Chapter 6

Recurrent Neural Networks

6.1 Introduction

Common for all neural network models we have studied so far is the lack of feedback connections. We will now study networks with such connections! Recurrent networks are typically used when we are dealing with sequence data. It can be text data, speech data or numerical times series data coming from eg. sensors or stock markets. The feedback connections are used to capture the short and long term temporal dependencies in the data. We will however also look into ways of dealing with sequence data using ordinary MLPs. Let's start with that!

6.2 Time delay networks

The time delay neural network is an ordinary MLP with no feedback connections. All of the possible temporal dependencies are modeled using a fixed number of delayed copies of the sequence data. Let's take an example! Suppose we have a single time series, eg. the yearly sunspot number time series, as shown in figure 6.1. The task is now to train a time delay network that can predict next years sunspot number. Suppose the sunspot data is given in $x(t)$. The approach is to use a fixed number of "previous" values of $x(t)$ in order to predict $x(t+1)$. From $x(t)$ we create the following input-output dataset that will be used to train the MLP.

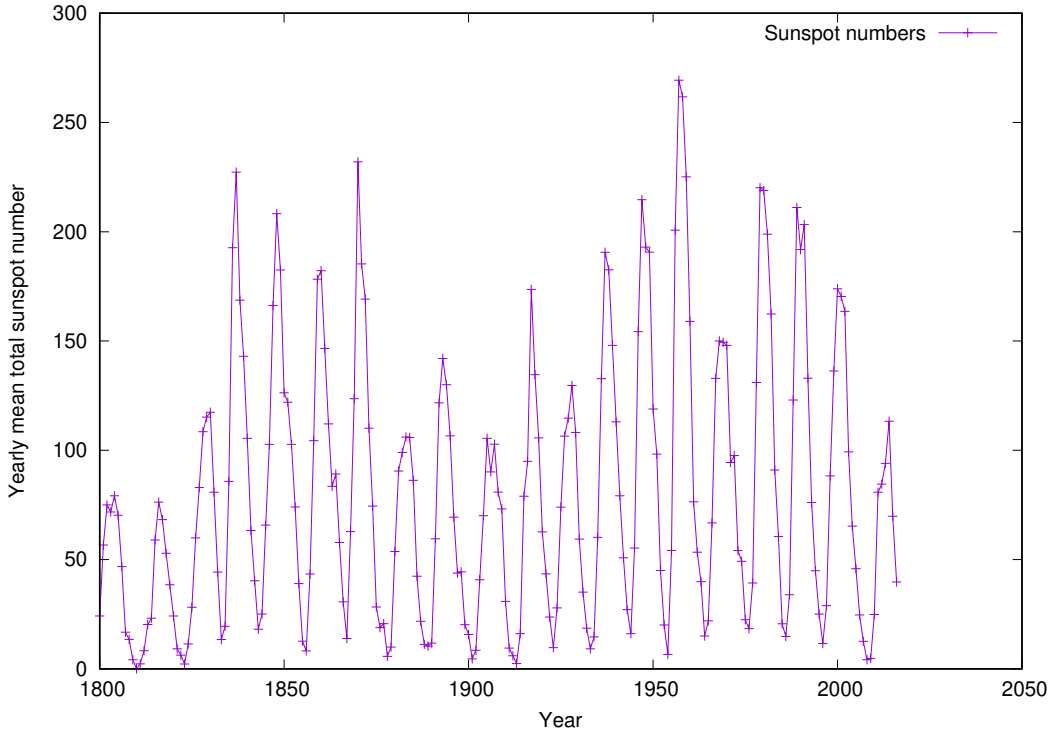

Figure 6.1: The sunspot number timeseries from 1800 to 2016.

Data no	Input	Target
1	$x(t-1), x(t-2), x(t-6), x(t-12)$	$x(t)$
2	$x(t-2), x(t-3), x(t-7), x(t-13)$	$x(t-1)$
3	$x(t-3), x(t-4), x(t-8), x(t-14)$	$x(t-2)$
...	...	...

Here we use four time delays, $t-1, t-2, t-6, t-12$, meaning that the MLP will have four inputs (and a single output). The possible temporal dependencies needed to model this time series are static since we have specified the exact nature of this dependence. The advantage of using time delay networks for sequence data is the usage of non-recurrent networks that can be simpler to train.

Figure 6.2 shows an example when training a time delay neural network for the sunspot data above. Here five inputs were used in an MLP with 4 hidden nodes and a single output for making one year into the future predictions. The inputs were $x(t-1), x(t-2, x(t-3), x(t-6), x(t-12)$ and the corresponding target value was $x(t)$. The data was divided into a training and a test set. Training data consisted of the years between 1700-1930 and test data between 1931-2016. The network was trained without any regularization until

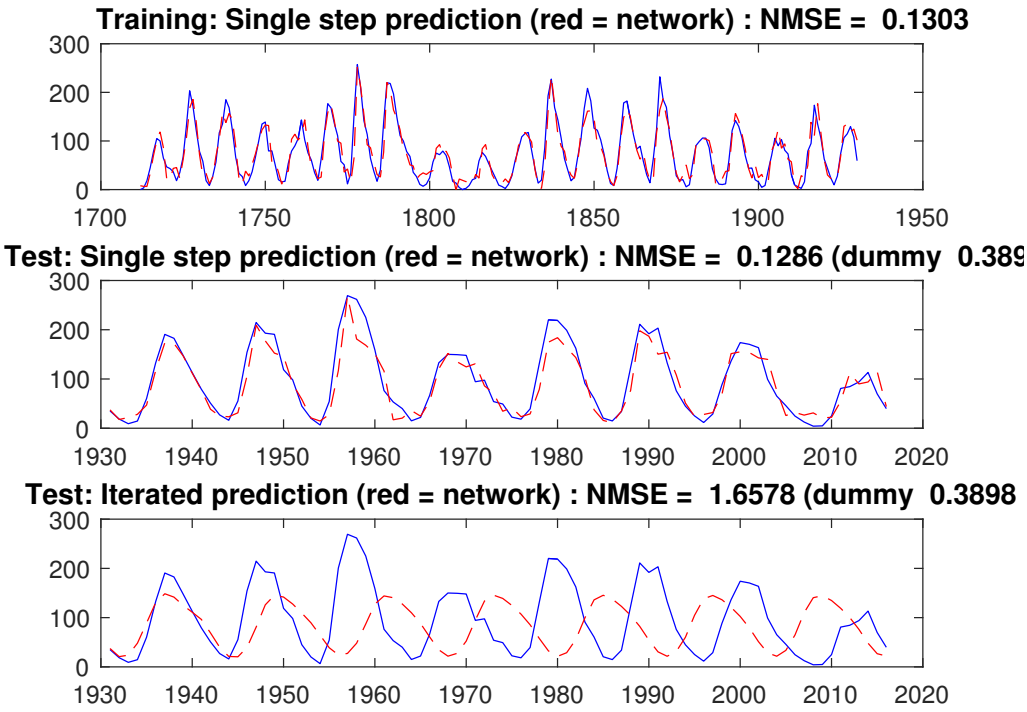

Figure 6.2: The result of training a time delay neural network to predict the yearly sunspot number. This network used 4 hidden nodes and 5 inputs consisting of time delayed inputs at $t-1, t-2, t-3, t-6, t-12$. The "NMSE" refers to the normalized mean square error.

convergence using SGD.

The top graph in figure 6.2 shows the training performance as single step prediction. Meaning that the network was always fed with correct input values and not predicted ones. The middle graph shows the test performance, again single step prediction. As a comparison we can use the very simple model,

$$y(t)_{\text{dummy}} = x(t-1)$$

meaning that the predicted yearly sunspot number is the same as previous year. The error for this model is referred to in the text as the "dummy" model. The lower graph in the figure shows so called iterated prediction, where test predictions are used as input to the network to predict coming years sunspot number. Here we can see that this is a much more difficult task!

So, one can use time delay networks if the there is a simple "static" dependence on a fixed number of previous time steps for the sequence data.

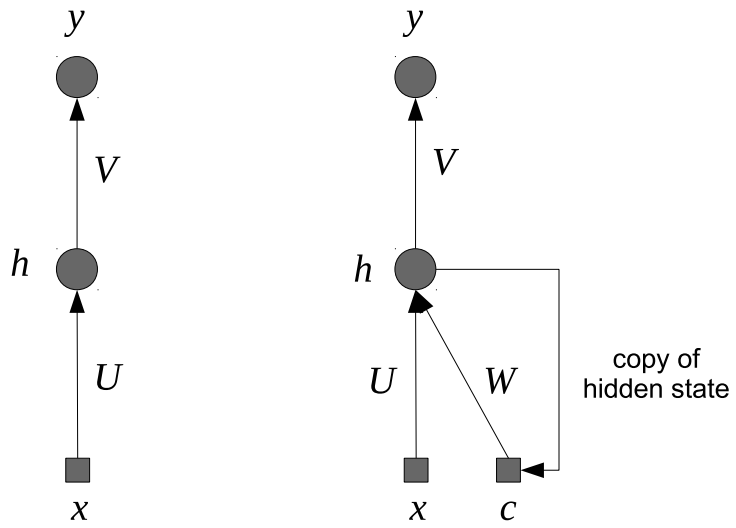

Figure 6.3: (Left) A simple 1-1-1 network. (Right) A simple recurrent network were a copy of the hidden state is used as an input to the network.

6.3 Simple recurrent networks, Elman type

We are now going to look at a class of networks that often goes under the name of "simple recurrent networks". They are also sometimes called *Elman* networks. Figure 6.3 shows such a simple network (right graph). For simplicity the bias node has been omitted. There is a reason for drawing this network in such a way, using the phrase "copy of hidden state". In principle we do not have any trainable feedback weights in this network, it only consists of feed-forward weights that can be trained using *almost* ordinary backpropagation.

So, how does the Elman network work? First we need to introduce a sequence index, let's use t. We have inputs in form a sequence $x(0), x(1), ..., x(T)$. The basic operation for the feedback network in figure 6.3 is,

$$y(t) = g_{\mathrm{o}}\left(Vh(t)\right) \tag{6.1}$$
$$h(t) = g_{\mathrm{h}}\left(Ux(t) + Wc(t)\right) \tag{6.2}$$
$$c(t) = h(t-1) \tag{6.3}$$

where g_{o} and g_{h} are the activation functions for the output and the hidden node respectively. So the input to hidden node is a weighted sum of the input and the **previous** value of the hidden node. Given the input sequence $x(t)$ one can produce an output sequence $y(t)$ according to eqn 6.1 above.

Training the above network can be done in many different ways. The idea presented below follows the original model proposed by Elman (more or less). The key point here is that we

forget that we have a dependence on the history through $c(t)$ and treat the training of this network as training for an ordinary MLP. Let's assume a simple mean square error function of the form,

$$E = \frac{1}{T}\sum_{t=0}^{T} E_t = \frac{1}{T}\sum_{t=0}^{T}(d(t) - y(t))^2$$

where $d(t)$ is the target sequence that we would like to model. The gradient descent approach means that we need, at some point, to compute the derivatives like $\partial y(t)/\partial W$. We have

$$\begin{aligned}
\frac{\partial y(t)}{\partial V} &= g_{\mathrm{o}}'(\cdot)h(t) \\
\frac{\partial y(t)}{\partial U} &= g_{\mathrm{o}}'(\cdot)V g_{\mathrm{h}}'(\cdot)x(t) \\
\frac{\partial y(t)}{\partial W} &= g_{\mathrm{o}}'(\cdot)V g_{\mathrm{h}}'(\cdot)c(t)
\end{aligned}$$

where the notation $g_{\mathrm{o}}'(\cdot)$ simply means the derivative of g_{o} evaluated at the current argument. But did we not miss something here? Since $c(t)$ in principle also depends on W through the relation $c(t) = h(t-1)$, meaning that we should treat the derivative,

$$\frac{\partial}{\partial W}(Wc(t))$$

as a derivative of a product. We will, for this version of the Elman network, simply ignore this dependence. This is why we use the phrase "copy of hidden state" in figure 6.3. We simply treat the network as a feed-forward network. The original Elman training was using "on-line" according to,

1. present an input $x(t)$
2. compute the hidden node value using $x(t)$ and the previous hidden node value $c(t)$. If $t = 0$ then $c(0)$ is typically set to zero
3. compute the output $y(t)$
4. perform all calculations (backpropagation) to compute all weight updates for this input "pattern"
5. update the weights V, U, W
6. go back to item 1, using the next input value in the sequence $(t \to t+1)$

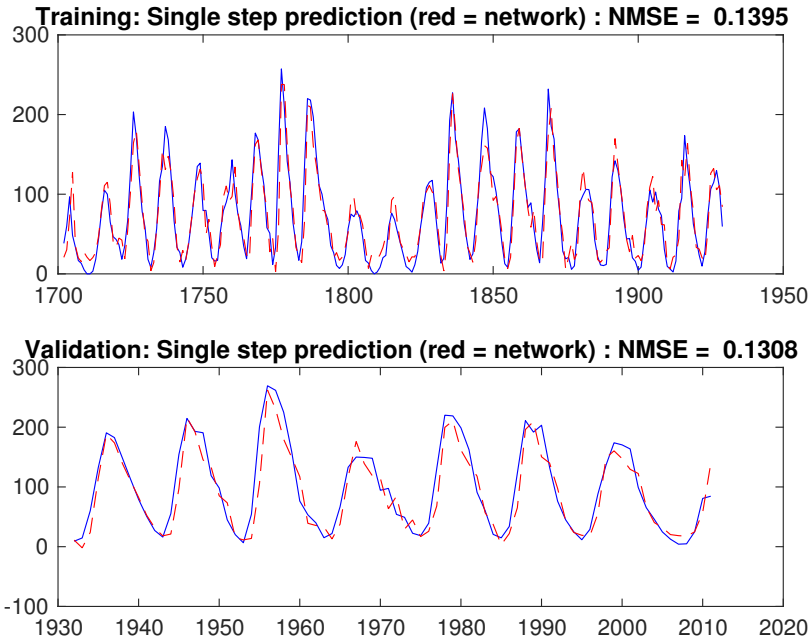

Figure 6.4: The result of training an Elman network for the sunspot number time series.

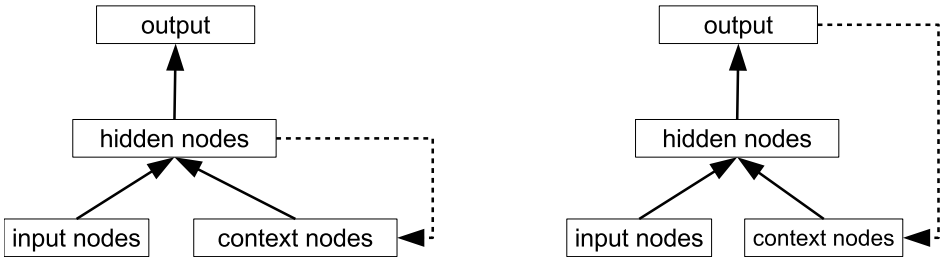

Figure 6.5: (Left) An Elman network. (Right) A Jordan network.

Figure 6.4 show the result of training an Elman network (using Matlab and the neural network toolbox). Here 5 hidden nodes were used, meaning that the Elman network had 6 inputs (the single time series input + 5 context nodes). The network was trained using SGD with no regularization.

Figure 6.5 shows the general Elman architecture together with the so called Jordan network where the feedback is coming from the output layer instead of the hidden layer. The extra input nodes are often called *context* nodes. Next we are still going to look at simple recurrent networks, but this time we are going to treat feedback weights as "real" feedback weights.

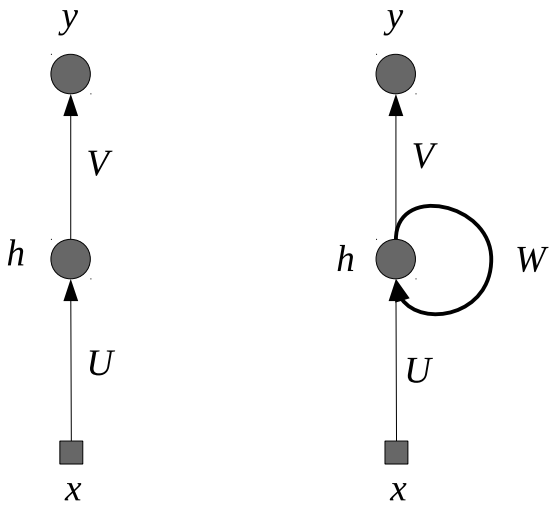

Figure 6.6: (Left) A simple 1-1-1 network. (Right) The same network but with an added feedback weight from the hidden node to itself.

6.4 Simple recurrent networks, general type

We will now start to look at networks were we have "true" feedback connections. Figure 6.6, again shows a very simple network with one input, one hidden node and one output node (left graph). For simplicity the bias weights are not shown in this plot. In the graph to the right we have now added a feedback weight W, feeding from the hidden node to itself. It is similar to the Elman network in figure 6.3, but we are going to treat this feedback weight in a proper way.

Again we have input data in form of a sequence $x(0), x(1), ..., x(T)$, with t being the sequence index. The operation of this network is now,

$$y(t) = g_{\mathrm{o}}\left(Vh(t)\right) \tag{6.4}$$
$$h(t) = g_{\mathrm{h}}\left(Ux(t) + Wh(t-1)\right) \tag{6.5}$$

again, g_{o} and g_{h} are the activation functions for the output and the hidden node respectively. Let's be explicit for a few sequence steps. (To avoid clutter I will use the notation $x(t) \to x_t$,

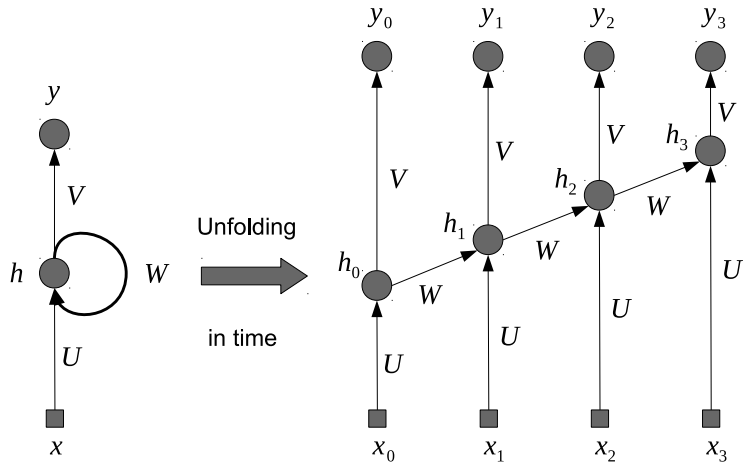

Figure 6.7: Unfolding in time. The simple feedback network to the left is "unfolded" for three time steps. The resulting network looks like a feed-forward network, although with a not so typical structure. Note that the weights are shared between the different layers.

$y(t) \to y_t$ and $h(t) \to h_t$ from now on. Also below, $g_{\mathrm{o}}(\cdot)$ and $g_{\mathrm{o}}[\cdot]$ means the same thing.),

$$
\begin{aligned}
y_0 &= g_{\mathrm{o}}\big[Vh_0\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_0\big)\big] \qquad \text{(using the initial condition } h(-1)=0\text{)} \\
y_1 &= g_{\mathrm{o}}\big[Vh_1\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_1 + Wh_0\big)\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_1 + Wg_{\mathrm{h}}\big(Ux_0\big)\big)\big] \\
y_2 &= g_{\mathrm{o}}\big[Vh_2\big] = g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_2 + Wh_1\big)\big] = \\
&= g_{\mathrm{o}}\big[Vg_{\mathrm{h}}\big(Ux_2 + Wg_{\mathrm{h}}\big(Ux_1 + Wh_0\big)\big)\big] = \\
&= g_{\mathrm{o}}\Big[Vg_{\mathrm{h}}\Big(Ux_2 + Wg_{\mathrm{h}}\big(Ux_1 + Wg_{\mathrm{h}}\big(Ux_0\big)\big)\Big)\Big]
\end{aligned}
$$

If we look at the last expression we see that it to some extent looks like an MLP with 3 hidden layers, although not fully connected and with shared weights. The procedure of turning a recurrent network into a MLP like structure is called *unfolding in time*. Figure 6.7 is showing our network unfolded in time for three time steps. Obviously this network will have many layers if we have long sequences.

6.4.1 Backpropagation through time (BPTT)

Let's look at the above simple network and see how it can be trained. First we need a training dataset! Again, assume that we have a an input sequence $x_t, t = 0, ..., T$ and a target sequence $d_t, t = 0, ..., T$. We can define an error function as

$$E(U, W, V) = \sum_{t=0}^{T} E_t(U, W, V)$$

Here E_t can be the usual square difference between y_t and d_t, but we can also imagine other possibilities such that we want to classify the input sequence into different classes (e.g. sequence of words that should be classified). To train using gradient descent we need to compute,

$$\begin{aligned}\frac{\partial E}{\partial V} &= \sum_t \frac{\partial E_t}{\partial V} \\ \frac{\partial E}{\partial W} &= \sum_t \frac{\partial E_t}{\partial W} \\ \frac{\partial E}{\partial U} &= \sum_t \frac{\partial E_t}{\partial U}\end{aligned}$$

The derivative of V is rather simple, since it sits at the topmost level. We have

$$\frac{\partial E_t}{\partial V} = \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial V} = \frac{\partial E_t}{\partial y_t} g_{\mathrm{o}}'(Vh_t)h_t$$

It is however more difficult for both W and U. Let's look at

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial W}$$

Now $h_t = g_{\mathrm{h}}(Ux_t + Wh_{t-1})$ (see eqn 6.5), where also h_{t-1} depends on W and so on, all the way down to the first "layer" (see figure 6.7). The correct way to write this is then,

$$\frac{\partial E_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W} \tag{6.6}$$

We have a similar expression for the derivative with respect to U. In principle what we have done here is exactly what we would have done to compute the gradients in a MLP with many hidden layer. The key difference is that we here share weights which means that we have to sum up the gradients for e.g. W along the layers (i.e. time steps). This way of training a recurrent network by unrolling it in time and do standard backpropagation is called *backpropagation through time (BPTT)*. This method is common when training simple recurrent networks, however as it stands (eqn 6.6) it has problems with gradients that can become small or become very large.

This vanishing or exploding gradient problem can be understood by looking at the equation 6.6 again. The derivative $\partial h_t/\partial h_k$ is in itself a chainrule. For example,

$$\frac{\partial h_3}{\partial h_1} = \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}$$

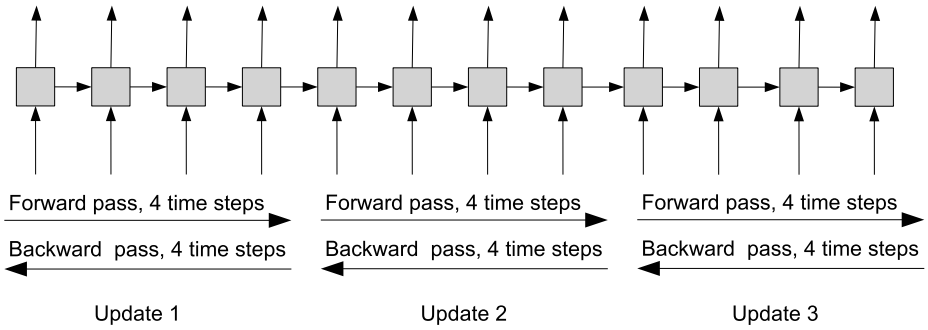

Figure 6.8: Truncated BPTT. Instead of making a full forward pass and a full backward pass, the sequence is divided into several smaller sequences. This will efficiently reduce the problem of vanishing gradients. It is common to remember the state of all nodes before starting the next sub-sequence.

Remember that $h_t = g_{\mathrm{h}}\left(Ux_t + Wh_{t-1}\right)$, leading to,

$$\frac{\partial h_{t+1}}{h_t} = g'_{\mathrm{h}}(\cdot)W$$

So for very long sequences we will get many multiplications of such gradient terms. Two numerical problems can occur:

- Vanishing graients
- Exploding gradients

If either or both the weights and the gradients are small then this will lead to the vansishing gradient problem. This causes training to be very slow, especially for sequence data with long time dependencies. There is also a risk of expoding gradients if the derivative of the activation function does not vanish, but the weight starts to grow, then multiplication of many terms can result in very large gradients causing the learning to be unstable. There are a few ways to combat the vanishing and the exploding gradient problem, one can make sure that all of the weights are properly initialized, use rectifier linear units instead of tanh() or sigmoid activation functions. But it is difficult to fully circumvent the problem. Another very common approach is to truncate the BPTT method. This means that we do not backpropagate for all time steps instead we run the BPTT method for a number of smaller sequences. Figure 6.8 is illustrating truncated BPTT. Another method to combat the vanishing gradient problem is to change the behavior of the hidden feedback nodes. Next section is looking at two such networks.

As a concluding remark, we have in all examples for this section (6.4) only had one single input and a signle hidden node. Of course in practical examples we may have multiple

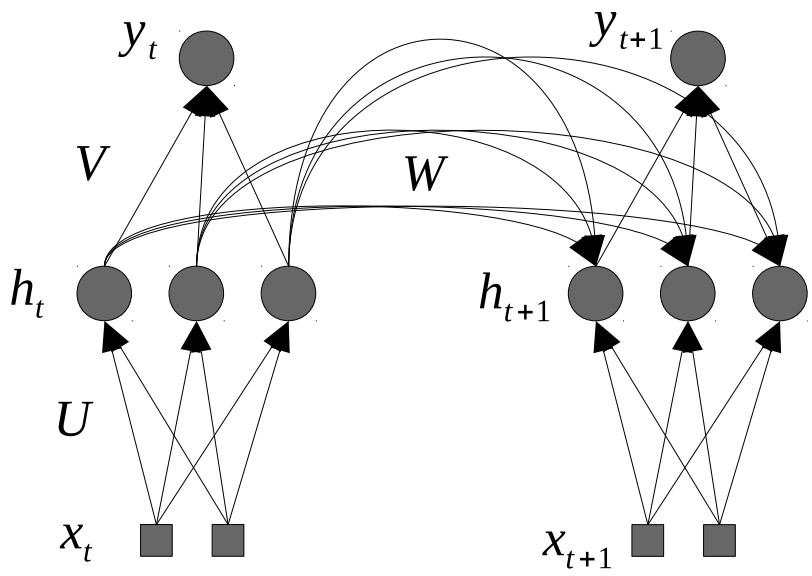

Figure 6.9: A more general recurrent network with three hidden nodes and two input nodes. Note that all hidden nodes have feedback connections to all other hidden nodes. This network is unfolded one time step.

inputs, many hidden nodes and even more than one hidden layer. This means that the weights U, W, V will be replaces by appropriate sized matrices instead, as illustrated by figure 6.9. However unfolding in time is as before, it is just that the unfolded network is a bit more complex.

6.5 LSTM networks

As we saw in the previous session the BPTT approach to training a recurrent networks will have problem for long sequences because of the vanishing gradient problem. To combat this problem and to come up with an architecture that could handle long time dependencies in an efficient way, the Long Short-Term Memory (LSTM) network was introduced (1997 by Schmidhuber).

To introduce the LSTM network we are going to start by looking at the simple network that we had before, see figure 6.10. The "U" box now represents the hidden node and consists of h_t and x_t feeding into a typical tanh() function. In the LSTM network we are going to replace the "U" box in figure 6.10 with a LSTM box as in figure 6.11. Apart from the extra c_t value we can just consider the "LSTM" box as another way of computing the new hidden value h_t. In this respect the LSTM is similar to the simple feedback node, however the

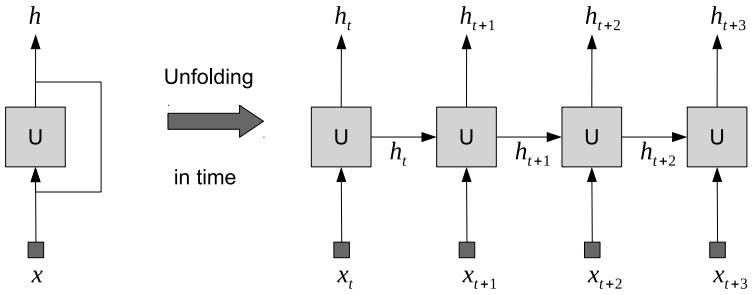

Figure 6.10: Unfolding in time. (Left) A single hidden node with a feedback weight. (Right) This hidden node unfolded in time.

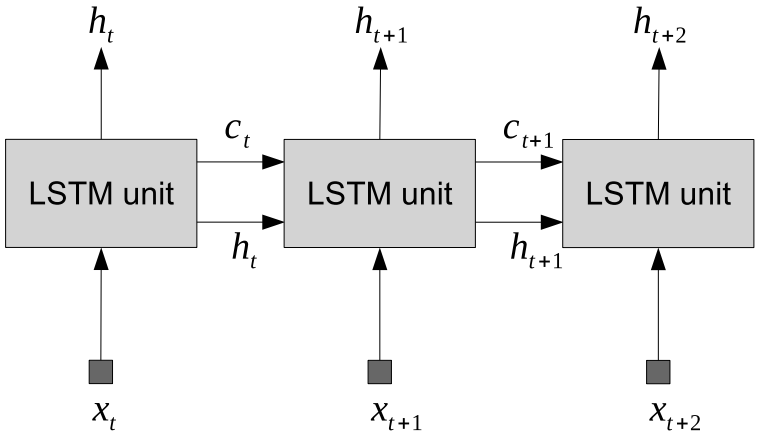

Figure 6.11: A LSTM network is basically the same as simple recurrent network (figure 6.10) where we have replaced the simple "U" box with a new "LSTM" box. We have also added a new value c_t.

computation inside the LSTM box is very different from the "U" box.

In addition to the node value (output from the node) h_t we have an extra value c_t. This is the internal memory of the node. There are also *gates*, denoted f, i, o and called forget, input and output gates, respectively. They are all numbers between 0 and 1, and will be used to "filter" the new values. They all have the same structure,

$$\begin{aligned} i_t &= \sigma(x_t U^i + h_{t-1} W^i) \\ f_t &= \sigma(x_t U^f + h_{t-1} W^f) \\ o_t &= \sigma(x_t U^o + h_{t-1} W^o) \end{aligned}$$

where (U^i, U^f, U^o) and (W^i, W^f, W^o) are new weights and $\sigma(\cdot)$ is the logistic function. Note that where appropriate bias weights should also be added, but I have removed them from

the formulas for simplicity. Given all these gates we will decide new values for the internal memory c_t and for the output h_t. We start by computing a candidate value $\tilde{c}_t$ for the memory,

$$\tilde{c}_t = \tanh(x_t U^c + h_{t-1} W^c)$$

This is exactly the same expression as we used to compute the output value for the simple recurrent networks (see eqn. 6.5). We have just renamed the weights U and W to U^c and W^c. We are now in the position of updating c_t and h_t using the gate values according to,

$$\begin{aligned} c_t &= c_{t-1} f_t + \tilde{c}_t i_t \\ h_t &= \tanh(c_t) o_t \end{aligned}$$

The new memory value c_t is a combination of the previous values "filtered" by the forget gate f, and the candidate value $\tilde{c}$ "filtered by the input gate i. Finally the new output value of the node h_t is the memory value c_t taken through tanh() to push it between -1 and 1, and filtered by the output gate o. It is the gating mechanism that allows the LSTM network to model long-term dependencies. These dependencies can be learned by training the various gating weights. For the equations above we have used a simple example of a single hidden node network, i.e one LSTM unit. In practical examples one typically uses many LSTM nodes. This means that the weights in the above equations should be replaced by appropriate matrices. The gates are then effectively vectors of values between 0 and 1.

As final remark let us try to understand why we can avoid vanishing or exploding gradients for the LSTM network. Previously we looked at $\partial h_t / \partial h_{t-1}$ to understand why gradients could vanish or explode. The relevant derivative for the LSTM node becomes $\partial c_t / \partial c_{t-1}$. We then have,

$$\begin{aligned} \frac{\partial c_t}{\partial c_{t-1}} =& c_{t-1} \sigma'(\cdot) W^f o_{t-1} \tanh'(c_{t-1}) \\ &+ \tilde{c}_t \sigma'(\cdot) W^i o_{t-1} \tanh'(c_{t-1}) \\ &+ i_t \tanh(\cdot) W^c o_{t-1} \tanh'(c_{t-1}) \\ &+ f_t \end{aligned}$$

The details of this expression are not important. Similar to previous expressions for the gradient of different weights (see e.g. 6.6), we will multiply many terms like $\partial c_t / \partial c_{t-1}$ above when computing the such gradients. What we can conclude from the above expression is that such terms can be both < 1 and > 1, hence avoiding gradients to vanish or explode. It is in principle possible for the network to learn when certain gradients should vanish.

6.5.1 Gated recurrent network (GRU) variant

The gated recurrent network (GRU) is a simpler version of the LSTM network. It is a rather new and was first mentioned in 2014. The core idea of the GRU network is the same as for

the LSTM, but it does it in a simpler way. While the LSTM have three gates the GRU node only has two, reset gate r and an update gate z. The reset gate determines how to combine the new input with the previous value and the update gate is used to determine how much of the current value to keep. The equations for them are,

$$\begin{aligned} z &= \sigma(x_t U^z + h_{t-1} W^z) \\ r &= \sigma(x_t U^r + h_{t-1} W^r) \end{aligned}$$

We then have a candidate for the output value $\tilde{h}$ and the new output value h_t according to,

$$\begin{aligned} \tilde{h} &= \tanh(x_t U^h + (h_{t-1} r) W^h) \\ h_t &= (1 - z)\tilde{h} + z h_{t-1} \end{aligned}$$

If we compare LSTM and GRU we can see,

- GRU has two gates and LSTM has three
- GRU does not have an extra internal memory c_t as the LSTM has.
- We do not use an extra nonlinear function to compute the output as the LSTM does.

Note if we set the reset gate r to 1 and the update gate z to 0, we end up with a simple recurrent node. Figure 6.12 shows and illustration of the LSTM and the GRU nodes.

As a final remark, we have not talked about how to train the LSTM/GRU networks. One can use the same idea as for simple recurrent networks, backpropagation through time! The problem of vanishing gradients are avoided using the gating mechanism. So we can train the LSTM/GRU networks using SGD and all the possible enhancements that are available to speed up the minimization process.

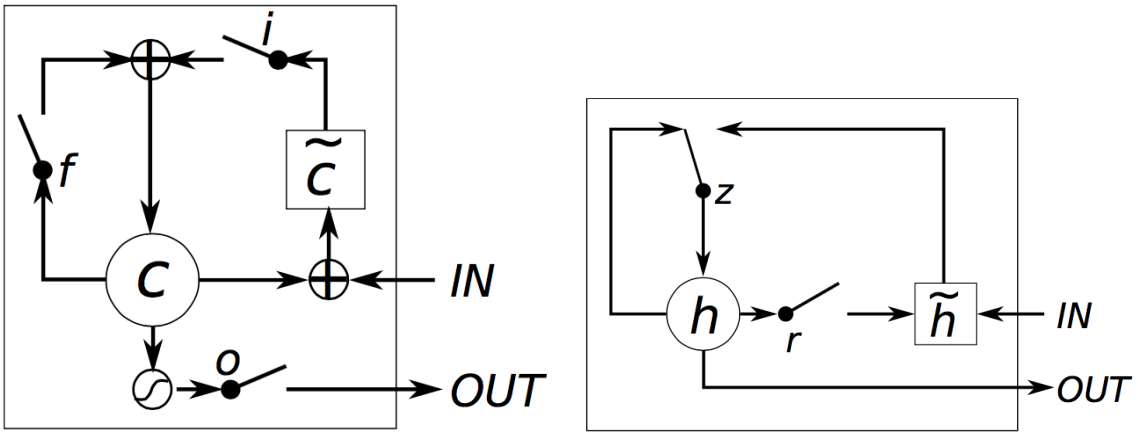

Figure 6.12: Images taken from *GRU Gating. Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." (2014).* (left) The LSTM node where we have three different gates and the extra memory c. (right) The GRU node where we only have two gates and no extra internal memory.

6.6 Recurrent backpropagation

We will now look at a more general formulation of recurrent networks in order to come up with a different approach of how to train a recurrent network. We will consider networks with N units v_i with connections ω_{ij} and activation functions $g(h)$. Some of the units may act as input units and we have input values $x_i(n)$ for pattern n. We set $x_i(n)$ to zero for units that are not input units. Some units may act as output units with associated target values $d_i(n)$. See figure 6.13 We have the following update equation

$$v_i = g(\sum_j \omega_{ij} v_j + x_i) \qquad \text{(I have dropped the pattern index } n\text{)}$$

We are now going to make a critical assumption. Updating the above equations will lead to a stable fix point. This means that we do not need the concept of sequence index. A given input pattern will lead to a fixed output "pattern". We can define an error function (for a single pattern):

$$E = \frac{1}{2}\sum_k e_k^2 \qquad \text{with} \tag{6.7}$$

$$e_k = \left\{ \begin{array}{rl} d_k - v_k & \text{if } k = \text{output unit} \\ 0 & \text{otherwise} \end{array} \right\} \tag{6.8}$$

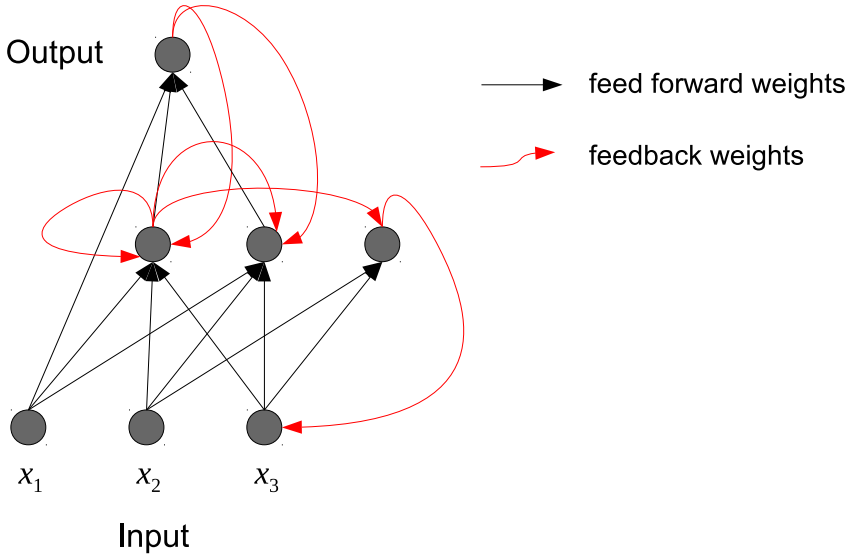

Figure 6.13: Example of a recurrent network. Red arrows denote feedback weights and black arrows are the usual feed-forward weights. Some nodes act as input nodes with external inputs and some are output nodes with known targets.

We will now try to update the weights using gradient descent, as usual,

$$\Delta\omega_{pq} = -\eta \frac{\partial E}{\partial \omega_{pq}} = \eta \sum_k e_k \frac{\partial v_k}{\partial \omega_{pq}} \tag{6.9}$$

The derivative of the node v_i is

$$\begin{aligned} \frac{\partial v_i}{\partial \omega_{pq}} &= g'(h_i)\left(\delta_{ip} v_q + \sum_j \omega_{ij} \frac{\partial v_j}{\partial \omega_{pq}}\right) \Leftrightarrow \\ \frac{\partial v_i}{\partial \omega_{pq}} - g'(h_i) \sum_j \sum_j \omega_{ij} \frac{\partial v_j}{\partial \omega_{pq}} &= g'(h_i)\delta_{ip} v_q \end{aligned} \tag{6.10}$$

where

$$h_i = \sum_j \omega_{ij} v_j + x_i$$

Now define the matrix $\mathbf{L}$ with as

$$L_{ij} = \delta_{ij} - g'(h_i)\omega_{ij}$$

This gives us the following relation from eqn 6.10

$$\sum_j L_{ij} \frac{\partial v_j}{\partial \omega_{pq}} = g'(h_i) v_q \delta_{ip}$$

This is simply a matrix equation where we can solve for $\partial v_j/\partial \omega_{pq}$, giving

$$\frac{\partial v_k}{\partial \omega_{pq}} = \left(L^{-1}\right)_{kp} g'(h_p) v_q$$

Using this expression in eqn 6.9 gives

$$\Delta\omega_{pq} = \eta \sum_k e_k \left(L^{-1}\right)_{kp} g'(h_p) v_q$$

The structure of this update equation is similar to that of ordinary backpropagation, except that we have to invert a matrix. Note that this derivation was based on a single pattern. We need to average the above update expression over several patterns if we want to use the stochastic gradient descent method. We can view this learning method as a way of using recurrent networks for solving problem where we previously used ordinary MLPs. The question is, if the feedback connections make solving the problem easier? The price to pay is a more complicated learning algorithm.

6.7 Realtime recurrent learning (RTRL)

In this section we will briefly look at what is called *realtime recurrent learning*. We will use the same network structure as of figure 6.13. But this time we will not assume that there is a fixpoint of the network for a given input. Instead we are interested in learning a target sequence given a number of input sequences, as we did for simple recurrent networks. We therefore introduce a discrete "time" index t. So the basic update equation for a node in the above network is given by,

$$v_i(t) = g(h_i(t-1)) = g\left(\sum_j \omega_{ij} v_j(t-1) + x_i(t-1)\right)$$

Here $x_i(t)$ is an input to node i at time t. If no such input exists then we put $x_i(t) = 0$. Similarly we introduce $d_i(t)$ to be the target for node i at time t, if it exists.

Again we define node error measure $e_k(t)$,

$$e_k(t) = \left\{ \begin{array}{ll} d_k(t) - v_k(t) & \text{if } d_k(t) \text{ is defined at time } t \\ 0 & \text{otherwise} \end{array} \right\}$$

From this we can define the total error for the full sequence $t = 1, ...T$, as,

$$E_T = \sum_{t=1}^{T} E(t) \qquad \text{with} \qquad E(t) = \frac{1}{2} \sum_k e_k(t)^2$$

A gradient descent approach will need to compute gradients of $E(t)$ with respect to any of the weights ω_{pq}. We can define the weight update at time t according to,

$$\Delta\omega_{pg}(t) = -\eta\frac{\partial E(t)}{\partial \omega_{pq}}$$

and

$$\frac{\partial E(t)}{\partial \omega_{pq}} = \sum_k e_k(t)\frac{\partial v_k(t)}{\partial \omega_{pq}}$$

where we have

$$\frac{\partial v_i(t)}{\partial \omega_{pq}} = g'(h_i(t-1))\left(\delta_{ip}v_q(t-1) + \sum_j \omega_{ij}\frac{\partial v_j(t-1)}{\partial \omega_{pq}}\right)$$

The above equation relates $\partial v_i/\partial\omega_{pq}$ at time t to those at time $t-1$. Using the initial condition $\partial v_i(0)/\partial\omega_{pq} = 0$ we can start to compute the derivatives for all other times $t = 1, ..., T$. The online version of this makes an update at each t,

$$\omega_{pq} \to \omega_{pq} + \Delta\omega_{pg}(t) \text{ for } t = 1, ...T$$

Repeat until the model learns the correct sequence associations. Williams and Zipser, who are among the ones that constructed this algorithm, showed that this works for sufficiently small η. It goes under the name *realtime recurrent learning* (RTRL). Still the major problem with this approach and vanilla BPTT is vanishing gradients. Learning takes long time for large problems and where there is "complicated" sequence dependencies. It is fair to say that truncated BPTT is more common then RTRL.

6.8 The Hopfield model

The Hopfield model is fully recurrent network, see figure 6.14, with the following properties:

- The nodes s_i in the network are binary with $s_i \in \{-1, 1\}$.
- We have symmetrical weights, i.e. $\omega_{ij} = \omega_{ji} \ \forall\, i, j$.
- No self feedback weights, i.e. $\omega_{ii} = 0 \ \forall\, i$.

The value of a given node follow the usual update equation for nodes, with a small change. Let h_i be net input to node i, $h_i = \sum_j \omega_{ij}s_j$,

$$s_i = \begin{cases} s_i & \text{if } h_i = 0 \\ \text{sgn}(h_i) & \text{if } h_i \neq 0 \end{cases} \quad \text{where} \tag{6.11}$$

$$\text{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \end{cases} \tag{6.12}$$

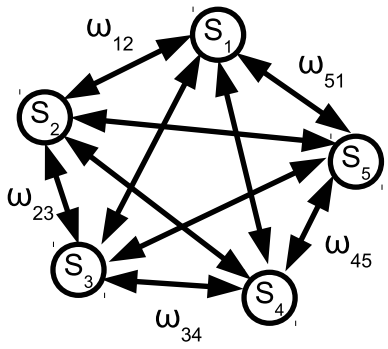

Figure 6.14: A Hopfield model with 5 neurons.

The updating of the nodes can be done in parallel or in serial. The most common mode is to use serial updating of the nodes.

6.8.1 Associative memory

The basic usage of the Hopfield model is to store patterns! What do we mean by that? Let's first introduce the state of the network. Our network has N nodes from which we can define the state vector $\mathbf{s} = (s_1, s_2, ..., s_N)$. A pattern $\boldsymbol{\xi}$ denotes one of the possible 2^N possible state vectors. If we want to separate between different patterns we use the notation $\boldsymbol{\xi}^\mu$ to denote pattern μ. Let also ξ_i^μ denote the i:th binary value of this pattern.

Now a pattern can be stable! By this we mean that if the Hopfield model is initialized with pattern μ ($\mathbf{s} = \boldsymbol{\xi}^\mu$) and we start to update the Hopfield model according to the above update formula (eqn. 6.11), nothing happens. This stability can be written as,

$$\text{sgn}(h_i^\mu) = \xi_i^\mu \qquad (i = 1, ..., N) \tag{6.13}$$

with $h_i^\mu = \sum_j^N \omega_{ij}\xi_j^\mu$. The phrase "associative memory" also means that if some of the "bits" of a stable pattern are changed, then when updating the Hopfield model it should retrieve the correct stable pattern. Next we need a way of training the Hopfield model to actually store patterns, i.e. making a number of patterns stable.

6.8.2 Storing patterns

We can call the process of storing a number of patterns in a Hopfield for training the model. As for other networks, training consists of finding appropriate values of the weights ω_{ij}. We are going to start with one pattern $\boldsymbol{\xi}$. For this case we propose the following,

$$\omega_{ij} = \frac{1}{N-1}\left(\xi_i\xi_j - \delta_{ij}\right) \tag{6.14}$$

where δ_{ij} is the Kronecker delta defined previously. We can easily check that this is working by looking at the stability condition.

$$\begin{aligned} \text{sgn}(h_i) &= \text{sgn}(\sum_j \omega_{ij}\xi_j) = \text{sgn}(\frac{1}{N-1}\sum_j (\xi_i\xi_j - \delta_{ij})\xi_j) = \\ &= \text{sgn}(\frac{1}{N-1}\sum_j \xi_i - \frac{1}{N-1}\xi_i) = \text{sgn}(\frac{N-1}{N-1}\xi_i) = \text{sgn}(\xi_i) = \xi_i \end{aligned}$$

is also easy to realize that if less than half of the bits of the starting pattern $\mathbf{s}$, is not equal to the stored pattern $\boldsymbol{\xi}$, then the update is going to result in $\boldsymbol{\xi}$. Figure 6.15 shows an example

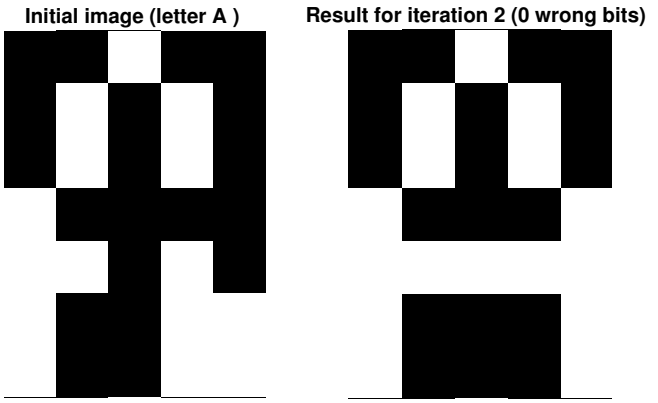

Figure 6.15: (left) A distorted letter "A" was used as the starting state of a Hopfield model that has been trained to store "A". (right) The result after 2 iterations, which is the original stored "A".

where the letter "A" (7x5 pixels) has been stored in a Hopfield model with $7 * 5 = 35$ nodes, giving rise to 35^2 weights. A white pixel corresponds to e.g. the $+1$ state of the nodes and a black pixel to the -1 state. The left figure shows a distorted "A" and in the right figure the result after 2 iterations of the model. As we can see it has recovered the stored image.

Generally we can say that $\boldsymbol{\xi}$ is an attractor, as illustrated in Figure 6.16. We also realize that $-\boldsymbol{\xi}$ is a stable pattern, by checking the stability condition.

To store more than one pattern we simply make a superposition of many patterns. Assume

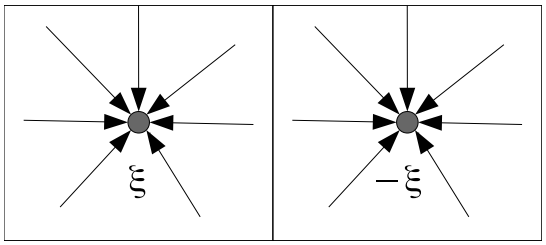

Figure 6.16: An illustration of the retrieving of one stored pattern. Both $\boldsymbol{\xi}$ and $-\boldsymbol{\xi}$ are stable patterns.

we have M patterns ($\boldsymbol{\xi}^\mu, \mu = 1, ..., M$), then we use eqn. 6.14 "many times",

$$\omega_{ij} = \frac{1}{N-1}\left(\sum_{\mu=1}^{M} \xi_i^\mu \xi_j^\mu - M\delta_{ij}\right) \tag{6.15}$$

The hope is now that all these patterns are stable and should act as attractors. It is reasonable to assume that we cannot store an unlimited number of patterns. In fact one can figure out this limit for random and uncorrelated patterns. It turns out to be $P_{\max} \approx 0.14N$, where N is the number of nodes in the Hopfield model.

6.8.3 The energy function

One of the most important contributions in the work of Hopfield was the energy function E of the Hopfield model. It is defined as,

$$E = -\frac{1}{2}\sum_i \sum_j \omega_{ij} s_i s_j \tag{6.16}$$

The energy function is a function of the state of the model $s_1, s_2, ..., s_N$. A central property of this energy function is that,

$$\Delta E \leq 0$$

when the system is updated according to the update rule (see eqn. 6.11). This means that the stored patterns corresponds to local minimum of E. We can imagine the energy landscape as a "hilly" surface where stored patterns represents local minima, see Figure 6.17.

As a final remark, not all local minima are stored patterns, there exists so called spurious patterns that also stable but do not correspond to any of the trained patterns. The energy landscape is however interesting from another perspective, we can imagine that retrieval of stored memories is just an energy minimization process.

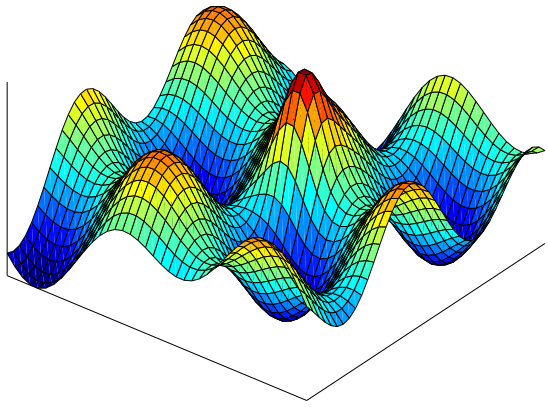

Figure 6.17: The z-axis represents the energy and the x-y plane all corners of the 2^N hyper cube that represents the states of the Hopfield model. A stored pattern often corresponds to a local minimum.