Multilayer Perceptrons in ANNs

R.K.Sharma*, Deepak Garg**, Amit Bhardwaj***
***Head, Computer Center**
****Computer Science & Engineering Department**
*****Electronics & Communication Engineering Department**
Thapar Institute of Engineering & technology
{rksharma,dgarg,abhardwaj} @mail.tiet.ac.in

Abstract

This is perhaps the most popular network architecture in use today. This is the type of network discussed briefly in previous sections: the units each perform a biased weighted sum of their inputs and pass this activation level through a transfer function to produce their output, and the units are arranged in a layered feedforward topology. The network thus has a simple interpretation as a form of input-output model, with the weights and thresholds (biases) the free parameters of the model. Such networks can model functions of almost arbitrary complexity, with the number of layers, and the number of units in each layer, determining the function complexity. Important issues in Multilayer Perceptrons (MLP) design include specification of the number of hidden layers and the number of units in these layers.

1. Introduction

The number of input and output units is defined by the problem (there may be some uncertainty about precisely which inputs to use, a point to which we will return later. However, for the moment we will assume that the input variables are intuitively selected and are all meaningful). The number of hidden units to use is far from clear. As good a starting point as any is to use one hidden layer, with the number of units equal to half the sum of the number of input and output units.

1.1 Training Multilayer Perceptrons

Once the number of layers, and number of units in each layer, has been selected, the network's weights and thresholds must be set so as to minimize the prediction error made by the network. This is the role of the *training algorithms*. The historical cases that you have gathered are used to automatically adjust the weights and thresholds in order to minimize this error. This process is equivalent to fitting the model represented by the network to the training data available. The error of a particular configuration of the network can be determined by running all the training cases through the network, comparing the actual output generated with the desired or target outputs. The differences are combined together by an *error function* to give the network error. The most common error functions are the *sum squared error* (used for regression problems), where the individual errors of output units on each case are squared and summed together, and the cross entropy functions (used for maximum likelihood classification).

In traditional modeling approaches (e.g., linear modeling) it is possible to algorithmically determine the model configuration that absolutely minimizes this error. The price paid for the greater (non-linear) modeling power of neural networks is that although we can adjust a network to lower its error, we can never be sure that the error could not be lower still.

A helpful concept here is the error surface. Each of the N weights and thresholds of the network (i.e., the free parameters of the model) is taken to be a dimension in space. The $N+1$th dimension is the network error. For any possible configuration of weights the error can be plotted in the $N+1$th dimension, forming an *error surface*. The objective of network training is to find the lowest point in this many-dimensional surface.

In a linear model with *sum squared error* function, this error surface is a parabola (a quadratic), which means that it is a smooth bowl-shape with a single minimum. It is therefore "easy" to locate the minimum.

Neural network error surfaces are much more complex, and are characterized by a number of unhelpful features, such as local minima (which are lower than the surrounding terrain, but above the global minimum), flat-spots and plateaus, saddle-points, and long narrow ravines.

It is not possible to analytically determine where the global minimum of the error surface is, and so neural network training is essentially an exploration of the error surface. From an initially random configuration of weights and thresholds (i.e., a random point on the error surface), the training algorithms incrementally seek for the global minimum. Typically, the gradient (slope) of the error surface is calculated at the current point, and used to make a downhill move. Eventually, the algorithm stops in a low point, which may be a local minimum (but hopefully is the global minimum).

2. The Back Propagation Algorithm

The best-known example of a neural network training algorithm is *back propagation*. Modern second-order algorithms such as *conjugate gradient descent* and *Levenberg-Marquardt* (both included in *ST Neural Networks*) are substantially faster (e.g., an order of magnitude faster) for many problems, but *back propagation* still has advantages in some circumstances, and is the easiest algorithm to understand. We will introduce this now, and discuss the more advanced algorithms later. There are also heuristic modifications of *back propagation* which work well for some problem domains, such as *quick propagation* (Fahlman, 1988) and *Delta-Bar-Delta* (Jacobs, 1988) and are also included in *ST Neural Networks*.

In *back propagation*, the gradient vector of the error surface is calculated. This vector points along the line of steepest descent from the current point, so we know that if we move along it a "short" distance, we will decrease the error. A sequence of such moves (slowing as we near the bottom) will eventually find a minimum of some sort. The difficult part is to decide how large the steps should be.

Large steps may converge more quickly, but may also overstep the solution or (if the error surface is very eccentric) go off in the wrong direction. A classic example of this in neural network training is where the algorithm progresses very slowly along a steep, narrow, valley, bouncing from one side across to the other. In contrast, very small steps may go in the correct direction, but they also require a large number of iterations. In practice, the step size is proportional to the slope (so that the algorithms settles down in a minimum) and to a special constant: the *learning rate*. The correct setting for the learning rate is application-dependent, and is typically chosen by experiment; it may also be time-varying, getting smaller as the algorithm progresses.

The algorithm is also usually modified by inclusion of a momentum term: this encourages movement in a fixed direction, so that if several steps are taken in the same direction, the algorithm "picks up speed", which gives it the ability to (sometimes) escape local minimum, and also to move rapidly over flat spots and plateaus.

The algorithm therefore progresses iteratively, through a number of *epochs*. On each epoch, the training cases are each submitted in turn to the network, and target and actual outputs compared and the error calculated. This error, together with the error surface gradient, is used to adjust the weights, and then the process repeats. The initial network configuration is random, and training stops when a given number of epochs elapses, or when the error reaches an acceptable level, or when the error stops improving (you can select which of these stopping conditions to use).

3. Over-learning and Generalization

One major problem with the approach outlined above is that it doesn't actually minimize the error that we are really interested in - which is the expected error the network will make when *new* cases are submitted to it. In other words, the most desirable property of a network is its ability to *generalize* to new cases. In reality, the network is trained to minimize the error on the training set, and short of having a perfect and infinitely large training set, this is not the same thing as minimizing the error on the real error surface - the error surface of the underlying and unknown model.

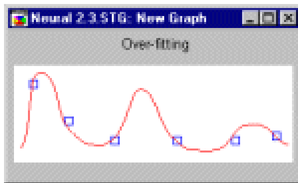
The most important manifestation of this distinction is the problem of over-learning, or over-fitting. It is easiest to demonstrate this concept using polynomial curve fitting rather

than neural networks, but the concept is precisely the same.

A polynomial is an equation with terms containing only constants and powers of the variables. For example:

$y=2x+3$
$y=3x^2+4x+1$

Different polynomials have different shapes, with larger powers (and therefore larger numbers of terms) having steadily more eccentric shapes. Given a set of data, we may want to fit a polynomial curve (i.e., a model) to explain the data. The data is probably noisy, so we don't necessarily expect the best model to pass exactly through all the points. A low-order polynomial may not be sufficiently flexible to fit close to the points, whereas a high-order polynomial is actually too flexible, fitting the data exactly by adopting a highly eccentric shape that is actually unrelated to the underlying function.



Neural networks have precisely the same problem. A network with more weights models a more complex function, and is therefore prone to over-fitting. A network with less weights may not be sufficiently powerful to model the underlying function. For example, a network with no hidden layers actually models a simple linear function.

How then can we select the right complexity of network? A larger network will almost invariably achieve a lower error eventually, but this may indicate over-fitting rather than good modeling.

The answer is to check progress against an independent data set, the selection set. Some of the cases are reserved, and not actually used for training in the *back propagation* algorithm. Instead, they are used to keep an independent check on the progress of the algorithm. It is invariably the case that the initial performance of the network on training and selection sets is the same (if it is not at least approximately the same, the division of cases between the two sets is probably biased). As training progresses, the training error naturally drops, and providing training is minimizing the true error function, the selection error drops too. However, if the selection error stops dropping, or indeed starts to rise, this indicates that the network is starting to overfit the data, and training should cease. When over-fitting occurs during the training process like this, it is called over-learning. In this case, it is usually advisable to decrease the number of hidden units and/or hidden layers, as the network is over-powerful for the problem at hand. In contrast, if the network is not sufficiently powerful to model the underlying function, over-learning is not likely to occur, and neither training nor selection errors will drop to a satisfactory level.

The problems associated with local minima, and decisions over the size of network to use, imply that using a neural network typically involves experimenting with a large number of different networks, probably training each one a number of times (to avoid being fooled by local minima), and observing individual performances. The key guide to performance here is the selection error. However, following the standard scientific precept that, all else being equal, a simple model is always preferable to a complex model, you can also select a smaller network in preference to a larger one with a negligible improvement in selection error.

A problem with this approach of repeated experimentation is that the selection set plays a key role in selecting the model, which means that it is actually part of the training process. Its reliability as an independent guide to performance of the model is therefore compromised - with sufficient experiments, you may just hit upon a lucky network that happens to perform well on the selection set. To add confidence in the performance of the final model, it is therefore normal practice (at least where the volume of training data allows it) to reserve a third set of cases - the test set. The final model is tested with the test set data, to ensure that the results on the selection and training set are real, and not artifacts of the training process. Of course, to fulfill this role properly the test set should be used only once - if it is in turn used to adjust and reiterate the training process, it effectively becomes selection data!

This division into multiple subsets is very unfortunate, given that we usually have less data than we would ideally desire even for a single subset. We can get around this problem by resampling. Experiments can be conducted using different divisions of the available data into training, selection, and test sets. There are a number of approaches to this subset, including random (monte-carlo) resampling, cross-validation, and bootstrap. If we make design decisions, such as the best configuration of neural network to use, based upon a number of experiments with different subset examples, the results will be much more reliable. We can then either use those experiments solely to guide the decision as to which network types to use, and train such networks from scratch with new samples (this removes any sampling bias); or, we can retain the best networks found during the sampling process, but average their results in an emsemble, which at least mitigates the sampling bias.

To summarize, network design (once the input variables have been selected) follows a number of stages:

- Select an initial configuration (typically, one hidden layer with the number of hidden units set to half the sum of the number of input and output units).
- Iteratively conduct a number of experiments with each configuration, retaining the best network (in terms of selection error) found. A number of experiments are required with each configuration to avoid being fooled if training locates a local minimum, and it is also best to resample.
- On each experiment, if under-learning occurs (the network doesn't achieve an acceptable performance level) try adding more neurons to the hidden layer(s). If this doesn't help, try adding an extra hidden layer.
- If over-learning occurs (selection error starts to rise) try removing hidden units (and possibly layers).
- Once you have experimentally determined an effective configuration for your networks, resample and generate new networks with that configuration.

4. Data Selection

All the above stages rely on a key assumption. Specifically, the training, verification and test data must be representative of the underlying model (and, further, the three sets must be independently representative). The old computer science adage "garbage in, garbage out" could not apply more strongly than in neural modeling. If training data is not representative, than the model's worth is at best compromised. At worst, it may be useless. It is worth spelling out the kind of problems which can corrupt a training set:

The future is not the past. Training data is typically historical. If circumstances have changed, relationships which held in the past may no longer hold.

All eventualities must be covered. A neural network can only learn from cases that are present. If people with incomes over $100,000 per year are a bad credit risk, and your training data includes nobody over $40,000 per year, you cannot expect it to make a correct decision when it encounters one of the previously-unseen cases. Extrapolation is dangerous with any model, but some types of neural network may make particularly poor predictions in such circumstances.

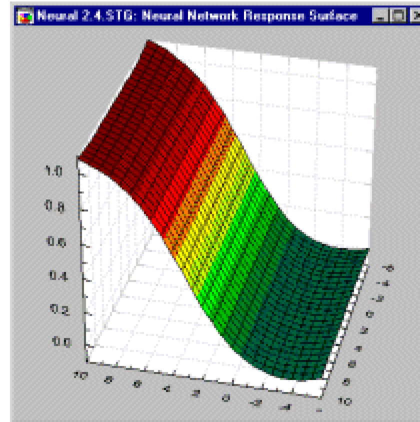
A network learns the easiest features it can. A classic (possibly apocryphal) illustration of this is a vision project designed to automatically recognize tanks. A network is trained on a hundred pictures including tanks, and a hundred not. It achieves a perfect 100% score. When tested on new data, it proves hopeless. The reason? The pictures of tanks are taken on dark, rainy days; the pictures without on sunny days. The network learns to distinguish the (trivial matter of) differences in overall light intensity. To work, the network would need training cases including all weather and lighting conditions under which it is expected to operate - not to mention all types of terrain, angles of shot, distances...

Unbalanced data sets. Since a network minimizes an overall error, the proportion of types of data in the set is critical. A network trained on a data set with 900 good cases and 100 bad will bias its decision towards good cases, as this allows the algorithm to lower the

overall error (which is much more heavily influenced by the good cases). If the representation of good and bad cases is different in the real population, the network's decisions may be wrong. A good example would be disease diagnosis. Perhaps 90% of patients routinely tested are clear of a disease. A network is trained on an available data set with a 90/10 split. It is then used in diagnosis on patients complaining of specific problems, where the likelihood of disease is 50/50. The network will react over-cautiously and fail to recognize disease in some unhealthy patients. In contrast, if trained on the "complainants" data, and then tested on "routine" data, the network may raise a high number of false positives. In such circumstances, the data set may need to be crafted to take account of the distribution of data (e.g., you could replicate the less numerous cases, or remove some of the numerous cases), or the network's decisions modified by the inclusion of a *loss matrix*. Often, the best approach is to ensure even representation of different cases, then to interpret the network's decisions accordingly.

Insights into MLP Training

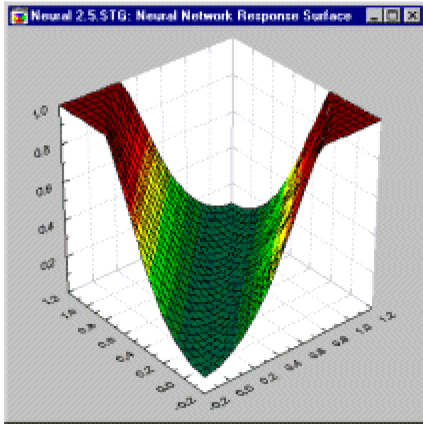
More key insights into MLP behavior and training can be gained by considering the type of functions they model. Recall that the activation level of a unit is the weighted sum of the inputs, plus a threshold value. This implies that the activation level is actually a simple linear function of the inputs. The activation is then passed through a sigmoid (S-shaped) curve. The combination of the multi-dimensional linear function and the one-dimensional sigmoid function gives the characteristic sigmoid cliff response of a first hidden layer MLP unit (the figure below illustrates the shape plotted across two inputs. An MLP unit with more inputs has a higher-dimensional version of this functional shape). Altering the weights and thresholds alters this response surface. In particular, both the orientation of the surface, and the steepness of the sloped section, can be altered. A steep slope corresponds to large weight values: doubling all weight values gives the same orientation but a different slope.



A multi-layered network combines a number of these response surfaces together, through repeated linear combination and non-linear activation functions. The next figure illustrates a typical response surface for a network with only one hidden layer, of two units, and a single output unit, on the classic XOR problem. Two separate sigmoid surfaces have been combined into a single U-shaped surface.

During network training, the weights and thresholds are first initialized to small, random values. This implies that the units' response surfaces are each aligned randomly with low slope: they are effectively uncommitted. As training progresses, the units' response surfaces are rotated and shifted into appropriate positions, and the magnitudes of the weights grow as they commit to modeling particular parts of the target response surface.

In a classification problem, an output unit's task is to output a strong signal if a case belongs to its class, and a weak signal if it doesn't. In other words, it is attempting to model a function that has magnitude one for parts of the pattern-space that contain its cases, and magnitude zero for other parts.

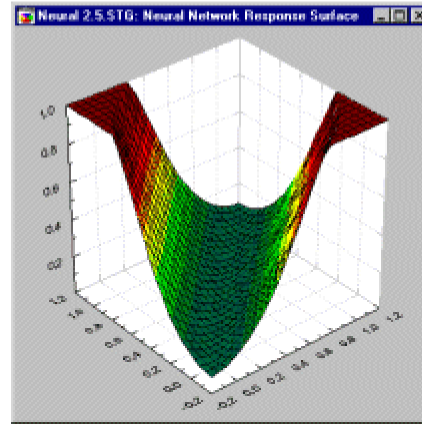


This is known as a *discriminant function* in pattern recognition problems. An ideal discriminant function could be said to have a plateau structure, where all points on the function are either at height zero or height one.

If there are no hidden units, then the output can only model a single sigmoid-cliff with areas to one side at low height and areas to the other high. There will always be a region in the middle (on the cliff) where the height is in-between, but as weight magnitudes are increased, this area shrinks.

A sigmoid-cliff like this is effectively a linear discriminant. Points to one side of the cliff are classified as belonging to the class, points to the other as not belonging to it. This implies that a network with no hidden layers can only classify linearly-separable problems (those where a line - or, more generally in higher dimensions, a hyperplane - can be drawn which separates the points in pattern space).

A network with a single hidden layer has a number of sigmoid-cliffs (one per hidden unit) represented in that hidden layer, and these are in turn combined into a plateau in the output layer. The plateau has a convex hull (i.e., there are no dents in it, and no holes inside it). Although the plateau is convex, it may extend to infinity in some directions (like an extended peninsular). Such a network is in practice capable of modeling adequately most real-world classification problems.



The figure above shows the plateau response surface developed by an MLP to solve the XOR problem: as can be seen, this neatly sections the space along a diagonal.

A network with two hidden layers has a number of plateaus combined together - the number of plateaus corresponds to the number of units in the second layer, and the number of sides on each plateau corresponds to the number of units in the first hidden layer. A little thought shows that you can represent any shape (including concavities and holes) using a sufficiently large number of such plateaus.

A consequence of these observations is that an MLP with two hidden layers is theoretically sufficient to model any problem (there is a more formal proof, the Kolmogorov Theorem). This does not necessarily imply that a network with more layers might not more conveniently or easily model a particular problem. In practice, however, most problems seem to yield to a single hidden layer, with two an occasional resort and three practically unknown.

A key question in classification is how to interpret points on or near the cliff. The standard practice is to adopt some confidence levels (the accept and reject thresholds) that must be exceeded before the unit is deemed to have made a decision. For example, if accept/reject thresholds of 0.95/0.05 are used, an output unit with an output level in excess of 0.95 is deemed to be on, below 0.05 it is deemed to be off, and in between it is deemed to be undecided.

A more subtle (and perhaps more useful) interpretation is to treat the network outputs as probabilities. In this case, the network gives

more information than simply a decision: it tells us how sure (in a formal sense) it is of that decision. There are modifications to MLPs that allow the neural network outputs to be interpreted as probabilities, which means that the network effectively learns to model the probability density function of the class. However, the probabilistic interpretation is only valid under certain assumptions about the distribution of the data (specifically, that it is drawn from the family of exponential distributions). Ultimately, a classification decision must still be made, but a probabilistic interpretation allows a more formal concept of minimum cost decision making to be evolved.

5. Other MLP Training Algorithms

Earlier in this section, we discussed how the *back propagation* algorithm performs gradient descent on the error surface. Speaking loosely, it calculates the direction of steepest descent on the surface, and jumps down the surface a distance proportional to the learning rate and the slope, picking up momentum as it maintains a consistent direction. As an analogy, it behaves like a blindfold kangaroo hopping in the most obvious direction. Actually, the descent is calculated independently on the error surface for each training case, and in random order, but this is actually a good approximation to descent on the composite error surface. Other MLP training algorithms work differently, but all use a strategy designed to travel towards a minimum as quickly as possible.

More sophisticated techniques for non-linear function optimization have been in use for some time. These methods include *conjugate gradient descent*, quasi-Newton, and *Levenberg-Marquardt*, which are very successful forms of two types of algorithm: line search and model-trust region approaches. They are collectively known as second order training algorithms.

A line search algorithm works as follows: pick a sensible direction to move in the multi-dimensional landscape. Then project a line in that direction, locate the minimum along that line (it is relatively trivial to locate a minimum along a line, by using some form of bisection algorithm), and repeat. What is a sensible direction in this context? An obvious choice is the direction of steepest descent (the same direction that would be chosen by *back propagation*). Actually, this intuitively obvious choice proves to be rather poor. Having minimized along one direction, the next line of steepest descent may spoil the minimization along the initial direction (even on a simple surface like a parabola a large number of line searches may be necessary). A better approach is to select conjugate or non-interfering directions - hence *conjugate gradient descent* .

The idea here is that, once the algorithm has minimized along a particular direction, the second derivative along that direction should be kept at zero. Conjugate directions are selected to maintain this zero second derivative on the assumption that the surface is parabolic (speaking roughly, a nice smooth surface). If this condition holds, N epochs are sufficient to reach a minimum. In reality, on a complex error surface the conjugacy deteriorates, but the algorithm still typically requires far less epochs than *back propagation*, and also converges to a better minimum (to settle down thoroughly, *back propagation* must be run with an extremely low learning rate).

Quasi-Newton training is based on the observation that the direction pointing directly towards the minimum on a quadratic surface is the so-called Newton direction. This is very expensive to calculate analytically, but quasi-Newton iteratively builds up a good approximation to it. Quasi-Newton is usually a little faster than conjugate gradient descent, but has substantially larger memory requirements and is occasionally numerically unstable.

A model-trust region approach works as follows: instead of following a search direction, assume that the surface is a simple shape such that the minimum can be located (and jumped to) directly - if the assumption is true. Try the model out and see how good the suggested point is. The model typically assumes that the surface is a nice well-behaved shape (e.g., a parabola), which will be true if sufficiently close to a minima. Elsewhere, the assumption may be grossly violated, and the model could choose wildly inappropriate points to move to. The model can only be trusted within a region of the current point, and the size of this region isn't known. Therefore, choose new points to test as a compromise between that suggested by the model and that suggested by a standard gradient-descent jump. If the new point is good, move to it, and strengthen the role of the model

in selecting a new point; if it is bad, don't move, and strengthen the role of the gradient descent step in selecting a new point (and make the step smaller). *Levenberg-Marquardt* uses a model that assumes that the underlying function is locally linear (and therefore has a parabolic error surface).

Levenberg-Marquardt is typically the fastest of the training algorithms, although unfortunately it has some important limitations, specifically: it can only be used on single output networks, can only be used with the *sum squared error* function, and has memory requirements proportional to W_2 (where W is the number of weights in the network; this makes it impractical for reasonably big networks). *Conjugate gradient descent* is nearly as good, and doesn't suffer from these restrictions.

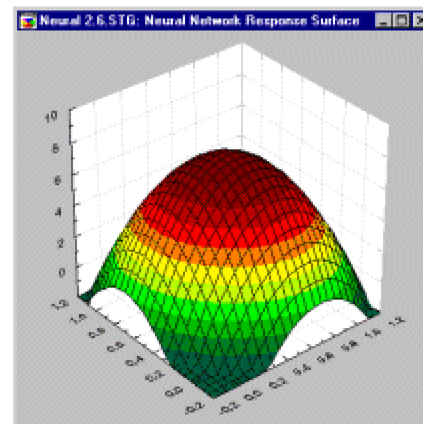
Back propagation can still be useful, not least in providing a quick (if not overwhelmingly accurate) solution. It is also a good choice if the data set is very large, and contains a great deal of redundant data. *Back propagation*'s case-by-case error adjustment means that data redundancy does it no harm (for example, if you double the data set size by replicating every case, each epoch will take twice as long, but have the same effect as two of the old epochs, so there is no loss). In contrast, *Levenberg-Marquardt*, quasi-Newton, and conjugate gradient descent all perform calculations using the entire data set, so increasing the number of cases can significantly slow each epoch, but does not necessarily improve performance on that epoch (not if data is redundant; if data is sparse, then adding data will make each epoch better). *Back propagation* can also be equally good if the data set is very small, for there is then insufficient information to make a highly fine-tuned solution appropriate (a more advanced algorithm may achieve a lower training error, but the selection error is unlikely to improve in the same way). Finally, the second order training algorithms seem to be very prone to stick in local minima in the early phases - for this reason, we recommend the practice of starting with a short burst of back propagation, before switching to a second order algorithm.

There are variations on *back propagation* (*quick propagation* and *Delta-bar-Delta*) that are designed to deal with some of the limitations on this technique. In most cases, they are not significantly better than *back propagation*, and sometimes they are worse (relative performance is application-dependent). They also require more control parameters than any of the other algorithms, which makes them more difficult to use, so they are not described in further detail in this section.

Radial Basis Function Networks

We have seen in the last section how an MLP models the response function using the composition of sigmoid-cliff functions - for a classification problem, this corresponds to dividing the pattern space up using hyperplanes. The use of hyperplanes to divide up space is a natural approach - intuitively appealing, and based on the fundamental simplicity of lines.

An equally appealing and intuitive approach is to divide up space using circles or (more generally) hyperspheres. A hypersphere is characterized by its center and radius. More generally, just as an MLP unit responds (non-linearly) to the distance of points from the line of the sigmoid-cliff, in a radial basis function network units respond (non-linearly) to the distance of points from the center represented by the radial unit. The response surface of a single radial unit is therefore a Gaussian (bell-shaped) function, peaked at the center, and descending outwards. Just as the steepness of the MLP's sigmoid curves can be altered, so can the slope of the radial unit's Gaussian.



MLP units are defined by their weights and threshold, which together give the equation of the defining line, and the rate of fall-off of the function from that line. Before application of the sigmoid activation function, the activation level of the unit is determined using a weighted sum,

which mathematically is the dot product of the input vector and the weight vector of the unit; these units are therefore referred to as dot product units. In contrast, a *radial unit* is defined by its center point and a radius. A point in N dimensional space is defined using N numbers, which exactly corresponds to the number of weights in a dot product unit, so the center of a radial unit is stored as weights. The radius (or deviation) value is stored as the threshold. It is worth emphasizing that the weights and thresholds in a radial unit are actually entirely different to those in a dot product unit, and the terminology is dangerous if you don't remember this: Radial weights really form a point, and a radial threshold is really a deviation.

A *radial basis function* network (RBF), therefore, has a hidden layer of radial units, each actually modeling a Gaussian response surface. Since these functions are nonlinear, it is not actually necessary to have more than one hidden layer to model any shape of function: sufficient radial units will always be enough to model any function. The remaining question is how to combine the hidden radial unit outputs into the network outputs? It turns out to be quite sufficient to use a linear combination of these outputs (i.e., a weighted sum of the Gaussians) to model any nonlinear function. The standard RBF has an output layer containing dot product units with indentity activation function.

RBF networks have a number of advantages over MLPs. First, as previously stated, they can model any nonlinear function using a single hidden layer, which removes some design-decisions about numbers of layers. Second, the simple linear transformation in the output layer can be optimized fully using traditional linear modeling techniques, which are fast and do not suffer from problems such as local minima which plague MLP training techniques. RBF networks can therefore be trained extremely quickly (i.e., orders of magnitude faster than MLPs).

On the other hand, before linear optimization can be applied to the output layer of an RBF network, the number of radial units must be decided, and then their centers and deviations must be set. Although faster than MLP training, the algorithms to do this are equally prone to discover sub-optimal combinations. Other features that distinguish RBF performance from MLPs are due to the differing approaches to modeling space, with RBFs "clumpy" and MLPs "planey."

Other features which distinguish RBF performance from MLPs are due to the differing approaches to modeling space, with RBFs "clumpy" and MLPs "planey."

The clumpy approach also implies that RBFs are not inclined to extrapolate beyond known data: the response drops off rapidly towards zero if data points far from the training data are used. Often the RBF output layer optimization will have set a bias level, hopefully more or less equal to the mean output level, so in fact the extrapolated output is the observed mean - a reasonable working assumption. In contrast, an MLP becomes more certain in its response when far-flung data is used. Whether this is an advantage or disadvantage depends largely on the application, but on the whole the MLP's uncritical extrapolation is regarded as a bad point: extrapolation far from training data is usually dangerous and unjustified.

RBFs are also more sensitive to the curse of dimensionality, and have greater difficulties if the number of input units is large: this problem is discussed further in a later section.

As mentioned earlier, training of RBFs takes place in distinct stages. First, the centers and deviations of the radial units must be set; then the linear output layer is optimized.

Centers should be assigned to reflect the natural clustering of the data. The two most common methods are:

Sub-sampling. Randomly-chosen training points are copied to the radial units. Since they are randomly selected, they will represent the distribution of the training data in a statistical sense. However, if the number of radial units is not large, the radial units may actually be a poor representation .

K-Means algorithm. This algorithm tries to select an optimal set of points that are placed at the centroids of clusters of training data. Given K radial units, it adjusts the positions of the centers so that:

- Each training point belongs to a cluster center, and is nearer to this center than to any other center;

- Each cluster center is the centroid of the training points that belong to it.

Once centers are assigned, deviations are set. The size of the deviation (also known as a smoothing factor) determines how spiky the Gaussian functions are. If the Gaussians are too spiky, the network will not interpolate between known points, and the network loses the ability to generalize. If the Gaussians are very broad, the network loses fine detail. This is actually another manifestation of the over/under-fitting dilemma. Deviations should typically be chosen so that Gaussians overlap with a few nearby centers. Methods available are:

Explicit. Choose the deviation yourself.

Isotropic. The deviation (same for all units) is selected heuristically to reflect the number of centers and the volume of space they occupy.

K-Nearest Neighbor. Each unit's deviation is individually set to the mean distance to its *K* nearest neighbors. Hence, deviations are smaller in tightly packed areas of space, preserving detail, and higher in sparse areas of space (interpolating where necessary).

Once centers and deviations have been set, the output layer can be optimized using the standard linear optimization technique: the pseudo-inverse (singular value decomposition) algorithm.

However, RBFs as described above suffer similar problems to Multilayer Perceptrons if they are used for classification - the output of the network is a measure of distance from a decision hyperplane, rather than a probabilistic confidence level. We may therefore choose to modify the RBF by including an output layer with logistic or softmax (normalized exponential) outputs, which is capable of probability estimation. We lose the advantage of fast linear optimization of the output layer; however, the non-linear output layer still has a relatively well-behaved error surface, and can be optimized quite quickly using a fast iterative algorithm such as conjugate gradient descent.

Radial basis functions can also be hybridized in a number of ways. The radial layer (the hidden layer) can be trained using the Kohonen and Learned Vector Quantization training algorithms, which are alternative methods of assigning centers to reflect the spread of data, and the output layer (whether linear or otherwise) can be trained using any of the iterative dot product algorithms.

Conclusion

Neural network error surfaces are much more complex, and are characterized by a number of unhelpful features, such as local minima (which are lower than the surrounding terrain, but above the global minimum), flat-spots and plateaus, saddle-points, and long narrow ravines.

Experience indicates that the RBF's more eccentric response surface requires a *lot* more units to adequately model most functions. Of course, it is always possible to draw shapes that are most easily represented one way or the other, but the balance does not favor RBFs. Consequently, an RBF solution will tend to be slower to execute and more space consuming than the corresponding MLP (but it was much faster to train, which is sometimes more of a constraint).

References

○ Kay(1994a) Information Theoretic Neural Networks for Contextually Guided Unsupervised Learning: Mathematical and Statistical Considerations. Research report.
○ Neural Networks : Simon Haykin, a comprehensive foundation
○ P.D. WasserMan, Neural Computing: Theory and Practice , Van Nortrand Reinhold
○ Bose, Neural Network fundamentals with graphs, algorithms and applications
○ R.H.Neilson, Neurocomputing, Addison Wesley
○ J. Anderson et al., Neurocomputing Vol. 1 & Vol 2, MIT Press, 1986 & 1988