CHAPTER 13

Recurrent Networks

13. 1 INTRODUCTION

The learning algorithms discussed in the previous chapter were applied to feed-forward networks: all data flows in a network in which no cycles are present.

But what happens when we introduce a cycle? For instance, we can connect a hidden unit with itself over a weighted connection, connect hidden units to input units, or even connect all units with each other. Although, as we know from the previous chapter, the approximation capabilities of such networks do not increase, we may obtain decreased complexity, network size, etc. to solve the same problem.

An important question we have to consider is the following: what do we want to learn in a recurrent network? After all, when one is considering a recurrent network, it is possible to continue propagating activation values until a stable point (attractor) is reached. As we will see in the sequel, there exist recurrent network, which are attractor based, i.e., the activation values in the network are repeatedly updated until a stable point is reached after which the weights are adapted, but there are also recurrent networks where the learning rule is used after each propagation (where an activation value is transversed over each weight only once), while external inputs are included in each propagation. In such networks, the recurrent connections can be regarded as extra inputs to the network (the values of which are computed by the network itself).

In this chapter, recurrent extensions to the feed-forward network will be discussed. The theory of the dynamics of recurrent networks extends beyond the scope of a one-semester course on neural networks. Yet the basics of these networks will be discussed.

Also some special recurrent networks will be discussed: the Hopfield network, which can be used for the representation of binary patterns; subsequently we touch upon Boltzmann machines, therewith introducing stochasticity in neural computation.

13.2 THE GENERALISED DELTA - RULE IN RECURRENT NETWORKS

The back-propagation learning rule, introduced in chapter 12, can be easily used for training patterns in recurrent networks. Before we will consider this general case, however, we will first describe networks

where some of the hidden unit activation values are fed back to an extra set of input units (the Elman network), or where output values are fed back into hidden units (the Jordan network).

A typical application of such a network is the following. Suppose we have to construct a network that must generate a control command depending on an external input, which is a time series $x(t)$, $x(t-1)$, $x(t-2)$, …. With a feed-forward network there are two possible approaches:

1. Create inputs $x_1, x_2, \ldots, x_n$ which constitute the last n values of the input vector. Thus a 'time window' of the input vector is input to the network.
2. Create inputs $x, x', x'', \ldots$. Besides only inputting $x(t)$, we also input its first, second, etc. derivatives. Naturally, computation of these derivatives is not a trivial task for higher-order derivatives.

The disadvantage is, of course, that the input dimensionality of the feed-forward network is multiplied with n, leading to a very large network, which is slow and difficult to train. The Jordan and Elman networks provide a solution to this problem. Due to the recurrent connections, a window of inputs need not be input anymore; instead, the network is supposed to learn the influence of the previous time steps itself.

13.2.1 The Jordan Network

One of the earliest recurrent neural networks was the Jordan network. An example of this network is shown in Fig. 13.1. In the Jordan network, the activation values of the output units are fed back into the

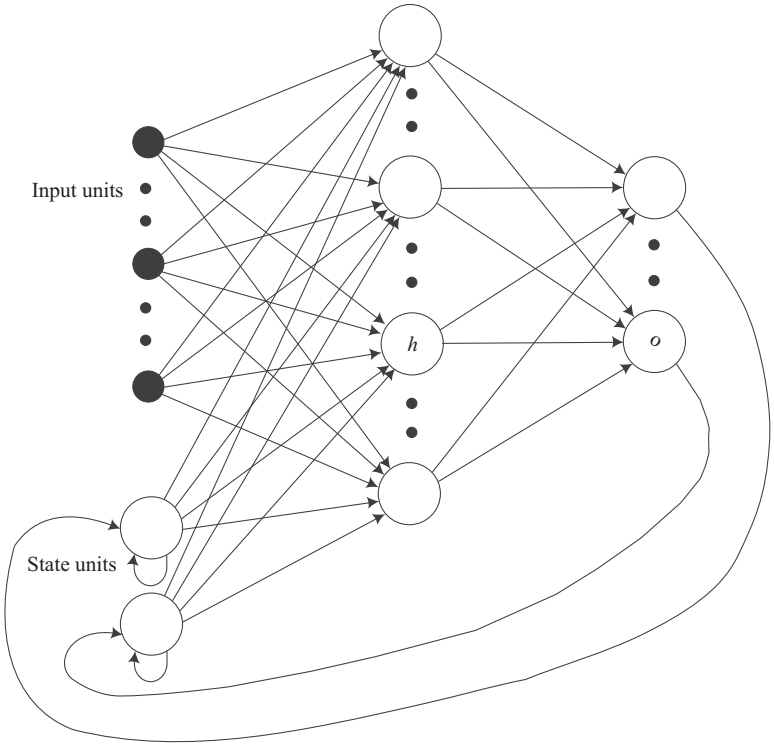

Fig. 13.1 The Jordan network. Output activation values are fed back to the input layer, to a set of extra neurons called the state units.

input layer through a set of extra input units called the state units. There are as many state units as there are output units in the network. The connections between the output and state units have a fixed weight of +1; learning takes place only in the connections between input and hidden units as well as hidden and output units. Thus all the learning rules derived for the multi-layer perceptron can be used to train this network.

13.2.2 The Elman Network

In the Elman network a set of context units are introduced, which are extra input units whose activation values are fed back from the hidden units. Thus the network is very similar to the Jordan network, except that (1) the hidden units instead of the output units are fed back; and (2) the extra input units have no self-connections.

The schematic structure of this network is shown in Fig. 13.2.

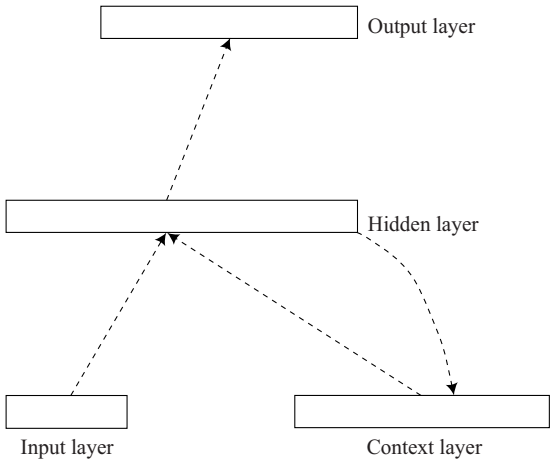

Fig. 13.2 The Elman network. With this network, the hidden unit activation values are fed back to the input layer, to a set of extra neurons called the context units.

Again the hidden units are connected to the context units with a fixed weight of value +1. Learning is done as follows:

1. The context units are set to 0; $t = 1$
2. Pattern x^t is clamped, the forward calculations are performed once;
3. The back-propagation learning rule is applied;
4. $t \leftarrow t + 1$; go to 2.

The context units at step t thus always have the activation value of the hidden units at step $t - 1$.

Example 13.1: As we mentioned above, the Jordan and Elman networks can be used to train a network on reproducing time sequences. The idea of the recurrent connections is that the network is able to 'remember' the previous states of the input values. As an example, we trained an Elman network on controlling an object moving in 1 D. This object has to follow a pre-specified trajectory x_d. To control the object, forces F must be applied, since the object suffers from friction and perhaps other external forces.

To tackle this problem, we use an Elman net with inputs x and x_d, one output F, and three hidden units. The hidden units are connected to three context units. In total, five units feed into the hidden layer.

The results of training are shown in Fig. 13.3. The same test can be done with an ordinary feed-forward network with sliding window input. We tested this with a network with five inputs, four of

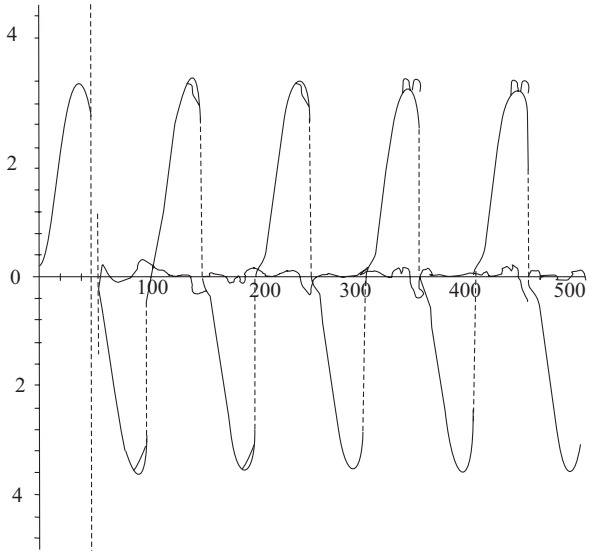

Fig. 13.3 Training an Elman network to control an object. The solid line depicts the desired trajectory x_d; the dashed line the realized trajectory. The third line is the error.

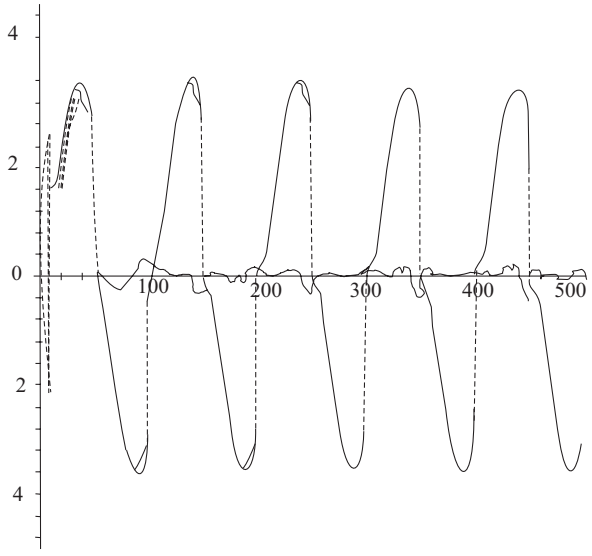

Fig. 13.4 Training a feed forward network to control an object. The solid line depicts the desired trajectory x_d; the dashed line the realized trajectory. The third line is the error.

which constituted the sliding window x_{-3}, x_{-2}, x_{-1} and x_0, and one the desired next position of the object. Results are shown in Fig. 13.4. The disappointing observation is that the results are actually better with the ordinary feed-forward network, which has the same complexity as the Elman network.

13.2.3 Back-Propagation in Fully Recurrent Networks

More complex schemes than the above are possible. For instance, independently of each other Pineda (1987) and Almeida (1987) discovered that error back-propagation is in fact a special case of a more general gradient learning method, which can be used for training attractor networks. However, also when a network does not reach a fixed point, a learning method can be used: back-propagation through time (Pearlmutter, 1989, 1990). This learning method, the discussion of which extents beyond the scope of our course, can be used to train a multi-layer perceptron to follow trajectories in its activation values.

13.3 THE HOPFIELD NETWORK

One of the earliest recurrent neural networks reported in literature was the auto-associator independently described by Anderson (1977) and Kohonen (1977). It consists of a pool of neurons with connections between each unit i and j, $i \neq j$ (see Fig. 15.5). All connections are weighted.

Hopfiled (1982) brings together several earlier ideas concerning these networks and presents a complete mathematical analysis.

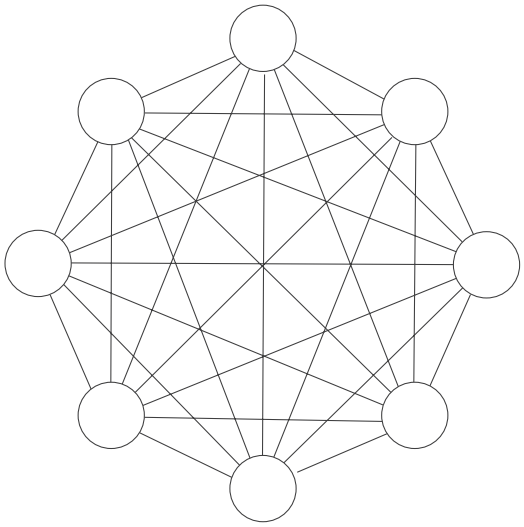

Fig. 13.5 The auto associator network. All neurons are both input and output neurons, i.e., a pattern is clamped, the network iterates to a stable state, and the output of the network consists of the new activation values of the neurons.

13.3.1 Description

The Hopfield network consists of a set of N interconnected neurons (Fig. 13.5), which update their activation values asynchronously and independently of other neurons. All neurons are both input and output neurons. The activation values are binary. Originally, Hopfield chose activation values of 1 and 0, but using values +1 and –1 presents some advantages discussed below. We will therefore adhere to the latter convention.

The state of the system is given by the activation values $Y = y(k)$. The net input $S_k(t + 1)$ of a neuron k at cycle $t + 1$ is a weighted sum

$$S_k(t + 1) = \sum_{j \neq k} y_j(t) w_{jk} + \theta_k \qquad ...(13.1)$$

A simple threshold function (Fig. 10.2) is applied to the net input to obtain the new activation value $y_i(t + 1)$ at time $t + 1$:

$$y_k(t + 1) = \begin{cases} +1 & \text{if } S_k(t+1) > U_k \\ -1 & \text{if } S_k(t+1) < U_k \\ y_k(t) & \text{otherwise} \end{cases} \qquad ...(13.2)$$

i.e., $yk(t + 1) = \text{sgn}(S_k(t + 1))$ For simplicity we henceforth choose $U_k = 0$, but this is of course not essential.

A neuron k in the Hopfield network is called stable at time t if, in accordance with equations (13.1) and (13.2),

$$y_k(t) = \text{sgn}(S_k(t - 1)) \qquad ...(13.3)$$

A state α is called stable if, when the network is in state α, all neurons are stable. A pattern x^p is called stable if, when x^p is clamped, all neurons are stable.

When the extra restriction $w_{jk} = w_{kj}$ is made, the behavior of the system can be described with an energy function

$$\varepsilon = -\frac{1}{2} \sum_{j \neq k} \sum \sum y_j \, y_k \, w_{jk} - \sum_k \theta_k y_k \qquad ...(13.4)$$

Theorem 13.1: A recurrent network with connections $w_{jk} = w_{kj}$ in which the neurons are updated using rule (13.2) has stable limit points.

Proof: First, note that the energy expressed in eq. (13.4) is bounded from below, since the y_k are bounded from below and the w_{jk} and θ_k are constant. Secondly, ε is monotonically decreasing when state changes occur, because

$$\Delta\varepsilon = -\Delta y_k \left(\sum_{j \neq k} y_j w_{jk} + \theta_k \right) \qquad ...(13.5)$$

is always negative when y_k changes according to eqs. (13.1) and (13.2).

The advantage of a + 1/–1 model over a 1/0 model then is symmetry of the states of the network. For, when some pattern x is stable, its inverse is stable, too, whereas in the 1/0 model this is not always true (as an example, the pattern 00 … 00 is always stable, but 11 … 11 need not be). Similarly, both a pattern and its inverse have the same energy in the +1/–1 model.

Removing the restriction of bidirectional connections (i.e., $w_{jk} = w_{kj}$) results in a system that is not guaranteed to settle to a stable state.

13.3.2 Hopfield Network as Associative Memory

A primary application of the Hopfield network is an associative memory. In this case, the weights of the connections between the neurons have to be thus set that the states of the system corresponding with the patterns which are to be stored in the network are stable. These states can be seen as 'dips' in energy space. When the network is cued with a noisy or incomplete test pattern, it will render the incorrect or missing data by iterating to a stable state, which is in some sense 'near' to the cued pattern.

The Hebb rule can be used to store P patterns:

$$w_{jk} = \begin{cases} \sum_{p=1}^{P} x_j^p x_k^p & \text{if } j \neq k \\ 0 & \text{otherwise} \end{cases} \qquad ...(13.6)$$

i.e., if x_j^p and x_k^p are equal, w_{jk} is increased, otherwise decreased by one (note that, in the original Hebb rule, weights only increase). It appears, however, that the network gets saturated very quickly, and that about $0{:}15N$ memories can be stored before recall errors become severe.

There are two problems associated with storing too many patterns:

1. The stored patterns become unstable;
2. Spurious stable states appear (i.e., stable states which do not correspond with stored patterns).

The first of these two problems can be solved by an algorithm proposed by Bruce et al. (Bruce, Canning, Forrest, Gardner, & Wallace, 1986).

Algorithm 13.1: Given a starting weight matrix $W = [w_{jk}]$, for each pattern x^p to be stored and each element x_k^p in x^p define a correction ε_k such that

$$\varepsilon_k = \begin{cases} 0 & \text{if } y_k \text{ is stable and } x^p \text{ is clamped} \\ 1 & \text{otherwise} \end{cases} \qquad ...(13.7)$$

Now modify w_{jk} by $\Delta w_{jk} = y_j\, y_k(\varepsilon_j + \varepsilon_k)$ if $j \neq k$. Repeat this procedure until all patterns are stable.

It appears that, in practice, this algorithm usually converges. There exist cases, however, where the algorithm remains oscillatory (try to find one)!

The second problem stated above can be alleviated by applying the Hebb rule in reverse to the spurious stable state, but with a low learning factor (Hopfield, Feinstein, & Palmer, 1983). Thus these patterns are weakly unstored and will become unstable again.

13.3.3 Neurons with Graded Response

The network described in section 13.3.1 can be generalized by allowing continuous activation values. Here, the threshold activation function is replaced by a sigmoid. As before, this system can be proved to be stable when a symmetric weight matrix is used (Hopfield, 1984).

13.3.4 Hopfield Networks for Optimization Problems

An interesting application of the Hopfield network with graded response arises in a heuristic solution to the NP-complete traveling salesman problem (Garey & Johnson, 1979). In this problem, a path of minimal distance must be found between n cities, such that the begin- and end-points are the same.

Hopfield and Tank (1985) use a network with $n \times n$ neurons. Each row in the matrix represents a city, whereas each column represents the position in the tour. When the network is settled, each row and each column should have one and only one active neuron, indicating a specific city occupying a specific position in the tour. The neurons are updated using rule (13.2) with a sigmoid activation function between 0 and 1. The activation value $y_{xj} = 1$ indicates that city X occupies the j^{th} place in the tour.

An energy function describing this problem can be set up as follows. To ensure a correct solution, the following energy must be minimized:

$$\varepsilon = \frac{A}{2}\sum_{X}\sum_{j}\sum_{k\neq j} y_{Xj}y_{Xk} + \frac{B}{2}\sum_{j}\sum_{X}\sum_{X\neq Y} y_{Xj}y_{Yj} + \frac{C}{2}\left(\sum_{X}\sum_{j} y_{Xj} - n\right)^2 \qquad ...(13.8)$$

where A, B, and C are constants. The first and second terms in equation (13.8) are zero if and only if there is a maximum of one active neuron in each row and column, respectively. The last term is zero if and only if there are exactly n active neurons.

To minimise the distance of the tour, an extra term

$$\varepsilon = \frac{D}{2}\sum_{X}\sum_{Y\neq X}\sum_{j} d_{XY}y_{Xj}\left(y_{Y,\,j+1} + y_{Y,\,j-1}\right) \qquad ...(13.9)$$

is added to the energy, where d_{XY} is the distance between cities X and Y and D is a constant. For convenience, the subscripts are defined modulo n.

The weights are set as follows:

$$\begin{aligned} w_{Xj,\,Yk} &= -A\delta_{XY}(1-\delta_{jk}) \text{ inhibitory connections within each row} \\ &= -B\delta_{jk}(1-\delta_{XY}) \text{ inhibitory connections within each column} \\ &= -C \text{ global inhibition} \\ &= -Dd_{XY}(\delta_{k,\,j+1} + \delta_{k,\,j\ 1}) \text{ data term} \end{aligned} \qquad ...(13.10)$$

where $\delta_{jk} = 1$ if $j = k$ and 0 otherwise. Finally, each neuron has an external bias input C_n.

Although this application is interesting from a theoretical point of view, the applicability is limited. Whereas Hopfield and Tank state that the network converges to a valid solution in 16 out of 20 trials while 50% of the solutions are optimal, other reports show less encouraging results. For example, (Wilson and Pawley, 1988) find that in only 15% of the runs a valid result is obtained, few of which lead

to an optimal or near-optimal solution. The main problem is the lack of global information. Since, for an N-city problem, there are $N!$ possible tours, each of which may be traversed in two directions as well as started in N points, the number of different tours is $N!/2N$. Differently put, the N-dimensional hypercube in which the solutions are situated is $2N$ degenerate. The degenerate solutions occur evenly within the hypercube, such that all but one of the final $2N$ configurations are redundant. The competition between the degenerate tours often leads to solutions which are piecewise optimal but globally inefficient.

13.4 BOLTZMANN MACHINES

The Boltzmann machine, as first described by Ackley, Hinton, and Sejnowski in 1985 is a neural network that can be seen as an extension to Hopfield networks to include hidden units, and with a stochastic instead of deterministic update rule. The weights are still symmetric. The operation of the network is based on the physics principle of annealing. This is a process whereby a material is heated and then cooled very, very slowly to a freezing point. As a result, the crystal lattice will be highly ordered, without any impurities, such that the system is in a state of very low energy. In the Boltzmann machine this system is mimicked by changing the deterministic update of equation (13.2) in a stochastic update, in which a neuron becomes active with a probability p,

$$p(y_k \leftarrow +1) = \frac{1}{1 + e^{-\Delta\varepsilon_k / T}} \qquad ...(13.11)$$

where T is a parameter comparable with the (synthetic) temperature of the system. This stochastic activation function is not to be confused with neurons having a sigmoid deterministic activation function.

In accordance with a physical system obeying a Boltzmann distribution, the network will eventually reach 'thermal equilibrium' and the relative probability of two global states α and β will follow the Boltzmann distribution

$$\frac{P_\alpha}{P_\beta} = e^{-(\varepsilon_\alpha - \varepsilon_\beta)/T} \qquad ...(13.12)$$

where P_α is the probability of being in the α^{th} global state, and ε_α is the energy of that state. Note that at thermal equilibrium the units still change state, but the probability of finding the network in any global state remains constant.

At low temperatures there is a strong bias in favor of states with low energy, but the time required to reach equilibrium may be long. At higher temperatures the bias is not so favorable but equilibrium is reached faster. A good way to beat this trade-off is to start at a high temperature and gradually reduce it. At high temperatures, the network will ignore small energy differences and will rapidly approach equilibrium. In doing so, it will perform a search of the coarse overall structure of the space of global states, and will find a good minimum at that coarse level. As the temperature is lowered, it will begin to respond to smaller energy differences and will find one of the better minima within the coarse-scale minimum it discovered at high temperature.

As multi-layer perceptions, the Boltzmann machine consists of a non-empty set of visible and a possibly empty set of hidden units. Here, however, the units are binary-valued and are updated stochastically and asynchronously. The simplicity of the Boltzmann distribution leads to a simple learning procedure, which adjusts the weights so as to use the hidden units in an optimal way (Ackley et al., 1985). This algorithm works as follows:

- First, the input and output vectors are clamped.
- The network is then annealed until it approaches thermal equilibrium at a temperature of 0. It then runs for a fixed time at equilibrium and each connection measures the fraction of the time during which both the units it connects are active. This is repeated for all input-output pairs so that each connection can measure $(y_j\, y_k)^{\text{clamped}}$, the expected probability, averaged over all cases, that units j and k are simultaneously active at thermal equilibrium when the input and output vectors are clamped.
- Similarly, $(y_j\, y_k)^{\text{free}}$ is measured when the output units are not clamped but determined by the network.
- In order to determine optimal weights in the network, an error function must be determined. Now, the probability $P^{\text{free}}(Y^p)$ that the visible units are in state $Y^p$when the system is running freely can be measured. Also, the desired probability $P^{\text{clamped}}(Y^p)$that the visible units are in state (Y^p) is determined by clamping the visible units and letting the network run.
- Now, if the weights in the network are correctly set, both probabilities are equal to each other, and the error E in the network must be 0. Otherwise, the error must have a positive value measuring the discrepancy between the network's internal mode and the environment. For this effect, the 'asymmetric divergence' or 'Kullback information' is used:

$$E = \sum_{p} P^{\text{clamped}}(Y^p) \log \frac{P^{\text{clamped}}(Y^p)}{P^{\text{free}}(Y^P)} \qquad ...(13.13)$$

Now, in order to minimize E using gradient descent, we must change the weights according to

$$\Delta w_{jk} = -\gamma \frac{\partial E}{\partial w_{jk}} \qquad ...(13.14)$$

It is not difficult to show that

$$\frac{\partial E}{\partial w_{jk}} = -\frac{1}{T}\left[(y_j y_k)^{\text{clamped}} - (y_j y_k)^{\text{free}}\right] \qquad ...(13.15)$$

Therefore, each weight is updated by

$$\Delta w_{jk} = \gamma\left[(y_j y_k)^{\text{clamped}} - (y_j y_k)^{\text{free}}\right] \qquad ...(13.16)$$