Appendix B

An Introduction to Neural Networks

Since introductions to neural networks can be found in many sources [9, 40], the treatment presented here will be very brief. Although neural networks are modeled after the interconnected networks of biological neurons found in living systems, the neural networks used in this thesis will be extremely simple by comparison. Each neuron is modeled as a device that has some number of inputs and a single output (see Figure B.1). Each input or output is considered to have an excitation level that can vary between -1 and 1.[1] For a particular neuron (call it neuron number j) with n_j inputs, the input excitation levels will be denoted as x_i, where $i = 1...n_j$, and the output excitation level will be denoted as y_j. Inside the neuron, each input has associated with it a *connection weight* w_{ij}. If the connection weight associated with a given input is negative, the input connection is said to be *inhibitory*, otherwise it is *excitatory*. To form the output, the neuron first calculates the *action potential* A_j according to

$$A_j = \sum_{i=1}^{n_j} w_{ij} x_i. \tag{B.1}$$

Then, a *squashing function* σ is applied to A_j to obtain the output y_j. The purpose of the squashing function is to keep the output within the range -1 to 1. Neurons based on many different squashing functions are possible, but the one used here is a *sigmoid function* that

[1]Limiting the range of excitation levels to lie between 0 and 1 is more common.

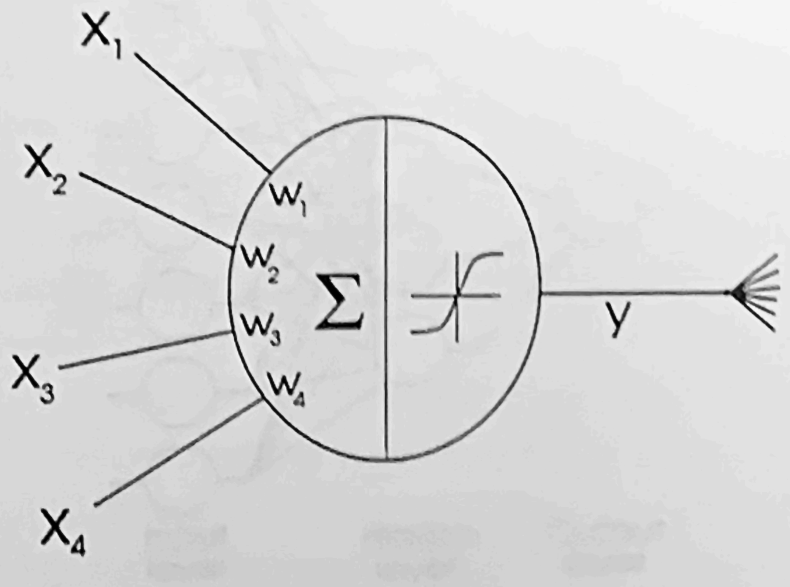

Figure B.1: A single neuron (with 4 inputs)

takes the form

$$y_j = \sigma(A_j) = \frac{2}{1+e^{-A_j}} - 1. \tag{B.2}$$

The value of y_j for large positive values of A_j will approach +1, while large negative A_js will produce y_js near -1. The output of an individual neuron thus depends only on the inputs x_i and on the internal state of the neuron, which is given by the connection weights w_{ij}.

In principle, the inputs and outputs of several neurons can be connected in any haphazard fashion to produce a neural network. However, the networks considered in this thesis are simple special cases called 3-layer feedforward networks. These networks consists of 3 layers of neurons; called the input layer, the hidden layer, and the output layer (see Figure B.2). Let the neurons in each layer be indexed by the subscript i for the input layer, j for the hidden layer, and k for the output layer, and let the number of neurons in each layer be N_{inp}, N_{hid}, and N_{out}. The numbers of neurons in each of the layers can be varied according to the demands of the problem that the network is designed to solve. In this thesis, the structure of the networks being considered will be further limited to those networks having a single output neuron ($N_{out} = 1$). The number of neurons in the hidden layer can be adjusted freely to optimize network performance. Having too few hidden layer neurons will prevent the network from converging (training), while having too many allows the network to simply "memorize" the training patterns instead of generalizing

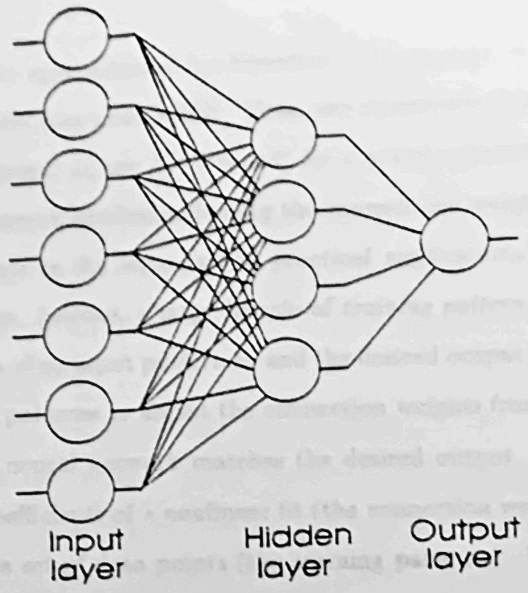

Figure B.2: A 3-layer feedforward neural network with 7 input, 4 hidden, and 1 output layer neuron

their features.

The input neurons do very little; they each accept one input value from the outside world, pass it straight through to their outputs (their single connection weight has a value of 1), and "fan-out" the value to an input on each of the neurons in the hidden layer. Each of the hidden layer neurons thus receives an input x_i from each input layer neuron. Each of the N_{hid} neurons in the hidden layer (indexed by j) then has an internal connection weight w_{ij} associated with each of the inputs x_i from the input layer neurons. Likewise, the single output layer neuron receives inputs x_j from the hidden layer neurons, and via the connection weights w_{jk}, produces the final network output y_k. The response of the neural network is thus given by

$$y_k = \sigma\left(\sum_{j}^{N_{hid}} w_{jk}\sigma\left(\sum_{i}^{N_{inp}} w_{ij}x_i\right)\right) \tag{B.3}$$

where the sums run only over the neurons in the appropriate layer.

Actually, there is an additional *bias neuron* added to the input and hidden layers whose output is fixed at 1.[2] Just as any complicated function can be represented as a sum of sines and cosines (plus a constant) in a Fourier series, it can be proved that any 'well-behaved' function

[2]The bias neuron can be incorporated into Equation B.3 by extending the sums over i and j by one, adding the appropriate connection weights, and fixing the extra x_i =1.

of N_{inp} variables can be approximated by Equation B.3 as a sum of sigmoid functions (with the offset provided by the bias neuron)[37]. Thus, any reasonable relation (mapping) between the inputs x_i and the output y_k can be "learned" by a neural network.

Training a neural network involves adjusting the connections weights to achieve the desired mapping from the inputs to the outputs. In practical applications, the desired mapping is not known ahead of time. Instead, a large sample of *training patterns* is often available. Each training pattern consists of an input pattern x_i and the desired output d. The goal of training is then to use the training patterns to adjust the connection weights from random starting values until the output of the neural network matches the desired output. In a sense, the training procedure adjusts the coefficients of a nonlinear fit (the connection weights in Equation B.3) to produce the best fit to a set of data points (the training patterns). From a more geometrical viewpoint, a *Network Energy* (or *Network Error*) function

$$E = \sum_{p=1}^{N_p} [y_p - d_p]^2 \tag{B.4}$$

can be defined, where the index p runs over the N_p training patterns, and y_p and d_p are the actual and desired outputs for pattern p. The Network Energy is related to how much the current network behavior differs from the desired network behavior. Since the output of the current network y depends on the connection weights w, the Network Energy is a function of the connection weights; $E = E(w)$. Thus, the Network Energy function forms a surface in a multidimensional weight space. During the training process, changing the connection weights corresponds to exploring regions of this surface in weight space. It would be desirable for a network to find the global minimum on the Network Energy surface, as this point corresponds to the situation where the actual network outputs agrees well with the desired outputs.

Many weight adjustment algorithms exist to train neural networks. Most are variants on the *backpropagation* algorithm. Training a network with the backpropagation algorithm is an iterative process that requires many *training iterations* or *epochs*. At each training epoch, the training patterns are presented to the network, and the connection weights are adjusted slightly so as to reduce the Network Energy. The backpropagation algorithm is most easily visualized as a simple gradient-descent procedure in weight space. Geometrically, the backpropagation algorithm adjusts the connection weights so as to move the state of the network down inclines on the Network Energy surface until a minimum is reached. Mathematically, the change in weights is given by

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} + \alpha(\text{ previous } \Delta w_{ij}). \tag{B.5}$$

The first term is the gradient-descent prescription, where η is a *learning constant*. The back-propagation algorithm, by virtue of the chain rule and the *generalized delta rule* described in [43], specifies how to evaluate the derivative of the Network Energy for neurons that are not directly connected to the outputs; thus it provides a recipe to propagate the weight corrections back through the network layers.

The second term, called the *momentum term*, is added to help the network avoid getting stuck in local minima of the Network Energy surface during training. The momentum term adds a portion α of the weight change from the previous epoch to the weight change for the current epoch. With a large enough value of α, the network can overshoot local minima during training. However, values of α or η that are too large can cause the training process to become choatic.

Figures B.3 and B.4 each show how the Network Energy and a selection of connection weights for a neural network evolved during training. Both training sessions used patterns from the TRASH application (see Chapter 7), and were started from the same set of initial weights. For Figure B.3, the training was carried out with $\alpha = 0$ and $\eta = 0.0005$. On the left hand plot, note that the Network Energy dropped rapidly after 30 training iterations, and thereafter improved only slowly with continued training. Each of the lines in the right hand plot shows the evolution of a single connection weight as training proceeded. The connection weights plotted happen to be for the first hidden layer neuron's connections to each of the 7 input layer neurons. The training session shown in Figure B.4 used $\alpha = 0.5$ and $\eta = 0.003$. Note that at these values, the training process is slightly unstable; after training rapidly, it underwent a violent oscillatory period, followed by lesser episodes. The disturbances in Network Energy are correlated with the disturbances in the connection weights. Note also that the network managed to train to a slightly smaller Network Energy. In general, it seemed that using training parameters that were borderline unstable resulted in better performance.

In summary, neural networks have several features that make them useful in high energy physics and other applications, namely:

- A trained neural network can often generalize its training. That is, if the network is

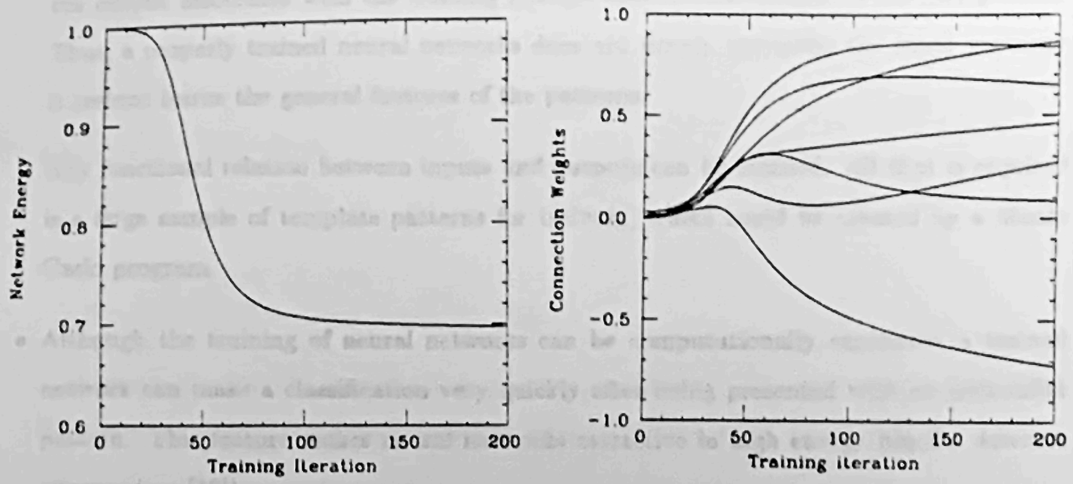

Figure B.3: Evolution of the Network Energy (left) and some connection weights (right) during a $\alpha = 0$, $\eta = 0.0005$ training session.

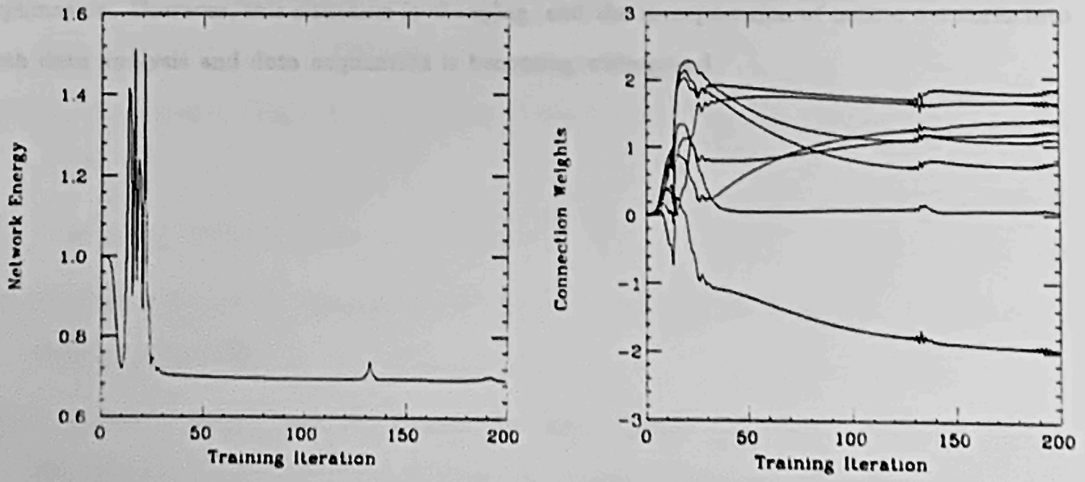

Figure B.4: Evolution of the Network Energy (left) and some connection weights (right) during a $\alpha = 0.5$, $\eta = 0.003$ training session.

presented with a pattern that was not in the training set, it will most likely produce the output associated with the training pattern that is most similar to the test pattern. Thus, a properly trained neural networks does not simply memorize the input patterns; it instead learns the general features of the patterns.

- Any functional relation between inputs and outputs can be learned. All that is required is a large sample of template patterns for training, which could be created by a Monte Carlo program.
- Although the training of neural networks can be computationally expensive, a trained network can make a classification very quickly after being presented with an unfamiliar pattern. This feature makes neural networks attractive in high energy physics detector triggers (see [10]).

The main disadvantage of neural networks is a problem of acceptance in the scientific community. It is sometimes somewhat difficult to understand what combinations of features in the training patterns that a neural network has "seized upon" to make its classifications. Thus, neural networks have a reputation of being incomprehensible "black boxes" whose actions defy explanation. However, this situation is changing, and the incorporation of neural networks into both data analysis and data acquisition is becoming widespread.