Chapter 3

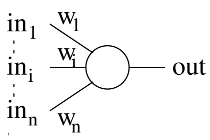
Connectionist Learning

Connectionist approaches have become increasingly popular in the field of natural language processing. These approaches use built-in learning mechanisms. Therefore they should be of interest to research in language acquisition and language learning as well. In this chapter we will examine the promises connectionist approaches, also called neural network approaches, have for natural language learning. We will concentrate on the language learning task which is central in this thesis: the acquisition of phonotactic structure. In the first section we will give an introduction to a class of neural networks: feed-forward networks. In the second section we will introduce the network we have chosen to experiment with: the Simple Recurrent Network (SRN) developed by Jeffrey Elman. The third section covers the acquisition experiments we have performed with this network. This set of experiments will not give us optimal results. We will use the fourth section for laying bare the problem that is the cause of these suboptimal results and presenting a method for restructuring the input data that could enable the networks to obtain better results. In the final section we will present a summary of this chapter with some concluding remarks.[1]

1 Feed-forward networks

In this section we will give an introduction to a specific type of neural network: the feed-forward network. We will start with a general description of this network type. After that we will show how feed-forward networks learn. We will conclude with showing how non-numeric data can be encoded in a feed-forward network.

[1] Some parts of this chapter have earlier been published in (Tjong Kim Sang 1995).



A possible computation performed by the cell: $out = 1$ iff $w_1 * in_1 + ... + w_i * in_i + ... + w_n * in_n \geq threshold$ else $out = 0$ Here in_i is the input value on link i, w_i is the weight corresponding to link i, out is the output value of the cell and $threshold$ is a threshold value.

Figure 3.1: An example of a cell in a neural network and the computation it performs.

1.1 General description of feed-forward networks

Artificial neural networks have been inspired by biological neural networks. They consist of cells which are connected to each other by weighted links. The weights of the links determine the function the network performs. Artificial neural networks usually contain a learning function which adapts the value of the weights to optimize the network's performance on a task.

Connectionist or neural networks have often been compared with rule-based models in cognitive science literature. Researchers have posed that neural networks are more suitable for modeling cognitive tasks such as language processing because they are robust and because they are biologically plausible. Robustness, the ability to deal with noisy data, is an interesting property. It will enable the networks to handle training data with errors. The errors will be infrequent and thus have little influence on the final model. Neural networks share the robustness feature with statistical models. However, robustness is not a feature that is often contributed to rule-based models.

There are many different types of neural networks available. A quick glance in a neural network introductory book like (Wasserman 1989) will reveal that there are about a dozen main types of neural networks and the main types have many variants. Some of these networks are biologically or psychologically plausible and others are just parallel cell models with smart learning algorithms. The cells used in the second network type have a far more simple structure than human brain cells (Zeidenberg 1990). We are interested in solving our problem, the acquisition of phonotactic structure, as well as possible. We feel that using biologically or psychologically inspired artificial neural networks is an interesting approach to solve the problem However, we refrain from committing ourselves to using only biologically or psychologically inspired artificial neural networks.

We have described a neural network as a collection of cells which are connected to each other with weighted links. The cells use these links to send signals to each other. The links are one-way links: signals can only travel through a link in one direction. Furthermore, signals are numbers between zero and one. The cells perform a simple function: they multiply incoming signals with the weight of the link the signals are on and compute the sum of all multiplications. When this result is larger than or equal to a threshold value, the cell will put a signal 1.0 at its output link otherwise it will put a signal 0.0 on its output link. The function that the cell performs to compute the output signal from the input signals is called the ACTIVATION FUNCTION.

in_1	in_2	$XOR(in_1, in_2)$
0	0	0
0	1	1
1	0	1
1	1	0

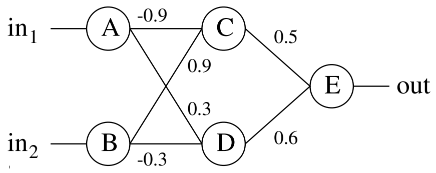
Figure 3.2: The XOR-function. It will produce a 1 if and only if exactly one of the two binary input parameters is equal to 1. Otherwise the function will produce 0.

When we connect these cells with each other and give cells input and output connections with the outside world, we have created a neural network. After presenting the network a set of input signals, it will produce a set of output signals. Sets of signals are also called patterns So it is capable of converting an input pattern to an output pattern. As an example we will design a network that computes the XOR-function (see figure 1.1).

We can try to build a network that performs this function by starting from a random collection of cells with a random number of links between them. However, we will to divide the network in groups of cells so that we can use the learning algorithm which will be described in the next section. We will call these cell groups layers. We will number the layers and impose a hierarchy on them. Cells in the first layer receive input from outside and send their output to cells in the second layer. Cells in the second layer receive input from the first layer and send their output to cells in the third layer, and so on. Cells in the final layer send output to the outside world. So the data will first enter the first layer, then it will move to the second layer then to the third and so on until it reaches the final layer via which it is sent to the outside world. Because the data flows from the back to the front of the network this network is called a feed-forward network.

Figure 3.3 shows a three-layer network which is able to compute the XOR-function, a binary function with two input parameters and one output parameter which is only equal to one if exactly one of the input parameters is equal to one. The cells in the network all have a threshold value of 0.1. The two input cells do not change their input: they simply pass it on to the second layer. The table presents an insight into the information flow in the network. For example, the [0,1] pattern (in_1=0 and $in_2 = 1$) will be passed to the second layer and result in a [1,0] output signal of the second layer. This will result in a [1] signal as output of the network.

This network prefectly computes the XOR-function because it contains the correct weights to do this. But now the question is: how do we get a network to compute an arbitrary function? In other words: how do we find weights that enable a network to compute a specific function? We will deal with this question in the next section.



in_1	in_2	out_A	out_B	out_C	out_D	out
0	0	0	0	0	0	0
0	1	0	1	1	0	1
1	0	1	0	0	1	1
1	1	1	1	0	0	0

Figure 3.3: A network that computes the XOR-function. Signals travel through the network from left to right. These signals have been represented by binary numbers in the table. The numbers next to the links in the network are the weights of the links. All cells have a threshold value of 0.1.

1.2 Learning in a feed-forward network

We want a feed-forward network to perform a task which is computing a specific function. This can be done by presenting the network the input and the corresponding output of the function and making it learn a weight configuration that can be used for simulating the function. This kind of learning is called SUPERVISED LEARNING: the learner is provided with both input and output for a particular task. In unsupervised learning the learner will only receive the task input (Zeidenberg 1990).

One of the learning algorithms which can be used for deriving a network weight configuration is the backpropagation algorithm described in (Rumelhart et al. 1986). In this algorithm the output of the network for the training patterns is compared with the required output. The difference between these two, the error, is used for improving the current weights. First the error of the network output is used for computing the weights of the links to the last layer, then for the weights to the next to last layer and so on. The error is propagated from the output layer of the network back to the input layer, hence the name of the algorithm 'backpropagation'. This learning algorithm is not biologically plausible. However it is still the most frequently used connectionist learning algorithm because it will produce a reasonable model for most data sets (Rumelhart et al. 1986).

Let us look at the backpropagation algorithm in more detail (Rumelhart et al. 1986). First we define 6 variables:

$target_{c,l}$	the target output value of cell c of layer l
$out_{c,l}$	the actual output value of cell c of layer l
$\delta_{c,l}$	the error in the output of cell c of layer l ($target_{c,l}$-$out_{c,l}$)
$w_{c_1c_2,l}$	the weight of the link between cell c_1 in layer *l-1* and cell c_2 in layer l
$\Delta w_{c_1c_2,l}$	the size of the change of $w_{c_1c_2,l}$ as computed by the algorithm
η	the learning rate, a parameter of the algorithm

Here layer 1 will be the input layer, layer 2 will be the first hidden layer, layer 3 will be the second hidden layer if present and so on. When there are n hidden layers, layer *n+2* will be the output layer. We will use one hidden layer and therefore in our networks layer 3 will be the output layer. The first step of the backpropagation algorithm consists of computing the $\delta_{c,l}$ values for the cells in the output layers:

$$\delta_{c,l} = (target_{c,l} - out_{c,l}) * (1 - out_{c,l}) * out_{c,l} \tag{3.1}$$

The simplest formula for computing a $\delta_{c,l}$ would have been $\delta_{c,l} = (target_{c,l} - out_{c,l})$ in which case the error would have been equal to the difference between the target output value and the actual output value. However, for the correctness proof of the backpropagation algorithm it is convenient to use the complex error equation that has been displayed. In this complex equation the error is the product of the simple error function and the derivative of the activation function with respect to the input.[2]

When we know the error values, we can use them for computing the weight change values for the output layer $\Delta w_{c_1c_2,l}$ and these can in turn be used for computing the new values of the weights of the links between the hidden layer cells and the output layer cells:

$$\Delta w_{c_1c_2,l} = \eta * \delta_{c_2,l} * out_{c_1,(l-1)} \tag{3.2}$$

$$new\ w_{c_1c_2,l} = old\ w_{c_1c_2,l} + \Delta w_{c_1c_2,l} \tag{3.3}$$

The change of a weight of a link $\Delta w_{c_1c_2,l}$ is proportional to the network parameter, learning rate η, the error value of the cell at the end of the link $\delta_{c_2,l}$ and the value of the signal on the link $out_{c_1,(l-1)}$. The equation contains an input signal value because multiple input signals determine the output value of a cell and the weights corresponding with the signals that contribute most to this output value should receive the largest increase or decrease. The new value of the weight will be the old value increased with the change.

The learning rate η is a model parameter that backpropagation has in common with hill-climbing algorithms (Rich et al. 1991). These algorithms can in general be used for solving the problem of a walker attempting to find the highest spot in a misty mountain area. In the case of backpropagation, a set of correct weights would be a representation of the location of the highest point of the mountain. Then each $\Delta w_{c_1c_2,l}$

[2]We are using the activation function defined in (Rumelhart et al. 1986), equation (15): $out_{c,l} = (1 + e^{in_{c,l}+\theta_{c,l}})^-1$ where $in_{c,l} = \sum_{c_1} w_{c_1c,l-1} * out_{c_1,l-1}$ and $\theta_{c,l}$ is the threshold value for cell c of layer l. The derivative of this function with respect to the input $in_{c,l}$ is $(1 - out_{c,l}) * out_{c,l})$.

corresponds to one step of the walker. A large value for η increases the size of the steps that the walker makes and enables him to reach the target faster. However, larger η values also introduce the danger that the walker might step over the target location and never be able to reach it (we assume that the walker is able to make steps of many kilometers). In terms of finding the correct weight configuration: a large η value enables backpropagation to move faster to the target weight configuration but it also increases the chance of never reaching this target.

By using equations 3.1, 3.2 and 3.3 we will be able to compute new values for the weights of the links between the hidden layer and the output layer. The latter two equations will also be used for the computation of the new values for the weights of the other links. Equation 3.1, however, cannot be used for that purpose because it contains the term $target_{c,l}$. While we know the correct output value for the output layer, it is impossible to tell what the correct output value for the cells in the hidden layer needs to be. Therefore we will use a different equation for computing $\delta_{c,l}$ for non-output cells:

$$\delta_{c_1,l} = (\sum_{c_2} \delta_{c_2,(l+1)} * w_{c_1c_2,(l+1)}) * (1 - out_{c_1,l}) * out_{c_1,l} \tag{3.4}$$

In this equation we have approximated $(target_{c_1,l} - out_{c_1,l})$ with the sum of the weights of the output links of cell c_1 $(w_{c_1c_2,(l+1)})$ multiplied with the error computed for the cell that the link provides input for $\delta_{c_2,(l+1)}$. The δ values computed for layer $l + 1$ will be used for computing the δ values for layer l. This operation can be performed for any number of network layers.

With the equations 3.1, 3.2, 3.3 and 3.4 we are able to update all the weights of the network. (Rumelhart et al. 1986) have introduced an improved version of equation 3.2, one which will increase the learning speed with minimal chance of the algorithm becoming instable[3]:

$$\Delta w_{c_1c_2,l}(t) = \eta * \delta_{c_2,l} * out_{c_1,(l-1)}(t) + \alpha * (\Delta w_{c_1c_2,l}(t-1)) \tag{3.5}$$

The weight change as defined in equation 3.2 is now increased with the previous value of the weight change $\Delta w_{c_1c_2,l}(t-1)$ multiplied with the momentum parameter α which is a value between 0 and 1. The idea behind this is that the process of reaching the correct weight values usually consists of a large number of small steps in the same direction. By adding to each step a portion of the value of the previous step, successive steps in the same direction will increase the size of the steps and thus enable the algorithm to reach the target weight configuration faster.

The backpropagation algorithm will have to perform a number of weight modifications before it reaches a good weight configuration. The number of modifications that are necessary will be determined by the values of the algorithm parameters η and α. We would like the algorithm to arrive at a correct network configuration as soon

[3] A network is instable when it is unable to converge to a more or less constant model for the training data in a finite amount of training rounds.

as possible and therefore we are interested in finding optimal values for these two parameters. Unfortunately, the optimal values of η and α are dependent on the task to be learned and the network initialization. Usually finding good values for these two algorithm parameters is a matter of trial and error.

In our experiments with feed-forward networks we will use an error function for measuring the performance of the network for a certain task. The error function takes all cell output values of all output patterns, subtracts from them the desired output values and adds the squares of these subtractions together:

$$E = \sum_{pat} \sum_{cell} (desired\ output_{pattern,cell} - actual\ output_{pattern,cell})^2 \qquad (3.6)$$

The result, the TOTAL SUM SQUARED ERROR, is an indication of the performance of the network. A small error value indicates that the network is performing the task well.

Now training the network is a four step process:

1. Present an input pattern to the network and make the network compute an output pattern.
2. Compare the actual output pattern of the network with the target output values and use the equations 3.1, 3.4, 3.5 and 3.3 for computing new values of the weights.
3. Repeat steps 1 and 2 for all patterns.
4. Repeat steps 1, 2 and 3 until the total sum squared error has dropped below a certain threshold or the number of times that these steps have been performed has reached some predefined maximum.

In an alternative schedule step 2 will only use the equations 3.1, 3.4 and 3.5 for computing the weight updates. These will be stored in some buffer and the actual update of the weights will be made in step 3 when all patterns have been processed.

1.3 Representing non-numeric data in a neural network

In the example in which we computed the XOR-function with a neural network, representing data was not difficult because the data was numeric and the network processed numeric data. But in language we want to process sounds and symbols and therefore we have to find out a way to represent non-numeric data in a neural network. When we consider using four characters *a*, *b*, *c* and *d* in a network, there are three basic ways for representing them in a network.

First we can represent the characters by using one signal and assigning numeric values to the characters, for example: $a = 0.0$, $b = 0.3$, $c = 0.6$ and $d = 0.9$. However, this approach will fail because imperfect data cannot be interpreted unambiguously.

If the network does not know which of two outputs is correct, it typically produces an intermediate value. A draw between *a* and *c* can result in an output value of 0.3, which actually is the representation of *b*. It is impossible to tell if a 0.3 output is the result of the network choosing for *b* or that the network cannot choose between *a* and *c*. The network output can not be analyzed because the representation of characters is inappropriate.

A second way to represent the characters is to use two binary signals: *a* = [0,0], *b* = [0,1], *c* = [1,0], *d* = [1,1]. However, this representation suffers from the same problem as the previous one: errors can occur when the network output needs to be interpreted. For example: if the network cannot choose between *b* and *c* it might produce a [0.5,0.5] output. However, this output can also be interpreted as the intermediate pattern for *a* and *d*. Furthermore, binary representations can invoke unintended similarities between the symbols. It is possible to use five bits for representing twenty six characters and use [0,0,0,0,0] (0) for *a* up to [1,1,0,0,1] (25) for *z*. It that case the representations for *e*, [0,0,1,0,0], and *m*, [0,1,1,0,0] are similar. The network will use this artificial similarity during processing. We might not want it to do that.

A third way to represent the characters in by using the localist mapping described in (Finch 1993) among others. Then every character will be linked to a particular cell in the representation and only one cell in the representation can contain a one: *a* = [1,0,0,0], *b* = [0,1,0,0], *c* = [0,0,1,0] and *d* = [0,0,0,1]. In this case we need four signals. This representation does not have the problem of the two previous ones. If the network produces an intermediate value the candidates that it suggests can unambiguously be pointed at, for example: if the network would generate a [0.5,0.5,0,0] output then we know that it had difficulty to choose between *a* and *b*. Furthermore all representation patterns are equally alike so the network cannot recognize non-intended similarities from the representations. However, the price we pay for this is that this way of representing characters requires more cells than the previous two. More cells means more links and more weights and, because computations will be performed for all weights during training, more cells means longer training times.

When a lot of symbols have to be processed, using the localist mapping can prove to be time-consuming during the training phase. In that case a balance has to be found between the two problems of large representations and non-intended similarities between patterns. A balance can be found by choosing some intermediate representation size and making the network itself find out meaningful representations of that size. (Blank et al. 1992), (Elman 1990) and (Miikkulainen et al. 1988) show three different methods for making networks build fixed length representations for symbols. The representations built by the network reflect the task the network learned to perform with the symbols. So a different network task will result in different representations.

2 The Simple Recurrent Network (SRN)

In this section we will present the network we will use in our phonotactic experiments: the Simple Recurrent Network (SRN). We will start with a general description and after that we will show how SRNs learn. The section will be concluded with a summary of language experiments performed with SRNs by others.

2.1 General description of SRNs

The three-layer feed-forward network we have presented in the previous section cannot perform all the tasks we would like to apply it to. An example of a task in which the network cannot achieve a score that is higher than 50% is predicting the next bit in a bit list like:

1 0 1 0 0 0 0 1 1 1 1 0 1 0 1 ...

Only one bit of the list is available at one time point. This bit list first appeared in (Elman 1990). At first sight the elements of the list seems to be chosen randomly and correctly predicting bits of the list seems impossible. However when we take a closer look at it, it turns out to be possible to discover three-bit patterns in the list. The first two bits of each pattern are random but the third is the XOR of the first two. In order to be able to predict this third bit correctly, the network must know the values of the previous two. However, as we said only one bit of the list is available at each moment and the network does not have memory to store the previous bit. Therefore the feed-forward network cannot predict the third bit of each three-bit pattern correctly. The network cannot do any better than guessing future bits which will result in a score of 50%.

The only way to improve the performance of the network is to give it memory. We can use a buffer to store input and only process the input when it is complete, an approach which has been chosen in (Weijters et al. 1990). To solve this problem we will need to a buffer size of one pattern. However, for general problems in which memory is required we might not know how large the buffer size needs to be. If we choose a buffer size that is too small, the network will not be able to solve the problem. Therefore we have not chosen this approach.

There are two other standard ways of adding memory to a three-layer feed-forward network. Both consist of adding a feedback loop to the network which enables it to store and use context information for the input sequence.

The first way of giving the network memory is by feeding the output of the network back to the input. This approach was developed by (Jordan 1986) and the resulting network has become known as the Jordan network. The input layer of the network is extended with the same number of cells as the output layer. These extra cells, the CONTEXT CELLS, contain a copy of the previous output layer activations. Like the input layer cells, they are connected with all cells in the hidden layer. Every output cell is connected with a context cell. Furthermore, each context cell is connected with itself and with every other context cell (see figure 3.4). The weights of the backward and the inter-context cell links are equal to one and these weights cannot be changed. Signals only flow through these links when the input of the network is updated.

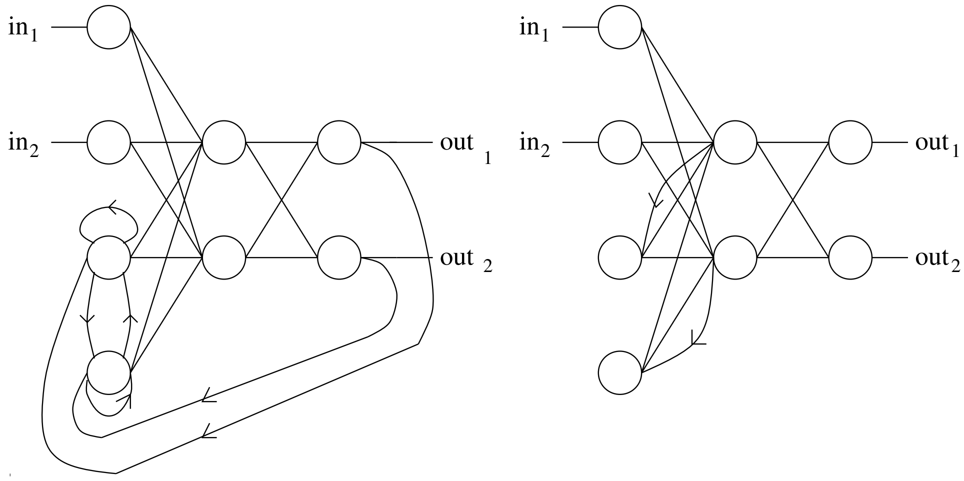


Figure 3.4: A Jordan network (left) and a Simple Recurrent Network (right). Except when arrows indicate otherwise, all connections are forward connections. The two context cells at the bottom left of each network make the difference between these two networks and the networks we have discussed in the previous section. These context cells implement the memory of the networks.

The second way of adding memory to the network is by feeding the output of the hidden layer back to the input layer. (Elman 1990) first used this approach and he named the network Simple Recurrent Network (SRN). Now the input layer is extended with as many context cells as the hidden layer. The other aspects of the network are the same as in the Jordan network apart from the internal context layer links which are present in the Jordan network but do not exist in the SRN (see figure 3.4).

Both networks are capable of performing the XOR-sequence task correctly. An example of the output of an SRN after training can be found in figure 3.5. The average error of the network for the third, the sixth and the ninth bits in the test list is 0.40 or lower but for other bits it is about 0.50. While this network does not perform perfectly, it seems to have recognized the structure of the bit lists.

For other tasks performance of these two network types can be different: sometimes the Jordan network performs better and sometimes the SRN. The Jordan network trains faster in a supervised task because the correct input values of the context cells are known during training (context cell input = previous network output in Jordan networks). The correct input values of the context cells in an SRN are unknown during training (context cell input = hidden layer output in SRNs). However, the hidden layer activation values may contain an interesting representation of features of the symbols and the task in which they are used (Elman 1990). Feeding back these activations can aid the network in performing well. This last feature is the prime reason for us for

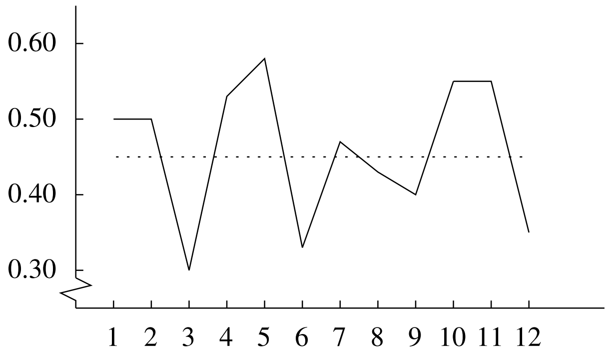


Figure 3.5: Average error in the SRN output for the XOR task for 12 consecutive inputs averaged over 10 trials. The network had to predict the next bit of a bit list of which each third bit was the XOR of the previous two. The overall average error is 0.45 (dotted line) but the error for the bits 3, 6, 9 and 12 was smaller (0.35 on average) which indicates that the network has recognized the structure of the bit list.

choosing for the SRN for our experiments. With this choice we join previous connectionist language research like (Elman 1990) and (Cleeremans 1993) which both use SRNs to model sequential language processing. Jordan networks are not used as often; an example of a Jordan networks application is the work of (Todd 1989) in music generation.

2.2 Learning in SRNs

An SRN can be trained by using the same equations and training schedule as was used for the feed-forward networks in section 1.2. The only differences are that some weights will not be changed during training (the weights of the backward links) and a part of the input data will be obtained from the hidden layer output for the previous pattern (the context cell input values).

Again training the network is a four step process:

1. Present an input pattern to the network and make the network compute an output pattern. The context cell input values are equal to the output values of the hidden layer for the previous pattern. If no previous pattern exists then the context cell input values will be equal to 0.

2. Compare the actual output pattern of the network with the target output values and use the equations 3.1, 3.4, 3.5 and 3.3 of the backpropagation algorithm for computing new values of the weights. The backward links should not be changed.

3. Repeat steps 1 and 2 for all patterns.
4. Repeat steps 1, 2 and 3 until the total sum squared error has dropped below a certain threshold or the number of times that these steps have been performed has reached some predefined maximum.

2.3 Using SRNs for language experiments

SRNs have been used successfully in different experiments. (Elman 1990) has used an SRN for analyzing simple sentences. From a small grammar he generated 10,000 two- and three-word sentences. The sentences were concatenated in a long sequence and an SRN was trained to predict the next word in the sequence. The network only had available the current word and a previous hidden layer activation which can be seen as a representation for the previous words in the sequence. The SRN cannot perform this task perfectly because the order of the words in the sequence is non-deterministic and the network is too small to memorize the complete sequence.

However, Elman was not interested in the output of the network. He was interested in the activation patterns which were formed in the hidden layer. Elman discovered that the average activation at the hidden layer for a word presented at the input reflected the way the word was used in the sentences. A cluster analysis of these average hidden representations divided the words in two groups: the verbs and the nouns. Within these groups some subgroups could be found: animals, humans, food, breakable objects and different verb types (Elman 1990). So the network developed internal representations for words which reflect their syntactic and semantic properties.

In another experiment Axel Cleeremans, David Servan-Schreiber and James McClelland (Cleeremans 1993) (Cleeremans et al. 1989) (Servan-Schreiber et al. 1991). have trained a network to recognize strings which were generated using a small grammar that was originally used in (Reber 1976) (see figure 3.6). Cleeremans et al. trained an SRN to predict the next character in a sequence of 60,000 strings which were randomly generated by the grammar. Again this task was non-deterministic and the size of the network was too small to memorize the complete sequence. So the network could not perform this task without making errors. For Cleeremans et al. it was sufficient that the network indicated in its output what characters are a valid successor of the sequence. They represented characters using the localist mapping and their aim was to make the network output at least 0.3 in the cells that correspond to valid successors. So when the output of the network is something like shown in figure 3.7, then valid successors are *S* and *X* because these have received an output value higher than 0.3. The network accepted a string if it considerated all characters of the string as valid characters in their context.

Cleeremans et al. have tested their network with 20,000 strings generated by the Reber grammar. For all characters in these strings the network output in the corresponding cells was 0.3 or higher. So the network accepted 100% of the grammatical strings. After this Cleeremans et al. fed their network 130,000 random strings built from the same characters. This time the network accepted 0.2% of the strings. It

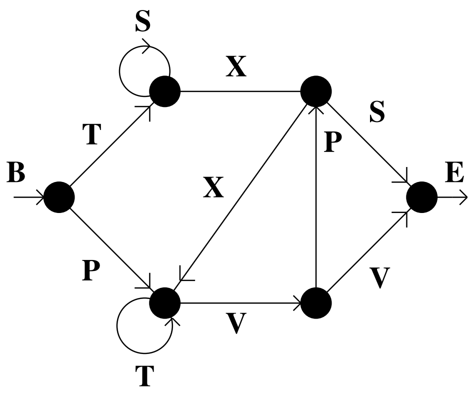


Figure 3.6: Finite state network representing the Reber grammar. Valid strings produced by this grammar will always start with *B* and end with *E*. Between these characters there will a string containing *T*'s, *S*'s, *X*'s, *V*'s and *P*'s. The finite state network indicates which characters are possible successors of a substring. For example: the substring *BT* can be followed by either *S* or *X*.

B	T	S	X	V	P	E
0.0	0.0	0.4	0.5	0.0	0.1	0.0

Figure 3.7: Possible network output of an SRN trained on the Reber grammar after presenting substring *BT* to the network. A number below a character indicates the output value of the network output cell corresponding with that character. The SRN attempts to predict the successor of the string. It considers *S* and *X* as valid successors of *BT* because the output values corresponding to these two characters are larger than 0.3.

turned out that all accepted strings were grammatical strings. All other strings were invalid according to the grammar. Thus the network separated perfectly the grammatical strings from the non-grammatical strings.

The results obtained by Cleeremans et al. are excellent. Their problem was related to ours so we hope to obtain results that are similar to theirs when we apply SRNs to our phonotactic data.

3 Experiments with SRNs

In this section we will describe the experiments we have performed with SRNs. We will start with a presentation of the general set-up of the experiments. After that we will discuss a small experiment which was used for determining optimal network parameters. The next two sections will describe the experiments that we have performed with orthographic data: one with random initialization and one with additional linguistic information. The linguistic information was supplied to the network by supplying the training data in an order based on the complexity of the training strings.

3.1 General experiment set-up

We will use the experiment of (Cleeremans 1993) with the Reber grammar as an example for the set-up of our own experiments involving the acquisition of phonotactic structure. We will use Simple Recurrent Networks (SRNs) which will process a list of words. The words will be presented to the networks character by character. The SRNs have no access to previous or later characters. They will have to use their internal memory for storing a representation of the previous characters. The SRNs will be trained to predict the next character of a word.

Like Cleeremans et al. we will use the localist mapping to represent characters. So in the SRN input and output layers every cell will correspond to one character. We will be using the localist mapping and this means that input and output patterns will contain 29 cells for the complete orthographic data. So the number of input and output cells for networks that process orthographic data will be 29. We will work with different hidden layer sizes: 3, 4, 10 and 21 cells. The number of cells in the context layer will be equal to the number of cells in the hidden layer.

We will adopt Cleeremans et al.'s measurement definition, the way the scores of characters and words are computed. The WORD SCORE is the value that is assigned by a network to the word. If the word score is high we will assume that the word is probable according to the training data. The CHARACTER SCORE is the value assigned by a network to the character in a certain context. If the character score in a certain context is high we will assume that the character is probable in that context. In the work of Cleeremans et al. these two scores are defined as follows:

> **Measure 1**
>
> The character score of character c in the context of some preceding substring s is the output value of the output cell corresponding to c of a network after the network has processed s.
>
> The word score of word w consisting of the characters $c_1...c_n$ is equal to the lowest value assigned by a network to any of these characters at their specific positions during the processing of w.

Note that these scores can only be interpreted in a reasonable way if they are computed by the same network. It makes no sense to compare scores of words that are computed by different networks. Because of the random initialization of SRNs before learning, two SRNs trained on the same problem with the same data may well assign different scores to the same word. We are only interested in the relation between scores of words computed by the same network.

We will compare the performance of measure 1 with two other measures. In our second measure the character scores will be computed in the same way as in measure 1 but the word scores will be equal to the product of the character scores:

> **Measure 2**
>
> The character score is computed in the same way as in measure 1.
>
> The word score of word *w* consisting of the characters $c_1...c_n$ is equal to the product of the scores assigned by the network to these characters at their specific positions during the processing of *w*.

In our third measure we will also multiply the character scores to get the word score but this time we will use the Euclidean distance between actual network output and target output as character score. This means that we compute character scores by comparing all network output values with the target values instead of only looking at one cell:

> **Measure 3**
>
> The character score of character *c* in the context of some preceding substring *s* is 1 minus the normalized Euclidean distance between the target output pattern of a network after processing *s* and the actual output pattern of the network after processing *s*:
>
> $score_c = 1 - \sqrt{\frac{1}{n}\sum_{i=1}^{n}(target_i(s) - output_i(s))^2}$
>
> in which $target_i(s)$ is the target output pattern of cell *i* and $output_i(s)$ is the actual output pattern.
>
> The word score is computed in the same way as in measure 2.

Since the patterns consist of values between 0 and 1, the largest difference sum between these two vectors is n so we divide the sum of products by n in order to obtain a value between 0 and 1.[4] This value is equal to 0 for a perfect match and equal to 1 for a non-match so we subtract it from 1 to obtain similar values as in the previous measures.

We have explained in the description of our work with Hidden Markov Models that it is necessary to append an end-of-word character to the words in our training corpus. We will also add an end-of-word character to the words we are processing in our SRN experiments. The end-of-word character is necessary to enable the network to recognize the start of a new word. We will present the words to the network character by character in one long list. If the network is unable to detect the start of a new

[4] Actually this maximum difference is smaller than n because the target vector has length one and we have observed that the output vectors of the network after training have about unit length. The Euclidean distance between two positive vectors of unit length has a maximal value of $\sqrt{2}$.

word then the scores of the characters of a word will be influenced by the characters of previous words.

We want the scores of the characters of a word to be influenced only by the characters of the same word. Therefore we have put the end-of-word characters as separators between words and made the networks reset their internal memory, the context cells, after processing such an end-of-word character. We have used the name end-of-word character rather than start-of-word character or word-separator to keep a consistent usage of terminology with the Hidden Markov Model chapter. Adding these characters to the data means adapting that task to the learning algorithm. We are not favoring that approach but in this particular case adding end-of-word characters is the best way to enable the networks to interpret the data in a reasonable way.

Our goal will be to obtain a Simple Recurrent Network which predicts the next character of a word given the previous characters. Because this task is nondeterministic and the network is too small to memorize all words, it cannot perform this task perfectly. We will be satisfied if, by using our measures, we can determine that the network has discovered the difference between words which are valid and words which are invalid in a certain language. The network will assign scores to all words. We will accept words which have a score above a certain threshold and reject all other words. The threshold value will be equal to the smallest word score occurring in the training set.

The parameters of the network will be initialized with the values mentioned in (Cleeremans 1993) (page 209): learning rate η between 0.01 and 0.02 and momentum α between 0.5 and 0.9. The number of cells used in the hidden layer influences the performance of the network. However, it is hard to find out the best size of the hidden layer. Cleeremans et al. have chosen a hidden layer size of three because the grammar they have trained their network with can be represented with a finite state network with a minimal number of six states. These six states correspond with the vectors which can appear at the hidden layer. If we assume that these vectors are binary, we need 3 cells to be able to represent 6 patterns ($2^3 > 6 > 2^2$). In our HMM experiments we have used HMMs with eight internal states. Therefore we will also start with a hidden layer of three cells.

3.2 Finding network parameters with restricted data

In our first experiment we will try to find out if our network set-up is correct. We are not interested in a network covering all data yet, so we start with training the SRN with part of the data: 184 words in orthographic representation starting with the character *t*. We trained the network 10,000 rounds with η=0.02 and $\alpha = 0.5$ and examined its performance every 1000 rounds. The network was tested with two word lists: one list consisting of 11 invalid t-words and one list of 16 valid Dutch t-words that were not present in the training corpus. All words receive a score. A word is rejected if it receives a score smaller than the smallest score that occurred in the training set. The target of the network is to reject all 11 negative words and accept all

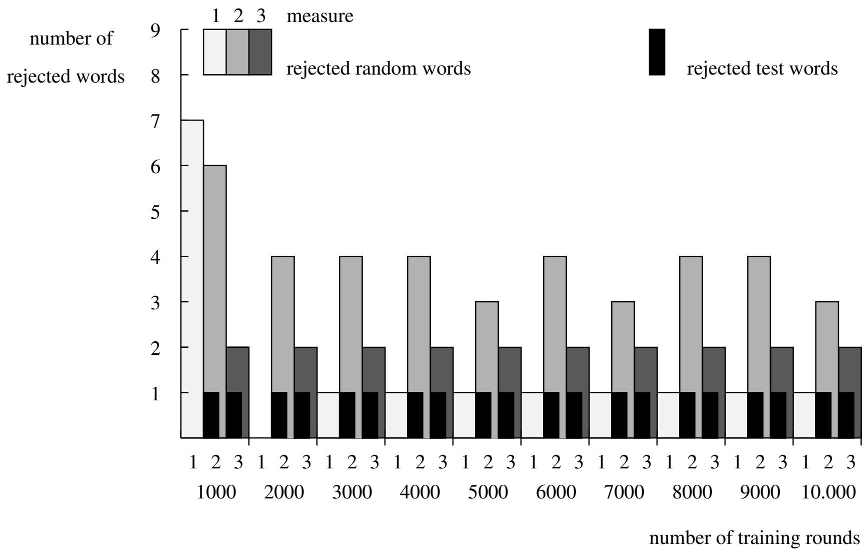


Figure 3.8: The performance of an SRN with three hidden cells for orthographically represented words starting with *t* using measures 1, 2 and 3. The target of the network was to reject 11 invalid t-words and accept all positive test words. The network reached the best performance after 1000 rounds of training. From 2000 rounds onwards the performance stabilized at a lower level.

16 test words. The behavior of the network is shown in figure 3.8.

The best results were achieved by measure 1 after 1000 rounds. At that point this measure rejected seven of the 11 negative words and accepted all positive words. The other two measures also performed best after 1000 rounds. At that point measure 2 rejected six negative words and one positive word and measure 3, which reached the same performance for all test points, rejected two negative words and one positive word. The performances of measure 1 and 2 were worse for all other test points: they accepted equally many positive words while rejecting fewer negative words.

Two observations can be made from the results shown in figure 2. First of all, the fact that the performance of measure 1 and 2 decreases after longer training is an example of OVERTRAINING. This is common behavior for SRNs: when they are trained too long they adapt themselves better to the training data and their generalization capabilities decrease. This will result in a poorer performance on patterns they have never seen.

We will attempt to avoid the overtraining problem by using a smaller learning rate η in our future experiments: 0.01 instead of 0.02. By decreasing the learning rate we will decrease the speed with which the network will approach the optimal set of weights. Another modification we will make in our future SRN experiments is that

word length	number of words	average scores for measure 2	factors for measure 2	average scores for measure 3	factors for measure 3
2	1	$4.432*10^{-4}$	-	$6.460*10^{-1}$	-
3	26	$7.401*10^{-4}$	$5.988*10^{-1}$	$5.532*10^{-1}$	1.168
4	81	$4.054*10^{-4}$	1.826	$4.903*10^{-1}$	1.128
5	61	$1.941*10^{-4}$	2.089	$4.219*10^{-1}$	1.162
6	13	$7.663*10^{-5}$	2.532	$3.537*10^{-1}$	1.193
7	2	$2.737*10^{-7}$	$2.800*10^{2}$	$2.694*10^{-1}$	1.313

Figure 3.9: The average scores for the orthographically represented training t-words of similar length (excluding the end-of-word character). The factors indicate average score for length divided by the average score for length n-1. Apart for the unique two character word the average score for words with length n+1 is always smaller than the average for words of length n.

we will test the performance of the network more frequently: every 50 rounds instead of every 1000 rounds. According to the results we have achieved here the network performed best after 1000 rounds but it might have been performing better after 550 rounds. By testing the performance of the network more often, we hope to determine the best performance point more precisely.

The question, of course, is when the training process of the SRN should be stopped. We will follow the solution the general literature has proposed for this and stop training when the total sum squared error (see section 1.2 of this chapter) stabilizes. We will stop training when the total sum squared error at a test point remains within a 1% distance of the error at the previous test point.

The second observation we can make from figure 3.8 is that measures 2 and 3 perform worse than measure 1. A possible explanation for this fact is that measure 2 and 3 compute word scores by multiplying character scores with each other. The character scores are values between 0 and 1. Almost all multiplications will make the word score smaller and this causes long words to have a smaller average score than short words.

The average word scores are shown in figure 3.9: in almost all cases words with length n receive a smaller average score than words with length $n + 1$. This is also the fact for the negative and the positive test set. The result of this is that a short negative word might receive a higher score than a long positive word which can result in the negative word being accepted and the positive word being rejected. We want to avoid this.

A possible solution to this problem is multiplying the scores with a value that depends on the length of the score. The longer the words are the larger the multiplication factor should be. The aim of this extra computation is making the word scores less dependent on the length of the words. We call this computation SCORE CORRECTION FOR LENGTH.

The question now is what the size of the multiplication factor should be. We chose as factor the quotient of the word average of length 4 and the word average of length 5 to the power of word length:

$$new\ word\ score = old\ word\ score * factor^{word\ length} \tag{3.7}$$

$$length\ correction\ factor = \frac{average\ score\ word\ length\ 4}{average\ score\ word\ length\ 5} \tag{3.8}$$

It seems reasonable to choose a value connected with the word lengths 4 and 5 because these are the two most frequently occurring lengths in this set of words. The quotients of the averages for word length n and word length *n-1* are more or less the same, so introducing a power function here seems reasonable as well. In this particular experiment we would have multiplied words with $2.089^{word\ length}$ for measure 2 and with $1.162^{word\ length}$ when we are using measure 3. No score correction for length is necessary for measure 1 because in this measure the word score dependents on one character score only.

We have repeated this experiment with words starting with *t* with the improved set-up: η is 0.01, network performance is tested after every 50 training rounds and score correction for length. After 350 rounds the error of the network stabilized. After training the network performed best with measure 3 rejecting seven of the eleven negative words and accepting all positive words. Measure 2 performed only slightly worse (rejecting six negative words and one positive word). Measure 1 performed poorly. It accepted all negative and all positive test words.

The performance of the network is not perfect. However, it seems reasonable to attempt to apply an SRN with this setup to the complete training data set.

3.3 Orthographic data with random initialization

We have trained SRNs to model the orthographic structure of the set of 5577 Dutch monosyllabic words described in section 2.2 of chapter 1. The networks have been trained with the setup which has been used in the previous section: learning rate η is 0.01, momentum α is 0.5, network performance is tested after every 50 training rounds and score correction for length. The length correction factors were equal to the average value of the words with length 4 divided by the average value of the words with length 5. These were the two most common lengths of the words in the training data. We have continued training until the change of the total sum squared error of the network resulting from a 50 rounds training period was smaller than 1%. We have explained that using an SRN for this experiment requires a separation character between the words but in order to make the comparison between these experiments and the bigram experiments of chapter 2 more fair we have provided all words with both a word start token and a word end token.

The model we are going to derive for the 5577 training words will be larger than the one for the 184 words starting with t. The size of the hidden layer in an SRN is

hidden cells	training rounds needed	total sum squared error	measure	accepted positive strings	rejected negative strings
4	250	26134	1	600 (100%)	0 (0%)
			2	599 (99.8%)	33 (5.5%)
			3	600 (100%)	50 (8.3%)
10	200	25711	1	600 (100%)	6 (1.0%)
			2	600 (100%)	3 (0.5%)
			3	600 (100%)	31 (5.2%)
21	250	24772	1	600 (100%)	0 (0%)
			2	600 (100%)	5 (0.8%)
			3	600 (100%)	6 (1.0%)

Figure 3.10: The performance of three different network configurations for the complete orthographic data set of monosyllabic Dutch words (5577 training words, 600 positive words and 600 negative words). The networks perform well with regards to accepting the words of the positive test set. However they reject far too few words of the negative test set.

closely related to the size of the model it can acquire. Therefore we expect that the number of hidden cells that was sufficient for the 184 words may be insufficient for the larger data set. But what would be the best size of the hidden layer? One can argue that the *t*-word SRN already contained a model for the words except for the first character (which always was a t), so adding one or two cells to the hidden layer should be enough to capture a complete model for monosyllabic words. At the other hand, someone might say that the training data increased with a factor 5577/184=30.3 so the number of weights should increase with a large factor. We tested both approaches and an intermediate one and performed additional experiments with SRNs with hidden layer size of 4, 10 and 21.[5]

We have noticed that the random initialization of the networks has some influence on their performance. In order to get a network performance that is reliable and stable, we performed five parallel experiments with each network configuration. Of these five experiments we chose the one with the lowest total sum squared error for the training data and tested its performance on the data set. Our motivation for choosing the performance of the network with the smallest error rather than the average performance of the networks is that we are interested in the best achievable performance of the network.

The network with 4 hidden cells in combination with measure 3 performed best,

[5]The number of four was derived by adding one to three (the number of hidden cells in the previous experiment). With 21 hidden cells we obtain a network that has approximately 30.3 times as many links as the t-words network. The value 10 lies somewhere between the other two numbers.

set 1:	V + , C V +	152 words
set 2:	V + , C V + , V + C , C V + C	1397 words
set 3:	the complete training set	5577 words

Figure 3.11: Increasingly complex training data sets. V is a vowel, C is a consonant and a character type followed by a plus means a sequence of 1 or more occurrences of that character type. The network will be trained with the first set first. When the network error stabilizes training will be resumed with the second data set. The training process will continue with the third set when the training process for the second set has stabilized.

rejecting 50 of 600 negative words (8.3%) while accepting all 600 positive words (see figure 3.10). The network tends to accept all words and this is not what we were aiming at. The performance even becomes worse when the number of hidden cells is increased. The total sum squared error decreases for a larger number of hidden cells which indicates that the network performs better on the training set. So a larger number of hidden cells makes it easier for the network to memorize features of the training set but it also degrades the network's generalization capabilities. This fact has already been recognized in other research using feed-forward networks.

If the SRNs are not able to learn the structure of our orthographic data with random initialization, we will have to rely on a linguistic initialization for a better performance.

3.4 Orthographic data with linguistic initialization

In the previous chapter we described how a Hidden Markov Model can start learning from some basic initial linguistic knowledge. Adding initialization knowledge to a neural network is more difficult. The most obvious way to influence the network training phase is by forcing network weights to start with values that encode certain knowledge. However, for a network of reasonable size it is hard to discover the exact influence of a specific weight on the performance of the network. Therefore it is difficult to find a reasonable set of initialization weights for a network.

Instead of trying to come up with a set of initialization weights, one can make the network discover such a set of weights. The idea would be to train the network on some basic problem and use the weights that are the result of that training phase as starting weights for a network that attempts to learn a complex problem. This approach to network learning was first suggested in (Elman 1991). In this work Jeffrey Elman described how a network that was not able to learn a complex problem was trained on increasingly more complex parts of the data. This approach was successful: after training the network was able to reach a satisfactory performance on the task.

We will describe an approach to our problem that is similar to the approach chosen by Elman. We divide our orthographically represented monosyllabic data in three sets of increasing complexity. The first set contains words without consonants and words containing one initial consonant and one or more vowels. The second set contains all words of the first set plus the words that consist of vowels and a one consonant suffix and consonant prefix that is either empty or contains one consonant. The third set contains all words (see figure 3.11).

This approach seems implausible from a cognitive point of view since children do not receive language input with simple words first and complex words later. However if we look at the language production of children we see that the complexity of the structure of their utterances increases when they grow up. Young children start with producing V and CV syllables (V: vowel and C: consonant) before they produce CVC syllables and progress to more complex syllables. While their language input is not ordered according to syllable complexity, their language production model seems to develop from simple syllable output to output of more complex syllables. With the incremental approach we attempt to make our networks go through the same development process.

Our definition of vowels contained *a*, *e*, *i*, *o*, *u*, *y* and the diphthong *ij*. All other characters including the *j* without a preceding *i* were regarded as consonants. We have considered the *ch* sequence as one consonant because the pronunciation of this consonant sequence in Dutch is equal to that of the *g*.

We have trained the network with orthographic data using an SRN with 4 hidden cells because that one has performed best in the previous experiments. We have used the same experiment set up as described in the previous section: learning rate η is 0.01, momentum α is 0.5, network performance is tested after every 50 training rounds and score correction for length. Training data was of increasing complexity but as test data we have used the complete positive and negative test data sets for all experiments. We have used the performance of the network on the different training data sets for determining the threshold value and the score correction for length values for measure 2 and 3.

We have performed 5 experiments for all data sets. When the performance of an SRN for data set 1 stabilized, we continued training with data set 2. When the performance of that SRN stabilized we continued training with data set 3. The network with the lowest error rate after training with data set 3 was chosen for the final performance tests. Its two predecessors were used for the two intermediate performance tests.

This learning procedure was not been successful. The network performed even worse on the data than the SRNs with random initialization (see 3.12). Measure 3 performs best by accepting all positive words and rejecting 29 negative words (4.8%, while the SRN with random initialization was able to reject 8.3%). Aiding the network by structuring the training data does not improve the final performance of the network. The results that the SRNs have achieved for orthographic data are disappointing. For the time being we will refrain from testing the performance of the SRNs on phonetic data. Instead we will try to find out why SRNs are performing much worse on our task than we had expected.

training data set size	training rounds needed	total sum squared error	measure	accepted positive strings	rejected negative strings
152	300	452	1	548 (91.3%)	121 (20.2%)
			2	104 (17.3%)	489 (81.5%)
			3	392 (65.3%)	209 (34.8%)
1397	100	4529	1	560 (93.3%)	150 (25.0%)
			2	481 (80.2%)	224 (37.3%)
			3	398 (66.3%)	269 (44.8%)
5577	100	25334	1	599 (99.8%)	0 (0%)
			2	600 (100%)	27 (4.5%)
			3	600 (100%)	29 (4.8%)

Figure 3.12: The performance of the network when trained with orthographic data of increasing complexity. Network weights were initialized by values obtained from the experiment with less complexity. The network was tested with the complete positive test set and the complete negative test set. The errors of the network have been computed for the training sets which has different sizes for the three parts of the experiment.

4 Discovering the problem

In this section we will explain why the SRNs perform worse on our data than on the data of Cleeremans et al. First we will explain the major difference between our experiments and the ones of (Cleeremans 1993). Then we will show that complicating the data used in the experiments by Cleeremans et al. will decrease the performance of the SRNs. After this we will examine a possible solution for improving the performance of SRNs on our phonotactic data and present the results of this method.

4.1 The influence of the number of valid successors of a string

In the experiments described in the previous section the SRNs have been unable to acquire a good model for the phonotactic structure of Dutch monosyllabic words. While our target was rejecting all negative test data our SRNs have never been able to reject more than 8.3%. The performance of the Cleeremans measure was very disappointing: the measure never rejected more than 1.0% of the negative data. This performance is a sharp contrast with the performance on grammaticality checking reported in (Cleeremans 1993) in which *all* ungrammatical strings were rejected by the network. This fact surprised us so we took a closer look at the differences between our experiments and the experiment described in (Cleeremans 1993) chapter 2.

The most obvious difference we were able to find was the maximal number of valid successors of grammatical substrings.[6] In the finite state model for the Reber grammar used by Cleeremans et al.(figure 3.6), at any point in a string the maximal number of valid successors is two. This is very important for the format of the network output. This output consists of a list of values between zero and one which are estimations for the probability that certain tokens are successors of a specific substring. For example, if a network trained with Reber grammar data is processing a string, it might present on its output the pattern [0.0 , 0.53 , 0.00 , 0.38 , 0.01 , 0.02 , 0.00] ((Cleeremans 1993), page 43). In this list the numbers estimate the probabilities that B, T, S, P, X, V, and E are the next token in some string. As we can see the two valid successors T (0.53) and P (0.38) receive an output value that is significantly larger than the output values of the other tokens. The network does not perfectly predict the correct successor but it outputs some average pattern that indicates possible successors. This network behavior was already recognized in (Elman 1990).

In the grammars for Dutch phonotactic structure the maximal number of valid successors is much larger than two. For example, according to our training corpus for the orthographic experiments the number of different tokens which are valid successors of the string *s* is 19. Some tokens occur frequently as successors of this character, for example *c* (23%) and *t* (24%) but others are very infrequent: *f*, *q* and *w* occur less than 1%. The frequency of the successors will be mirrored by the network output, as in the example from Cleeremans et al. But the network output contains random noise and therefore it will be difficult to distinguish between low frequency successors and invalid successors, like *b* and *x* in this example. For example, figure 3.7 shows network output in which *P* receives value 0.1 (ideally this should be 0.0) and *S* receives value 0.4 (ideally this should be 0.5). Cleeremans et al. have detected in the output of their network values of 0.3 where 0.5 was expected. One can understand that in an environment where signals have an absolute variation of 0.2 the difference between a signals 0.01 and 0.00 are in practice impossible to detect.

An alternative approach to choosing a threshold value is to take a larger threshold value in advance in order to reject more non-grammatical strings. This will not bring us closer to a solution because a larger threshold value will immediately reject the element with the smallest value of the training set. It might even cause the rejection of other grammatical elements from the test set and the training set. This is not acceptable to use: we want our models to accept all training data.

4.2 Can we scale up the Cleeremans et al. experiment?

We decided to test the influence of the number of valid successors by repeating the experiment described in (Cleeremans 1993) chapter 2. Apart from training an SRN to decide on the grammatically of strings according to the Reber grammar shown

[6]The standard term for the *average* number of valid successors is PERPLEXITY: PP(W) = $2^{H(W)}$ where W is some data set and H(W) is the entropy of this data set (Rosenfeld 1996).

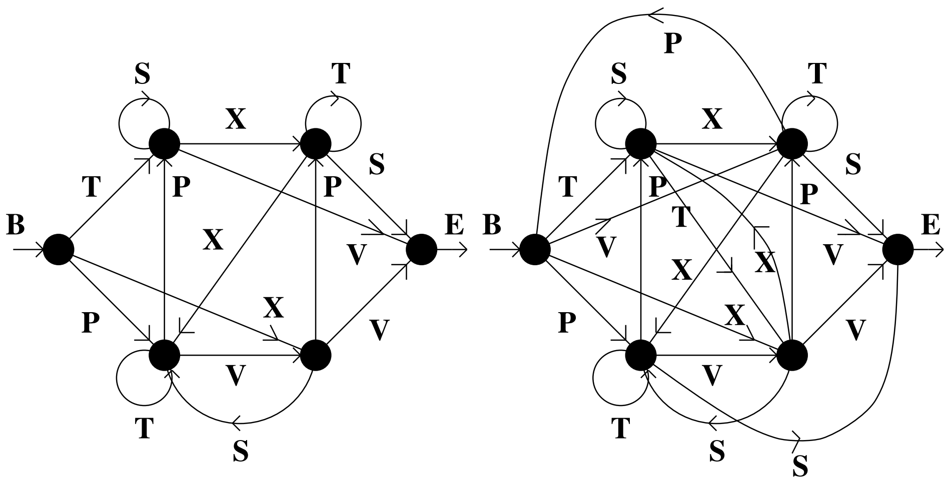


Figure 3.13: Finite state networks representing a Reber grammar with three valid successors for every valid substring (left) and one with four valid successors (right). The valid successors of, for example, substring *BT* are *S*, *X* and *V* for the three-successor grammar and *S*, *X*, *V* and *T* for the four-successor grammar.

in figure 3.6, we trained SRNs with strings from two alternative grammars. The first alternative grammar was an extension of the Reber grammar such that valid substrings have three possible successors (see figure 3.13). The second alternative grammar was an extension of the first and here valid substrings have four possible successors (see figure 3.13). For all grammars we have generated 3000 training words, 300 positive test words and 300 random words. Of the 300 random words for the Reber grammar 1 was grammatical and for the other two grammars 5 and 8 strings were valid.

We have trained the SRNs in steps of 50 training rounds using learning rate η=0.01, momentum α=0.5 and a network configuration that was similar to the one of Cleeremans et al. (7 input cells, 3 hidden cells, 3 context cells and 7 output cells). After each step we checked if the total sum squared error of the network had changed more than 1%. If this was the fact we continued training otherwise training was stopped. Only one experiment was performed for each grammar. As in the previous experiments we will reject words when their score is below the lowest score of the training data.

As we can see in figure 3.14 the performance of the SRN decreases when the complexity of the grammar increases. The SRN accepted all valid words for all grammars but the rejection rate of the random data decreased when the number of valid successors increased: for the grammar with two valid successors (one valid string in the random data) 100% of the invalid random strings were rejected, for three valid successors (5 valid strings) this dropped to 92.2% and for four valid successors (8

maximum number of successors	training rounds needed	total sum squared error	measure	accepted positive strings	rejected random strings
2	150	9079	1	300 (100%)	299 (99.7%)
			2	300 (100%)	296 (98.7%)
			3	300 (100%)	299 (99.7%)
3	250	12665	1	300 (100%)	272 (90.7%)
			2	300 (100%)	254 (84.7%)
			3	300 (100%)	241 (80.3%)
4	100	13944	1	300 (100%)	240 (80.0%)
			2	298 (99.3%)	172 (57.3%)
			3	299 (99.7%)	119 (39.7%)

Figure 3.14: The performance of the SRN for Reber grammars of increasing complexity. Some random strings are valid: 1 for the first grammar, 5 for the second grammar and 8 for the most complex grammar. The actual rejection percentages for measure 1 for the invalid data are 100.0% for the first grammar, 92.2% for the second grammar and 82.2% for the third grammar. The performance of the network deteriorates when the number of valid successors increases.

valid strings) the rejection rate was 82.1%. We can conclude that the number of valid successors influences the difficulty of the problem. (Cleeremans et al. 1989), (Servan-Schreiber et al. 1991) and (Cleeremans 1993), showed that SRNs are capable of acquiring finite state grammars in which a grammatical substring has two valid successors. We have showed that the performance of the SRNs will deteriorate rapidly when the maximal number of successors increases.

4.3 A possible solution: IT-SRNs

The output patterns of the SRNs after training shown in (Cleeremans 1993) page 43 show that the sum of the output signals is approximately equal to 1. One of the patterns shown is the output pattern for input *B* in an SRN modeling the Reber grammar: [0.0, 0.53, 0.00, 0.38, 0.01, 0.02, 0.00] (sum is 0.94). We have observed the same behavior from our SRNs. This behavior can be explained by the fact that the patterns we use as required output patterns during training consist of a single 1 and zeroes for the other values. For every input pattern there are usually different required output patterns in the training data and the output pattern of a network after training will be an average of all possible output patterns for that particular input. The values of each pattern sum up to one and therefore the values of an exact average will also sum up to one.

In an earlier section we have argued that it is difficult to distinguish a small output value (caused by an infrequent successor) from network output noise. This is the cause of poor performance of the SRNs in the previous experiments. We would like to increase the difference between the output values for infrequent successors and the network noise. We cannot do much about the network noise but we can try to increase the output values for the successors.

We will change the required SRN output patterns during training in such a way that they will satisfy the following requirement:

> INCREASED THRESHOLD REQUIREMENT
> If SRN training data that has been encoded by using the localist mapping allows token Y as the successor of token X then the required value of the output cell which is 1 for token Y should be equal to or larger than some threshold value during training whenever X is presented as input.

We will call SRNs which satisfy this requirement INCREASED THRESHOLD SRNS or in short IT-SRNs. Let us look at how this requirement influences the training process. We will examine a network trained to model the standard Reber grammar with two valid successors (figure 3.6) and look at the valid successors for the start character *B*.

B can be followed by *T* or *P*. In the output pattern of the network *T* is represented by the second value and *P* is represented by the sixth value. Therefore a *B* at the input of a standard SRN should require [0.0 , 1.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0] on the output when its successor is *T* and [0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 1.0 , 0.0] when its successor is *P*. However, in a IT-SRN the output values corresponding with all valid successors of a pattern should at least be equal to some threshold value. If we choose 0.5 as threshold value and present *B* as input the IT-SRN should require [0.0 , 1.0 , 0.0 , 0.0 , 0.0 , 0.5 , 0.0] as output when the successor is *T* and [0.0 , 0.5 , 0.0 , 0.0 , 0.0 , 1.0 , 0.0] when the successor is *P*.

By making a network satisfy this increased threshold requirement it should become easier to recognize the difference between low frequency output values and noise. All valid successors of a character will be represented in every output pattern of this character during training. The average output values of the network will increase and so will the values assigned by the network to the training and the test data. We assume that IT-SRNs will be able to reject more negative test data than standard SRNs. In the next section we will test this assumption.

4.4 Experiments with IT-SRNs

We have trained the IT-SRNs to model the orthographic data first with random initialization and after that with linguistic initialization. Data and experiment set up were equal to the experiments described in section 3.3 and 3.4: learning rate η is 0.01, momentum α is 0.5, network performance is tested after every 50 training rounds and score correction for length. We have chosen threshold value 0.5 for valid successors and used IT-SRNs with four hidden cells because SRNs with 4 hidden cells have achieved the best result in the experiments described in section 3.3. The training data consisted of 5577 monosyllabic Dutch words and tests were performed with 600

initialization	training rounds needed	total sum squared error	measure	accepted test strings	rejected random strings
random	200	163496	1	600 (100%)	15 (2.5%)
			2	600 (100%)	26 (4.3%)
			3	600 (100%)	20 (3.3%)
linguistic: 157 training strings	800	1631	1	11 (1.8%)	593 (98.8%)
			2	11 (1.8%)	594 (99.0%)
			3	242 (40.3%)	410 (68.3%)
linguistic: 1397 training strings	300	26587	1	468 (78.0%)	247 (41.2%)
			2	434 (72.3%)	409 (68.2%)
			3	600 (100%)	6 (1.0%)
linguistic: 5577 training strings	250	164042	1	600 (100%)	0 (0%)
			2	600 (100%)	2 (0.3%)
			3	600 (100%)	21 (3.5%)

Figure 3.15: The performance of the 4 hidden cell IT-SRN with random initialization and IT-SRNs with linguistic initialization for the orthographic data. The total sum squared errors are smaller for the first two linguistic experiments because they were computed for the small training sets. The IT-SRNs performed best with random initialization for measure 2 but it only rejected 4.3% of the negative test data. Neither randomly initialized nor linguistically initialized IT-SRNs were able to reject more negative words than randomly initialized SRNs.

test words which were not present in the training data and 600 ungrammatical strings. In the experiments with linguistic initialization the training data was divided in three groups of increasing complexity just like in section 3.4. In both experiment groups 5 parallel experiments were performed and the IT-SRN which achieved the lowest error for the training data was used for the tests.

The results of the two experiments can be found in figure 3.15. The error of the network has become much larger than in the SRN experiments because the IT-SRN output patterns during training are different from the output patterns during testing. IT-SRNs do not perform better on this data than SRNs. The randomly initialized IT-SRN has achieved the best performance with measure 2 accepting all positive test data but rejecting only 26 negative strings (4.3% while the randomly initialized SRNs have achieved 8.3%). The linguistically initialized IT-SRN has achieved the best performance with measure 3 accepting all positive test data but rejecting only 21 negative strings (3.5% while the linguistically initialized SRNs have achieved 4.8%).

The performance difference between IT-SRNs and SRN can only be made visible if we change the threshold used for deciding if a string is grammatical or not. We have defined this threshold value as being equal to the lowest score assigned to a word in

network	measure	100%		99%		95%		90%	
		positive	negative	pos.	neg.	pos.	neg.	pos.	neg.
SRN	1	0	0	5	14	28	105	61	208
	2	1	33	12	207	31	325	53	395
	3	0	50	7	270	25	399	65	469
IT-SRN	1	0	15	12	306	35	371	52	438
	2	0	26	11	316	32	405	58	449
	3	0	20	3	56	24	110	53	156

Figure 3.16: Number of words that were rejected in the orthographic positive and negative test data in the randomly initialized SRN and IT-SRN when the acceptance threshold is increased and only part on the training data will be accepted. The percentage indicates the amount of training data that will be accepted. Now the network is able to reject 78.2% of the negative words (SRN, measure 3, 90% column) but the price is high: 10.8% of the positive test data and 10.0% of the training data will be rejected as well.

the training data. In that way we have made sure that all words in the training data will be accepted. If we allow some words of the training data to be rejected then we can raise the value of the threshold. The result of this has been made visible in figure 3.16.

When we allow 1% of the training data being rejected, the IT-SRN will reject at best 316 words of the negative test data (52.7%, measure 2). The SRN will reject at best 270 words of the training data (45.0%, measure 3) when we apply the same constraint. Increasing the acceptance threshold seems to be the only way to improve the performance of the SRNs and the IT-SRNs on the negative data. However, it is not a good solution. The best performance that we have achieved by increasing the acceptance threshold was rejection of 469 words of the negative test data (78.2%, SRN with measure 3) but then we also had to reject 10% of the training data and 10.8% of the positive test data.

Our conclusion is that neither IT-SRNs nor an increasement of the acceptance threshold will produce acceptable behavior of the networks on our learning task.

5 Concluding remarks

In this chapter we have described our attempts to solve the central problem in this thesis, automatic acquisition of the phonotactic structure of Dutch monosyllabic words, by using connectionist techniques, in particular the Simple Recurrent Network (SRN) developed in (Elman 1990). We compared two of our word evaluation measures with a word evaluation measure used in (Cleeremans 1993), (Cleeremans et al. 1989) and

(Servan-Schreiber et al. 1991). In the experiments of Cleeremans et al. the latter measure reached a 100% performance in a grammaticality checking task using a small artificial grammar (the Reber grammar, see figure 3.6). Our experiments resulted in SRNs which performed well with respect to accepting the positive orthographic test data. However, they performed poorly in rejecting negative orthographic data: they never rejected more than 8.3% of our 600 negative strings. In these experiments the measure used by Cleeremans et al. came out as the worst of the three measures used. It never rejected more than 1.0% of the negative data.

We have shown that the reason for the difference in performance lies in the complexity of the grammars that we are teaching the networks. Cleeremans et al. have used a grammar in which for every grammatical substring the maximal number of valid successors never is larger than two. In the grammars we use this number is about 10 times as large. Therefore the relevant output values of the network become smaller and it is more difficult to distinguish them from random network noise. We have shown that even a small increase of the complexity of the grammar that models the training data will lead to a deterioration of the behavior of SRN models trained on this data. Increasing the maximal number of valid successors in the Reber grammar to three resulted in the SRN being unable to reach perfect performance in a grammaticality checking task.

We have attempted to solve this problem by changing the required output patterns during training. We have forced the network to use information about all valid successors of a character instead of just learning the relation between two characters at each training step. However the resulting modified SRN, Increased Threshold SRN (IT-SRN), did not perform better than the standard SRN. At best it rejected only 4.3% of the negative test data.

It seems that the only way to make the SRNs and the IT-SRNs reject more invalid data is increasing the acceptance threshold value and thus allowing that a part of the training data will be rejected. But even when we allow 10% of the training data to be rejected we cannot make the networks reject more than 78.2% of the invalid data This is a low percentage compared with the performance of the HMMs. Furthermore this solution is not acceptable to us: we want our models to accept all training data.

We conclude from the experiments described in this chapter that Simple Recurrent Networks do not seem to be able to reach a good performance on our learning task.[7]

[7] Some alternative approaches with SRNs to this learning task will be discussed in section 2 of chapter 5.