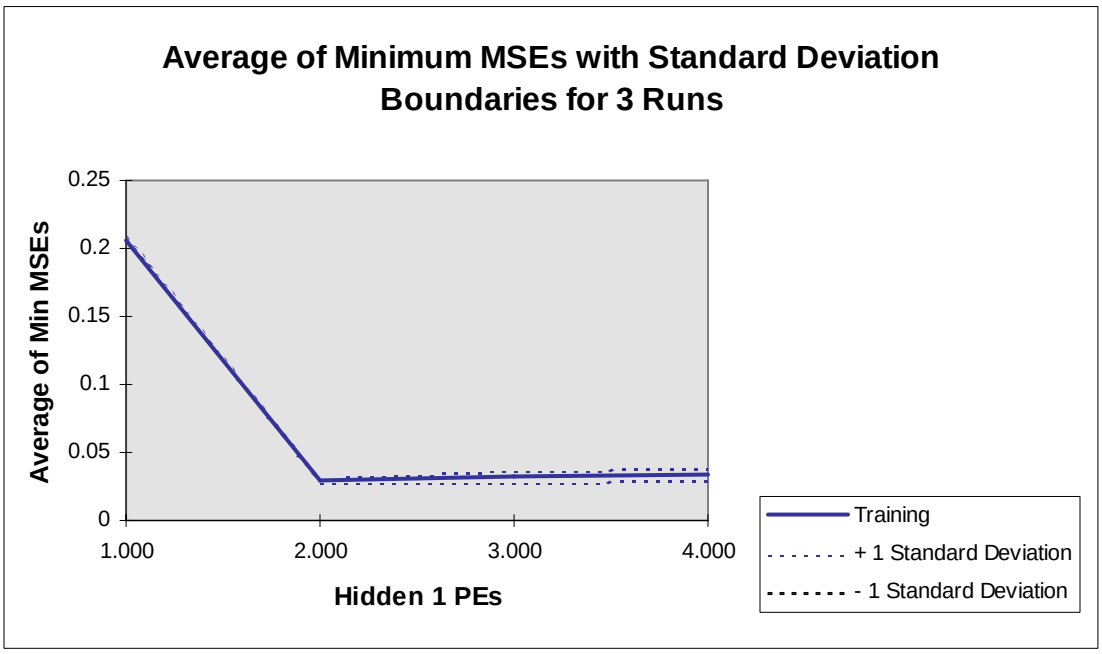
NeuroSolutions 19

4.19. Iris data classification

In order to see how data dependent is the performance of a classifier, let us change the data set to the iris data. The inputs to the network are sepal length, sepal width, petal length, and petal width (i.e. 4 inputs) and the three classes of Iris plants (which are used as the desired outputs) are Setosa, Versicolour, and Virginica. There are 50 samples of each class for a total of 150 samples. The 150 samples have been pre-randomized. Furthermore, 100 samples will be used for "Training" and 50 samples for "Testing". This data is contained within the worksheet named "Iris Data Randomized".

Starting with the perceptron we see that the result is not satisfactory. There is a large confusion between the classes of Virginica and Versicolour. So we need to go to a one hidden layer MLP. The issue is how to choose the number of hidden PEs.

Let us run the MLP first with one PE in the hidden layer for several runs and record the minimum MSE value. Then increase this number to 2, 4, 6 and run the network several times for each. Finally we should plot the mean MSE as a function of the number of PEs. The MSE error curve stabilizes after 2 PEs so we will put 2 PEs in the hidden layer.



The classification performance for this network is much improved (only 3 mistakes instead of 19). We can then conclude that the data set is not linearly separable since the perceptron could not solve it but the one hidden layer MLP can.

NeuroSolutions Example

Neural Networks for decision making

The next problem we will discuss is an interesting one. Assume that we want to create a computer aided diagnostic tool for breast cancer using a neural network. Ten features (radius, texture, perimeter, area, smoothness, compactness, concavity, concave points, symmetry, and fractal dimension) have been computed from a digitized image of a fine needle aspirate of a breast mass. The inputs to the neural network model consist of the mean, standard error, and "worst" (mean of the three largest values) for each of these 10 features resulting in 30 total inputs for each image. This data is contained within the file

named “Breast Cancer Data”. We have only data from 150 images. A two hidden layer MLP configured with 10 inputs and 2 outputs will be used as the neural network model.

The difficulty in this application is that we want to estimate a decision (sick or healthy) based on the outcome of the neural network, i.e., we have to estimate the “probability” of a decision. How can a neural network do that? We saw that the MLP can be trained to give us the *a posteriori* probability of a class given the particular data example. The a *posteriori* probability interpretation means that when the net output for class 1 is 0.9 this can be interpreted as saying that the a posteriori probability of that example belong to class 1 is 0.9. This is an important step, but from here to a diagnostic many other things must be taken into consideration.

First, we used 50% of sick and 50% of normals to train the system, but the ratio of sick to normals in the population is (fortunately) very different from this. So we have to compensate for the a priori probabilities when we interpret the results. If you recall from Chapter II (Figure 2), when the a priori probabilities for each class are the same, the decision that minimizes the probability of misclassification is directly given by the a posteriori probability. In general, when the *a priori* probabilities are different they multiply the corresponding likelihoods, so the decision boundary is moved proportionally to the ratio of the logarithm of the *a priori* probabilities.

The advantage of ANNs is that *we do not need to retrain the network*. We can still interpret the outputs as likelihoods, and in order to obtain the *a posteriori* probability simply multiply the outputs by the priors (Bayes theorem) when the training set is constructed with equal number of exemplars from each class. For example, suppose that the net output for the class sick was 0.9 and we know that the probability of being sick in the population is 0.2, and equal number of cases were used in the training set. So the probability is 0.18 that the subject has the disease. If the training set is not formed equally by the two classes, then we should also divide the outputs by the relative frequency of training exemplars in the respective class.

The other difficulty is that there is a risk in making a decision, and the two types of errors (the subject is normal and the network says sick - called a *false positive*; or the subject is sick and the net says normal - called a *false negative*) have different costs. It is preferable to initially tell a normal patient that she needs to make further exams than simply state that she is normal when in fact she has breast cancer since the implications are vastly different.

This means that minimizing the probability of misclassification may not be the best strategy. *We should weight the a posteriori probabilities by the risk of making the decisions.* A matrix of penalties L_{ij} of making the wrong decisions must first be constructed. Normally in medicine this is rather subjective, but the idea is the following. We would like to penalize much more the false negatives (calling a sick subject normal) than the false positives (calling a normal patient sick). The penalties are numbers between 0 and 1, where the penalty for being correct is 0, while the penalty to being wrong is 1. Let us call sick the hypothesis 1, and healthy the hypothesis 2. Here let us say that the penalty of the false negative is L12=0.3 and the penalty of the false positive is L21=0.5. So the matrix of penalties is

$$L = \begin{bmatrix} 0 & 0.3 \\ 0.5 & 0 \end{bmatrix}$$

We compute the average penalty by

$$R_i = \sum_{j=1}^{c} L_{ij}\, p(x_j \mid C_i)$$

where *c* is the number of classes and *x* is the random variable. We call *risk* the expected value of the penalties, i.e.

$$R = \sum_{i=1}^{c} R_i\, P(C_i)$$

where P(Ci) are the *a priori* probabilities. So the best decision should be done by minimizing the risk (instead of minimizing the probability of misclassification as we did when we applied the Bayes rule), as

$$\sum_{i=1}^{c} L_{ik}\, p(x|C_i)P(C_i) < \sum_{i=1}^{c} L_{ij}\, p(x|C_i)P(C_i) \qquad for\ all \quad k \neq j$$

Notice that when the network provides $p(x|C_i)$ it is trivial to make the decision based on risks.

NeuroSolutions 20

4.20 Risk decision in the Cancer data

In order to solve this problem we will start with a MLP and we should make sure that the network is well configured (try several number of hidden PEs as we did above) and it is well trained. So we should use crossvalidation to stop at the point of maximum generalization. We should also try different initial conditions to make sure that we are really converging to the absolute minimum.

Then the next step is to create a DLL that will compute the risk based on the above formulas. The computer aided diagnostic is then made based on the class that produces the smallest risk.

NeuroSolutions Example

Go to next section

9. Conclusion

In this Chapter we covered the most fundamental practical aspects of applying MLPs to real world problems. We learned about new search procedures, how to control the initialization, the stopping criterion, and how to effectively decide if a reasonably solution was obtained (the confusion matrix).

We also discussed ways to build more flexibility into the solution in the form of different norms. One of the fundamental problems of MLPs (and other learning machines) is that they can not control their generalization ability. This is crucial for good results, so we

present a method (weight decay) to set to zero unnecessary weights and so provide tighter topologies that tends to generalize better. We also present the idea of committees of networks as a form to decrease the variance of performance in the test set. We end the chapter with the application of MLPs to several interesting and real world problems.

List of NeuroSolutions Examples

4.1. Visualization of Learning

4.2. Learning as a stochastic process (XOR)

4.3. Learning rate scheduling

4.4. Flow of errors across MLP layers

4.5. Effect of initial conditions on adaptation

4.6. Momentum Learning

4.7 Adaptive Stepsizes

4.8. Adaptation with noise in the desired signal

4.9. Stopping based on MSE value

4.10 Stopping with Crossvalidation

4.11 Sparse connectivity in the input layer

4.12 Confusion matrix for classification performance

4.13 Regression performance as a function of the norm

4.14 Classification performance as a function of the norm

4.15 Cross-entropy training

4.16. Weight-decay

4.17 Committees

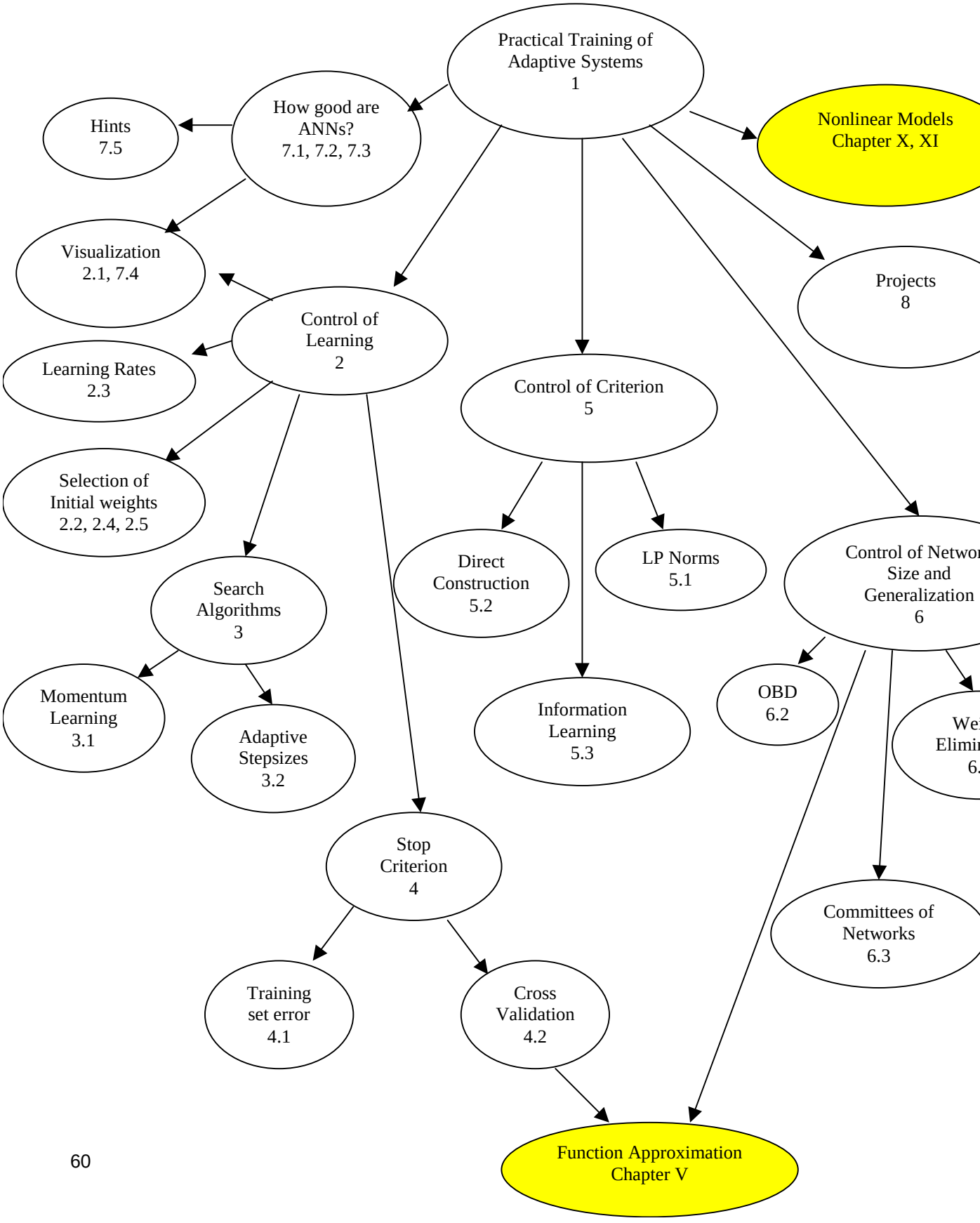
4.18 Crab data classification

4.19. Iris data classification

4.20 Risk decision in the Cancer data

Concept Maps for Chapter IV

Chapter IV


Practical Training of Adaptive Systems 1
How good are ANNs? 7.1, 7.2, 7.3
Hints 7.5
Nonlinear Models Chapter X, XI
Visualization 2.1, 7.4
Projects 8
Control of Learning 2
Learning Rates 2.3
Control of Criterion 5
Selection of Initial weights 2.2, 2.4, 2.5
Direct Construction 5.2
LP Norms 5.1
Control of Networ Size and Generalization 6
Search Algorithms 3
Momentum Learning 3.1
Adaptive Stepsizes 3.2
Information Learning 5.3
OBD 6.2
Stop Criterion 4
Committees of Networks 6.3
Training set error 4.1
Cross Validation 4.2
Function Approximation Chapter V

Go to Next Chapter
Go to the Table of Contents

algorithm locality and distributed systems

One of the appealing issues of gradient descent learning is the locality of the computation. Recall that any of the LMS, delta rule and backpropagation use only local quantities available at the weight to perform its adaptation. This is crucial for the efficiency of the algorithms in particular for distributed system implementations. If the algorithm was not local, then the necessary variables would have to be fetched from intermediate storage or across the neural network. This means that there would be lots of overhead for algorithm implementation, and more importantly, that a centralize control was necessary. Adaptation would loose the biological appeal that locality provides.

Complexity imposes stringent constraints in the system design. One of the luxuries that the designer must give up is centralize control, because otherwise most of the system resources will be eaten up by the control, not to perform the required function. This is the beauty of distributed systems with local rules of interactions: they do not require centralized control. They may potentially be the only way to break the barrier of system complexity as our own brain exemplifies.

Going back to the learning rules, lets for a minute appreciate the formidable task of moving a distributed system with thousands of parameters to its optimum parameter set. There is no one that is orchestrating the change in weights. Each weight receives an error and an activation and independently of all the other weights, changes its own value using only two multiplications. Overall the system approaches the optimum operating point, but we can not finger where the control is.

This is a great model to construct optimal complex systems.

Return to text

system identification versus modeling

Not all the problems of interest can be formulated in terms of an external, input-output constraint. But the class of these problems is very large indeed, ranging from system identification, prediction, classification, etc. For instance, when we are predicting the stock market, we are not interested in preserving the individual components of the economy, we are simply interested in a small prediction error.

On the other hand, there are problems that require more than an input-output relation, such as modeling. In modeling we are interested not only in good predictions but also in preserving some (or all) of the underlying features of the phenomenon we are interested in. Examples of modeling appear in the physical and biological sciences and in engineering when applied to these problems.

Return to text

good initial weight values

Very little work has been done in good initial weight values for training, since the problem is very difficult to formulate. Intuitively we can see that if the initial weights can be chosen close to the optimal, then search will be fast and reliable. Unfortunately, we do not know where the optimal weights are in pattern space (this is the reason we are adapting the system…). And faster and more reliable methods to find the optimum are unknown. Some work has been done proposing the linear solution as the initial weight values, but this does not always work. In many cases the linear solution is exactly the point that search should avoid because it is a strong local minimum.

Return to Text

Minskowski measures

An alternate interpretation of the error norms is provided in the statistical literature as *Minskowski measures*. In this perspective, the L_2 norm appears as the maximum likelihood solution when the instantaneous errors are Gaussian distributed. When the data set is such that they produce error *pdf*s that deviates from the Gaussian distribution, the L_2 norm does not provide the maximum likelihood solution. So, if one has a priori information about the error distributions, the appropriate Minskowski measure can be used to establish the best performance.

In classification problems the most reasonable distribution seems to be the *Bernoulli* distribution, which points to the use of the Kullback-Leibler criterion as we will see.

Return to Text

cross entropy criterion

In order to understand cross-entropy we have to give a short definition of what is information and the related concept of entropy. Information theory was invented by Shannon in the late 40's to explain the content of messages and how they are corrupted through communication channels. The key concept in information theory is that of information. Information is a measure of randomness of a message. If the message is totally predictable, it contains no information. On the other hand something very unexpected has a high information content. So the concept is inversely associated with the probability of an event. We can define the amount of information in a random event xk as

$$I(x_k) = \log(\frac{1}{p(x_k)})$$

Entropy then becomes the mean value of I(x) over the complete range of discrete

messages (2K+1) with probabilities pk as

$$H(x) = \sum_{k=-K}^{K} p_k I(x_k)$$

Entropy is a measure of the average amount of information contained in the event.

Now assume that we have two probability distributions {pk} and {qk}. The relative entropy of the probability distribution P (function of some event r) with respect to the second distribution Q is given by

$$L = \sum_r P_r \log(\frac{P_r}{Q_r})$$

This concept called relative entropy was introduced by Kullback, and is also called Kullback-Leibler information criterion Cover .

Return to Text

early stopping and model complexity

We saw before in section 4.2 that early stopping provides a criterion to stop training at the point of smallest error in the validation set, i.e. the point of best generalization for that particular combination topology/training set. From the point of view of model complexity, *early stopping is effectively controlling the complexity of the model*, which may seem strange since the number of free parameters is constant. It turns out that in nonlinear systems the model complexity depends not only on the number of parameters (as in linear systems) but also on the actual value of the free parameters, so it may change during training.

Early stopping does not address the size of the learning machine, which is also a determining factor to control the model complexity. In linear learning machines, model size turns out to be the only way to control model complexity. So parsimonious architectures should be a design goal. The full discussion of this topic is rather theoretical

and will be left to Chapter V. Here we will use a heuristic approach.

Return to text

learning rate annealing

is the progressive decrease of the learning rate across iterations.

shallow networks

are networks with few layers, i.e. typically one of two hidden layers.

Eq.6

$$N > \frac{W}{\delta}$$

outliers

erroneous samples produced by observation noise.

Eq.8

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_k} \frac{\partial y_k}{\partial w_{ij}}$$

activation

is the PE output in the MLP topology.

dual

or transpose network is the network obtained from the original MLP by reversing the signal flow, and changing summing junctions with splitting nodes and vice-versa.

fan-in

is the number of inputs that feed a given PE.

Simon Haykin

Artificial Neural Networks: A Comprehensive Foundation, IEEE Press 1995.

nonconvex

a surface is nonconvex when it has multiple minima.

confusion matrix

is a matrix where the man made (rows) and the machine classification (columns) per class is entered. A perfect classifier will have only the diagonal populated. Errors appear in non-diagonal positions. The confusion matrix is an efficient way to observe the confusion between classes.

generalization

is the ability to correctly classify samples unknown to the learning machine

Vladimir Vapnik

The Nature of Statistical Learning Theory, Springer Verlag, 1995.

Barron

Universal approximation bounds for superpositions of sigmoid functions, IEEE Trans. Information Theory 39, #3, 930-945, 1993.

saliency

is the importance of a weight for the overall input-output map

Hessian

is the matrix of the second derivative of the cost with respect to the weights

committees

are ensembles of ANNs trained in the same data, eventually with different topologies, whose output is interpreted as a vote for the classification. Committees have the appeal that they decrease the variance of the final decision which is considered one of the most severe problems in semi-parametric classification.

simulated annealing

is a global search criterion where the space is searched with a random rule. In the beginning the variance of the random jumps is very large. Every so often the variance is decreased and a more local search is undertaken. It has been shown that if the decrease

of the variance is set appropriately, then the global optimum can be found with probability one. The method is called simulated annealing because it is similar to the annealing process of creating crystals from a hot liquid). See van Laarhoven and Aarts, Simmulated Annealing: theory and applications, Kluwer, 1988.

first order

Gradient descent is called a first order method because it only uses information of the performance surface tangent space. Newton is a second order method because it uses information of the curvature.

validation

Validation set is the ensemble of samples that will be used to validate the parameters used in the training (not to be confused with the test set, which assesses the performance of the classifier).

classification error

The classification error is the number of samples that were incorrectly classified -misclassifications- normalized by the total number of samples

robust

A method is robust if it is not very sensitive to outliers. The term was coined in statistics and control theory to represent methods that have low sensitivity to perturbations.

Occam

William of Occam was a monk that lived in the XIV century England and enunciated a principle that has echoed across the scientific circles for centuries: He said that a scientific model should favor simplicity. Hence the name for the principle: Occam’s razor (shave-off the fat in the model) .

VC dimension

Stands for Vapnik-Chervonenkis dimension and measures the capacity of a learning machine . See Vapnik's book, The nature of statistical learning theory, Springer, 1995.

Genetic Algorithms

are global search procedures proposed by John Holland that search the performance surface concentrating on the areas that provide better solutions. They use "generations" of search points computed from the previous search points using the operators of crossover and mutation (hence the name). See Goldberg, Genetic Algorithms in search, optimization and machine learning, Addison Wesley, 1989.

Luenberger

Linear and Nonlinear Programming, Addison Wesley, 1986

Scott Fahlman

Fast learning variations of backpropagation: an empirical study, Proc. 1988 Connectionist Models Summer School, 38-51, Morgan Kaufmann.

Campbell

Campbell N., Mahon R., A multivariate study of variation in two species of rock crabs of genus Leptograpsus", Australian Journal of Zoology, 22, 417-425, 1974.

R. A. Fisher

The use of multiple measurements in taxonomic problems, Annal Eugenics 7, Pt II, 179-188, 1936.

line search methods

A principled approach to find the optimal stepsize at each iteration is to minimize $J(w_k + \mu_k s_k)$ with respect to μ, where sk represents the direction of the search at sample k. For quadratic surfaces, there is an analytic solution given by

$$\mu_k = \frac{-\nabla J_k^T s_k}{s_k^T H_k s_k}$$ **Equation 29**

where H is the Hessian. Since the Hessian for quadratic surfaces is independent of the point, this expression needs to be computed just once. We can prove that the algorithm zig-zags to the minimum (orthogonal directions) with this choice of the stepsize. Note that at each iteration we need to perform vector computations, which makes the adaptation computationally heavier.

In general the optimal value of m must be solved by line search since the performance surfaces are not quadratic. This gives rize to the conjugate gradient algorithms.

Return to Text

Bishop

Neural Networks for Pattern Recognition, Oxford, 1995, pp 347.

Eq. 24

$$\mu_k = \frac{-\nabla J_k^T s_k}{s_k^T H_k s_k}$$

Fletcher

Practical Methods of Optimization, John Wiley, 1987

Horst, Pardalos and Thoai

Introduction to Global Optimization, Kluwer, 1995

Shepherd

Second-order methods for neural networks, Springer, 1997.

Pearlmutter

Fast exact multiplications by the Hessian, Neural Computation 6 (1), 147-160, 1994

Hertz, Krogh, Palmer

Introduction to the theory of neural computation, Addison_Wesley, 1991

LeCun, Denker and Solla

Optimal Brain Damage, Advances in Neural Information Processing Systems, vol 2,

598-605, 1993.

Perrone

General averaging results for convex optimization, Mozer et al (Ed), Proc. 1993 Connectionist Models Summer School, 364-371, Lawrence Erlbaum, 1994.

Cover

Elements of Information Theory, Wiley, 1991.

LeCun, Simard, Pearlmutter

Automatic learning rate maximization by on-line estimation of the Hessian eigenvectors, in Advances of Neural Information Processing Systems, vol 5, 1556-163, 1993, Morgan Kaufmann.

Silva e Almeida

Acceleration techniques for the backpropagation algorithm, Almeida and Wellekens (Ed), Neural Networks, Lecture notes in Computer Science, 110-119, 1990, Springer.

Almeida's adaptive stepsize

The idea is very close to the adaptive stepsize algorithm described above, but it is more stable due to the inclusion of nonlinearity. As before there will be a different stepsize per weight. The reasoning is as follows:

1- If the gradient component has the same size in two consecutive iterations, the stepsize should be increased.

2- 2- If the gradient alternates sign, the stepsize should be decreased.

So the weight update is

$$w_{kj}(n) = w_{kj}(n-1) + \eta_{kj}(n)\nabla_{kj}C(n)$$

where $\nabla_{kj}C(n)$ is each gradient component, and at each iteration

$$\eta(n) = \begin{cases} u\eta_{kj}(n-1) & if \quad \nabla_{kj}C(n)\nabla_{kj}C(n-1) > 0 \\ d\eta_{kj}(n-1) & if \quad \nabla_{kj}C(n)\nabla_{kj}C(n-1) < 0 \end{cases}$$

Equation 30

where u and d are positive constants with values slightly above and below unity, respectively. They suggest to make $d \approx 1/u$ .The initial value of each stepsize is set the same for all weights. Note that unlike the delta-bar-delta, here the update of the stepsize is geometric in both directions (decrease or increase).

There are 3 heuristics to help control the growth of the stepsize:

1- The error obtained at each iteration should be compared with the previous error. The new weight vector is accepted only if the new error is at most larger than the previous by 1 or 2.5%.

2- If the new error is higher than the previous by more than this margin, then compute a new stepsize as Eq. 30, but apply it to the old weights.

3- If #2 does not decrease the error in 2 to 3 iterations, then reduce all the stepsizes by a constant factor α, until the error starts to decrease again.

4- At this point restart with the normal adaptation of stepsizes.

This method works well when the gradient is known with high precision as in batch learning. The method can be also applied to on-line adaptation, but we can not utilize the instantaneous estimates of the gradient since they are too noisy. The authors propose to keep a running estimate of the gradient in each epoch (P is the number of training set exemplars)

$$\nabla_{kj}C(n) = \sum_{p=1}^{P} \frac{\partial J(e_p)}{\partial w_{kj}}$$

to be used in the adaptive stepsize computation.

Return to text

Index