THEORETICAL REPORT IFS-TR-020

THE MULTI-LAYER PERCEPTRON AND ITS USE WITHIN AMBER

ABSTRACT

The Multi-Layer Perceptron (MLP) and its descendants play an integral part as one of the basic time series prediction models used within Amber. This report will provide a basic introduction to some of the theory behind the Multi-Layer Perceptron. We shall illustrate the training and operation of an MLP using a time series prediction problem as an example.

There is a great deal of established theory surrounding the use (and potential misuse) of MLPs in the technical literature. This report is intended to serve as a gentle introduction to the MLP. More detailed and thorough reports into different aspects of MLP's are given in Theoretical Reports IFS-TR-21 through to IFS-TR-33.

REPORT

HISTORICAL BACKGROUND TO THE MLP

The Multi-Layer Perceptron began receiving a great deal of wide-spread popular attention in the middle to late 1980s. Since then it has found successful applications in many different fields. The original 1986 MLP [1] is essentially an extension of the linear Two Layer Perceptron developed during the 1960s by Rosenblatt [2] and others. The key difference between the MLP and the older, Two Layer Perceptron, was the inclusion of *hidden* layers of processing units and non-linear activation functions into the model. The two architectures are shown below in Figure 1.

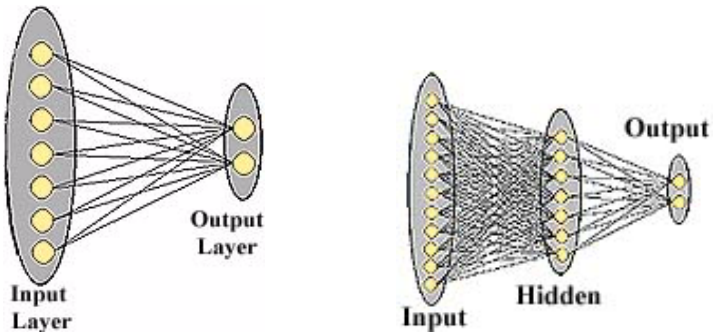
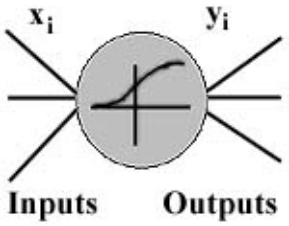


Figure 1: Two-layer perceptron and MLP

THE NEURON



The basic building blocks of the MLP and many other neural networks models are the *neurons*. The basic neuron, shown to the left, is connected to other neurons in the network architecture by a set of weighted links $\omega_{i,j}$. The neuron's role in the architecture is to sum the weighted signals being input into it (this summation is often referred to as the *activation* of the neuron) and to then output a function of this summation to other neurons via further weighted links. The exact function being used by a particular neuron (normally referred to as the *activation function* of the neuron) varies from architecture to architecture, but a popular choice is the sigmoidal function. The functioning of a sigmoidal neuron may be summarised by writing the equation for the output of the neuron o_i as

$$o_i = \Phi\left(\sum_{j=1}^{N} \omega_{j,i}\, y_j + \alpha\right)$$

$$\Phi(x) = \frac{1}{1+e^{-\frac{x}{x_0}}}$$

where $\omega_{i,j}$ is the weighted connection between neurons i and j, y_j is the output of the j^{th} connected neuron and α is the *bias* of the neuron. The slope of the sigmoidal function is controlled by x_0.

THE OPERATION OF THE MLP

The basic manner in which the MLP functions is as follows. An *input vector* (often referred to as an *input pattern*) is fed into the first, or *input layer* of the architecture. This signal then propagates through the *hidden layers* of the MLP via sets of inter-connected weights and activation functions on each neuron. The signal eventually emerges from the final or *output layer* of the network as a vector of values usually referred to as the *response*, or *output vector*.

The manner in which the MLP is most often used in applications is as a mapping from a set of input data onto a classification or *answer* vector with regards to the original data. For example, the input vector may contain an encoding of a digital image of a human face and the output response may indicate whether the face is male or female. Another example is for the input vector to contain recent historical price data concerning a financial market and the output response a prediction of the future direction of that market.

Mathematically, we can express this mapping of input vectors onto output vectors by the MLP as a function,

$$F : R^{I_N} \Rightarrow R^{O_N}$$

where I_N is the number of input neurons and O_N is the number of output neurons. For the case of a Three-Layer MLP the response of the k^{th} output node o_k produced by an input vector $\underset{\sim}{x}$ can be written out explicitly,

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

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 for the activation functions to be fixed for the MLP (i.e. they are not altered during training). For a given architecture then, the MLP's mapping is often summarised as a function of the input pattern, $\underset{\sim}{x}$, and the set of all weighted links $\underset{\sim}{w}$ in the network i.e.

$$y = f(\underset{\sim}{x}, \underset{\sim}{w})$$

For notational convenience, we shall assume from now on that there is only a single output neuron. Extension to multiple output neurons is trivial.

TRAINING THE MLP

In the previous section we saw that the neural network essentially performs a mapping from an input vector to an output vector. We can control the exact mapping it performs by changing the values of the weighted links between neurons.

In a particular application, we will wish to find the best mapping for the problem we wish to solve. For instance, suppose that the input vector $\underset{\sim}{x}$ is a 30-dimensional vector, representing the last 30 daily returns of the Nikkei-225 index. We may wish that the output vector $\underset{\sim}{o}$ is then a one-dimensional vector, representing the expected 5 day future return of the index (given a 5-day working week). This output response could then be used as the basis of a trading model for that market. The question is how can we find a suitable set of weighted

links in the network such that the network will perform the desired mapping (or at least perform the mapping to the best of its ability)?

The most usual method of finding a good set of weights $\underset{\sim}{w}$ for a given problem is to use a *training set.* A training set is a set of example input vectors and their associated required response vectors (i.e. the answer we would wish the network to produce). Obviously, we can only obtain such a set of vectors by using historical price data of the Nikkei-225 index. If, for instance, we have the last five years of historical daily price data for the Nikkei-225, we could obtain approximately 1500 training examples from this time period. We shall denote the training set, Ψ as a set as T input patterns, $\underset{\sim}{x}_i$ along with their associated required outputs, t_i, i.e.

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

The network is then trained to perform the required mapping using the training data at all times to assess its performance. We usually define the error of the network in terms of its classifying of the training data as a simple squared difference of required and actual responses, i.e.

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

As can be seen above, we have written the error function as a function of the weights in the network $\underset{\sim}{\omega}$. The goal of training will be to find the set of weights $\underset{\sim}{\omega}_{opt}$ that minimises $E(\underset{\sim}{\omega})$.

This is usually achieved through the use the *Backpropagation of Error* algorithm (backprop for short). The backprop algorithm is essentially a gradient descent technique that operates by finding the partial derivatives of each individual weight with respect to the Error Function, $\frac{\partial E(\underset{\sim}{\omega})}{\partial \omega_i}$. For a given set of weights, these partial derivatives are obtained by calculating the error of each training vector in the training set individually and then summing the responses (this is referred to as *batch* mode training). The individual weights of the network are then updated using the simple gradient descent learning rule,

$$\omega_i \quad \rightarrow \quad \omega_i - \varepsilon \frac{\partial E(\underset{\sim}{\omega})}{\partial \omega_i}$$

where ε is the *learning rate.* This is the simplest possible form of the learning algorithm and is rarely used in practice. Other, faster techniques utilising pseudo-second derivative information or conjugate gradient techniques tend to be used. Details about these techniques may be found in technical reports IFS-TR-022 and IFS-TR-023.

FURTHER INFORMATION

This report is intended only to act as a simple introduction to the MLP architecture. We have only touched lightly on the basic aspects of the MLP and left out or simplified many aspects. There are a number of other Theoretical Reports that deal more thoroughly with the MLP. Below we shall give an indication of the contents of some of these reports organised by subject.

Learning and Training Rules

For more detail on the derivation and use of the *backprop* algorithm see Theoretical Report IFS-TR-022. For a description of other, faster learning rules see Theoretical Report IFS-TR-023. For a discussion of the use of *early stopping* using a validation set during training see Theoretical Report IFS-TR-025.

Regularisation

For a discussion of the need for regularisation when training neural networks see Theoretical Report IFS-TR-027. Theoretical Report IFS-TR-028 describes the use of weight decay as a regularisation policy. Theoretical Reports IFS-TR-029 and IFS-TR-030 describe a more theoretical approach to regularisation using Bayesian methods.

Inputs and Outputs

Theoretical Report IFS-TR-026 contains a discussion of the need for appropriately scaled inputs to MLPs and also discusses the choice of activation function for hidden neurons. Reports IFS-TR-031 and IFS-TR-032 discuss the different output targets that may be used in time series prediction problems.

BIBLIOGRAPHY

[1] D.E.Rummelhart, G.E. Hinton, R.J.Williams **Parallel Distributed Processing** MIT Press 1986.

[2] F. Rosenblatt **Principles of Neurodynamics** Spartan Books, New York.

Author: Darren Toulson
Revision: v 1.00 4 January, 1997
See ALSO: Theoretical Reports IFS-TR-021 toIFS-TR-033 , The Amber Reference Guide