Chapter 5

Neural Networks

Neural networks [75, 76, 77, 78] (NN) are a powerful tool for pattern recognition, and are widely applied in several fields, ranging from financial predictions to weather forecasts to character recognition. In particle physics they are mostly used for event or particle classification tasks.

The analysis described in this dissertation makes use of a neural network to separate the $g \to c\bar{c}$ events from the backgrounds.

5.1 What is a Neural Network

Several definitions exist of what is meant by “neural network”. One of the most agreed on is the following [76]:

A neural network is an interconnected assembly of simple processing elements, **units** *or* **nodes***, whose functionality is loosely based on the animal neuron. The processing ability of the network is stored in the inter-unit connection strengths, or weights, obtained by a process of adaptation to, or* **learning from***, a set of training patterns.*

The “neuron” is the fundamental processing element of a neural network. It takes its name from the analogy with the cells responsible of signal transmission in the brain.

The information processing performed by the real (biological) neuron may be crudely summarized as follows: it receives inputs from other neurons (or from the surrounding environment), combines them in some way, performs a generally non-linear operation on the result, and then outputs the final result. The artificial neuron mimics the real one by multiplying the input values by some number or weight to indicate the strength of the link (the “synapse”, following the biological similitude); the weighted signals are then summed to produce an overall unit activation. If this activation exceeds a certain threshold the unit produces an output response. This functionality is captured in the artificial neuron known as the Threshold Logic Unit (TLU) (see Fig. 5.1).

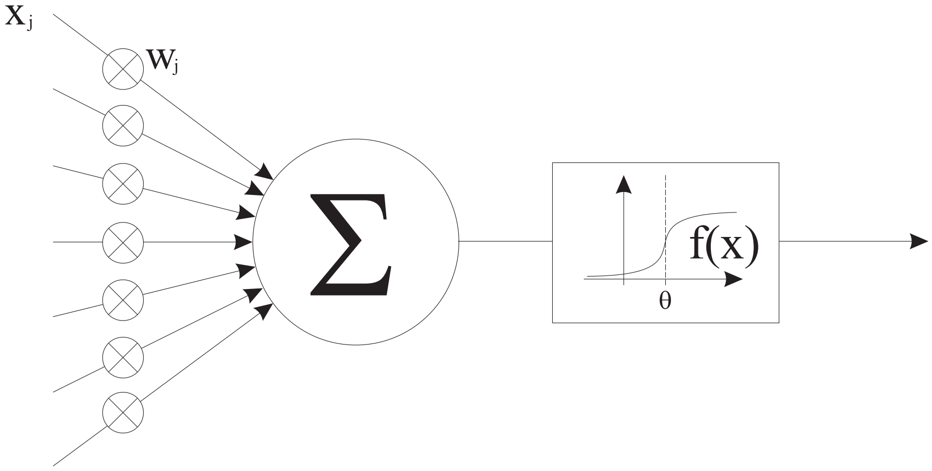

Figure 5.1: Schematic structure of a neuron.

In principle, neural networks can compute any *computable function* (i.e. they can do everything a digital computer does), under some assumptions. In practice, NNs are particularly useful for classification and function approximation/mapping problems which are tolerant of some imprecision, which have lots of training data available, but to which hard and fast rules cannot be applied easily. Almost any finite-dimensional vector function on a compact set can be approximated by a NN to arbitrary precision if enough data and enough computing resources are available.

Practical applications of NNs most often employ *supervised learning*. In supervised learning, training data are provided that include both the input and the desired result (the *target value*). After successful training, input data alone are presented to the NN (that is, input data without the desired result), and the NN will compute an output value that approximates the desired result. However, for training to be successful, lots of training data and lots of computer time for the training are often required. This topic will be best covered in Sec. 5.2.

Neural nets without supervised learning are not provided with target values during training. Unsupervised NNs are used to recognize regularities (a task which is present also in supervised nets, but usually considered secondary or auxiliary with respect to the classification, fitting or forecasting task for which the network is designed) and are usually employed for data compression, such as dimensionality reduction or clustering.

Neural networks can have any topology suitable to the problem to which they are tailored (bi- or three-dimensional, layered or unlayered, and so on). The two major kinds of network topology are *feedforward* and *feedback* nets. In a feedforward NN, the connections between units do not form cycles, while in a feedback or recurrent NN there are cycles in the connections. In some feedback NNs, each time an input is presented, the NN must iterate for a potentially long time before it produces a response, while feedforward NNs usually produce a response to an input quickly. Feedback NNs are usually more difficult to train than feedforward NNs. The topology chosen for this analysis is a subcategory of the feedforward networks, namely the Multi-Layer Perceptron (MLP). MLPs are the logical extension of the "Perceptron", i.e. the simplest possible neural network topology: a single neuron which combines several inputs to obtain its output. MLP's structure consists of one "input layer", made of neurons having one single input each from the environment, one "output layer" whose neurons communicate their outputs to the user, and one or more "hidden layers" whose neurons have as input the outputs of all the neurons of the preceding layer and give their outputs as inputs of the following layer. No "horizontal" (i.e. between neurons of the same layer) communication is allowed. A scheme of a typical MLP is shown in Fig. 5.2.

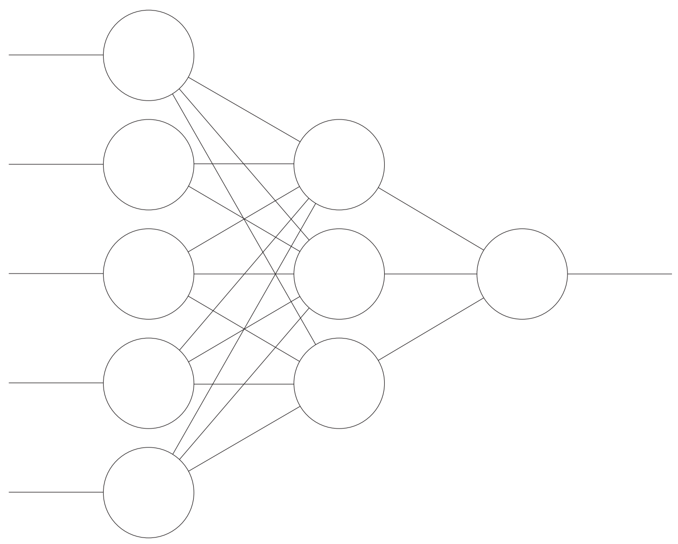

Figure 5.2: Typical architecture of a Multi-Layer Perceptron with a single hidden layer.

5.2 Supervised learning

In supervised training both the inputs and the desired outputs (*target values*) are provided. To start this process the initial weights are chosen randomly; then the network processes the inputs and compares its resulting outputs against the desired outputs. An "error" is defined, as the "distance" (in some metric) between the NN output and the target value. Typical error definitions are functions of $|O_i - T_i|$ (where O_i is output of the net for the i-th event, and T_i is the corresponding target value). Errors are then propagated back through the system, causing the system to adjust the weights which control the network, with the aim of minimizing the expected error at the next iteration. This process occurs over and over as the weights are continually tweaked. The set of data which enables the training is called the *"training sample"*. During the training of a network the same set of data is processed many times as the connection weights are ever refined.

It is important to have a second data set, the *"test sample"*, to which apply the neural network during the training phase to test its performance, with a given set of weights, on events that have not been used to adjust those weights. This is of great importance since a neural network is completely useless if it is not able to generalize. For example, if the training has gone too far the weights can be optimally adapted to the data set used to learn but the network can have poor performance on any other analogous data set; this is called *"over-learning"* (or *"over-fitting"*) and is usually avoided choosing a statistical criterion to stop the learning based on the performances on the test set, instead of the training set.

In "batch learning", in every learning cycle (or "epoch") the total error is summed over all the events in the training set and is the quantity to minimize; in "incremental learning", instead, the NN weights are updated after every single event.

Apart from the learning algorithm, supervised training methods can be classified in two subvarieties: auto-associative and hetero-associative. In auto-associative learning, the output variables are the same as the inputs, whereas in hetero-associative learning, the outputs are generally different from the inputs. For example, in most forecast problems one tries to predict the future values of some variables according to their history; in this case auto-associative learning is appropriate. An example of a problem in which hetero-associative training is used is the fitting of a function $y = f(x)$: if a data set of N values for the x variable is given, being f known, in every training cycle the network computes a single value as its best guess for y; this is compared with the target value $y_{true} = f(x)$, and the distance $y_{guess} - y_{true}$ is used to decide the adjustment of the synaptic weights.

A less trivial problem, in which neural nets are particularly useful, is the case in which the function f is completely unknown, but both x and the corresponding y are available as an experimental data set; the network, trained with N pairs (x_i, y_i), $i = 1, ..., N$, can "generalize", i.e. make a fair guess of the value y_{N+1} corresponding to x_{N+1}. An interesting feature of neural networks is that under certain assumptions they can approximate with arbitrary accuracy any continuous function. The method, of course, usually gives fairly good results even for quasi-continuous curves: the result will be an approximation of a continuous curve that approximates the discontinuous one.

Classification problems, when events have to be classified in one of two mutually exclusive categories, can be viewed as the fitting of a function having value *1* for one category (the "signal") and *0* for its complementary ensemble (the "background"). When N categories have to be recognized, it is customary to build neural networks with $N - 1$ outputs in order to map the categories in different regions of the $(N-1)$-dimensional space of the outputs. For example, to classify three categories of events A, B, C it is possible to assign target values $(0, 0)$ to A events, $(0, 1)$ to B events, and $(1, 1)$ to C events; after training, the A events should gather in the proximity of the point $(0, 0)$, and so on.

In classification problems, events of all the categories (which in this thesis are the signal, i.e. the $g \to c\bar{c}$ events, and the background consisting of all the other events) have to be shown to the net. Training a net with signal events only, for example, would yield a net which recognizes them with very high probability of success, but is completely unable to reject the background. A careful balance of the number of events for every category in the training sample is needed, to reach a fair separation between the different categories and so an optimal balance between high efficiency (i.e. low percentage of false negatives) and high purity (i.e. low percentage of false positives).

5.3 Training algorithms

The error (summed over the data set, or concerning a single event) is considered as a function of the synaptic weights of all the neurons, so that the learning process is essentially a minimization problem in the space of the weights. To perform this minimization several algorithms are available. Some of the most popular belong to the family of the "backpropagation algorithms", and are modified versions of the so called "standard backpropagation" [79].

"Standard backpropagation" is an application to the NNs of the *generalized delta rule*, also called the "heavy ball method" in the numerical analysis literature [80]: the variation of the weight for the link (synapsis) between unit j and unit i (with i being in the layer forward, in the net, with respect to j) is:

$$\Delta w_{ij} = \eta \delta_i o_j \tag{5.1}$$

where η is the "learning rate" (which is fixed in the standard backpropagation, but not in most of its variants), o_j is an output of the unit j (which is an input to i), and δ_i is a function of the outputs of the preceding layer's neurons which is calculated slightly differently for output and hidden units. For output units, under the assumption of a sigmoid[1] activation function, it is

$$\delta_i = o_i(1 - o_i)E_i \tag{5.2}$$

(where E_i is the error, as said above, calculated according to some distance definition), and for hidden units

$$\delta_i = o_i(1 - o_i) \sum_k \delta_k w_{ki} \tag{5.3}$$

In other words, the learning cycle in a backpropagation network involves the repetitive simultaneous presentation of matching input and output patterns while the weights are adjusted using a *gradient descent* search, i.e. the minimum is looked for by calculating the direction and magnitude of the gradient of the "error surface" in the space of the weights, $\frac{\partial E}{\partial w_{ij}}$. Thus a neural network can also be viewed as a non-linear numerical optimization procedure. Backpropagation can be used for both batch training (in which the weights are updated after processing the entire training set) and incremental training (in which the weights are updated after processing each case). For batch training, standard backpropagation usually converges to a local minimum, if one exists. For incremental training, standard backpropagation does not converge to a stationary point of the error surface. To obtain convergence, the learning rate must be slowly reduced, so that η in

[1] Best known to physicists as the Fermi-function: $f(x) = \frac{1}{1+e^{-x}}$. This function is approximately linear around $x = 0$ and reaches 0 or 1 asymptotically for large negative or large positive values of the argument, respectively. Due to this fact it is not possible for a single neuron to have an excessively large influence.

Eq. 5.1 is not constant anymore. This methodology is called "stochastic approximation" or "annealing".

Other variants are designed to avoid as much as possible to be trapped in a small local minimum failing to reach the absolute minimum. For example, a *momentum term* can be added in order to accelerate convergence when large plateaus of the error surface are present. At the n-th epoch the weights are updated as follows:

$$w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}(n) + \alpha \Delta w_{ij}(n-1) \tag{5.4}$$

where Δw_{ij} is the variation defined in Eq. 5.1, and $0 < \alpha < 1$ (typically $\alpha \approx 0.9$) is the momentum term, whose effect is to impart enough "momentum" to "roll" quickly over the plateau.

In this thesis the Resilient Propagation algorithm, or RPROP [81], one of the most popular variants of the Standard Backpropagation, has been used as training algorithm. In contrast to other gradient descent algorithms, RPROP does not use the magnitude of the gradient $\frac{\partial E}{\partial w_{ij}}$, but only its direction. It begins with an initial small update value, and then increases this value if the current gradient points towards the same direction as the previous gradient, but decreases it if the gradient has the opposite direction. This update value is then added to the weight, if the gradient points towards the minimum, and subtracted from the weight in the opposite case.

5.4 Optimal number of layers in a Multi-Layer Perceptrons

No general rule exists about the optimal number of units per (hidden) layer in a MLP for a given problem [77], so that it is customary to proceed by trial and error: beginning with only one unit per layer, the number of units is increased until the performance of the network ceases to improve. When the number of units is too large, the number of connections, and consequently the number of weights to variate, makes the training very slow and a larger training sample becomes necessary to avoid over-fitting.

Instead, some considerations can be made about the number of layers needed to properly treat a problem, according to its complexity. These extra layers are needed to take into account the nonlinearities of the function to approximate (see Ref. [75, 76]). Consider Fig. 5.3a, which shows a classification problem with two input variables: it is possible to find a single hyper-plane (in the bidimensional case a straight line) separating class "A" from class "B". The task of finding this hyper-plane is called a "linearly separable problem". This problem can be solved by a MLP with no hidden layer (a Perceptron), which essentially performs a *linear regression* [82]. After proper training with a target function which takes on 0 for "A" events and 1 for "B" events, the output of the NN will be close to 0 on the "A" side, to 1 on the other side, and will have intermediate values in the proximity of the separating line. Even if the separation hyper-surface (called *decision surface*) to learn is mildly nonlinear, a better generalization can be obtained with a simple Perceptron than with a multilayered net if there is too little data or too much noise to estimate the nonlinearities accurately.

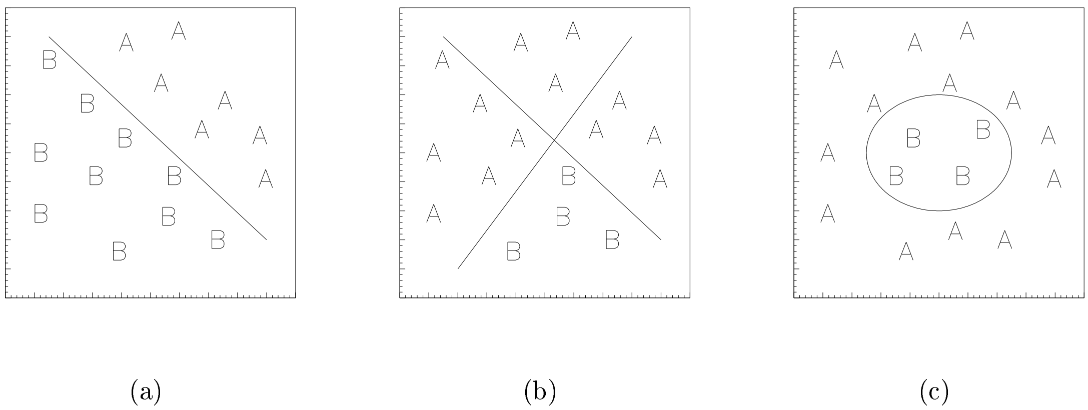

Figure 5.3: (a) Linearly separable problem, (b) non-linearly separable problem, (c) non-linearly separable problem with a closed contour.

Fig. 5.3b shows an example of "non-linearly separable problem": no straight line can properly separate the two categories of events. The introduction of a hidden layer overcomes the problem [83]. Intuitively, dividing the neurons in the hidden layer in two independent sub-groups, each one can be used by the net to find a straight line which confines all of the "B" events on one side, even if this means to have also a lot of "A" events on the same side. Working independently, the two groups will find, in general, different lines which obtain the same result in different ways; the neuron on the output layer can then make use of both the results. Other decision surfaces having more complex shapes can need more sub-groups, and so more units in the hidden layers, to be approximated. Problems with higher complexity (like the closed contour in Fig. 5.3c) can be better approximated with the help of a second hidden layer [84]. More than two hidden layers are shown to be unnecessary [77].

[73] The ALEPH collaboration, *The Forward-Backward Asymmetry for Charm Quarks at the Z*, Phys.Lett. **B 434** (1998) 415.

[74] The ALEPH collaboration, *Production of K^0 and Λ in hadronic Z decays*, Z.Phys. **C64** (1994) 361.

[75] S. Haykin, *Neural Networks, a comprehensive foundation* (2nd ed., 1999), Prentice Hall, Upper Saddle River (NJ).

[76] K. Gurney, *An Introduction to Neural Networks* (2002), UCL Press, London.

[77] W.S. Sarle, *Neural Network FAQ*, periodic posting to the Usenet newsgroup comp.ai.neural-nets, URL: ftp://ftp.sas.com/pub/neural/FAQ.html

[78] Andreas Zell et al., *Stuttgart Neural Network Simulator, User Manual, Version 4.1*, URL: http://www-ra.informatik.uni-tuebingen.de/SNNS/UserManual/UserManual.html

[79] D.E. Rumelhart, J.L. McClelland, eds. *Parallel Distributed Processing* (1986), MIT Press, Massachusetts.

[80] B.T. Polyak, Some methods of speeding up the convergence of iterative methods, Zh.Vychisl.Mat.Mat.Fiz. **4** (1964) 1.

[81] M. Riedmiller and H. Braun, *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm* (1993), Proc. ICNN, San Francisco.

[82] N. Draper, H. Smith *Applied Regression Analysis* (1991), John Wiley ed., New York.

[83] K. Hornik, M. Stinchcombe, H. White, *Multilayer feedforward networks are universal approximators*, Neur.Net., **2** (1989) 359.

[84] E.D. Sontag, *Feedback stabilization using two-hidden-layer nets*, IEEE Transactions on Neural Networks, **3** (1992) 981.

[85] A. Stocchi, *Experimental results on heavy quark hadrons*, Proceedings of the 31st International Conference on High Energy Physics, Amsterdam, 24-31 July 2002.