THEORETICAL REPORT IFS-TR-022

THE BACKPROPAGATION ALGORITHM AND SOME OF ITS VARIANTS

ABSTRACT

One of the most popular methods of training the Multi-Layer Perceptron in recent years has been the Backpropagation Of Error algorithm (*backprop* for short). Backprop has its origins in work carried out by Werbos [1] in 1973 but the popularisation of the algorithm is usually accredited to the PDP group and their publication in 1986 of Parallel Distributed Processing [2].

In this report we shall examine both the derivation and practical use of the algorithm to train MLP's. We shall then discuss a few of it's limitations and review some of the faster, more efficient techniques that are currently available to train MLP's.

REPORT

BACKGROUND

As discussed in Theoretical Report IFS-TR-020, the Multi-Layer Perceptron (MLP) performs a mapping between an input vector, $\underline{x}$ and an output vector, $\underline{o}$. The explicit mapping performed by the MLP may be written in terms of the inputs to the network, the activation functions of the neurons and the weighted connections between neurons,

$$o_k = \Phi^{(o)}\left(\sum_{i=1}^{H}\left(\omega^{(2)}{}_{i,k}\,\Phi^{(h)}\left(\sum_{j=1}^{I_N}(\omega^{(1)}{}_{i,j}\,x_j)+\alpha_i^{(1)}\right)\right)+\alpha_k^{(2)}\right)$$

where $\omega^{(1)}{}_{i,j}$ is the weighted connection between the i^{th} hidden neuron and the j^{th} input neuron, $\omega^{(2)}{}_{i,k}$ is the weighted connection between the k^{th} output node and the i^{th} hidden

node, $\alpha_i^{(1)}$ is the bias on the i^{th} hidden node, $\alpha_k^{(2)}$ is the bias on the k^{th} output node, $\Phi^{(o)}$ is the output function of the output neurons, $\Phi^{(h)}$ is the activation function of the hidden neurons and finally x_j is the j^{th} component of the input vector.

It is usual to keep the activation functions of the network fixed during training, The mapping is then seen to be purely a function of the set of weights in the network $\underset{\sim}{w}$. We shall denote this mapping as $y = f(\underset{\sim}{x}, \underset{\sim}{\omega})$. The objective of training an MLP is to find a set of weights $\underset{\sim}{\omega}_{opt}$ that performs the desired input-output mapping for the problem we wish to solve. This mapping is generally found through the use of a collection of example input and output vectors called the *training set*. Suppose that we have a set of T example vectors contained in the set Ψ, i.e.

$$\Psi = \left\{ \underset{\sim}{x}_i, t_i \quad i = 1..T \right\}$$

where $\underset{\sim}{x}_i$ is the i^{th} example input pattern and t_i is the required response of the network to this input. For notational convenience we assume a scalar output, however, extension to multiple dimensional output vectors is trivial.

The misclassification error or *training error* of the network in classifying the training examples with a given set of weights $\underset{\sim}{\omega}$ is generally chosen to be the sum of squares of the differences between the actual and desired responses of the network taken over the entire training set, i.e.

$$E(\underset{\sim}{\omega}) = \frac{1}{2} \sum_{i=1}^{T} \left(t_i - f\left(\underset{\sim}{x}_i, \underset{\sim}{\omega} \right) \right)^2$$

The objective of training will be to minimise the above function, $E(\underset{\sim}{\omega})$ with respect to the weights of the network, $\underset{\sim}{\omega}$. To do this, it is usual to use a gradient descent based technique. This will require the calculation of the partial derivatives of the error function, $E(\underset{\sim}{\omega})$ with respect to each of the individual weights of the network $\omega_{i,j}^{(n)}$. The derivation of these partial derivatives is usually performed using what has become known as the Backpropagation Of Error algorithm.

Once the partial derivatives of the Error Function with respect of the weights have been found, gradient descent or variations on that method can be applied to devise a learning rule to find a set of weights that produce a (locally) minimum value for $E(\underset{\sim}{\omega})$. We shall begin by deriving the partial derivatives for a particular type of MLP.

THE BACKPROP ALGORITHM

We shall derive the backprop algorithm for the case of a three-layer MLP with sigmoidal hidden and sigmoidal output units. Extension to other types of network (for instance an MLP with linear output units) is straightforward. Let us assume that we have just presented the input pattern $\underset{\sim}{x}$ to the network and it has produced the response $f(\underset{\sim}{x}, \underset{\sim}{\omega})$ and that the desired response was t.

Hidden To Output Weights

For the weights connecting the final layer and the hidden layer we wish to calculate

$$\frac{\partial E(\underset{\sim}{\omega})}{\partial \omega^{(2)}{}_{i,j}} = \frac{\partial E(\underset{\sim}{\omega})}{\partial o^{(o)}{}_{j}} \cdot \frac{\partial o^{(o)}{}_{j}}{\partial y^{(o)}{}_{j}} \cdot \frac{\partial y^{(o)}{}_{j}}{\partial \omega^{(2)}{}_{i,j}}$$

where $\omega^{(2)}{}_{i,j}$ is the weight connecting the i^{th} hidden neuron and the j^{th} output neuron, $o^{(o)}{}_{j}$ is the output of the j^{th} output neuron and $y^{(o)}{}_{j}$ is the input to the j^{th} output neuron given by

$$y_j^{(o)} = \sum_{i=1}^{H} \omega^{(2)}{}_{i,j}\, o^{(h)}{}_{i}$$

where $o^{(h)}{}_{i}$ is the output of the i^{th} hidden neuron.

We must therefore evaluate the three partial derivatives contained on the right hand side of the equation above.

1) Evaluating $\dfrac{\partial E(\underset{\sim}{\omega})}{\partial o^{(o)}{}_{j}}$

Recalling that the error function we are using is the least squares function then the above term is simply

$$\frac{\partial E(\underset{\sim}{\omega})}{\partial o^{(o)}{}_{j}} = o^{(o)}{}_{j} - t$$

where t is the target value for the output neuron.

2) Evaluating $\dfrac{\partial o^{(o)}{}_{j}}{\partial y^{(o)}{}_{j}}$

The output of the j^{th} neuron (for the case of sigmoidal activation functions) is given by

$$o^{(o)}{}_{j} = \frac{1}{1+e^{-y^{(o)}{}_{j}}}$$

so the partial derivative $\dfrac{\partial o^{(o)}{}_{j}}{\partial y^{(o)}{}_{j}}$ may be written

$$\frac{\partial o^{(o)}{}_j}{\partial y^{(o)}{}_j} = o^{(o)}{}_j\left(1-o^{(o)}{}_j\right)$$

due to the fact that for sigmoidal functions, $f'(x) = f(x)\left(1-f(x)\right)$.

3) Evaluating $\frac{\partial y^{(o)}{}_j}{\partial \omega^{(2)}{}_{i,j}}$

Recalling that the input to an output node is the weighted sum of the outputs of the nodes in the hidden layer, i.e.

$$y^{(o)}{}_j = \sum_{i=1}^{H} o^{(h)}{}_i \omega^{(2)}{}_{i,j}$$

then the partial derivative we require is simply

$$\frac{\partial y^{(o)}{}_j}{\partial \omega^{(2)}{}_{i,j}} = o^{(h)}{}_i$$

Putting together the three partial derivatives we have obtained expressions for above we have a complete expression for the partial Error derivatives for the hidden to output weights in the network,

$$\begin{aligned}\frac{\partial E(\underset{\sim}{\omega})}{\partial \omega^{(2)}{}_{i,j}} &= \frac{\partial E(\underset{\sim}{\omega})}{\partial o^{(o)}{}_j} \cdot \frac{\partial o^{(o)}{}_j}{\partial y^{(o)}{}_j} \cdot \frac{\partial y^{(o)}{}_j}{\partial \omega^{(2)}{}_{i,j}} \\ &= \left(o^{(o)}{}_j - t\right) . o^{(o)}{}_j\left(1-o^{(o)}{}_j\right) . o^{(h)}{}_i\end{aligned}$$

Input to Hidden Weights

We now must determine the values of partial derivative of error with respect to the weights between the input and hidden layers. This is done is a similar manner to the case of the hidden to output units. The only problem we are faced with is how to determine the partial derivative $\frac{\partial E(\underset{\sim}{\omega})}{\partial o^{(h)}{}_j}$, since we do not have a target value for the output of a hidden node $o_j^{(h)}$. It turns out that we can *backpropagate* this partial derivative from its evaluation for the output layer neurons using the backpropagation rule

$$\frac{\partial E(\underset{\sim}{\omega})}{\partial o^{(h)}{}_j} = \sum_{i=1}^{O} \omega^{(2)}{}_{j,i} \frac{\partial E(\underset{\sim}{\omega})}{\partial o^{(o)}{}_i}$$

where O is the number of output units. Let the output of the *j*th hidden node be given by

$$y_j^{(h)} = \sum_{i=1}^{I_n} x_i \omega_{i,j}^{(1)}$$

Then the derivation proceeds in the same manner as before to yield:

$$\frac{\partial E(\underset{\sim}{\omega})}{\partial \omega^{(1)}{}_{i,j}} = \frac{\partial E(\underset{\sim}{\omega})}{\partial o^{(h)}{}_j} \cdot \frac{\partial o^{(h)}{}_j}{\partial y^{(h)}{}_j} \cdot \frac{\partial y^{(h)}{}_j}{\partial \omega^{(1)}{}_{i,j}}$$

$$= o^{(h)}{}_j \left(1 - o^{(h)}{}_j\right) . x_i \sum_{k=1}^{O} \omega^{(2)}{}_{j,k} \frac{\partial E(\underset{\sim}{w})}{\partial o^{(o)}{}_k}$$

where x_i is the i^{th} component of the input vector $\underset{\sim}{x}$.

GRADIENT DESCENT

In the previous section, we have derived a method of finding the partial derivatives of data misfit error with respect to each of the individual weights from both *layers* of the network. We can now use this gradient to minimise the error function. The simplest possible way of achieving this would be to use the update rule

$$\omega_{i,j}^{(t+1)} = \omega_{i,j}^{(t)} + \Delta_{i,j}^{(t)}$$

$$\Delta_{i,j}^{(t)} = -\varepsilon \frac{\partial E(\underset{\sim}{\omega})}{\partial \omega_{i,j}^{(t)}}$$

where $\omega_{i,j}^{(t)}$ is the value of the weighted link at time t and ε is the *learning rate* or *step size*.

The major problem with the above learning rule is the speed at which it operates. It can take many tens of thousands of training iterations to achieve a minimum of the misfit function using the above on even quite simple problems. Because of this, simple gradient descent on its own is rarely used in practice.

A popular way of *speeding up* backprop is to use an additional *momentum* term, α. Momentum may be added to the simple gradient descent algorithm by modifying the update rule,

$$\omega_{i,j}^{(t+1)} = \omega_{i,j}^{(t)} + \Delta_{i,j}^{(t)}$$

$$\Delta_{i,j}^{(t)} = \alpha \Delta_{i,j}^{(t-1)} - \varepsilon \frac{\partial E(\underset{\sim}{\omega})}{\partial \omega_{i,j}^{(t)}}$$

In this way the update step for the current iteration is made to include a fraction of the update step from the previous iteration as well as the current negative gradient. Thus, if the

sign of the gradient is consistently above or below zero then the rate of change will *gather momentum* increasing the speed at which the error misfit function is minimised.

BATCH MODE OR ON-LINE LEARNING ?

We have derived above the expressions to evaluate the partial derivatives of the error function with respect to the weights of the network **for a single training example.** The error function is defined as the sum of the training errors over all training examples. Thus, if we wish to be sure of minimising the global error function then we should sum all of the partial derivatives of error with respect to weight for all of the training examples and then update the weights using these summed derivatives. This method is often referred to in the neural network literature as *batch mode* learning.

However, in cases where the training data is quite *similar* it is possible to update the weights on the basis of the partial derivatives found individually for the separate training patterns. Thus, weight updates occur after the presentation of each training example. This method of training is usually referred to as *on-line* learning. On-line learning is often much faster than batch mode learning, however, it is also potentially unstable as we are no longer directly optimising on the basis of the true error function we wish to minimise.

It is possible to compromise between the extremes of on-line and batch learning and update the weights after, for instance, each 100 training patterns. Again, the success of this method will depend greatly on the degree to which the training data is homogenous. It is our experience that, in general, batch mode training is better for financial time series prediction as the training sets usually display a degree of non-stationarity, reflecting the non-stationarity of the underlying time series themselves.

VALUES FOR α AND ε

An obvious question to ask is what are the appropriate values of α and ε to use for a particular problem? There are no firm theoretical rules for this as the optimal α and ε will generally depend on the complexity of the error surface which will be problem dependent. However, IFS have found the following *rules of thumb*.

- Step size ε - When using batch mode training, this should be set such that the number of training patterns in the training set times ε is in the range [1.0 ,10.0].

- Momentum α - This should be set in the range [0.80,0.99]. It is generally advisable to set α to be very small (even 0.00) at the beginning of training to avoid *rushing* into a local minimum.

BETTER LEARNING RULES

The backprop algorithm has been used to train neural networks in a number of successful real world applications. It is however, a severely sub-optimal approach to the optimisation of the error function. Many better and faster, second order techniques have been proposed in the literature, ranging from pseudo-second derivative techniques such as Quickprop or delta-bar-delta to more theoretically sound techniques such as Conjugate Gradients or Newton-Raphson.

The MLPs in Amber contain the Quickprop technique (IFS-TR-023) and a simple Conjugate Gradients scheme (IFS-TR-024). We have found Quickprop to be the most robust and efficient for time series prediction problems.

Author: Darren Toulson
Revision: v 1.00 4 January, 1997
See ALSO: Theoretical Reports IFS-TR-023,IFS-TR-024 , The Amber Reference Guide