Slovak Technical University
Faculty of Electrical Engineering and Information Technology
Department of Computer Science and Engineering

Recurrent Neural Networks

Michal Čerňanský

Written Part of the Ph.D. Examination
Supervised by Ľubica Beňušková, Ph.D.

Bratislava, December 2001

Abstract

Dynamical neural networks have much larger potential than classical feed-forward neural networks. Their output responses depend also on time position of given input and they can be successfully used in spatio-temporal task processing. This work describes basic architectures of dynamical neural networks. Training methods and algorithms are also discussed. Widely used architecture - Elman's simple recurrent network is chosen to explain its behavior by means of the theory of iterated function system. Novel methods making use of this type of dynamics are described. Simple recurrent network can also acquire behavior of regular grammar and work as finite state automaton. Also it is able to process strings generated by simple context-free grammars by creating counting mechanisms. Our preliminary experiments are described and future research activities are suggested.

Contents

List of Figures

Chapter 1

Introduction

Artificial neural networks are inspired by biological neural networks such as human brain. According to [12], human brain is an extremely powerful and complex information-processing system. It works in an entirely different way than the computer does. Rate of neural events is much more slower than the rate of operations in logical gates. In spite of its relatively slow operation rate the computational power of the human brain is enormous, many times superior to that of nowadays computers. This computational power is the result of cooperation between huge amount of relatively simple processing elements - neurons. Interactions between neurons are determined by the neural interconnections that are called synapses. Plasticity - the ability to adapt to its surrounding environment can be met by two mechanisms: by creating new synaptic interconnections and by modifying the strength of existing ones. In this manner an artificial neural network resembles the brain, it is processing device composed of simple elements working in parallel way. Knowledge is acquired by the process called learning and information obtained during this process is stored in strengths of neural interconnections.

Concepts of the theory of artificial neural networks can be found far in the past. However, the pioneering work of this field was achieved by McCulloch and Pitts in 1943. They introduced simplified model of neural cell - binary threshold unit. Later, models of units organized in layers were developed and a problem, how to find appropriate synaptic weights arose. Rosenblatt (1962) was able to prove convergence of a learning algorithm for networks without intermediate layers. But in 1969 Minski and Papert pointed out that some rather elementary computations cannot be done by this learning algorithm. XOR problem is typical example of the tasks that cannot be solved by Rosenblatt's neural network. Since that, various people have developed an appropriate learning algorithm for multi layer neural network. It is known as back-propagation and was introduced by Rumelhart, Hinton and Williams in 1986. Much

activity was centered on back-propagation and its extensions or modifications. Still it is the most widely used algorithm for training multi-layer neural network.

All these models are extremely simplified from the biological point of view. Still, many researcher believe that they are able to make us understand better the principles of biological computation. Of course, neural networks are successfully used in many practical applications such as object recognition, sound and signal processing, robotics, etc.

Chapter 2

Feed-Forward Neural Networks

Feed-forward neural networks are usually composed of multiple layers. These networks are also called perceptrons or multi-layer perceptrons. Every layer is composed of simple processing elements called neurons (Figure 2.1). Usually, there are no connections leading from units to previous layer or to units in the same layer nor the units more than one layer ahead. Every unit feeds only the units in the next layer. Networks, that are not strictly feed-forward are called recurrent networks.

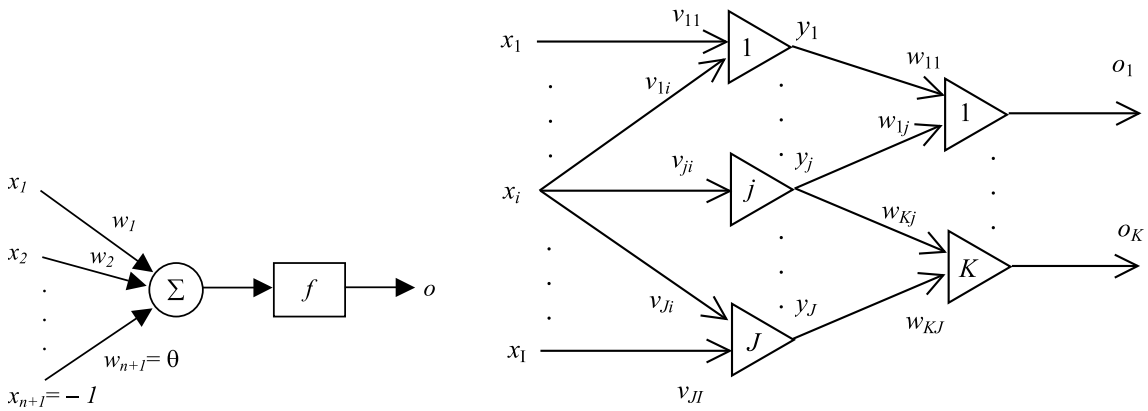

Figure 2.1: Model of a neuron and two-layer feedforward neural network.

The units in the intermediate layer are called hidden units, they have no direct connection to the outside world, neither input or output. Inputs from the environment are provided by input units. Output values calculated by the network are delivered by output units. Computation of a single processing element - neuron is very simple. For every input x_i there is a

weight w_i. One input is usually hardwired to be -1 and the corresponding weight is called threshold or bias. Output o is calculated according to these equations:

$$o = f(net) \tag{2.1}$$

$$net = \sum_{i=1}^{n} w_i x_i = \sum_{i=1}^{n-1} w_i x_i - \theta \tag{2.2}$$

where net is an internal activity level of the neuron, also called net output, θ is a threshold and f stands for the so-called activation function. Common activation functions are the step function (McCulloch and Pitts binary threshold unit - figure 2.2 (a)), linear and piecewise linear functions (2.2 figure (b)) and sigmoid function (2.2 figure (c)).

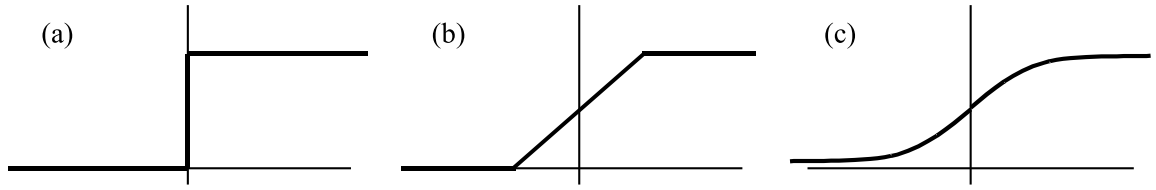

Figure 2.2: Common activation functions.

There are two main paradigms how to adjust the strengths of weights iteratively. The first is called supervised learning or learning with a teacher. Current output of neural network is compared with a desired output and appropriate weight changes are computed. A "teacher" is an external entity that knows the correct output values. Error Back-propagation is the commonest supervised learning algorithm. On the other hand neural networks trained in unsupervised manner are expected to find the correlations in the input data and to produce corresponding output signal. No desired output signal is given to them. An example of unsupervised learning algorithm is the BCM learning rule proposed by Bienenstock, Cooper and Munroo [4].

A mechanism of weight adjustment for simple neuron (or single-layer network) was proposed by Rosenblatt in 1962. But such a network is able to solve only linearly separable problems. These limitations do not apply to feed-forward networks with hidden layers between input and output layer. Although the greater power of the multi-layer network was realized long ago, a learning algorithm was invented 20 years after Rosenblatt's work. The algorithm gives a prescription for changing weights to learn a training set of input output pairs. The basis

is a simple gradient descent and back-propagation of error signals in multi-layer feed-forward network. It is called error back-propagation algorithm or simply back-propagation.

2.1 Error Back-Propagation Algorithm

Back-propagation is an example of supervised learning. It is based on minimization of error by gradient descent. Consider two layer feedforward network (Figure 2.3).

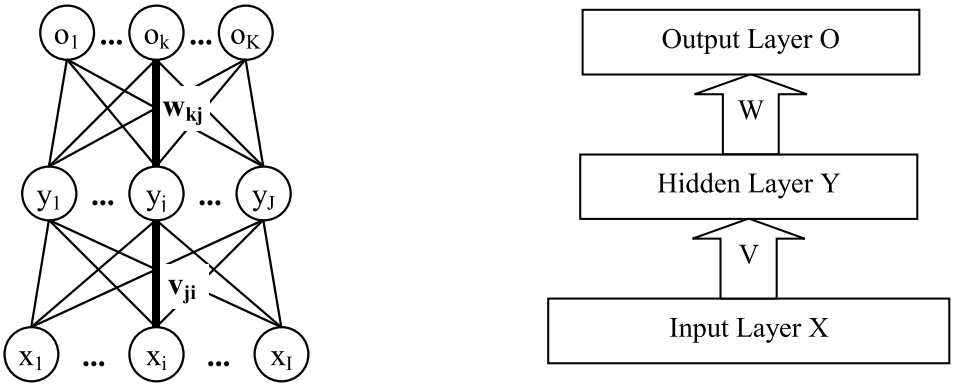

Figure 2.3: Two layer feedforward neural network showing notation used for units and weights.

2.1.1 Forward pass

Given input patter $\overline{x} = (x_1, ..., x_I)$ output vector $\overline{o} = (o_1, ..., o_K)$ is calculated as forward pass of signal. Simple processing elements - neurons calculate their outputs according to the equation 2.1. Hidden unit j calculates its net input $\widetilde{y}_j$:

$$\widetilde{y}_j = \sum_{i=1}^{I} v_{ji} x_i \tag{2.3}$$

and produces output y_j:

$$y_j = f(\widetilde{y}_j). \tag{2.4}$$

Output unit k calculates its net input $\widetilde{o}_k$ and output value o_k in the same way:

$$\widetilde{o}_k = \sum_{j=1}^{J} w_{kj} y_j, \tag{2.5}$$

$$o_k = f(\widetilde{o}_k), \tag{2.6}$$

where v_{ji} is weight connecting hidden unit j with input unit i and w_{kj} is weight connecting output unit k with hidden unit j. Special weights (units' thresholds $v_{j(I+1)}$ and $w_{k(J+1)}$) from constant input hardwired to -1 are not considered, but they follow exactly same rules as described here.

2.1.2 Backward pass

For given input pattern $\overline{x}$ output $\overline{o}$ is calculated (equations 2.4 and 2.6). But desired-expected output values $\overline{d} = (d_1, ..., d_K)$ are unlikely to be the same as computed ones $\overline{o} = (o_1, ..., o_K)$. This difference is expressed through error function. Commonly used error function is the L2 norm:

$$E = \frac{1}{2} \sum_{k=1}^{K} (d_k - o_k)^2. \tag{2.7}$$

The aim of the back-propagation algorithm is to minimize the error E. Weights are updated by gradient descent method. Weight changes for units in layers are calculated in backward order - from output layer to first hidden layer.

Weight changes for output units are calculated first:

$$\Delta w_{kj} = -\alpha \frac{\partial E}{\partial w_{kj}} = \alpha \delta_k y_j, \tag{2.8}$$

where error signal δ_k is defined as

$$\delta_k = f'(\widetilde{o}_k)(d_k - o_k). \tag{2.9}$$

f' is an activation function derivative and α is a learning rate. Weight changes for units in hidden layer are calculated:

$$\Delta v_{ji} = -\alpha \frac{\partial E}{\partial v_{ji}} = \alpha \delta_j x_i, \tag{2.10}$$

where error signal δ_j of hidden neuron j is defined as

$$\delta_j = f'(\widetilde{y}_j) \sum_{k=1}^{K} w_{kj} \delta_k. \tag{2.11}$$

This equation allows us to determine the error signal of hidden unit in terms of the error signal of units that the hidden unit feeds. Error signal is propagated backwards, that is why the algorithm is called error back-propagation. Two layer network has only one hidden layer, but in general, number of hidden layers can be higher. This calculation can be easily generalized to multi-layer network. Error signals for units in all hidden layers are calculated exactly as shown in equation 2.11.

An activation function usually used is sigmoid function defined as

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{2.12}$$

A derivative of it can be simply computed as

$$f'(x) = f(x)(1 - f(x)) \tag{2.13}$$

and therefore error signals δ_k (equation 2.9) and δ_j (equation 2.11) can be rewritten as

$$\delta_k = f(\widetilde{o}_k)(1 - f(\widetilde{o}_k))(d_k - o_k) \tag{2.14}$$

and

$$\delta_j = f(\widetilde{y}_j)(1 - f(\widetilde{y}_j)) \sum_{k=1}^{K} w_{kj}\delta_k. \tag{2.15}$$

2.1.3 Momentum

Back-propagation algorithm can be very slow. Other common problem while using gradient descent is its susceptibility to local minima. Instead of finding global minimum of error function back-propagation can get stuck in one of local minima. Simple but very useful method to avoid this problems (at least partially) is using momentum. Actual weight changes take into account weight changes from previous time step:

$$\Delta v_{ji}(t) = \alpha\delta_j x_i + \beta\Delta v_{ji}(t-1) \tag{2.16}$$

and

$$\Delta w_{kj}(t) = \alpha\delta_k y_j + \beta\Delta w_{kj}(t-1), \tag{2.17}$$

where β is momentum parameter.

2.1.4 Weight update

After weight changes are calculated, weights can be updated:

$$w_{kj} = w_{kj} + \Delta w_{kj} \tag{2.18}$$

and

$$v_{ji} = v_{ji} + \Delta v_{ji}. \tag{2.19}$$

2.1.5 Batch mode

According to this version of weight update, after a pattern is presented to the network at the input, weights are updated before the next pattern is considered. It works well for small learning rate. An alternative batch mode can be used. In this case weights are updated after all patterns have been presented. Batch mode is a result of minimization of total error function (over all patterns).

2.1.6 Parameters and other notes

Untrained network is usually initialized by random weights from small interval. It is important to initialized weights with random values, both positive and negative ones, for example by using uniform distribution over interval $[-1, 1]$. Specific weights - thresholds although not considered in equations, obey the same rules as described. They are randomly initialized and trained exactly as other weights. Learning rate α is value usually chosen from interval $(0, 2]$ and momentum parameter β from interval $[0, 1]$ often value 0.9 is used for β. While using described incremental mode of back-propagation algorithm (not batch mode), it is good idea to chose patterns in random order. Frequently used activation function is sigmoid (equation 2.12, figure 2.2 (c)).

Chapter 3

Dynamical Neural Networks

The back-propagation algorithm has become the most popular method for training neural networks. Neural network trained according to this algorithm can learn only a static input-output mapping. This is well suited for, for example, a pattern recognition tasks, where input and output represent spatial patterns, patterns that are independent of time. In this case an output of the network depends entirely on the current input, a position of input in time is not relevant. But time is important in many tasks such as signal-processing or speech. There is a need for computational system whose response behavior varies over time. Neural network has to be provided with dynamic properties, that would allow the network acquire a knowledge from spatio-temporal tasks. The network must be given a memory for storing temporal context.

3.1 Time Delay Neural Network

One way of how to accomplish this task is to introduce time delays into a network structure [13]. Such approach was suggested by Hinton in 1988 and this technique is called Time Delay Neural Network (TDNN). The network is provided not only with the current input vector but also with input vectors from previous discrete time steps. Extended input is called the "time window".

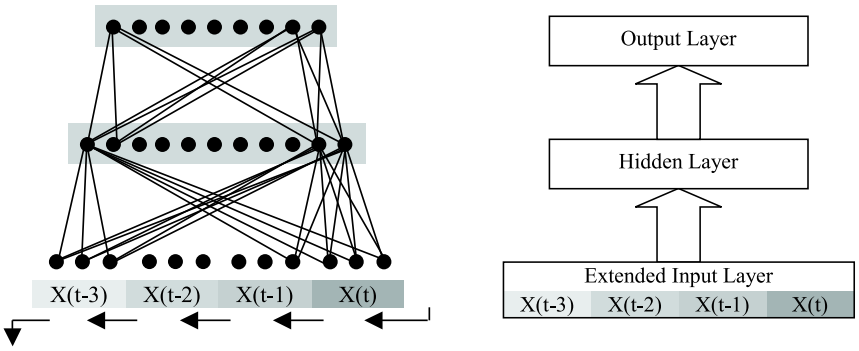

Figure 3.1: Simple time delay neural network.

The simplest variant is represented in the figure 3.1 (a). All layers are fully connected, only some connections are shown. Figure 3.1 (b) shows simplified schematic representation of TDNN. An advantage of this TDNN approach is that the network can be trained by common error back-propagation method. Even such a simple technique can be successful in some cases and can acquire the structure hidden in temporal data. But the way how this type of dynamical network represents temporal context can be insufficient.

To illustrate a potential of TDNN consider two simple automata. They characterize simple association tasks with temporal context. With given state and a given input symbol is uniquely associated with the next state and an output symbol.

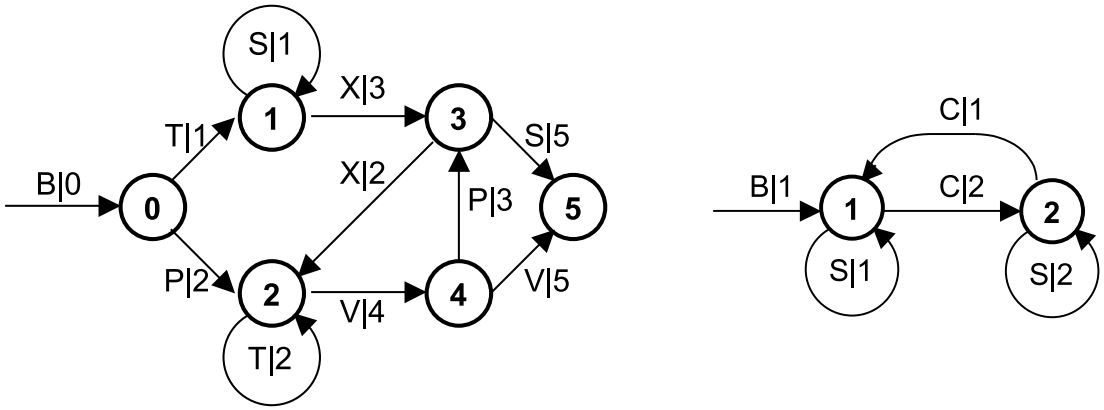

Figure 3.2: Two simple Mealy automata.

First Mealy automaton represented in figure 3.2 (a) was motivated by well known Reber automaton. The aim of the neural network is to associate given input symbol with an output symbol. In this specific case output symbol describes the next state of the automaton. Each automaton state is uniquely determined by the last two input symbols. TDNN with time window of length 2 can easily create appropriate mapping between the last pair of input symbols and a corresponding output symbol. An automaton in figure 3.2 (b) is also relatively simple. There are only three input symbols: one starting symbol, one symbol for "changing" automaton state and one for "staying" in current state. Input symbol sequence can be infinitely long and current automaton state depends also on the very first input symbol. It seems, that TDNN architecture with time window of any finite length would fail in solving this task.

In previous very simple example of TDNN time window was used only for input units. This time window technique can also be extended for hidden units and whole network can consist of multiple copies of hidden and output layers. Figure 3.3 shows TDNN replicated in time. Output layer consists of 2 copies of output neurons and hidden layer consists of 4 copies of hidden neurons. Output units apply their weights to three-time-step window of hidden units. Hidden units apply their weights to four-time-step window of input units.

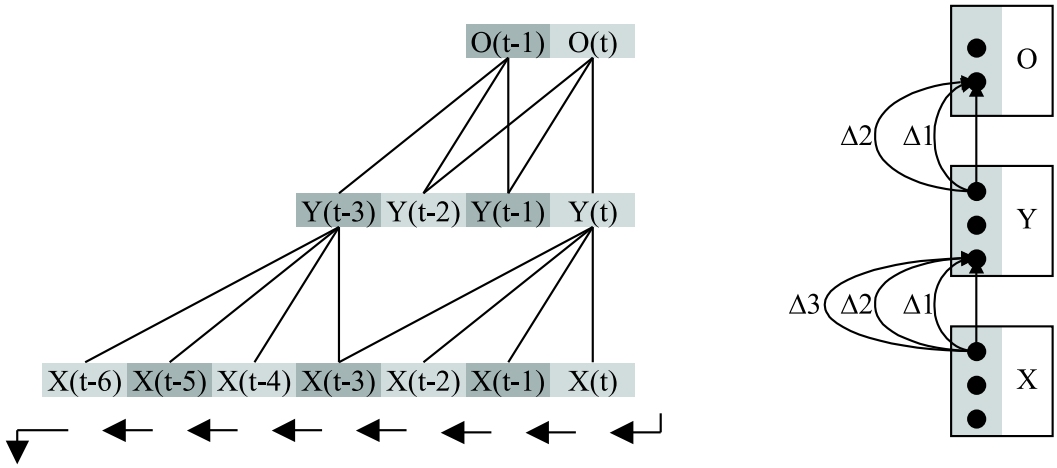

Figure 3.3: Extended TDNN.

3.2 Recurrent Neural Networks

Another way how to incorporate dynamical properties into a neural network is to built feedback into its design through recurrent connections. One well known architecture was introduced by Elman [10] and is called Simple Recurrent Network (SRN). This architecture is presented in 3.4 (a). Network input layer is extended with so called recurrent or context neurons. They hold activations of hidden neurons from previous time step. Other interesting architectures are modified Elman architecture (figure 3.4 (b)), architectures proposed by Jordan (figure 3.4 (c)) and Bengio (figure 3.4 (d))

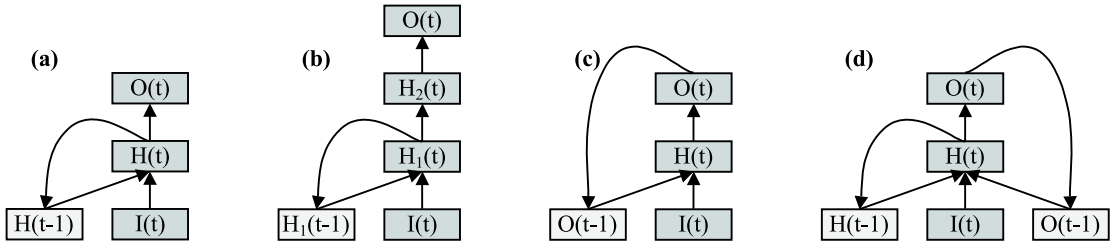

Figure 3.4: Architectures of recurrent networks.

There are two main approaches how to train recurrent networks. First is called Back-propagation through time and the second is called Real Time Recurrent Learning.

3.3 Back Propagation Through Time

Back propagation through time is based on the idea that every recurrent network can be unfolded in time and an appropriate feedforward network with identical behavior over a particular time interval can be created [33]. Unfolded recurrent neural network is multilayer feedforward neural network with extra layers for every time step. Unfolded Elman network is represented in figure 3.5. This method can be successfully used when it is possible to partition the input data into epochs. For each epoch recurrent network is unfolded in time and then weight changes of neuron replicas are computed by the error back-propagation algorithm. Specific neuronal weight modification is calculated as a sum of weight changes calculated for all replicas of given neuron. This epoch by epoch approach is not suitable for real-time operation of recurrent network.

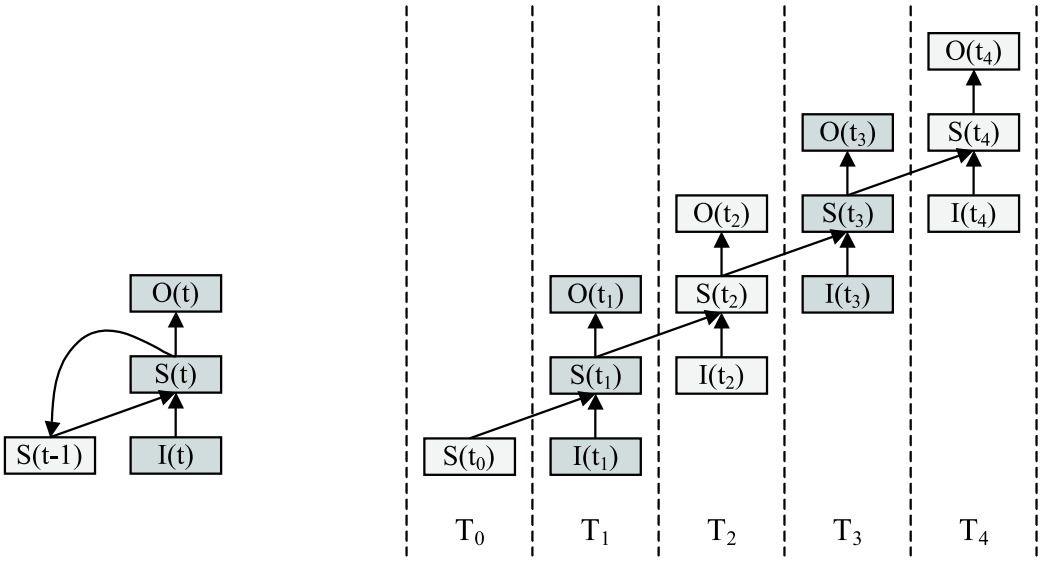

Figure 3.5: Elman SRN unfolded in 4 time steps into feedforward neural network.

3.4 Real Time Recurrent Learning

Real time recurrent learning [40] was proposed to address continuously running recurrent network. There is no need for duplicating hidden neurons and sequences of arbitrary length can be presented to a network. The principle lies in updating the weights after each time step although weight changes are not derived from exact gradient of total error function. For small learning rate this approach works well.

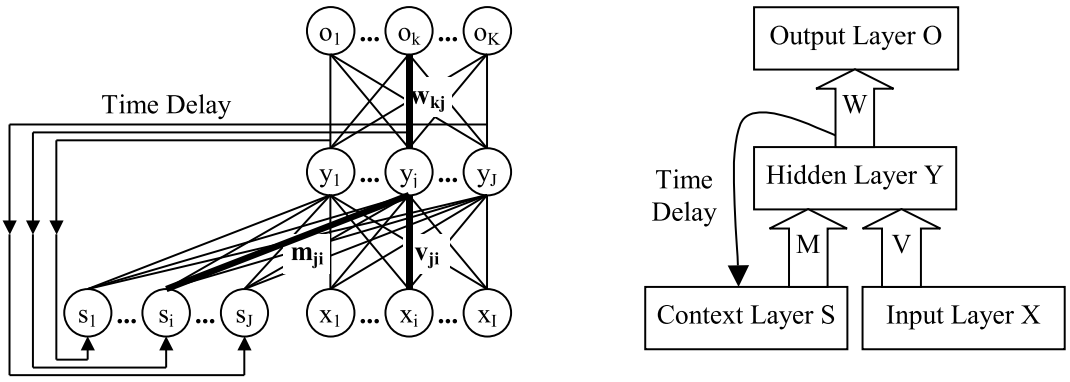

Figure 3.6: Simple Recurrent Network showing the notation of units and weights.

3.4.1 Forward pass

Given input patter $\overline{x}(t) = \Big(x_1(t), ..., x_I(t)\Big)$ output vector $\overline{o}(t) = \Big(o_1(t), ..., o_K(t)\Big)$ is calculated as forward pass of signal. Hidden unit j calculates its net input $\widetilde{y}_j(t)$:

$$\widetilde{y}_j(t) = \sum_{i=1}^{I} v_{ji} x_i(t) + \sum_{i=1}^{J} m_{ji} y_i(t-1) \tag{3.1}$$

and produces output $y_j(t)$:

$$y_j(t) = f\Big(\widetilde{y}_j(t)\Big). \tag{3.2}$$

Output unit k calculates its net input $\widetilde{o}_k(t)$ and output value $o_k(t)$ as:

$$\widetilde{o}_k(t) = \sum_{j=1}^{J} w_{kj} y_j(t), \tag{3.3}$$

$$o_k(t) = f\Big(\widetilde{o}_k(t)\Big), \tag{3.4}$$

where v_{ji} is weight connecting hidden unit j with input unit i, m_{ji} is weight connecting hidden unit j with context unit i and w_{kj} is weight connecting output unit k with hidden unit j.

3.4.2 Backward pass

The difference between $\overline{d}(t) = \Big(d_1(t), ..., d_K(t)\Big)$ and $\overline{o} = \Big(o_1(t), ..., o_K(t)\Big)$ is expressed through error function

$$E = \frac{1}{2} \sum_{k=1}^{K} \Big(d_k(t) - o_k(t)\Big)^2. \tag{3.5}$$

The aim is to minimize the error E. Weights are updated by gradient descent method. Weight changes for output units are given by:

$$\Delta w_{kj} = -\alpha \frac{\partial E}{\partial w_{kj}} = \alpha f'(\widetilde{o}_k)\Big(d_k(t) - o_k(t)\Big) y_j, \tag{3.6}$$

Modifications of weights from input to hidden units are calculated:

$$\Delta v_{ji} = -\alpha \frac{\partial E}{\partial v_{ji}} = \alpha \sum_{k=1}^{K} \left[\Big(d_k(t) - o_k(t)\Big) f'\Big(\widetilde{o}_k(t)\Big) \sum_{h=1}^{J} w_{kh} \frac{\partial y_h(t)}{\partial v_{ji}} \right], \tag{3.7}$$

where

$$\frac{\partial y_h(t)}{\partial v_{ji}} = x_i(t)\delta_{hj}^{\text{kron}} + \sum_{l=1}^{J} m_{hl} \frac{\partial y_h(t-1)}{\partial v_{ji}}. \tag{3.8}$$

Modifications of weights from context to hidden units are calculated:

$$\Delta m_{ji} = -\alpha \frac{\partial E}{\partial m_{ji}} = \alpha \sum_{k=1}^{K} \left[\Big(d_k(t) - o_k(t)\Big) f'\Big(\widetilde{o}_k(t)\Big) \sum_{h=1}^{J} w_{kh} \frac{\partial y_h(t)}{\partial m_{ji}} \right], \tag{3.9}$$

where

$$\frac{\partial y_h(t)}{\partial m_{ji}} = y_i(t)\delta_{hj}^{\text{kron}} + \sum_{l=1}^{J} m_{hl} \frac{\partial y_h(t-1)}{\partial m_{ji}}. \tag{3.10}$$

3.4.3 Weight update

After weight changes are calculated, weights can be updated:

$$w_{kj} = w_{kj} + \Delta w_{kj}, \tag{3.11}$$

$$v_{ji} = w_{ji} + \Delta w_{ji}, \tag{3.12}$$

$$m_{ji} = m_{ji} + \Delta m_{ji}. \tag{3.13}$$

3.5 Other Approaches

Computational power of recurrent networks is very high [35], but common training methods based on gradient descent are inappropriate [3]. Error signal flowing backwards in time tend to vanish. Novel RNN architecture called long short-term memory (LSTM) was introduced [11, 14]. It uses recurrent connections of constant strength called constant error carrousel to guarantee constant error flow.

Chapter 4

RNNs and IFSs

State units of recurrent networks show considerable amount of structural differentiation [27] before learning. It means, that even in an untrained - randomly initialized recurrent neural network activities of recurrent neurons can be grouped in clusters. [19, 20]. This phenomenon can be explained by means of the Iterated Function System theory. IFS theory was originally developed by Barnsley (Fractals Everywhere 1988) as a method of describing the limit behavior of systems of transformations.

4.1 Iterated Function System

An iterated function system is a finite set of contraction transformations

$$\Omega = \{\omega_i | \omega_i : X \longrightarrow X, i \leq n\} \tag{4.1}$$

Limit behavior of a single transformation can be a single point in the space. Limit set over the union of transformations can be extremely complex with recursive structures. This limit behavior of composite mapping is called the IFS attractor. IFS address is defined for every point of an attractor. This address is the infinite sequence of transformations whose limit is a point on an attractor when starting point is the entire space. An example of an IFS is set of these three transformations over state space $X = [0, 1]^2$:

$$\begin{aligned} \omega_a(x, y) &= (0.5x + 0.5, 0.5y) \\ \omega_b(x, y) &= (0.5x, 0.5y + 0.5) \\ \omega_c(x, y) &= (0.5x, 0.5y) \end{aligned} \tag{4.2}$$

Limit behavior of a single transformation is a point in the corner of the state space. Limit behavior of the composition of all three transformations is a complex set representation known as the Sierpinski triangle (figure 4.1).

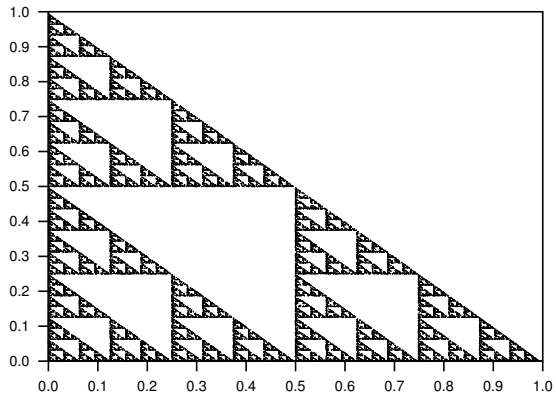

Figure 4.1: Sierpinski triange created by random iteration of transformations decribed by equation 4.2.

4.2 Architectural bias of RNNs

Behavior of recurrent networks in symbolic processing can be explained by IFS theory. For example the dynamics of SRN can be expressed by equation

$$S^{(t+1)} = f(MS^{(t)} + WI^{(t)}), \tag{4.3}$$

where f stands for activation function. M and W are matrices with recurrent and input weights respectively. Having finite input alphabet this dynamics can be rewritten to

$$S^{(t+1)} = f(V_i S^{(t)}). \tag{4.4}$$

For each input vector i from the input alphabet $A = \{a_1, \ldots, a_n\}$ corresponding weight matrix V_i can be found. Both approaches for the next state $S^{(t+1)}$ calculation are identical. When an input symbol appears, corresponding weight matrix V_i is applied to the current state $S^{(t)}$. In other words, IFS transformations are represented by weight matrices V_i and specific input vector selects, which transformation is applied to the current state. Recurrent networks initialized with small weights have similar behavior as IFSs. Interesting properties of IFS are present in RNNs prior to learning. This phenomenon can be called architectural bias [27].

4.3 Properties of IFSs

Let us return to the notion of the address. Next IFS is a modification of the previous one. The forth transformation was added whose attractor is the top right-hand corner of state space:

$$\begin{aligned} \omega_a(x,y) &= (0.5x + 0.5, 0.5y) \\ \omega_b(x,y) &= (0.5x, 0.5y + 0.5) \\ \omega_c(x,y) &= (0.5x, 0.5y) \\ \omega_d(x,y) &= (0.5x + 0.5, 0.5y + 0.5) \end{aligned} \tag{4.5}$$

Single transformation shrinks the entire image into one-forth sized copy of the original. A position of a point is mostly determined by the last performed transformation. This last performed transformation corresponds to the last symbol presented to the network. Next input symbol will release corresponding transformation and again, whole state space is mapped into a specific subspace. But its position within subspace is determined by the second last transformation. With an infinite precision, current point in the state space reflects all performed transformations, i.e. its position is determined by all input symbols. This notion is illustrated in figure 4.2. An IFS address of a point within an IFS attractor is the infinite sequence of indices of transformations, that map whole state space into the point. Hence, the indices of IFS address correspond to the input symbols. An IFS address is the inverse of string presented to the network, the first index of IFS address correspond to the last symbol presented to the network, the second index corresponds to the second last symbol, etc. A region of points that share common prefix of their IFS addresses gets smaller with the length of the prefix. Consequently, the longer the common suffix of input sequences, the nearer the corresponding points in the state space of RNN are.

Consider the starting point $x^* = (0.5, 0.5)$ and the input sequence $S = bcccca\ldots$ over the input alphabet $A = \{a, b, c, d\}$. Each symbol corresponds to one transformation of IFS given by set of transfromations $\Omega = \{\omega_a, \omega_b, \omega_c, \omega_d\}$ described by equation 4.5. The sequence S corresponds to concatenation of transformations $\ldots(\omega_a(\omega_c(\omega_c(\omega_c(\omega_b(x^*))))))$. Trajectory of points x_i is shown in figure 4.3.

The attractor of the composition of all four transformations is the whole state space. An approximation to the IFS attractor is easy to construct. Random iteration of transformations can produce attractor very rapidly. This spatial representation of points within the state space is also called a chaos game. Random iteration of transformations corresponds to the random sequence presented to the network. Resulting representation is shown in figure 4.4 (a). along with the spatial representation of Laser sequence. While random sequence produces uniformly covered state space, representation of Laser sequence shows some structure.

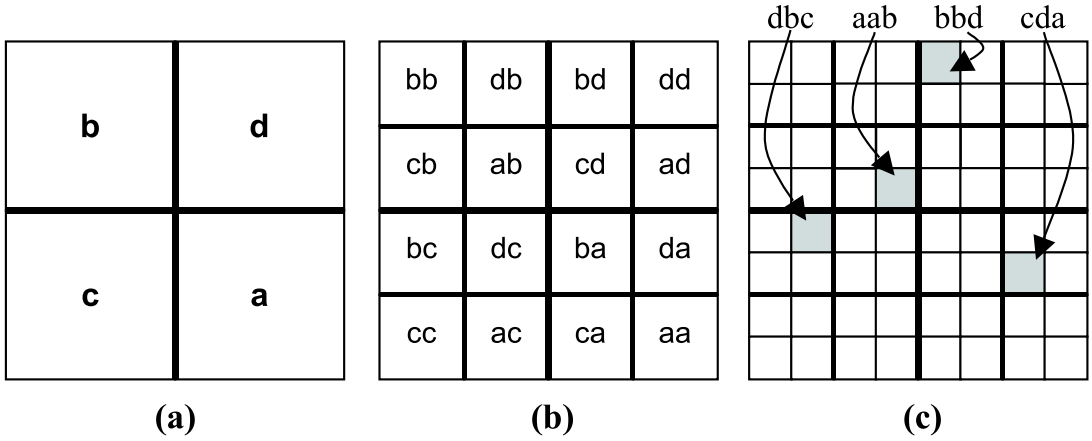

Figure 4.2: Regions of points with common IFS address prefixes.

Laser data set is the sequence of differences between successive activations of a real laser in chaotic regime [25]. The sequence was quantized to form symbolic sequence over four-symbol alphabet $A = \{a, b, c, d\}$. Each symbol corresponds to one of IFS transformations from $\Omega = \{\omega_a, \omega_b, \omega_c, \omega_d\}$. Symbolic stream of quantized Laser activations corresponds to concatenation of transformations specified by symbols. Resulting figure shows considerable amount of clustering.

Concatenation of random transformations represented by a random sequence over four symbol alphabet results in the state space regularly covered by points. It is not the case of Laser sequence. Similar subsequences correspond to points that are closer in the state space. The longer the common suffix, the nearer the points are in the state space. Frequent subsequences of longer length produce clusters.

This behavior has led to the idea described in [25]. Novel RNN architecture called IFSN was proposed.

4.4 RNNs with IFS dynamics

Interesting properties of IFS are used in special types of architecture called IFSN [25]. RNN state part is not trained, recurrent weights are fixed.

Simple and useful notation was established. Consider sequence $s = s_1 s_2 s_3 \ldots$ over finite alphabet alphabet $A = \{\omega_1, \omega_2, \ldots, \omega_{|A|}\}$ of $|A|$ symbols. N-block subsequence $u = u_1 u_2 \ldots u_n$

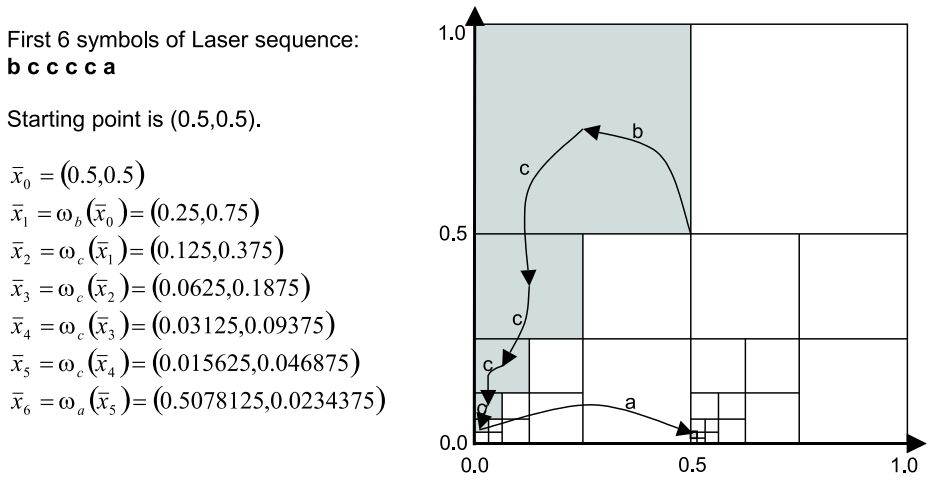

Figure 4.3: Detailed dynamics in state space for first 6 symbols of Laser sequence.

within sequence s is represented as a set of points $u(\overline{x}) = u_n(u_{n-1}(u_{n-2}(\ldots(u_1(\overline{x}))\ldots)))$ and $\overline{x} \in X$, where $X = [0,1]^N$ is a multi-dimensional hypercube with dimension $N = log(|A|)$. $u(X) = \{u(\overline{x})|\overline{x} \in X)\}$. Transformations $\omega_1, \omega_2 \ldots, \omega_{|A|}$ are

$$\omega_i = k\overline{x} + (1-k)\overline{t}_i, \tag{4.6}$$

what is the generalization of IFS described previously (equations 4.2 and 4.5). Parameter $k \in (0, 0.5]$ is a contraction coefficient and $\overline{t}_i$ are corners of hypercube X, $\overline{t}_i \neq \overline{t}_j$ for $i \neq j$. Given a sequence $s = s_1 s_2 \ldots$ its chaos game representation $CGR(s)$ is a sequence of points $CGR(s) = \{s_1^i(\overline{x}^*)\}_{n\geq 1}$ where $s_i^j = s_i s_{i+1} \ldots s_j$ and $\overline{x}^* = (0.5, \ldots, 0.5)$ is the center of state space. Hence, figures 4.4 (a) and (b) are examples of CGRs of random and Laser sequences.

Now consider two sequences $s = tv$ and $p = qv$ that share common suffix v of length $|v|$. End points of $CGR(s)$ amd $CGR(p)$ are $s(\overline{x}^*)$ and $t(\overline{x}^*)$. The difference between them is:

$$s(\overline{x}^*) - t(\overline{x}^*) = v(p(\overline{x}^*)) - v(q(\overline{x}^*)) = k^{|v|}(p(\overline{x}^*) - q(\overline{x}^*)) \tag{4.7}$$

This is a formal explanation of the claim intuitively explained in the section above. The longer the common suffix v of the sequences, the closer the end points are. Subspace $v(X)$ is an N-dimensional hypercube of side length $k^{|v|}$ [25]. The longer the sequence v, the smaller the region corresponding to v.

CGR represents statistical properties of the input sequence. Rigorous analysis can be found in [22], where direct correspondence between statistical characterization of symbolic sequences and multifractal characteristics was established.

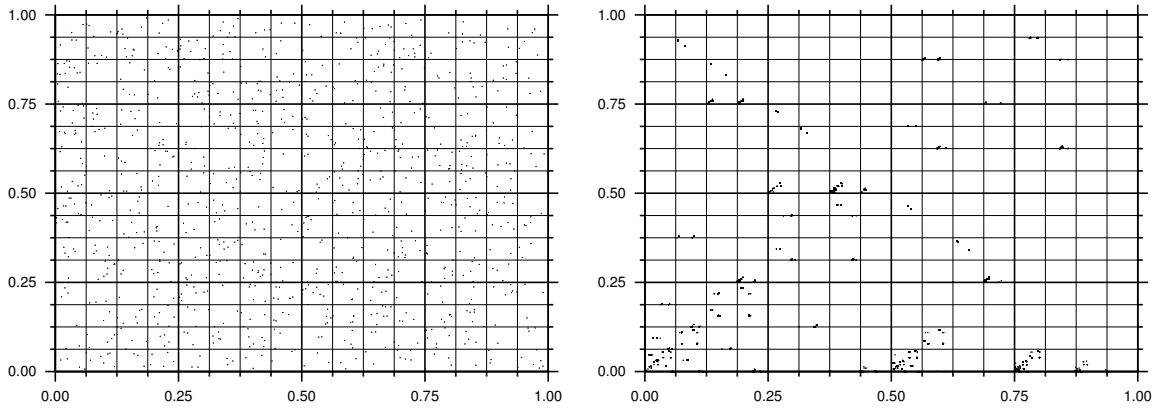

Figure 4.4: Chaos game representations of sequence created of random symbols and Laser sequence (by IFS represented by equation 4.5).

IFSN is a recurrent network whose recurrent part matches IFS described by equation 4.6. Learning process is fast, less problematic and dynamics are easy to understand. It was also introduced as an alternative to the problematic learning based on gradient descent [3]. In some cases IFSN are comparable with RNNs. IFSN architecture proposed in [25] is shown in figure 4.5 (a).

Other method used for symbolic sequence processing based on IFS dynamics was proposed. This approach is called Fractal prediction machine (FPM, in figure 4.5 (b)) [29] and was used for language modeling. Training sequence is presented to the FPM and activations of the state units are recorded. The Next vector quantization over recorded activations is performed to split data into classes. Each class is represented by the center vector. Points in the classes correspond to "similar" subsequences - subsequences with potentially long common suffix. For each class, the next symbol probabilities are computed. Classes are regarded as prediction contexts. Sliding over the training sequence one can calculate the probabilities of desired responses with respect to which cluster the current state belongs to. Testing consists of sliding over the test sequence and choosing response with the highest probability for the FPM current state class. Yet again, very good results were obtained when comparing FPMs and RNNs.

Formal description of other computing devices called dynamical automata was established in [36, 37]. They are also based on fractal-like representations in the state space. This method enables using fractal sets to organize infinite state computations in a bounded state space.

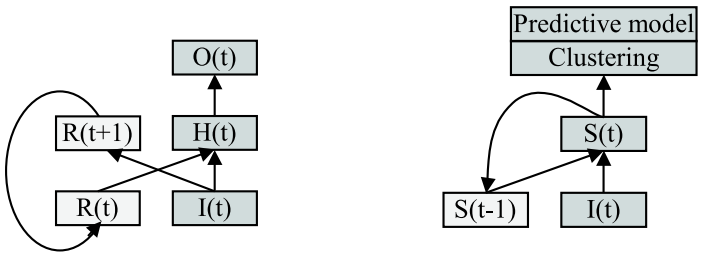

Figure 4.5: RNN architecture proposed for IFSN and fractal prediction machine (FPM).

Chapter 5

RNNs and Finite State Machines

Several researchers have explored behavior of RNNs for predicting successive elements of a sequence [7, 26]. When a network is trained with strings from a particular regular grammar, it can learn to become perfect finite state recognizer for the grammar. Let us take SRN as a typical representative of the RNN architectures. It uses hidden units' activations from previous step together with current input symbol to form hidden patterns and to predict the next symbol. In this way SRN seems to work as an finite-state automaton. It can be in one of its internal configurations called "state". State holds a contextual information, i.e. it reflects past events. The next input symbol together with the current state are transformed into the new state and the corresponding output symbol is released through the output layer. Inferring underlying regular grammar from a set of examples has been studied in many works, where various architectures and approaches have been used [7, 31, 39]. It was found, that SRN can learn to behave as a finite-state automaton and also can create automaton-like state representations within its context space (also called the state space). After successful learning, the internal states are localized in separate clusters corresponding to automaton states.

5.1 Finite State Machines (FSMs) and Regular Grammars

Regular languages are one part of Chomsky hierarchy of languages [15]. They are generated by regular grammars. A grammar is the 4-tuple $G = \{V_N, V_T, P, S\}$ where V_N and V_T are non-terminal and terminal symbols respectively, $S \in N$ is the starting symbol and P is a finite set of production rules in form of $\alpha \rightarrow \beta$ where $\alpha \in V^+$ and $\beta \in V^*$. All strings generated by grammar G form language $L(G)$.

Regular grammar G is grammar, where all production rules $\alpha \to \beta$ have form of $A \to \alpha$ or $A \to \alpha B$ where $A, B \in N$ and $\alpha \in T$. All strings generated by regular grammar create regular language $L(G)$. With each regular language a finite-state automaton is associated. This automaton M is the acceptor of language L. M accepts only strings of regular language L , which is denoted by $L(M) = L(G)$.

Deterministic finite-state automaton (FSA) is the 5-tuple $M = \{X, Q, \delta, q_0, F\}$ where X is the finite input alphabet, Q is a finite set of automaton states, map $\delta : Q \times X \to Q$ defines state transitions in M, $q_0 \in Q$ is the initial state and $F \in Q$ is a set of accepting states. String x is accepted by M if an accepting state has been reached after string x was read by automaton. Finite-state automaton can be also considered as a generator of regular language $L(M)$. FSAs are usually represented as directed graphs called state transition diagrams, where edges correspond to state transitions. One variant of FSA will be called the stochastic automaton. Each transition (edge of graph) is evaluated by the probability of the state transition. Hence, stochastic automaton describes also some statistical properties of generated language. Initial mealy automaton is the 6-tuple $M = \{X, Y, Q, \delta, \lambda, q_0, \}$, where Y is the finite output alphabet and map $\lambda : Q \times X \to Y$ defines output function of M. Output symbol is released when state transition occurs.

5.2 Grammatical Inference

Grammatical inference is the problem of inferring grammar from samples of strings of an unknown regular language. Consider regular grammar:

$$\begin{gathered} G = \{\{A, B, C, A\}, \{s, a, b, c\}, P, C\} \\ P = \{A \to sA, B \to sB, C \to sC, A \to bB, B \to cC, C \to aA\}. \end{gathered} \tag{5.1}$$

An example of string generated by G is $x = absssc sssasbsscsabcss \ldots$ SRN can be trained for predicting the next symbol of string generated by the grammar G. Number of input and output unit is given by the cardinality of the set of terminal symbols (thus 4 input and output units were used), number of hidden units was set to 2. After each input symbol the error between prediction computed by SRN and the actual successor was minimized by RTRL. Figure 5.1 (a) shows the state space of an untrained SRN. 1000 symbols were presented to the network. Four clusters around the center of state space correspond to the four symbols of an input alphabet. Why this structural differentiation can be present in an untrained networks was intuitively and also formally described in previous chapter. Figure 5.1 (b) shows a state space of a SRN trained for prediction task. Also 1000 symbols were presented to the network.

Points are grouped within clusters in the corners of the state space. Extraction of finite state automaton from the trained network can be performed. It would lead to the automaton represented in figure 5.2 (a). This automaton fully describes the grammar G.

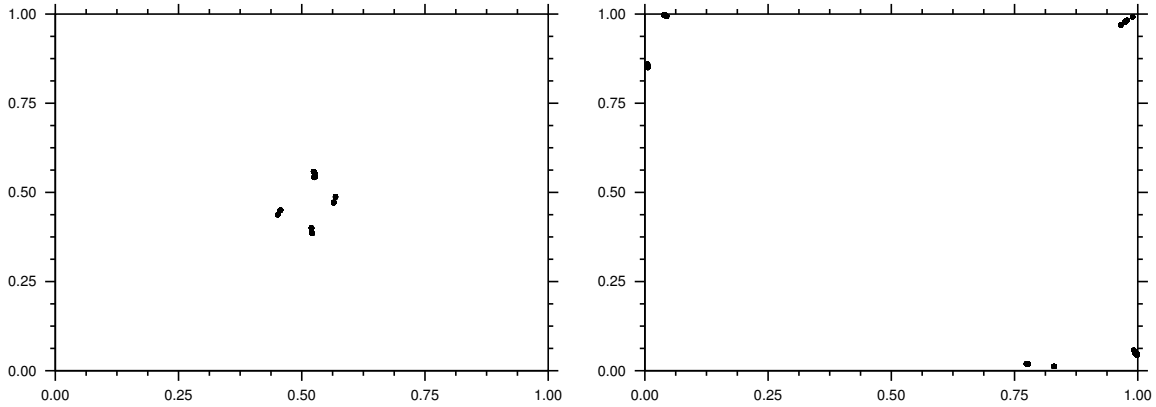

Figure 5.1: State space of an untrained and trained SRN.

This was one example of grammatical inference. Simple grammar was recognized only by presenting the SRN with generated string. Clusters corresponding to the automaton states tend to move to the corners of the state space hypercube. SRN with only two hidden units can imitate this three state automaton thus giving us the possibility to see its state space as a 2D plot. Similar experiment with Reber grammar (figure 5.2 (b)) was described in [7].

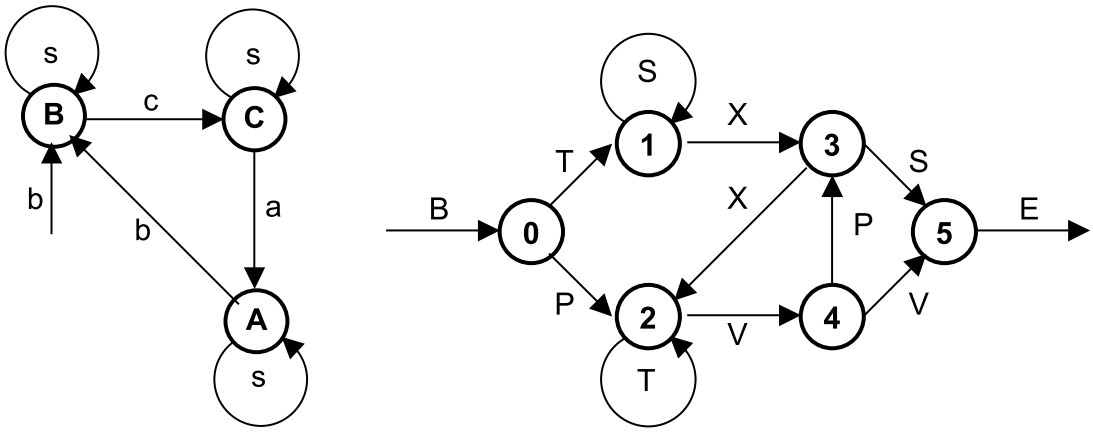

Figure 5.2: Automaton associated with grammar described by equation 5.1 and well known Reber grammar automaton.

Much effort has been devoted to the clustering and minimization technique to extract finite state automaton that approximates the behavior from an trained recurrent network. Relationship between the internal state representation and the stable behavior of RNN as an automaton was examined [1, 8], complexity requirements of SRN implementing finite state machines in [16]. The simplest techniques such as K-means clustering algorithm based on expectation-minimization (EM) approach or hierarchical clustering proposed in [8] can be successful as shown in the one of our experiments. More complex algorithms for rules extraction were introduced in [5, 9, 28].

Chapter 6

Beyond Finite State Representation

Previous chapter described recurrent neural networks working as finite-state machines in order to create regular language representation. On the other hand, processing strings of context-free languages requires a stack or a counter memory device. Next section shows, that RNNs can learn to imitate push-down automatons.

6.1 RNN as Counters

One dimensional up-down counter for two symbols can be created using two maps:

$$\begin{aligned} \omega_a(x) &= (0.5x + 0.5) \\ \omega_b(x) &= (2x - 1) \end{aligned} \tag{6.1}$$

where x is the state variable and a and b are symbols for counting up and down respectively. A sequence of inputs *aaabbb* will correspond to states: $0.75, 0.875, 0.9375, 0.875, 0.75, 0.5$ where $x_0 = 0.5$ was the starting state. Counting up gradually increases the state variable x towards the corner of state space $x_a = 1$. With an infinite precision for state representation of counting device every finite sequence of a symbols is uniquely described by one point in the state space. Counting down will gradually decrease state x in order to return to the original state $x_0 = 0.5$, what indicates that the same number of b symbols was presented to the device.

This $a^n b^n$ task is the simplest context-free language requiring counting-like representation. After the sequence of a symbols, the same number of b symbols follows. Grammar generating this language could be

$$\begin{aligned} G &= \{\{X\}, \{a, b\}, P, X\} \\ P &= \{X \to ab, X \to aXb\} \end{aligned} \tag{6.2}$$

Simple Recurrent Networks were trained for next symbol prediction in $a^n b^n$ strings with n varying from 1 to 11 [31, 39]. Some networks that successfully learnt the task were able to generalize prediction up to $n = 16$. An interesting oscillation dynamics of SRN can be represented by an idealized solution:

$$X_t = f\left(\begin{bmatrix} 0.5 & 0 \\ 2.0 & 2.0 \end{bmatrix} X_{t-1} + \begin{bmatrix} 0.5 & -5 \\ -5 & -1 \end{bmatrix} I_t\right), \tag{6.3}$$

where f stands for the piecewise-linear activation, X_t is the state vector in time step t and I_t is the input vector, $a = [1, 0]$ and $b = [0, 1]$. This solution can be rewritten to:

$$\begin{aligned} \omega_a(x_1, x_2) &= f(0.5x + 0.5, 0) \\ \omega_b(x_1, x_2) &= f(0, 2x_1 + 2x_2 - 1.0) \end{aligned} \tag{6.4}$$

Starting point is $\overline{x}_{start} = (0, 0)$. A sequence of inputs *aaabbb* will correspond to states: $(0.5, 0), (0.75, 0), (0.875, 0), (0, 0.75), (0, 0.5), (0, 0)$. SRN works as an up-down counter. First dimension variable x_i is used for counting up the number of a symbol. The state is approaching towards a fixed point of ω_a, what is an attractive point $\omega_a^{\infty}(\overline{x}) = (1, 0)$. This attractive fixed point lies on stable manifold of saddle fixed point of ω_b transformation. The expansion and contraction rates are inversely proportional. Figure 6.1 represents dynamics of trained SRN. Stable and unstable manifolds of ω_b transformation's saddle point are denoted by dashed lines.

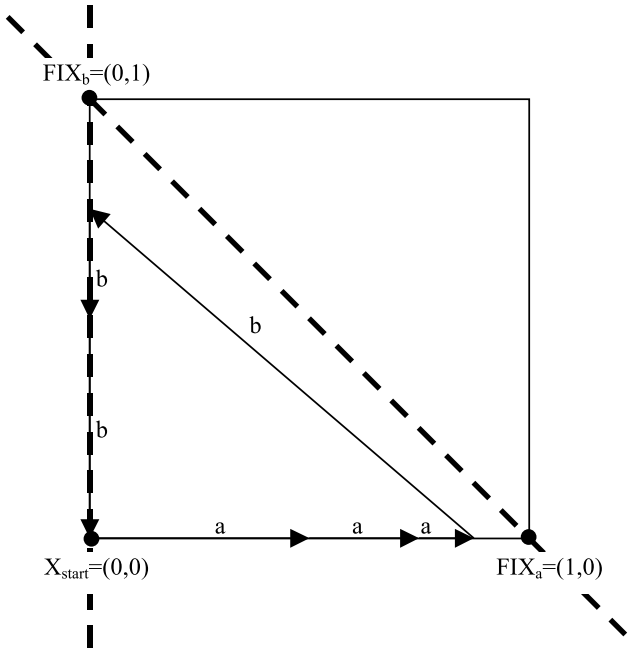

Figure 6.1: Idealized representation of dynamics of trained SRN.

How the SRN can learn to process more complex context-free and context sensitive languages was studied in [30]. It is important to note that after extensive training process only a portion of trained SRNs has developed good internal representation of these languages.

6.2 Chaotic RNN behavior

Some experiments show, that recurrent neural network can learn to acquire chaotic behavior. In [38], SRN without an input layer was trained as to reconstruct the target process represented by stochastic finite state automaton. The results revealed the capability of RNN to evolve towards chaos in order to mimic a target stochastic process. Internal RNN dynamics proceeded from fixed point through limit cycling to chaos.

Chapter 7

Our Preliminary Research

7.1 Evolution of clusters in state space of SRN trained by RTRL

7.1.1 Introduction

Next experiment investigates the evolution of performance of finite-context predictive models during learning. SRN was trained by Real Time Recurrent Learning algorithm for next symbol prediction of symbols in strings randomly generated according to the Reber grammar (figure 5.2) as described in [7]. Then predictive models were built upon the recurrent activations of the 2nd-order version of Elman Simple Recurrent Network.

7.1.2 Experiment

The predictive models based on internal state vectors are constructed as follows. After each training epoch, all synaptic weights are fixed and stored. Next, sliding through the training sequence, for each symbol in the training data set, recurrent activations are recorded. Vector quantization for given number of centers is performed over recorded activations. In predictive models, the quantization centers are identified with predictive states. To calculate the state-conditional probabilities, we associate with each center counters, one for each symbol from the input alphabet. Sliding through the recurrent activations' sequence, the closest center is found for the current activation vector. Then next symbol is identified. For the closest center the counter associated with the symbol is raised by one. After seeing the whole training sequence,

for each quantization center (prediction state) the conditional next-symbol probabilities are calculated by normalization of counters.

On the test sequence the next-symbol probabilities are determined as follows: for time step given the network state the actual symbol drives the network to a new recurrent activation vector. The closest quantization center is found and the next-symbol probabilities are considered.

These predictive models based on activations of RNN units are also called neural prediction machines (NPMs).

Predictive model's performance is evaluated by means of the normalized negative log-likelihood NNL on the whole test sequence:

$$\mathrm{NNL} = -\frac{\sum_i log_{|A|} P_t}{n}, \tag{7.1}$$

where the base of the logarithm is the number of symbols $|A|$ in the input/output alphabet A. P_t is the next-symbol probability chosen with respect of the closest quantization center at time step i and the correct symbol (output symbol that should be predicted). The higher the next correct-symbol probabilities the smaller is NNL, with NNL= 0 corresponding to the 100% correct next-symbol prediction.

The metric in vector quantization and nearest-center detection is Euclidean. Hierarchical vector quantization inspired by [8] was used. The first activation vector becomes the first quantization center. Given a certain number of centers, maximal cluster radius is iteratively found. For every internal state vector in turn, we find the closest center. If their distance is less than cluster radius, the vector belongs to the center. Otherwise, this vector becomes a new center. Evolution of performance of predictive models built upon the activation of the recurrent layer during training as a function of the amount of training and the number of quantization centers in the recurrent state space is shown in figure 7.1.

We trained the networks on randomly generated strings. The activations of recurrent neurons were reset to the same small random values at the beginning of each string. Before training and after each 10000 symbols we evaluated the quality of the next-symbol prediction by means of NNL. For the NNL calculation we used the test set of the length 20000 newly generated symbols. The input and output had the dimension of 6 with the one-hot encoding of symbols. "E" is equally coded as "B". The values of parameters were set to: the number of (recurrent) neurons to 6 and 8, learning rate and momentum to 0.03, and unipolar '0-1' sigmoid activation parameter (slope) to 1.0. The initial (reset) activations of recurrent neurons and initial weights were randomly generated from a uniform distribution over [-0.5, 0.5], for both RNNs.

7.1.3 Results

NNL results for the next-symbol prediction of the RNN for 6 and 8 recurrent neurons are shown in figure 7.1. NNL at time 0 expresses the prediction performance of an untrained network. Length of training set means the total number of symbols over all training strings. Center count means the number of quantization centers in the hierarchical vector quantization. The lowest NNL matches the theoretically calculated NNL for the occurence of symbols in Reber strings, e.g. 0.331.

RNN behaves as a kind of nonlinear IFS consisting of a sequence of transformations that map the state space represented by activations of recurrent neurons into separate subspaces of the state space. This phenomenon is called architectural bias and was explained in details in previous chapter. Each next state of an RNN, represented by the recurrent activations, is mostly determined by the last performed transformation, e.g. by the last presented symbol (input). Within the subspace belonging to the last transformation, the network state is determined by the last previous transformation, and thus by the last previous symbol (input). In this way, the RNN "theoretically" codes an infinite time window to the past, e.g. its current state uniquelly represents the history of inputs. The more distant the input is in the past the less it determines the current RNN state. It turned out, that two subsequences with the common suffix will correspond to the two states that are close to each other in the state space. The longer the common suffix the smaller the distance between the corresponding states in the state space.

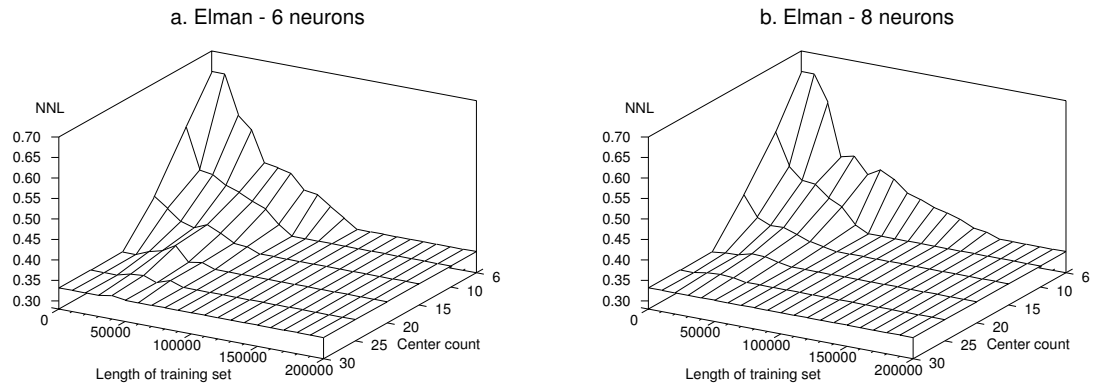

Figure 7.1: NNL results for the next-symbol prediction of the RNN for 6 and 8 recurrent neurons.

All the Reber states are uniquely determined by each pair of symbols in the Reber string. There are 20 of these pairs including the "SE" and "VE" pairs. Thus, even in the untrained

RNN, the 20 quantization centers obtained by means of the hierarchical clusterization described above, correspond to the RNN states determined by the last 2 symbols. It means that there is a unique mapping between the automaton states and the RNN states and the prediction model gives the minimal possible NNL, in this case 0.331 (Figure fig:elmanresults). In an untrained case, the individual states (recurrent activities) evoked by the last two symbols are well separated in the state space. Given more than 20 (e.g. 25 or 30) quantization centers, these recurrent activities can become split, due to the third last input. Therefore, the quality of the next-symbol prediction does not get worse. The RNN is trained by RTRL, thus the weights are updated after each symbol in turn. The errors that backpropagate from the output layer cause the weights to modify in such a way that the recurrent activities which ought to belong to the same automaton state get closer to each other step by step. They get closer, thus reflecting the probability that the continuation of the sequence will be the same. This movement may cause a temporary disturbance of the coding based on the last pair of symbols as can be observed for the large number of quantization centers. After the training, the network state reflects not only the history of inputs but, which is perhaps more important, also the probability of a certain continuation. Each network state belongs to a certain automaton state. It can be shown that the optimal prediction model can be created with only 6 clusters in the state space (see also [7]).

7.2 Evolution of clusters in state space of BCM RNN

7.2.1 Introduction

In previously described experiment we were interested in evolution of performance of finite-context predictive models built upon recurrent part of SRN. An interesting alternative to supervised RTRL training is Bienenstock, Cooper and Munro (BCM) [4, 17] learning rule adopted for RNNs [2]. BCM theory is with high correspondence with biological observations [34]. Networks based on BCM theory were also successfully applied to real-life problems [2, 18].

Recent results of experiments with the chaotic time series are promising [23, 24]. In our experiment RNN trained by the BCM rule processed strings from simple regular grammar described by the Reber automaton (figure 5.2)[21].

7.2.2 Experiment

Prediction models were created and evaluated in the same manner as in previous experiment in order to be able to compare the results. The same quantization algorithm was used.

Recurrent BCM networks were trained and prediction models were created on the same data sets as described in previous experiment. The values of parameters were set to: the number of (recurrent) neurons to 8 and 12, learning rate to 0.001, and unipolar '0-1' sigmoid activation parameter (slope) of 0.4 was used.

7.2.3 Results

The BCM RNN behaves also like a kind of IFS, albeit different from the previous one because of the lateral inhibition. In this case, we can also obtain the optimal prediction with a large number of quantization centers (e.g. 25) for the predictive model built upon the untrained network (figure 7.2). Then the weights are modified after each presentation of input according to the BCM rules for the recurrent network [2]. Recurrent neurons become sensitive to particular statistical characteristics of the input set and they become selective only to individual last symbols. Thus we are not able to observe the improvement of prediction for the number of quantization centers smaller than 20.

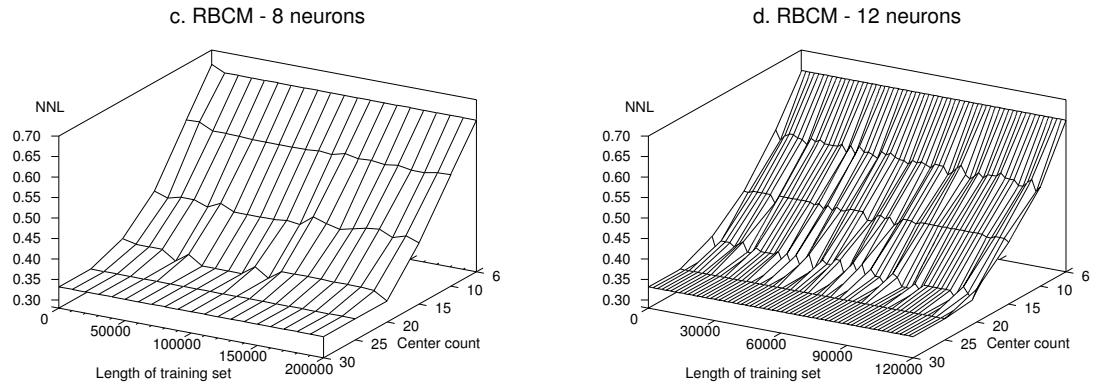

Figure 7.2: NNL results for the next-symbol prediction of the BCM RNN for 8 and 12 recurrent neurons.

7.3 Processing Language Structures by SRN

7.3.1 Introduction

Various problems related to different aspects of human or animal behavior are often modeled by neural networks. At the firs glance, the aim is not to obtain a connectionist model with "perfect" properties as in technically oriented domains. Researchers are seeking for models with high correspondence to real life problems. In cognitive science community RNN are

often used for language processing. Recently, connectionist networks were used for processing complex recursive structures represented by recursive languages directly inspired by Chomsky [6]. Levels of embedding of recursive structures that RNN was able to process was in correspondence with human ability to process recursive structures. Our aim was to point out the existence of the architectural bias in RNN and to show its importance by comparing results of NPM created from untrained and trained networks.

7.3.2 Experiment

We used three data sets described by [6]. Three artificial languages representing different types of recursion were created. They were composed of symbols representing four grammatical categories: singular nouns, singular verbs, plural nouns and plural verbs. Each language used in [6] involves one of three complex recursions taken from Chomsky, interleaved with right-branching recursions (RBR). The latter is generated by a simple iterative process to obtain constructions like: $P_N P_V S_N S_V$, where P stands for plural, S for singular, N for noun and V for verb category. The three complex recursions are:

1. Counting recursion (CR): $\{\}, NV, NNVV, NNNVVV, ...,$ while ignoring singulars and plurals.
2. Center-embedding recursion (CER): $\{\}, ..., S_N P_N P_V S_V, P_N S_N S_V P_V, ...$ Example: "the boy girls like runs".
3. Identity (cross-dependency) recursion (IR): $\{\}, ..., S_N P_N S_V P_V, P_N S_N P_V S_V, ...$ Example: "the boy girls runs like".

Thus, our three benchmark recursive languages were: CRandRBR, CERandRBR, IRandRBR. Each language had 16 word vocabulary with 4 words from each category, i.e. 4 singular nouns, 4 singular verbs, 4 plural nouns and 4 plural verbs. The RNN had 17 input and output units, where each unit represented one word or the end of sentence mark. There were 10 hidden and 10 recurrent neurons. The networks were trained in 10 training runs starting from different random weight initializations (from $[-0.5, 0.5]$). The training set of each language consisted of 5000 sentences and the test set of 500 novel sentences. One half of each set was comprised of RBR constructions and another half of appropriate complex recursions. Depths of embedding ranged from 0 to 3, with the following distributions: depth 0 – 15 %, depth 1 – 27.5 %, depth 2 – 7 %, depth 3 – 0.5 % (together 50 %). The mean sentence length was approximately 4.7 words.

7.3.3 Results

In figure 7.3 we show the mean (across 10 training runs) normalized (per symbol) negative log likelihoods (NNL) achieved on the test set by FPMs, RNNs and the corresponding NPMs. In the 2-D plots, we also show the corresponding standard deviations. Standard deviations in the 3-D plots are not shown, but generally are less than 5 % of the mean value.

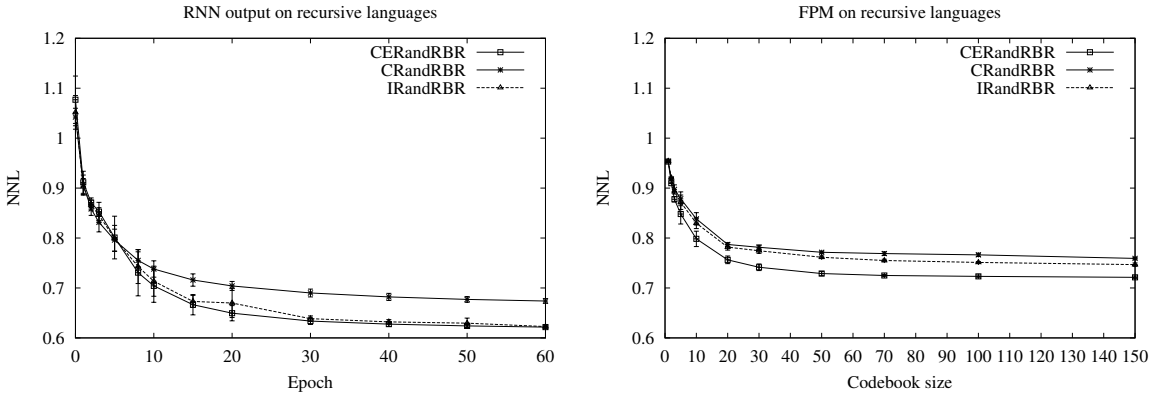

Figure 7.3: NNLs achieved on Christiansen and Chater recursion data sets by RNNs and FPMs.

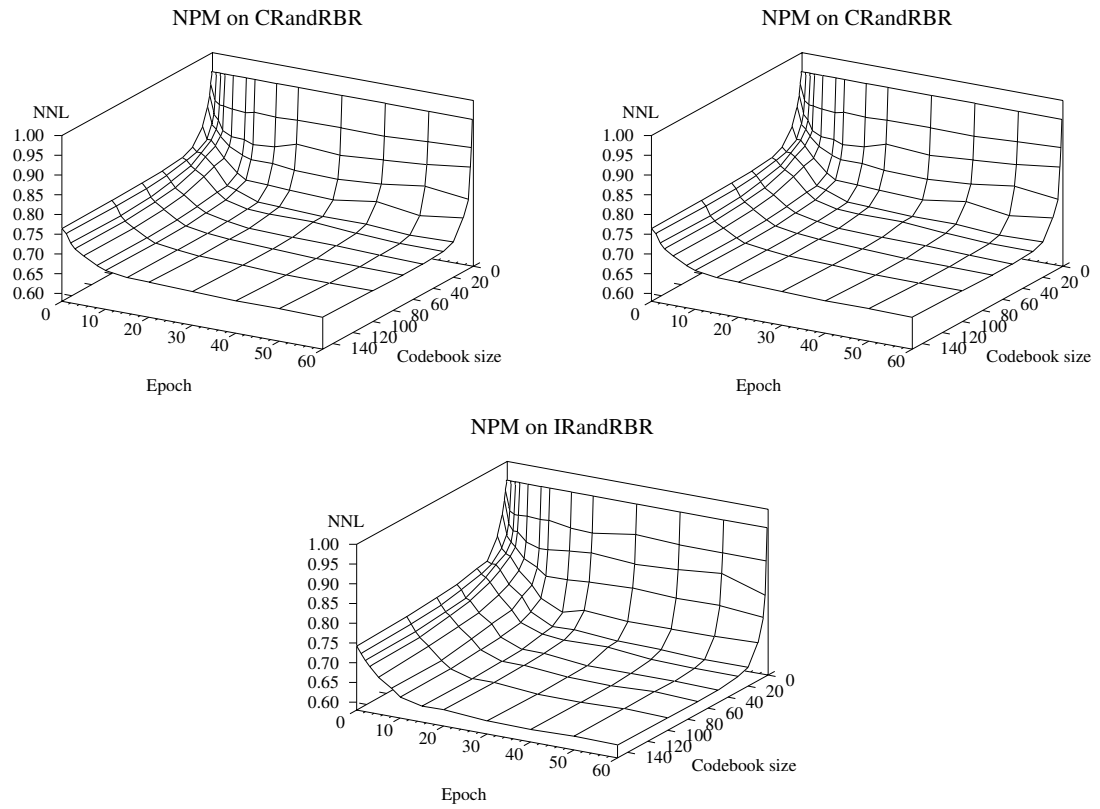

Figure 7.4: NNLs achieved on Christiansen and Chater recursion data sets by NPMs.

The importance of architectural bias is clearly visible. While NNL based on outputs of SRN (figure 7.3 (a)) followed significant improvement during training, NNL for NPM for given number of clusters remain almost still during training (figure 7.4) despite small improvements in first 10 epochs of training. FPM or NPM are closely related to variable memory length Markov models (VLMM) [32]. While Markov models use fixed memory depth for all prediction contexts, VLMMs use flexible memory length depending on context's importance. Improvements of NPMs during first 10 epoch of training will be object of future research.

Chapter 8

Future Work

In our preliminary experiments we observed behavior of recurrent network trained by the BCM algorithm on regular language strings. BCM RNN did not create automaton-like state representation of regular grammar. Instead BCM neurons became selective to input symbols and attractive fixed points dynamics present also in an untrained RNN was dominant. On the other hand, BCM RNNs trained on other symbolic sequences seemed to build more interesting internal dynamics valuable for NPM creation [23, 24]. These results should be reconsidered and experiments reevaluated with respect to architectural bias of RNN. If more than attractive fixed point dynamics is present, its usefulness should be experimentally shown and explained if possible. Extraction and understanding of rules from trained neural networks is not a trivial task.

RNNs can acquire variety of dynamical behaviors. RNNs can successfully solve simple tasks by creating automaton-like finite state representation or infinite state counting devices. But training RNNs on complex sequences seems to be difficult. Our preliminary experiments revealed the importance of architectural bias in RNN. But NPMs on trained RNNs seemed to organize the state space "a little bit better" than FPMs or NPMs on untrained RNNs. Where did this even small improvement came from? Yet again, deeper analysis of internal dynamics should be performed.

Importance of the architectural bias is omnipresent throughout this work. Fractal-like behavior of attractive fixed point dynamics can be intuitively understood. New RNN architectures and devices based on this IFS dynamics (fractal prediction machine, neural prediction machine [22, 25, 29]) were already proposed and it seems that they may offer valuable insight into some problems studied by connectionist cognitive science community. Spatial representation of device's state space reflects statistical properties of underlying input sequence. Points

in the state space also represent history of inputs. Both these properties encourage a practical application of devices based on fractal representation. Real life technical problems surely require computational inexpensive means of representing the past. Similar approaches already exist (VLMM [32]), maybe others can be found, studied and potential improvements can be suggested. We also want to explore this direction.

Chapter 9

Conclusion

Feed-forward neural networks are successfully used in many non-temporal tasks. Spatio-temporal tasks require some sort of memory for holding temporal context. Recurrent networks were introduced to handle such types of problems. Recurrent connections are used to give a context information for actual input processing.

Recurrent neural networks can have different behaviors from dynamical point of view. This work described some types of RNN behavior. Dynamics of RNNs initialized with small weights is based on attractive fixed points of transformations corresponding to different inputs. Properties of spatial representations of RNN state space were intuitively and also formally explained. RNNs can acquire regular languages through training process. Then RNNs behave as finite state machines, i.e. the internal states of RNN can be identified with the states of finite state automaton describing the regular grammar. Finally RNNs can also acquire context free languages and context sensitive languages. They can learn to construct mechanisms such as counting in order to mimic behavior of lower level grammars.

Our preliminary experiments were described. They are oriented towards architectural bias of RNNs and interesting unsupervised variant of learning algorithm based on Bienenstock, Cooper and Munro [2, 4].

Last chapter tries to offer potential problems for further work, namely several directions of investigation of evolution of dynamics of RNNs during training when trained on real life tasks.

Bibliography

[1] K. Arai and R. Nakano. Stable behavior in a recurrent network for a finite state machine. *Neural Networks*, 13:667–680, 2000.

[2] C. M. Bachman, S.A. Musman, D. Luong, and A. Shultz. Unsupervised BCM projection pursuit algorithms for classification of simulated radar presentations. *Neural Networks*, 7:709–728, 1994.

[3] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[4] E. L. Bienenstock, L. N. Cooper, and P. W. Munro. Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. *Journal of Neuroscience*, 2(1):32–48, 1982.

[5] A. D. Blair and J. B. Pollack. Analysis of dynamical recognizers. *Neural Computation*, 9(5):1127–1142, 1997.

[6] M.H. Christiansen and N. Chater. Toward a connectionist model of recursion in human linguistic performance. *Cognitive Science*, 23:417–437, 1999.

[7] A. Cleeremans, D. Servan-Schreiber, and J. L. McClelland. Finite state automata and simple recurrent networks. *Neural Computation*, 1(3):372–381, 1989.

[8] S. Das and R. Das. Induction of discrete state-machine by stabilizing a continuous recurrent network using clustering. *Computer Science and Informatics*, 21(2):35–40, 1991.

[9] S. Das and M. Mozer. Dynamic on-line clustering and state extraction: An approach to symbolic learning. *Neural Networks*, 11(1):53–64, 1998.

[10] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[11] F. A. Gers, J. Schmidhuber, and F. A. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

[12] S. Haykin. *Neural Networks-A Comprehensive Foundation.* Prentice-Hall, Upper Sadle River, New Jersey 07458, 1994.

[13] J. Hertz, A. Krough, and R. G. Palmer. *Introduction to the theory of neural computation.* Addison-Wesley, 1991.

[14] J. Hochreiter and J. Schmidhuber. Long short term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[15] J. E. Hopcroft and J. D. Ulman. *Formal languages and their relations to automata.* Addison-Wesley, 1969.

[16] B. G. Horne and D. R. Hush. Bounds on the complexity of recurrent neural network implementations of finite state machines. *Neural Networks*, 9(2):243–252, 1996.

[17] N. Intrator and L.N. Cooper. Objective function formulation of the BCM theory of visual cortical plasticity: statistical connections, stability conditions. *Neural Networks*, 5:3–17, 1992.

[18] N. Intrator and J. I. Gold. Three-dimensional object recognition of gray level images: The usefulness of distinguishing features. *Neural Computation*, 5:61–74, 1993.

[19] J.F. Kolen. The origin of clusters in recurrent neural network state space. In *Proceedings from the Sixteenth Annual Conference of the Cognitive Science Society*, pages 508–513. Hillsdale, NJ: Lawrence Erlbaum Associates, 1994.

[20] J.F. Kolen. Recurrent networks: state machines or iterated function systems? In M. C. Mozer, P. Smolensky, D. S. Touretzky, J. L. Elman, and A.S. Weigend, editors, *Proceedings of the 1993 Connectionist Models Summer School*, pages 203–210. Erlbaum Associates, Hillsdale, NJ, 1994.

[21] M. Čerňanský and L. Beňušková. Finite-state reber automaton and the recurrent neural networks trained in supervised and unsupervised manner. In H. Bischof Lecture Notes in Computer Science 2130. G. Dorffner and K. Hornik (Eds), editors, *Artificial Neural Networks - ICANN'2001*, pages 737–742. Springer-Verlag, 1998.

[22] P. Tiňo. Spatial representation of symbolic sequences through iterative function system. *IEEE Transactions on Systems, Man, and Cybernetics Part A: Systems and Humans*, 29(4):386–392, 1999.

[23] P. Tiňo, M. Stančík, and L. Beňušková. Building predictive models on complex symbolic sequences via a first-order recurrent BCM network with lateral inhibition. In P. Sinčak and

J. Vaščak, editors, *Quo Vadis Computational Intelligence? New Trends and Approaches in Computational Intelligence*, pages 42–50. Physica-Verlag, Heidelberg, 2000.

[24] P. Tiňo, M. Stančík, and L. Beňušková. Building predictive models on complex symbolic sequences with a second-order recurrent BCM network with lateral inhibition. In *Proc. Int. Joint Conf. Neural Networks*, pages 265–270, 2000.

[25] P. Tiňo and G. Dorffner. Recurrent neural networks with iterated function systems dynamics. In *International ICSC/IFAC Symposium on Neural Computation*, 1998.

[26] P. Tiňo and J. Šajda. Laearning and extracting initial mealy automata with a modular neural network model. *Neural Computation*, 7(4):822–844, 1995.

[27] P. Tiňo, M. Čerňanský, and L. Beňušková. Markovian architectural bias of recurrent neural networks. Submitted to 2nd Euro-International Symposium on Computational Intelligence, June 16 - 19, 2002, Košice, Slovakia., 2002.

[28] C. W. Omlin and C. L. Giles. Extraction of rules from discrete-time recurrent neural networks. *Neural Networks*, 9(1):41–51, 1996.

[29] S. Parfitt, P. Tiňo, and G. Dorffner. Graded gramaticality in prediction fractal machines. In *Advances in Neural Information Processing Systems 12*, pages 52–58, 2000.

[30] P. Rodriguez. Simple recurrent networks learn contex-free and contex-sensitive languages by counting. *Neural Computation*, 13:2093–2118, 2001.

[31] P. Rodriguez, J. Wiles, and J. L. Elman. A recurrent neural network that learns to count. *Connection Science*, 11:5–40, 1999.

[32] D. Ron, Y. Singer, and N. Tishby. The power of amnesia. *Machine Learning*, 25, 1996.

[33] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. I: Foundations*, pages 318–362. Bradford Books/MIT Press, Cambridge, MA., 1986.

[34] H. Shouval, N. Intrator, and L. Cooper. BCM network develops orientation selectivity and ocular dominance from natural scenes environment. *Neural Computation*, 37(23):3339–3342, 1997.

[35] H. T. Siegelmann and E. D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991.

[36] W. Tabor. Dynamical automata. Technical Report TR98-1694, Department of Psychology, Uris Hall, Cornell University, Itaca, NY14853, July, 20 1998.

[37] W. Tabor. Fractal encoding of context free grammars in connectionist networks. *Expert Systems: The International Journal of Knowledge Engineering and Neural Networks*, 17(1):41–56, 2000.

[38] J. Tani and N. Fukurama. Embedding a grammatical description in deterministic chaos: an experiment in recurrent neural learning. *Biological Cybernetics*, 72:365–370, 1995.

[39] J. Wiles and J. Elman. Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pages 482 – 487, 1995.

[40] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, 1989.