Neural Networks: Automata and Formal Models of Computation

Mikel L. Forcada
Universitat d'Alacant,
Dept. Llenguatges i Sistemes Informàtics,
E-03071 Alacant (Spain).

January 21, 2002

Contents

Preface

Status of this document

The document you are reading was originally conceived (in 1995) as a reprint collection; the text was intended to serve as an introduction to what I believed to be some of the important papers in the field. A number of personal and editorial circumstances have prevented me from finishing this work; the LaTeX files have been frozen in my hard disk since May 2000, and, expectedly, reflect the perception of the field I had at that time. Therefore, this document can only be seen as some kind of draft, besides being somewhat outdated. Moreover, you will also find passages which may sound odd without the reprinted papers: perhaps you can make up your own reprint collection by binding the papers together with these pages (when I have found a *featured* paper to be freely available on the web, a link to a locally-cached online version of the paper is given for convenience; authors of missing papers are kindly requested to contribute them if they are allowed to do so: `mailto:mlf@dlsi.ua.es`).

Some of the material has already been published as journal papers or book chapters (Carrasco et al., 2000; Forcada and Carrasco, 2001), but the rest was not available in any way. A couple of months ago it occurred to me that this document might be useful to other people working in related fields, and, since I am basically switching to a different field of computer science, I have decided to hang it up on the web for whatever you wish to do with it. In fact, if you are really interested in using some of this material for your own publications or course notes, just contact me and I will be glad to pass the LaTeX sources on: you are just kindly asked to mention me and this website in the derivative work you produce.

Motivation[1]

The fields of artificial neural networks and theoretical computer science have been linked since their inception. As Perrin (1990) says:

> The first historical reference to finite automata is a paper of S.C. Kleene of 1956 in which the basic theorem, now known as Kleene's theorem, is

[1] A slightly edited version of the original preface

> already proved [...]. Kleene's paper was actually a mathematical reworking of the ideas of two reseachers from the MIT, W. McCulloch and W. Pitts, [...] who had presented as early as 1943 a logical model for the behaviour of nervous systems that turned out to be the model of a finite-state machine [...].

In the paper by McCulloch and Pitts, the first model of a finite automaton was indeed a network of idealized neuron-like elements.

The objective of this document is to comment a series of papers showing the intellectual strands from this early work to the current theoretical developments and potential applications in various areas. Neural networks are becoming more and more popular as tools in time-series prediction, pattern recognition, process control, signal and image processing, etc. One of the new exciting and growing directions in neural networks are models that process temporal signals and have feedback. Fully understanding these models means understanding their relationships and similarities to other models of mathematics, logic and computer science. It also means understanding what the neural network models are capable of computing and representing and what computational insights can be gained from new models of neural networks. Because of the increased importance of computational models in all fields, the interest in this area in neural networks and computer science will continue to grow.

In particular, the papers selected explore the mentioned relationship between neural networks and the foundations of computer science: automata (from finite-state to Turing machines), the classes of formal languages they define (from regular to unrestricted), and the underlying models for computation. Neural networks have been used both to represent these models and to learn them from examples; this growing field of research has produced an interesting corpus of scientific work which, in addition to strengthening the view of neural networks as implementations of theoretical computational devices, has found numerous practical applications. I have tried to survey in a single volume the most important work relating neural networks to formal language and computation models, which may be found scattered —as many other works on neural networks— throughout a wide variety of journals as well as in technical reports.

The hardest part of compiling a document like this is the process of selecting the papers. Some papers have not been included because of their extension. Sometimes I have decided not to include the earliest account of a new idea or concept, but have included a later, but clearer paper; in these cases, earlier works are referred and briefly discussed in the introductions.

To my knowledge, this is the first document covering this relationship; however, it has an orientation which is very similar to other books such as *Discrete Neural Computation* by Siu et al. (1995) and *Circuit Complexity and Neural Networks* by Parberry (1994) which explore related subjects. What makes this document different is its emphasis in dynamic models of computation (i.e., automata) and dynamic neural networks (i.e., recurrent neural networks).

The featured papers will be grouped in chapters. Each group of papers will share an introduction that will allow the reader to comprehend the relationships among the different approaches and the main emerging results.

Chapter introductions have been written having in mind two (apparently conflicting) goals:

- on the one hand, an effort has been made to define clearly and formally (and motivating the need for) the main theoretical and neural concepts and constructions discussed in the papers and their relationship so that introductions may almost be read independently of the papers and convey a reasonably comprehensive view of the field;
- on the other hand, I have tried not to give away too much but instead have tried to "advertise" the papers in such a way that the reader cannot resist the temptation to delve into the individual papers.

In this way, the two main intended uses of the document are covered: either as a textbook in a graduate course or as a reference book for researchers active in the fields involved. Chapter introductions provide the reader with a unified vision of the whole field since its inception up to the most recent work. For this purpose:

- a homogeneous notation is throughout the whole document, a notation that may serve as a reference to understand the particular notations used by the authors of the particular papers.
- special care has been taken to provide extensive references to the most relevant work in the field, to make it useful to any reader willing to start or pursue research in the expanding field of neural networks, automata and formal models of computation.

The intended readership of this document includes active researchers, both in academia and industry (mainly in computer science, computer engineering, electrical engineering, and applied mathematics departments or in R&D laboratories involved in computing, control, signal processing, etc.) of all levels, both coming from the fields of formal languages and computational theory and from the broader neural networks community; graduate students in these fields; and research planners or evaluators. Readers are expected to hold (or be about to obtain) a bachelor's degree in computer science, computer engineering, electrical engineering or applied mathematics and to be interested in fields such as computer science (formal languages and models of computation, pattern recognition, machine learning), computer engineering, electrical engineering, applied mathematics, control, signal processing, and complex systems.

Acknowledgements: This document was conceived as a reprint collection by C. Lee Giles, whose contributions to it have been crucial; my sincerest thanks go to him. This document has also benefited enormously from comments by Rafael C. Carrasco, Juan Antonio Pérez Ortiz and Asun Castaño and from technical help by Ana Isabel Guijarro, all of whom I thank. Partial support through grant TIC97-0941 of the Spanish Comisión Ministerial de Ciencia y Tecnología is also acknowledged.

Mikel L. Forcada
January 2002

Chapter 1

Introduction

1.1 Neural networks and formal models of language and computation

The last decade has witnessed a surge of interest in the relationships between the behavior of artificial neural networks and models used in formal language theory and theory of computation, such as automata or Turing machines. As a result, an increasing number of researchers are focusing their attention in questions such as

- "what can a neural network compute?", and —since neural networks are by nature trainable systems—
- "what can a neural network *learn* to compute?"

Distinguishing these two apparently equivalent questions is crucial in the field of artificial neural networks. It is often the case that neural network architectures that may be shown to be capable of a certain kind of computation cannot easily be trained from examples to perform it. Unfortunately, one may found examples in the literature where researchers attribute an observed problem to the architecture used when it may also be due to the training algorithm (learning algorithm) used.

These questions —which will be called the *main questions* in this Introduction— tie together the cores of two broad fields of basic knowledge: computer science and neuroscience. The convergence of their daughter disciplines (such as pattern recognition, artificial intelligence, neurocognition, etc.) in the interdisciplinary arena of artificial neural networks may be reasonably expected to have a great technological impact and a wealth of applications.

Indeed, on the one hand, in computer science, the formal theories of language and computation are so intimately related that they may be considered to form a single body of knowledge. One may just take a look at the titles of highly cited and well known books on the subject; as an example, take the *Introduction to automata theory, languages and computation* by Hopcroft and Ullman

(1979). There is a well-established relationship between the levels in Chomsky's hierarchy of language complexity (regular, context-free, context-sensitive and unrestricted, in order of increasing complexity) and the classes of idealized machines, (correspondingly, finite-state automata, pushdown or stack automata, linearly bounded automata and Turing machines, in increasing order of computational power). This equivalence allows us to view computational devices —that is, automata— as language acceptors or language recognizers and their computational power in terms of the complexity of the language class they accept or recognize.

On the other hand, the advances in neuroscience have inspired idealized models of the brain and the nervous system. Extremely simplified models of the behavior of an isolated neuron and of the interaction between neurons have been used to construct the concept of *artificial neural networks*, systems that are capable of acting as universal function approximators (Hornik et al., 1989), amenable to be trained from examples without the need for a thorough understanding of the task in hand, and able to show surprising generalization performance and predicting power, thus mimicking some interesting cognitive properties of evolved natural neural systems such as the brains of mammals.

In the light of the equivalence between language and automata classes, and the wide variety of neural architectures —more specifically, discrete-time recurrent neural networks— that can be used to emulate the behavior of sequential machines, the basic questions stated at the beginning of this introduction may be given a more concrete formulation: on the one hand,

- can a neural network of architecture class A perform the same computation as an automaton of class M?
- can a neural network of architecture class A be a recognizer for languages of language class L?

and on the other hand,

- can a neural network of architecture class A *be trained* [1] *to* perform the same computation as an automaton of class M, from a set of examples?
- can a neural network of architecture class A *be trained to* recognize a language of class L from a set of examples?

Viewing computing devices as language recognizers or language acceptors allows researchers to view examples as strings, belonging or not to the language —to the computation— that is to be learned by the neural net. This brings the research in this field very close to another important area of computer science: *grammatical inference*, that is learning grammars from examples. Grammatical inference is relevant because a wide variety of phenomena may be represented and treated as languages after discretization of the input sequences as strings of symbols from an appropriate alphabet.

[1] Of course, in reasonable time, as Marvin Minsky remarks in his foreword to the book by Siu et al. (1995).

One of the first contacts of these two wide streams of research may be traced to a date as early as 1943, when two researchers from MIT, Warren S. McCulloch and Walter Pitts, published their paper "A logical calculus of the ideas immanent in nervous activity" (McCulloch and Pitts, 1943). This paper is considered to be seminal to both the field of artificial neural networks and to that of automata theory. As Perrin (1990) put it, McCulloch and Pitts presented "a logical model for the behaviour of nervous systems that turned out to be the model of a finite-state machine". This contact was followed a decade later by the work of Kleene (1956) and Minsky (1956), who gradually moved away from the neural formalization toward a logical and mathematical layout that would be called "finite automata". It was Minsky (1967) who made the often-quoted statement "every finite-state machine is equivalent to, and can be simulated by, some neural net".

A long hiatus of more than three decades followed that early junction. The field of artificial neural networks grew falteringly in the beginning and then bloomed exuberantly in the late seventies and the early eighties with the work of Werbos (1974), Kohonen (1974), Hopfield (1982), and McClelland et al. (1986). The formal theory of language and computation has been maturing steadily during this time. It was not until the late eighties that both fields made contact again, this time for a long and fertile relationship. This document is a selection of what I believe to be some of the best representatives of the work that has been done in the field that connects artificial neural networks, automata and formal models of language and computation.

1.2 Organization of the document

The document is organized in four chapters in addition to this Introduction. Each chapter contains extensive introductory material that will allow the reader to comprehend the relationships among the different approaches and the main emerging results; these chapter introductions include the tutorial material needed to put the papers in context.

Chapter 2 illustrates the beginning of the relationship between neural networks, automata and formal models of computation by presenting two pioneering papers; one is the seminal paper "A logical calculus of ideas immanent in nervous activity", (McCulloch and Pitts, 1943), and the other is a book chapter by Minsky (1967), which discusses in detail the implementation of finite-state automata in terms of McCulloch-Pitts neurons. The introduction of the chapter includes a discussion on the computational capabilities of a single neuron as well as an introduction to finite-state machines.

Chapter 3 addresses the use of neural networks as sequence processors and collects four papers (Jordan, 1986; Elman, 1990; Pollack, 1990; Bengio et al., 1994) in which discrete-time recurrent neural networks are trained to process temporal sequences. The introduction describes discrete-time recurrent neural network architectures in detail, viewing them as neural automata, and discusses briefly the learning algorithms used for training them and the problems that

may be encountered.

Chapter 4 deals with the theoretical work concerning the computational (symbolic sequence processing) capabilities of discrete-time recurrent neural networks and collects a number of representative papers (Alon et al., 1991; Siegelmann and Sontag, 1991; Goudreau et al., 1994; Siegelmann, 1995; Kremer, 1995; Horne and Hush, 1996; Omlin and Giles, 1996b; Alquézar and Sanfeliu, 1995). The introduction contains a tutorial describing formal grammars, Chomsky's hierarchy, Turing machines, and super-Turing computation.

The field of grammatical inference (that is, learning a language from examples) is the main theme of the papers collected in Chapter 5, which deals with the inference of regular languages and finite-state automata (Cleeremans et al., 1989; Pollack, 1991; Giles et al., 1992; Manolios and Fanelli, 1994; Tiňo and Sajda, 1995) and with the inference of context-free languages and pushdown automata (Zeng et al., 1994; Mozer and Das, 1993; Giles et al., 1990). The chapter starts with an introduction to grammatical inference in general and in particular with neural networks.

Chapter 2

Finite-state machines and neural nets: the inception

This chapter collects two papers: one is the seminal paper by McCulloch and Pitts (1943), where the authors decided to explore the question "what can a neural network compute?" in terms of a very idealized model of the brain which was in turn based on a very simplified model of neuron: a threshold unit. The second paper, by Minsky (1967), discusses in detail the implementation of finite-state machinesin terms of McCulloch-Pitts neurons.

2.1 McCulloch and Pitts' neural logical calculus

The paper by McCulloch and Pitts (1943) is commonly regarded as the inception of two fields of research. One is the theory of finite-state machines as a model of computation. The other one is the field of artificial neural networks. This is an important and widely cited paper, yet a difficult one to understand. the notation used by McCulloch and Pitts (1943) is hard to follow for us nowadays, but it also was for Kleene (1956), who expresses this very clearly:

> The present article is partly an exposition of their results; but we found the part of their [McCulloch and Pitts'] paper dealing with arbitrary nerve nets obscure, so we have proceeded independently there.

Later on, having found an apparent flaw in one of the results by McCulloch and Pitts (1943), Kleene (1956) says:

> This apparent counterexample discouraged us from further attempts to decipher part III of McCulloch-Pitts [1943].

The paper by McCulloch and Pitts (1943) paper is properly titled "A logical calculus of the ideas immanent in nervous activity": after stating a careful and well argumented selection of simplifications of the behavior of real neurons, they develop a logical apparatus to define:

- The concepts of *solution* of a net and *realizability* of a logical predicate by a net. After dividing the neurons in one net in two groups, that is, input neurons ("peripheral afferents") that do not get signals from any other neuron in the net, and the rest of neurons, they go on to define a method to answer the following two questions in the most general way possible:
  - What does a given net compute?
  - Can a given net compute a given logical sentence?

  Neurons are in two possible states: firing and not firing, and thus, they define for each neuron i a predicate that is true when the neuron is firing at a given time t: $N_i(t)$. They define the *solution* of a net as a set of logical sentences of the form "neuron i is firing if and only if" a given logical combination of the firing predicates of input neurons at previous times and some constant sentences including firing predicates of these same neurons at $t = 0$ is true. These sentences are a solution for a net if they are all true for it. In other words, the sentences describe what the net computes. Conversely, such an "if and only if" sentence is called *realizable* by a net if it is true for that net; that is, when the net can compute it.

- A class of logical expressions (including predicates which have predicates as functions) which they call *temporal propositional expressions* (TPE). These predicates have a single free variable (which will be identified as discrete time), and are recursively defined such that (a) any predicate of one argument is a TPE; (b) the logical conjunction (and), disjunction (or) and negated conjunction (and not) of any two TPEs with the same variable is a TPE; (c) a TPE in which the value of the time variable is substituted by its predecessor (time delay) is a TPE; and nothing else is a TPE.

The main result is that any TPE is realizable by a non-recurrent neural net, that is, there is always a net, whose synapses or connections do not form cycles, which can compute any given TPE (McCulloch and Pitts (1943) call non-recurrent nets "nets without circles"). This result follows from the fact that the neurons in McCulloch and Pitts (1943) are described by a TPE (their eq. (1)) which basically states that a neuron i is firing at time t if and only if none of the neurons having an inhibitory synapse towards it was firing at time $t - 1$ and more than θ_i neurons having an excitatory synapse towards it were firing at time $t - 1$. The positive integer θ_i is called the *excitation threshold* of neuron i.

To understand the computational behavior of nets *with* circles, one can follow Kleene (1956)[1]. His notation is closer to the one used nowadays, and the results are more general. Kleene (1956) characterized with mathematical expressions the set of all input neuron activation sequences that bring a given net with circles to a particular state after they have been completely processed, and discovered

[1]Available as a PDF file at `http://www.dlsi.ua.es/~mlf/nnafmc/papers/kleene56representation.pdf`, retyped by Juan Antonio Pérez-Ortiz

interesting regularities among them. Kleene (1956) called them "regular events"; we currently call them "regular expressions" in language theory (Hopcroft and Ullman, 1979, 28).

2.2 What functions can a neuron compute?

From the statements in McCulloch and Pitts (1943) it follows naturally that their neurons are equivalent to a model commonly used nowadays, where:

- Each neuron i is in either of two states at time t: $x_i[t] = 1$ or "firing" and $x_i[t] = 0$ or "not firing";
- all synapses (connections) are equivalent and characterized by a real number (their strength), which is positive for excitatory connections and negative for inhibitory connections;
- a neuron i becomes active when the sum of those connections w_{ij} coming from neurons j connected to it which are active, plus a bias b_i, is larger than zero.

This is usually represented by

$$x_i[t] = \theta \left(b_i + \sum_{j \in C(i)} w_{ij} x_j[t-1] \right), \tag{2.1}$$

where $\theta(x)$ is the step function: 1 when $x >= 0$ and 0 otherwise and $C(i)$ is the set of neurons that impinge on neuron i. This kind of neural processing element is usually called a *threshold linear unit* or TLU. The time indexes are dropped when processing time is not an issue (Hertz et al., 1991, 4).

If all inputs (assume there are n of them) to a TLU are either 0 or 1, the neuron may be viewed as computing a logical function of n arguments. The truth table of an arbitrary, total logical function of n arguments has 2^n different rows, and the output for any of them may be 0 or 1. Accordingly, there are $2^{(2^n)}$ logical functions of n arguments. However, there are logical functions a TLU cannot compute. For $n = 1$ all 4 possible functions (identity, negation, constant true and constant false) are computable. However, for $n = 2$ there are two noncomputable functions, corresponding to the exclusive or and its negation. The fraction of computable functions cannot be expressed as a closed-form function of n but vanishes as n grows (Horne and Hush, 1996)). The computable functions correspond to those in which the set of all input vectors corresponding to true outputs and the set of all input vectors corresponding to false outputs are separable by a n-dimensional hyperplane in that n-dimensional space. This follows intuitively from eq. (2.1): the equation of the hyperplane is the argument of function θ equated to zero.

The computational limitations of TLUs have a radical consequence: to compute a general logical function of n arguments, one needs a cascade of TLUs.

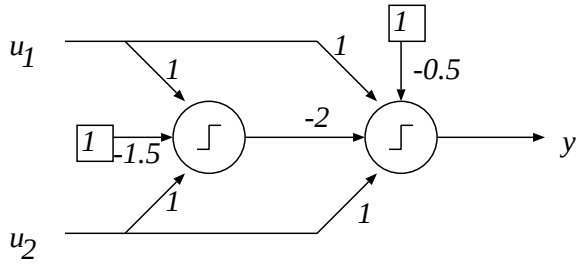

Figure 2.1: Two TLUs may be used to compute the exclusive-or function (u_1 and u_2 are the inputs, y is the output, and biases are represented as connections coming from a constant input of 1).

For example, to compute the *exclusive-or* function one needs at least two TLUs, as shown in figure 2.1. A common layout is the so-called *multilayer perceptron* (MLP) or *layered feedforward neural net* (Haykin (1998), ch. 4; Hertz et al. (1991), ch. 6). In this layout:

- Each neuron belongs to a subset called *layer*.
- If neuron i belongs to layer I then all neurons j sending their output to neuron i belong to layer $I - 1$.
- Layer 0 is the input vector.

The backpropagation (BP) learning algorithm (Haykin (1998), sec. 4.3; Hertz et al. (1991), ch. 6) is usually formulated for the MLP.

2.3 Nets with circles and finite-state machines

The second document featured in this section, a chapter in *Infinite and Finite Machines* (Minsky, 1967), is well known because it establishes the equivalence between neural nets with cyclic connections and a class of abstract computing devices called *finite-state machines* or *finite automata*. The following sections define three main classes of finite-state machines (FSM): Mealy machines, Moore machines and deterministic finite automata .

2.3.1 Mealy machines

Mealy machines (Hopcroft and Ullman (1979, 42), Salomaa (1973, 31)) are finite-state machines that act as transducers or translators, taking a string on an input alphabet and producing a string of equal length on an output alphabet. Formally, a *Mealy machine* is a six-tuple

$$M = (Q, \Sigma, \Gamma, \delta, \lambda, q_I) \tag{2.2}$$

where

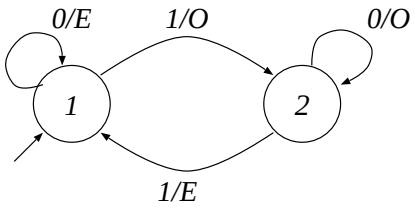

Figure 2.2: A Mealy machine that outputs an `E` if the number of `1`s read so far is even and an `O` if it is odd. Transition labels are σ/γ where $\sigma \in \Sigma$ is the input and $\gamma \in \Gamma$ is the output.

- $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$ is a finite set of *states*;
- $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$ is a finite *input alphabet*;
- $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_{|\Gamma|}\}$ is a finite *output alphabet*;
- $\delta : Q \times \Sigma \to Q$ is the *next-state function*, such that a machine in state q_j, after reading symbol σ_k, moves to state $\delta(q_j, \sigma_k) \in Q$.
- $\lambda : Q \times \Sigma \to \Gamma$ is the *output function*, such that a machine in state q_j, after reading symbol σ_k, writes symbol $\lambda(q_j, \sigma_k) \in \Gamma$; and
- $q_I \in Q$ is the *initial state* in which the machine is found before the first symbol of a string is processed.

As an example, let $M = (Q, \Sigma, \Gamma, \delta, \lambda, q_I)$ such that:

$Q = \{q_1, q_2\}$, $q_I = q_1$.

$\Sigma = \{\texttt{0}, \texttt{1}\}$

$\Gamma = \{\texttt{E}, \texttt{O}\}$

$\delta(q_1, 0) = q_1$, $\delta(q_1, 1) = q_2$
$\delta(q_2, 0) = q_2$, $\delta(q_2, 1) = q_1$

$\lambda(q_1, 0) = \texttt{E}$, $\lambda(q_1, 1) = \texttt{O}$
$\lambda(q_2, 0) = \texttt{O}$, $\lambda(q_2, 1) = \texttt{E}$

This machine, which may also be represented as in figure 2.2, outputs an `E` if the number of `1`s read so far is even and an `O` if it is odd; for example, the translation of `11100101` is `OEOOOEEO`.

2.3.2 Moore machines

A *Moore machine* (Hopcroft and Ullman, 1979, 42) may be defined by a similar six-tuple, with the only difference that symbols are output after the transition to a new state is completed, and the output symbol depends only on the state just reached, that is, $\lambda : Q \rightarrow \Gamma$.

The class of translations that may be performed by Mealy machines and the class of translations that may be performed by Moore machines are identical. Indeed, given a Mealy machine, it is straightforward to construct the equivalent Moore machine and vice versa (Hopcroft and Ullman, 1979, 44) .

2.3.3 Deterministic finite-state automata

Deterministic finite-state automata (DFA) (Hopcroft and Ullman (1979, 16),Salomaa (1973, 26)) may be seen as a special case of Moore machines. A DFA is a five-tuple

$$M = (Q, \Sigma, \delta, q_I, F) \tag{2.3}$$

where Q, Σ, δ and q_I have the same meaning as in Mealy and Moore machines and $F \subseteq Q$ is the set of *accepting states*. If the state reached by the DFA after reading a complete string in Σ^* (the set of all finite-length strings over Σ, including the empty string ϵ) is in F then, the string is *accepted*; if not, the string is *not accepted* (or *rejected*). This would be equivalent to having a Moore machine whose output alphabet has only two symbols, Y (yes) and N (no), and looking only at the last output symbol (not at the whole output string) to decide whether the string is accepted or rejected.

As an example, let $M = (Q, \Sigma, \delta, q_I, F)$ be a DFA such that:

$Q = \{q_1, q_2\}$, $F = \{q_1\}$, $q_I = q_1$.

$\Sigma = \{\texttt{0}, \texttt{1}\}$

$\delta(q_1, 0) = q_1$
$\delta(q_1, 1) = q_2$
$\delta(q_2, 0) = q_2$
$\delta(q_2, 1) = q_1$

This automaton, which may also be represented as in figure 2.3, accepts only those strings of `0`s and `1`s having an even number of `1`s. Another example of DFA is given by the following definition

$Q = \{q_1, q_2, q_3, q_4\}$, $F = \{q_1, q_2, q_3\}$, $q_I = q_1$.

$\Sigma = \{\texttt{0}, \texttt{1}\}$

$\delta(q_1, 0) = q_2$, $\delta(q_1, 1) = q_1$
$\delta(q_2, 0) = q_3$, $\delta(q_2, 1) = q_1$
$\delta(q_3, 0) = q_4$, $\delta(q_3, 1) = q_1$
$\delta(q_4, 0) = q_4$, $\delta(q_4, 1) = q_4$.

This automaton, which may also be represented as in figure 2.4, accepts only those strings of `0`s and `1`s not including the substring `000`.

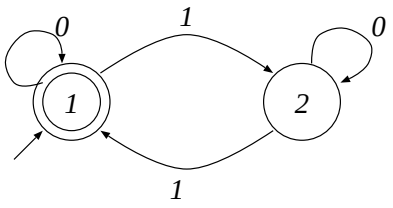

Figure 2.3: A deterministic finite-state automaton that accepts only binary strings having an even number of 1s.

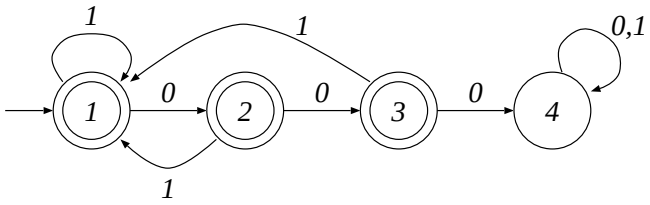

Figure 2.4: A deterministic finite-state automaton that accepts only binary strings not containing the sequence "000".

2.3.4 Minsky's neural automata

This chapter discusses the implementation of finite-state machines (formulated as Mealy machines) using McCulloch and Pitts (1943)'s neurons ; it starts by building very simple automata such as gates, switches, impulse counters, and simple arithmetic circuits, and finally proves a theorem which is crucial to the field that connects artificial neural networks and the formal theories of language and computation. In Minsky's own words, "every finite-state machine is equivalent to, and can be simulated by, some neural net". The theorem constructs a recurrent neural net in which there are $|Q||\Sigma|$ units which detect a particular combination of state and input symbol and $|\Gamma|$ units which compute outputs. The neural net is shown to exhibit the same behavior as the corresponding Mealy machine.

It is worth mentioning that the net itself is a neural Moore machine, architecturally very similar to Elman (1990)'s nets and that Minsky's construction has inspired other work such as Kremer (1995)'s or Alquézar and Sanfeliu (1995)'s .[2]

[2]Some of these architectures use sigmoid units whose output is real and continuous instead of TLU whose output is discrete; for details, see section 3.2.

2.4 Finite-state automata and regular languages

Each given DFA, as defined in eq. 2.3, can *accept* a certain set of strings of symbols. A (finite or infinite) set of strings over a given alphabet is usually called a *language*. The class of languages that DFAs can accept is called the class of *regular languages*, where the word *regular* is borrowed from Kleene (1956)'s "regular events"; in other words, if a language is accepted by a DFA, it is a regular language, and *vice versa*. These languages (also called *regular sets*) may be represented by *regular expressions* based on Kleene (1956)'s regular events. Each regular expression represents a set of strings. Now, for any two expressions r and s, rs represents the set of all strings that may be obtained by concatenating strings of r with strings of s and $r|s$ represents the union of r and s; in addition, for any set s, s^* represents the set of all strings that may be obtained by concatenating strings of s to themselves zero (that is, the empty string ϵ) or more times (this operation is called the Kleene closure and the symbol is called the Kleene star); using these operations (concatenation, union, and Kleene closure) on regular languages always yields regular languages. The basic regular languages are the empty set $\{\}$ (represented $\emptyset$), the languages containing a single one-symbol string ($\{\sigma\}$ for all $\sigma \in \Sigma$, represented simply by σ), and the language containing only the empty string $\{\epsilon\}$ (represented by ϵ itself). It is easy to show that any finite language is regular. Kleene (1956) showed for the first time that the set of languages expressible by regular expressions is exactly the one that may be accepted by a net with circles, that is, by a finite-state machine (see also Hopcroft and Ullman (1979, 218), Salomaa (1973, 27)).

See section 4.1.1 for an introduction to grammars as an alternative way of defining formal languages.

Chapter 3

Sequence processing with neural networks

This chapter collects a number of early papers in which neural networks are trained to be sequence processors. The notions of sequence or time are consubstantial to the concept of computation as a sequential behavior, and indeed, the question explored by papers in this chapter is the second "main question" mentioned in the introduction, that is, "what can a neural network *learn* to compute?", in terms of sequence processing and recognition.

The introductory material discusses briefly what is sequence processing (section 3.1); then, the following sections illustrate how neural networks may be used as sequence processors. The use of neural networks for sequence processing tasks has a very important advantage: neural networks are adaptive devices that may be trained to perform sequence processing tasks from examples. Section 3.2 gives a general introduction to a kind of neural networks which is very relevant to sequence processing, namely, discrete-time recurrent neural networks, under the paradigm of "neural state machines"; section 3.3 briefly reviews some applications of recurrent neural networks to some real-world sequence processing tasks; section 3.4 gives an outline of the main learning (also *training*) algorithms; and the learning problems that may be observed are discussed in section 3.5. Finally, a brief introduction to each one of the featured papers is given in section 3.6

3.1 Processing sequences

The word *sequence* (from Latin *sequentia*, i.e., "the ones following") is used to refer to a series of data items, each one taken from a certain set of possible values U, so that each one of them is assigned an index (usually consecutive integers) indicating the order in which the data items are generated or measured.[1] Since

[1] Other authors (Stiles et al., 1997; Stiles and Ghosh, 1997) prefer to see sequences as functions that take an integer and return a value in U.

the index usually refers to time, some researchers like to call sequences *time series* as in "time series prediction" (Janacek and Swift, 1993; Weigend and Gershenfeld, 1993; Box et al., 1994). In the field of signal processing, this would usually be called a discrete-time sampled signal; researchers in this field would identify the subject of this discussion as that of *discrete-time signal processing* (Oppenheim and Schafer, 1989; Mitra and Kaiser, 1993; Ifeachor and Jervis, 1994).

In most of the following, it will be considered, for convenience, that U is a vector space in the broadest possible sense. Examples of sequences would be:

- strings (words) on an alphabet (where U would be the alphabet of possible letters and the integer labels $1, 2, \ldots$ would be used to refer to the first, second, etc. symbol of the string)
- acoustic vectors obtained every T milliseconds after suitable preprocessing of a speech signal (here U would be a vector space and the indices would refer to sampling times)

What can be done with sequences? Far from having the intention of being exhaustive and formal, one may classify sequence processing operations in the following broad classes:[2]

Sequence classification, sequence recognition: in this kind of processing, a whole sequence $u = u[1]u[2]\ldots u[L_u]$ is read and a single value, label or pattern (not a sequence) y, taken from a suitable set Y, is computed from it. For example, a sequence of acoustic vectors such as the one mentioned above may be assigned a label that describes the word that was pronounced, or a vector of probabilities for each possible word. Or a string on a given alphabet may be recognized as belonging to a certain formal language. For convenience, Y will also be considered to be some kind of vector space.

Sequence transduction or translation, signal filtering: In this kind of processing, a sequence $u = u[1]u[2]\ldots u[L_u]$ is transformed into another sequence $y = y[1]y[2]\ldots y[L_y]$ of data items taken from a set Y. In principle, the lengths of the input L_u and the output L_y may be different. Processing may occur in different modes: some sequence processors read the whole input sequence u and then generate the sequence y. Another mode is *sequential* processing, in which the output sequence is produced incrementally while processing the input sequence. Sequential processing has the interesting property that, if the result of processing of a given sequence u_1 is a sequence y_1, then the result of processing a sequence that starts with u_1 is always a sequence that starts with y_1 (this is sometimes called the *prefix property*). A special case of sequential processing is *synchronous* processing: the processor reads and writes one data item at a time, and therefore, both sequences grow at the same rate during processing. For example, Mealy and Moore machines, defined in section 2.3,

[2]This classification is inspired in the one given by Hertz et al. (1991, 177).

are sequential, finite-memory, synchronous processors that read and write symbol strings. Examples of transductions and filtering include machine translation of sentences and filtering of a discrete-time sampled signal. Note that sequence classification applied to each prefix $u[1]$, $u[1]u[2]$, etc. of a sequence $u[1]u[2]u[3]\ldots$ is equivalent to synchronous sequence transduction.

Sequence continuation or prediction: In this case, the sequence processor reads a sequence $u[1]u[2]\ldots u[t]$ and produces as an output a possible continuation of the sequence $\hat{u}[t+1]\hat{u}[t+2]$.... This is usually called *time series prediction* and has interesting applications in meteorology or finance, where the ability to predict the future behavior of a system is a primary goal. Another interesting application of sequence prediction is predictive coding and compression. If the prediction is good enough, the difference between the predicted continuation of the signal and its actual continuation may be transmitted using a channel with a lower bandwidth or a lower bit rate. This is extensively used in *speech coding* (Sluijter et al., 1995), for example, in digital cellular phone systems.

Sequence generation: in this mode, the process generates an output sequence $y[1]y[2]\ldots$ from a single input u or no input at all. For example, a phone number inquiry system may generate a synthetical pronunciation of each digit.

3.1.1 State-based sequence processors

Sequence processors may be built around a *state*; state-based sequence processors maintain and update at each time t a state $x[t]$ which stores the information about the input sequence they have seen so far $(u[1],\ldots,u[t])$ which is necessary to compute the current output $y[t]$ or future outputs. State is *recursively* computed: the state at time t, $x[t]$, is computed from the state at time $t-1$, $x[t-1]$, and the current input $u[t]$ using a suitable *next-state* function:

$$x[t] = f(x[t-1], u[t]). \tag{3.1}$$

The output is then computed using an *output* function, usually from the current state,

$$y[t] = h(x[t]), \tag{3.2}$$

but sometimes from the previous state and the current input, like current state itself

$$y[t] = h(x[t-1], u[t]). \tag{3.3}$$

Such a state-based sequence processor is therefore defined by the set of available states, by its initial state $x[0]$, and by the next-state (f) and output (h) functions (the nature of inputs and outputs is defined by the task itself). For example, Mealy and Moore machines (sections 2.3.1 and 2.3.2) and deterministic finite-state automata (section 2.3.3) are sequence processors having a finite set of

available states. As will be seen in the following section, neural networks may be used and trained as state-based adaptive sequence processors. .

3.2 Discrete-time recurrent neural networks

When neural networks are used to do sequence processing, the most general architecture is a recurrent neural network (that is, a neural network in which the output of some units is fed back as an input to some others), in which, for generality, unit outputs are allowed to take any real value in a given interval instead of simply two characteristic values as in threshold linear units. In particular, since sequences are discrete in nature (that is, they are made of data indexed by integers), the processing occurs in discrete steps, as if the network were driven by an external clock, and each of the neurons is assumed to compute its output instantaneously, hence the name *discrete-time recurrent neural networks* to account for this fact. There is another wide class of recurrent neural networks in which inputs and outputs are functions of a continuous time variable and neurons have a temporal response (relating state to inputs) that is described by a differential equation in time (Pineda, 1987). These networks are aptly called continous-time recurrent neural networks (for an excellent review, see Pearlmutter (1995)).

Discrete-time recurrent neural networks are adaptive, state-based sequence processors that may be applied to any of the four broad classes of sequence processing tasks mentioned in section 3.1: in sequence classification, the output of the DTRNN is examined only at the end of the sequence; in synchronous sequence transduction tasks, the DTRNN produces a temporal sequence of outputs corresponding to the sequence of inputs it is processing; in sequence continuation or prediction tasks, the output of the DTRNN after having seen an input sequence may be interpreted as a continuation of it; finally, in sequence generation tasks, a constant or no input may be applied in each cycle to generate a sequence of outputs.

In this document, it has been found to be convenient to see discrete-time recurrent neural networks (DTRNN) (see Haykin (1998), ch. 15; Hertz et al. (1991), ch. 7; Hush and Horne (1993); Tsoi and Back (1997)) as *neural state machines* (NSM), and to define them in a way that is parallel to the definitions of Mealy and Moore machines given in section 2.3. This parallelism is inspired in the relationship established by Pollack (1991) between deterministic finite-state automata (DFA) and a class of second-order DTRNN,[3] under the name of *dynamical recognizers*. A *neural state machine* N is a six-tuple

$$N = (X, U, Y, \mathbf{f}, \mathbf{h}, \mathbf{x}_0) \tag{3.4}$$

in which

- $X = [S_0, S_1]^{n_X}$ is the state space of the NSM, with S_0 and S_1 the values defining the range of values for the state of each unit, and n_X the number

[3] Defined in section 3.2.1.

of state units;[4]

- $U = \mathbb{R}^{n_U}$ defines the set of possible input vectors, with n_U the number of input lines;
- $Y = [S_0, S_1]^{n_Y}$ is the set of outputs of the NSM, with n_Y the number of output units;
- $\mathbf{f} : X \times U \to X$ is the *next-state function* a feedforward neural network which computes a new state $\mathbf{x}[t]$ from the previous state $\mathbf{x}[t-1]$ and the input just read $\mathbf{u}[t]$[5].:

$$\mathbf{x}[t] = \mathbf{f}(\mathbf{x}[t-1], \mathbf{u}[t]); \tag{3.5}$$

- $\mathbf{h}$ is the *output function*, which in the case of a Mealy NSM is $\mathbf{h} : X \times U \to Y$, that is, a feedforward neural network which computes a new output $\mathbf{y}[t]$ from the previous state $\mathbf{x}[t-1]$ and the input just read $\mathbf{u}[t]$:

$$\mathbf{y}[t] = \mathbf{h}(\mathbf{x}[t-1], \mathbf{u}[t]), \tag{3.6}$$

  and in the case of a Moore NSM is $\mathbf{h} : X \to Y$, a feedforward neural network which computes a new output $\mathbf{y}[t]$ from the newly reached state $\mathbf{x}[t]$:

$$\mathbf{y}[t] = \mathbf{h}(\mathbf{x}[t]); \tag{3.7}$$

- and finally, $\mathbf{x}_0$ is the initial state of the NSM, that is, the value that will be used for $\mathbf{x}[0]$.

Most classical DTRNN architectures may be directly defined using the NSM scheme; the following sections show some examples (in all of them, weights and biases are assumed to be real numbers). The generic block diagrams of neural Mealy and neural Moore machines are given in figures 3.1 and 3.2 respectively.

3.2.1 Neural Mealy machines

Omlin and Giles (1996a) and Omlin and Giles (1996b) have used a second-order[6] recurrent neural network (similar to the one used by Giles et al. (1992), Pollack (1991), Forcada and Carrasco (1995), Watrous and Kuhn (1992), and Zeng et al. (1993)) which may be formulated as a Mealy NSM described by a next-state function whose i-th coordinate ($i = 1, \ldots, n_X$) is

$$f_i(\mathbf{x}[t-1], \mathbf{u}[t]) = g\left(\sum_{j=1}^{n_X}\sum_{k=1}^{n_U} W_{ijk}^{xxu} x_j[t-1]u_k[t] + W_i^x\right), \tag{3.8}$$

[4] All state units are assumed to be of the same kind.

[5] Unlike in eqs. (3.1–3.3), bold lettering is used here to emphasize the vectorial nature of states, inputs, outputs, and next-state and output functions.

[6] *Second-order* neural networks operate on products of *two* inputs, each input taken from a different subset; first-order networks instead, operate on raw inputs.

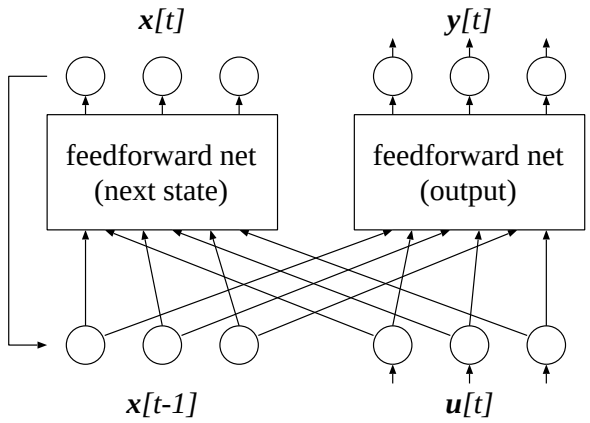

Figure 3.1: Block diagram of a neural Mealy machine.

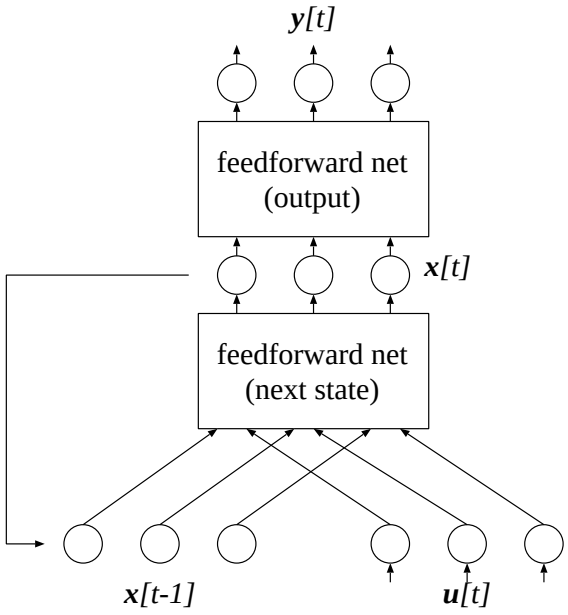

Figure 3.2: Block diagram of a neural Moore machine.

where $g : \mathbb{R} \to [S_0, S_1]$ (usually $S_0 = 0$ or -1 and $S_1 = 1$) is the activation function[7] of the neurons, and an output function whose i-th coordinate ($i = 1, \ldots, n_Y$) is

$$h_i(\mathbf{x}[t-1], \mathbf{u}[t]) = g\left(\sum_{j=1}^{n_X}\sum_{k=1}^{n_U} W_{ijk}^{yxu} x_j[t-1]u_k[t] + W_i^y\right). \tag{3.9}$$

Throughout this document, a homogeneous notation will be used for weights. Superscripts indicate the computation in which the weight is involved: the xxu in W_{ijk}^{xxu} indicates that the weight is used to compute a state (x) from a state and an input (xu); the y in W_i^y (a bias) indicates that it is used to compute an output. Subscripts designate, as usual, the particular units involved and run parallel to superscripts.

Activation functions $g(x)$ are usually required to be real-valued, monotonously growing, continuous (very often also differentiable), and bounded; they are usually nonlinear. Two commonly used examples of differentiable activation functions are the logistic function $g_L(x) = 1/(1 + \exp(-x))$, which is bounded by 0 and 1, and the hyperbolic tangent $g_T(x) = \tanh(x) = (1 - \exp(-2x))/(1 + \exp(-2x))$, which is bounded by -1 and 1. Activation functions are usually required to be differentiable because this allows the use of learning algorithms based on gradients. There are also a number of architectures that do not use sigmoid-like activation functions but instead use *radial basis functions* ((Haykin, 1998), ch. 5; Hertz et al. (1991, 248)), which are not monotonous but instead are Gaussian-like functions that reach their maximum value for a given value of their input. DTRNN architectures using radial basis functions have been used by Cid-Sueiro et al. (1994); Frasconi et al. (1996).

Another Mealy NSM is that defined by Robinson and Fallside (1991) under the name of *recurrent error propagation network*, a first-order DTRNN which has a next-state function whose i-th coordinate ($i = 1, \ldots, n_X$) is given by

$$f_i(\mathbf{x}[t-1], \mathbf{u}[t]) = g\left(\sum_{j=1}^{n_X} W_{ij}^{xx} x_j[t-1] + \sum_{j=1}^{n_U} W_{ij}^{xu} u_j[t] + W_i^x\right), \tag{3.10}$$

and an output function $\mathbf{h}(\mathbf{x}[t-1], \mathbf{u}[t])$ whose i-th component ($i = 1, \ldots, n_Y$) is given by

$$h_i(\mathbf{x}[t-1], \mathbf{u}[t]) = g\left(\sum_{j=1}^{n_X} W_{ij}^{yx} x_j[t-1] + \sum_{j=1}^{n_U} W_{ij}^{yu} u_j[t] + W_i^y\right). \tag{3.11}$$

Jordan (1986) nets may also be formulated as Mealy NSM. Both the next-state and the output function use an auxiliary function $\mathbf{z}(\mathbf{x}[t-1], \mathbf{u}[t])$ whose i-th

[7] Also called *transfer function*, *gain function* and *squashing function* (Hertz et al., 1991, 4).

coordinate is

$$z_i((\mathbf{x}[t-1],\mathbf{u}[t]) = g\left(\sum_{j=1}^{n_X} W_{ij}^{zx} x_j[t-1] + \sum_{j=1}^{n_U} W_{ij}^{zu} u_j[t] + W_i^z\right), \tag{3.12}$$

with $i = 1,\ldots,n_Z$. The i-th coordinate of the next-state function is

$$f_i(\mathbf{x}[t-1],\mathbf{u}[t]) = \alpha x_i[t-1] + g\left(\sum_{j=1}^{n_Z} W_{ij}^{xz} z_j(\mathbf{x}[t-1],\mathbf{u}[t]) + W_i^x\right) \tag{3.13}$$

(with $\alpha \in [0,1]$ a constant) and the i-th coordinate of the output function is

$$h_i(\mathbf{x}[t-1],\mathbf{u}[t]) = g\left(\sum_{j=1}^{n_Z} W_{ij}^{xz} z_j(\mathbf{x}[t-1],\mathbf{u}[t]) + W_i^x\right). \tag{3.14}$$

3.2.2 Neural Moore machines

Elman (1990)'s *simple recurrent net*, a widely-used Moore NSM, is described by a next-state function identical to the next-state function of Robinson and Fallside (1991), eq. (3.10), and an output function $\mathbf{h}(\mathbf{x}[t])$ whose i-th component ($i = 1,\ldots,n_Y$) is given by

$$h_i(\mathbf{x}[t]) = g\left(\sum_{j=1}^{n_X} W_{ij}^{yx} x_j[t] + W_i^y\right). \tag{3.15}$$

However, an even simpler DTRNN is the one used by Williams and Zipser (1989c,a), which has the same next-state function but an output function that is simply a projection of the state vector $y_i[t] = x_i[t]$ for $i = 1,\ldots,n_Y$ with $n_Y \leq n_X$. This architecture is also used in the encoder (or compressor) part of Pollack (1990)'s RAAM (see page 34) when encoding sequences.

The second-order counterpart of Elman (1990)'s simple recurrent net has been used by Blair and Pollack (1997) and Carrasco et al. (1996). In that case, the i-th coordinate of the next-state function is identical to eq. (3.8), and the output function is identical to eq. (3.15).

The second-order DTRNN used by Giles et al. (1992) , Watrous and Kuhn (1992), Pollack (1991) , Forcada and Carrasco (1995), and Zeng et al. (1993) may be formulated as a Moore NSM in which the output vector is simply a projection of the state vector $h_i(\mathbf{x}[t]) = x_i[t]$ for $i = 1,\ldots,n_Y$ with $n_Y \leq n_X$, as in the case of Williams and Zipser (1989c) and Williams and Zipser (1989a). The classification of these second-order networks as Mealy or Moore NSM depends on the actual configuration of feedback weights used by the authors. For example, Giles et al. (1992) use one of the units of the state vector $\mathbf{x}[t]$ as an output unit; this would be a neural Moore machine in which $y[t] = x_1[t]$ (this unit is part of the state vector because its value is also fed back to form $\mathbf{x}[t-1]$ for the next cycle).

3.2.3 Other architectures without hidden state

There are a number of discrete-time neural network architectures that do not have a hidden state (their state is observable because it is simply a combination of past inputs and past outputs) but may still be classified as recurrent. One such example is the NARX (Nonlinear Auto-Regressive with eXogenous inputs) network used by Narendra and Parthasarathy (1990) and then later by Lin et al. (1996) and Siegelmann et al. (1996) (see also (Haykin, 1998, 746)), which may be formulated in state-space form by defining a state that is simply a window of the last n_I inputs and a window of the last n_O outputs. Accordingly, the next-state function simply incorporates a new input (discarding the oldest one) and a freshly computed output (discarding the oldest one) to the windows and shifts each one of them one position. The $n_X = n_I n_U + n_O n_Y$ components of the state vector are distributed as follows:

- The first $n_I n_U$ components are allocated to the window of the last n_I inputs: $u_i[t-k]$ $(k = 0 \dots n_I - 1)$ is stored in $x_{i+kn_U}[t]$;
- The $n_O n_Y$ components from $n_I n_U + 1$ to n_X are allocated to the window of the last n_O outputs: $y_i[t-k]$ $(k = 1 \dots n_O)$ is stored in $x_{n_I n_U + i + (k-1)n_Y}[t]$.

The next-state function $\mathbf{f}$ performs, therefore, the following operations:

- Incorporating the new input and shifting past inputs

$$\begin{array}{lll} f_i(\mathbf{x}[t-1], \mathbf{u}[t]) = & u_i[t], & 1 \le i \le n_U; \\ f_i(\mathbf{x}[t-1], \mathbf{u}[t]) = & x_{i-n_U}[t-1], & n_U < i \le n_U n_I \end{array} \tag{3.16}$$

- Shifting past outputs:

$$f_i(\mathbf{x}[t-1], \mathbf{u}[t]) = x_{i-n_Y}[t-1], \;\; n_U n_I + n_Y < i \le n_X; \tag{3.17}$$

- Computing new state components using an intermediate hidden layer of n_Z units:

$$f_{i+n_U n_i}(\mathbf{x}[t-1], \mathbf{u}[t]) = g\left(\sum_{j=1}^{n_Z} W_{ij}^{xz} z_j[t] + W_i^x\right), \; 1 \le i \le n_Y \tag{3.18}$$

with

$$z_i[t] = g\left(\sum_{j=n_U+1}^{n_X} W_{ij}^{zx} x_i[t-1] + \sum_{j=1}^{n_U} W_{ij}^{zu} u_j[t] + W_i^z\right), \; 1 \le i \le n_Z. \tag{3.19}$$

The output function is then simply

$$h_i(\mathbf{x}[\mathbf{t}]) = x_{i+n_U n_I}[t], \tag{3.20}$$

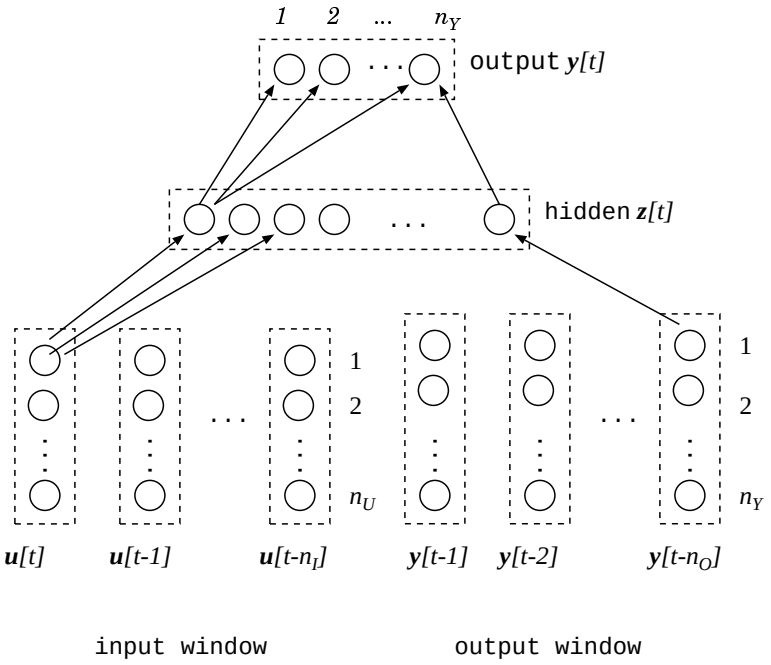

Figure 3.3: Block diagram of a NARX network (the network is fully connected but all arrows have not been drawn for clarity).

with $1 \leq i \leq n_Y$. Note that the output is computed by a two-layer feedforward neural network. The operation of a NARX network N may then be summarized as follows (see figure 3.3):

$$\mathbf{y}[t] = N(\mathbf{u}[t], \mathbf{u}[t-1], \ldots, \mathbf{u}[t-n_I], \mathbf{y}[t-1], \mathbf{y}[t-2], \ldots, \mathbf{y}[t-n_O]). \quad (3.21)$$

Their operation is therefore a nonlinear variation of that of an ARMA (*Auto-Regressive, Moving Average*) model or that of an IIR (*Infinite-time Impulse Response*) filter.

When the state of the discrete-time neural network is simply a window of past inputs, we have a network usually called a *time delay neural network* (TDNN) (see also (Haykin, 1998, 641)). In state-space formulation, the state is simply the window of past inputs and the next-state function simply incorporates a new input to the window and shifts it one position in time:

$$\begin{array}{lll} f_i(\mathbf{x}[t-1], \mathbf{u}[t]) = & x_{i-n_U}[t-1], & n_U < i \leq n_U n_I; \\ f_i(\mathbf{x}[t-1], \mathbf{u}[t]) = & u_i[t], & 1 \leq i \leq n_U, \end{array} \quad (3.22)$$

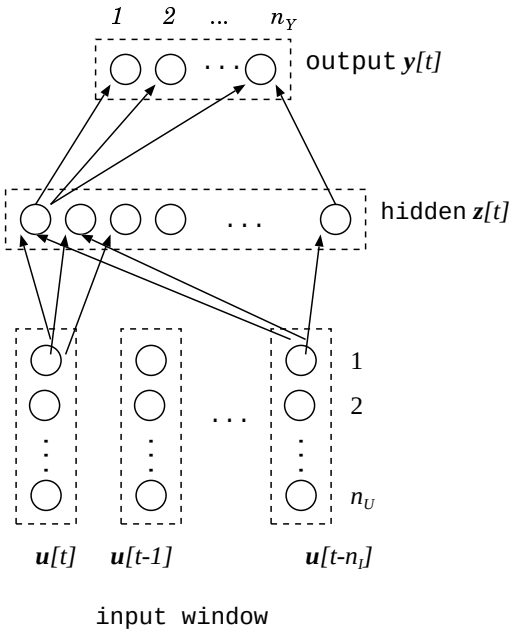

Figure 3.4: Block diagram of a TDNN (the network is fully connected but all arrows have not been drawn for clarity).

with $n_X = n_U n_I$; and the output is usually computed by a two-layer perceptron (feedforward net):

$$h_i(\mathbf{x}[t-1], \mathbf{u}[t]) = g\left(\sum_{j=1}^{n_Z} W_{ij}^{yz} z_j[t] + W_i^y\right),\ 1 \le i \le n_Y \tag{3.23}$$

with

$$z_i[t] = g\left(\sum_{j=1}^{n_X} W_{ij}^{zx} x_i[t-1] + \sum_{j=1}^{n_U} W_{ij}^{zu} u_j[t] + W_i^z\right),\ 1 \le i \le n_Z. \tag{3.24}$$

The operation of a TDNN network N may then be summarized as follows (see figure 3.4):

$$\mathbf{y}[t] = N(\mathbf{u}[t], \mathbf{u}[t-1], \ldots, \mathbf{u}[t-n_I]). \tag{3.25}$$

Their operation is therefore a nonlinear variation of that of an MA (*Moving Average*) model or that of a FIR (*Finite-time Impulse Response*) filter.

The weights connecting the window of inputs to the hidden layer may be organized in blocks sharing weight values, so that the components of the hidden layer retain some of the temporal ordering in the input window. TDNN have

been used for tasks such as phonetic transcription (Sejnowski and Rosenberg, 1987), protein secondary structure prediction (Qian and Sejnowski, 1988), or phoneme recognition (Waibel et al., 1989; Lang et al., 1990). Clouse et al. (1997b) have studied the ability of TDNN to represent and learn a class of finite-state recognizers from examples (see also (Clouse et al., 1997a) and (Clouse et al., 1994))[8].

3.3 Application of DTRNN to sequence processing

DTRNN have been applied to a wide variety of sequence-processing tasks; here is a survey of some of them:

Channel equalization: In digital communications, when a series of symbols is transmitted, the effect of the channel may yield a signal whose decoding may be impossible without resorting to a compensation or reversal of these effects at the receiver side. This sequence transduction task (which converts the garbled sequence received into something as similar as possible to the transmitted signal) is usually known as *equalization*. A number of researchers have studied DTRNN for channel equalization purposes (Kechriotis et al., 1994; Ortiz-Fuentes and Forcada, 1997; Bradley and Mars, 1995; Cid-Sueiro and Figueiras-Vidal, 1993; Cid-Sueiro et al., 1994; Parisi et al., 1997).

Speech recognition: Speech recognition may be formulated either as a sequence transduction task (for example, continuous speech recognition systems aim at obtaining a sequence of phonemes from a sequence of acoustic vectors derived from a digitized speech sample) or as a sequence recognition task (for example, as in isolated-word recognition, which assigns a word in a vocabulary to a sequence of acoustic vectors). Discrete-time recurrent neural networks have been extensively used in speech recognition tasks (Robinson and Fallside, 1991; Robinson, 1994; Bridle, 1990; Watrous et al., 1990; Chiu and Shanblatt, 1995; Kuhn et al., 1990; Chen et al., 1995).

Speech coding: Speech coding aims at obtaining a compressed representation of a speech signal so that it may be sent at the lowest possible bit rate. A family of speech coders are based in the concept of *predictive coding*: if the speech signal at time t may be predicted using the values of the signal at earlier times, then the transmitter may simply send the prediction error

[8]The class of finite-state recognizers representable in TDNN is that of *definite-memory machines* (DMM), that is, finite-state machines whose state is observable because it may be determined by inspecting a finite window of past inputs (Kohavi, 1978). When the state is observable but has to be determined by inspecting both a finite window of past inputs and a finite window of past outputs, we have a *finite-memory machine* (FMM); the neural equivalent of FMM is therefore the NARX network just mentioned.

instead of the actual value of the signal and the receiver may use a similar predictor to reconstruct the signal; in particular, a DTRNN may be used as a predictor. The transmission of the prediction error may be arranged in such a way that the number of bits necessary is much smaller than the one needed to send the actual signal with the same reception quality (Sluijter et al., 1995). Haykin and Li (1995), Baltersee and Chambers (1997), and Wu et al. (1994) have used DTRNN predictors for speech coding.

System identification and control: DTRNN may be trained to be models of time-dependent processes such as a stirred-tank continuous chemical reactor: this is usually referred to as *system identification*. Control goes a step further: a DTRNN may be trained to drive a real system (a "plant") so that the properties of its output follows a desired temporal pattern. Many researchers have used of DTRNN in system identification (Adalı et al., 1997; Cheng et al., 1995; Werbos, 1990; Dreider et al., 1995; Nerrand et al., 1994), and control (Li et al., 1995; Narendra and Parthasarathy, 1990; Puskorius and Feldkamp, 1994; Chovan et al., 1994, 1996; Wang and Wu, 1995, 1996; Zbikowski and Dzielinski, 1995).

Time series prediction: The prediction of the next item in a sequence may be interesting in many other applications besides speech coding. For example, short-term electrical load forecasting is important to control electrical power generation and distribution. Time series prediction is a classical sequence prediction application of DTRNN. See, for example, Connor and Martin (1994); Dreider et al. (1995); Draye et al. (1995); Aussem et al. (1995).

Natural language processing: The processing of sentences written in any natural (human) language may itself be seen as a sequence processing task, and has been also approached with DTRNN. Examples include discovering grammatical and semantic classes of words when predicting the next word in a sentence (Elman, 1991), learning to assign thematic roles to parts of Chinese sentences (Chen et al., 1997), or training a DTRNN to judge on the grammaticality of natural language sentences (Lawrence et al., 1996).

3.4 Learning algorithms for DTRNN

When we want to train a DTRNN as a sequence processor, the usual procedure is to choose the architecture and parameters of the architecture: the number of input lines n_U and the number of output neurons n_Y will usually be determined by the nature of the input sequence itself and by the nature of the processing we want to perform[9]; the number of state neurons n_X will have to be determined through experimentation or used to act as a computational bias restricting the

[9]Although the choice of a particular encoding scheme or a particular kind of preprocessing —such as e.g. a normalization— may indeed affect the performance of the network.

computational power of the DTRNN when we have a priori knowledge about the computational requirements of the task. Since DTRNN are state-based sequence processors (see section 3.1.1), the choice of the number of state units is crucial: the resulting state space has to be ample enough to store all the information about an input sequence that is necessary to produce a correct output for it, assuming that the DTRNN architecture is capable of extracting that information from inputs and computing correct outputs from states; it is also possible to modify the architecture as training proceeds (see e.g. Fahlman (1991)), as will be mentioned later.

Then we train the DTRNN on examples of processed sequences; training a DTRNN as a discrete-time sequence processor involves adjusting its learnable parameters. In a DTRNN these are the weights, biases and initial states[10] ($\mathbf{x}_0$). To train the network we usually need an *error* measure which describes how far the actual outputs are from their desired targets; the learnable parameters are modified to minimize the error measure. It is very convenient that the error is a differentiable function of the learnable parameters which is to be *minimized* (this is usually the case with most sigmoid-like activation functions, as has been discussed in the previous section). A number of different problems may occur when training DTRNN —and, in general, any neural network— by error minimization. These problems are reviewed in section 3.5.

Learning algorithms (also called *training algorithms*) for DTRNN may be classified according to diverse criteria. All learning algorithms (except trivial algorithms such as a random search) implement a heuristic to search the many-dimensional space of learnable parameters for minima of the error function chosen; the nature of this heuristic may be used to classify them. Some of the divisions that will be described in the following may also apply to non-recurrent neural networks.

A major division occurs between *gradient-based* algorithms, which compute the gradient of the error function with respect to the learnable parameters at the current search point and use this vector to define the next point in the search sequence, and *non-gradient-based* algorithms which use other (usually local) information to decide the next point. Obviously, gradient-based algorithms require that the error function be differentiable, whereas most non-gradient-based algorithms may dispense with this requirement. In the following, this will be used as the main division.

Another division relates to the schedule used to decide the next set of learnable parameters. *Batch* algorithms compute the total error function for all of the patterns in the current learning set and update the learnable parameters only after a complete evaluation of the total error function has been performed. *Pattern* algorithms compute the contribution of a single pattern to the error function and update the learnable parameters after computing this contribution. This formulation of the division may be applied to most neural network learning algorithms; however, in the case of DTRNN used as sequence proces-

[10]Learning the initial state is surprisingly not too common in DTRNN literature, because it seems rather straightforward to do so; we may cite the papers by Bulsari and Saxén (1995) Forcada and Carrasco (1995), and Blair and Pollack (1997).

sors, targets may be available not only for a whole sequence (as, for instance, in a classification task) but also for parts of a sequence (as would be the case in a synchronous translation task in which the targets are known after each item of the sequence). In the second case, a third learning mode, *online learning*, is possible: the contribution to the error function of each partial target may be used to update some of the learnable parameters even before the complete sequence has been processed. Online learning is the only possible choice when the learning set consists of a single sequence without a defined endpoint or when patterns can only be presented once.[11]

A third division has already been mentioned. Most learning algorithms for DTRNN do not change the architecture during the learning process. However, there are some algorithms that modify the architecture of the DTRNN while training it (for example, the *recurrent cascade correlation* algorithm by Fahlman (1991) adds neurons to the network during training).

3.4.1 Gradient-based algorithms

The two most common gradient-based algorithms for DTRNN are *backpropagation through time* (BPTT) and *real-time recurrent learning* (RTRL). Most other gradient-based algorithms may be classified as using an intermediate or hybrid strategy combining the desirable features of these two canonical algorithms.

The simplest kind of gradient-based algorithm —used also for feedforward neural networks— is a gradient-descent learning algorithm which updates each learnable parameter p of the network according to the rule

$$p_{\text{new}} = p_{\text{old}} - \alpha_p \frac{\partial E}{\partial p} \tag{3.26}$$

where α_p is a positive magnitude (not necessarily a constant) called the *learning rate* for parameter p and E is either the total error for the whole learning set (as in batch learning) or the error for the pattern just presented (as in pattern learning). Most gradient-based algorithms are improvements of this simple scheme (for details see e.g. (Haykin, 1998, 220,233ff); (Hertz et al., 1991, 103ff,123ff,157)); all of them require the calculation of derivatives of error with respect to all of the learnable parameters. The derivatives for a DTRNN may be computed (or approximated) in different ways, which leads to a variety of methods.

Backpropagation through time

Backpropagation through time (BPTT) may be considered as the earliest learning algorithm for DTRNN. The most commonly used reference for BPTT is the book chapter by Rumelhart et al. (1986), although an earlier description of BPTT may be found in Werbos (1974)'s PhD dissertation (see also Werbos

[11] For a detailed discussion of gradient-based learning algorithms for DTRNN and their modes of application, the reader is referred to an excellent survey by Williams and Zipser (1995), whose emphasis is on continuously running DTRNN.

(1990)). The central idea to BPTT is the *unfolding* of the discrete-time recurrent neural network into a multilayer feedforward neural network (FFNN) each time a sequence is processed. The FFNN has a layer for each "time step" in the sequence; each layer has n_X units, that is, as many as there are state units in the original networks. It is as if we are using time to index layers in the FFNN. Next state is implemented by connecting state units in layer $t-1$ and inputs in time t to state units in layer t. Output units (which are also repeated in each "time step" where targets are available) are connected to state units (and input lines when the DTRNN is a Mealy NSM) as in the DTRNN itself.

The resulting FFNN is trained using the standard backpropagation (BP) algorithm, but with one restriction: since layers have been obtained by replicating the DTRNN over and over, weights in all layers should be the same. To achieve this, BPTT updates all equivalent weights using the sum of the gradients obtained for weights in equivalent layers, which may be shown to be the exact gradient of the error function for the DTRNN.

In BPTT, weights can only be updated after a complete forward step and a complete backward step, just as in regular backpropagation. When processing finite sequences, weights are usually updated after a complete presentation of the sequence.

The time complexity of BPTT is one of its most attractive features: for first-order DTRNN in which the number of states is larger than the number of inputs ($n_X > n_U$), the temporal cost of the backward step used to compute the derivatives grows as n_X^2, that is, the same as the cost of the forward step used to process the sequence and compute the outputs. The main drawback of BPTT is its space complexity, which comes from the need to replicate the DTRNN for each step of the sequence. This also makes it a bit trickier to program than RTRL.

For more details on BPTT the reader is referred to Haykin (1998, 751) and Hertz et al. (1991, 182).

Real-time recurrent learning

Real-time recurrent learning (RTRL) has been independently derived by many authors, although the most commonly cited reference for it is Williams and Zipser (1989b) (for more details see also Hertz et al. (1991, 184) and Haykin (1998, 756)). This algorithm computes the derivatives of states and outputs with respect to all weights as the network processes the sequence, that is, during the forward step. No unfolding is performed or necessary. For instance, if the network has a simple next-state dynamics such as the one described in eq. (3.10), derivatives may be computed together with the next state. The derivative of states with respect to, say, state-state weights at time t, would be computed from the states and derivatives at time $t-1$ and the input at time t as follows:

$$\frac{\partial x_i[t]}{\partial W_{kl}^{xx}} = g'(\Xi_i[t]) \left(\delta_{ik} x_l[t] + \sum_{j=1}^{n_X} W_{ij}^{xx} \frac{\partial x_j[t-1]}{\partial W_{kl}^{xx}} \right), \tag{3.27}$$

with $g'()$ the derivative of the activation function, δ_{ik} Kronecker's delta (1 if $i = k$ and zero otherwise) and

$$\Xi_i[t] = \sum_{j=1}^{n_X} W_{ij}^{xx} x_j[t-1] + \sum_{j=1}^{n_U} W_{ij}^{xu} u_j[t] + W_i^x \tag{3.28}$$

the net input to state unit i. The derivatives of states with respect to weights at $t = 0$ are initialized to zero.[12]

Since derivatives of outputs are easily defined in terms of state derivatives for all architectures, the learnable parameters of the DTRNN may be updated after every time step in which output targets are defined, (using the derivatives of the error for each output), therefore even after having processed only part of a sequence. This is one of the main advantages of RTRL in applications where online learning is necessary; the other one is the ease with which it may be derived and programmed for a new architecture; however, its time complexity is much higher than that of BPTT; for first-order DTRNNs such as the above with more state units than input lines ($n_X > n_U$) the dominant term in the time complexity is n_X^4. A detailed derivation of RTRL for a second-order DTRNN architecture may be found in (Giles et al., 1992).

The reader should be aware that the name RTRL (Williams and Zipser, 1989c) is applied to two different concepts: it may be viewed solely as a method to compute the derivatives or as a method to compute derivatives *and* to update weights (in each cycle). One may use RTRL to compute derivatives and update the weights after processing a complete learning set made up of a number of sequences (batch update), after processing each sequence (pattern update), and after processing each item in each sequence (online update). In these last two cases, the derivatives are not exact but approximate (they would be exact for a zero learning rate). For batch and pattern weight updates, RTRL and BPTT are equivalent, since they compute the same derivatives. The reader is referred to Williams and Zipser (1995) for a more detailed discussion.

Other derivative-based methods

It is also possible to train DTRNN using the *extended Kalman filter* (EKF, see e.g. (Haykin, 1998, 762ff)) of which RTRL may be shown to be a special case (Williams, 1992); the EKF has been successfully used in many applications, such as neurocontrol (Puskorius and Feldkamp, 1994). The EKF is also related to RLS (*recursive-least squares*) algorithms.

3.4.2 Non-gradient methods

Gradient-based algorithms are the most used of all learning algorithms for DTRNN. But there are also some interesting non-gradient-based algorithms,

[12] Derivatives with respect to the components of the initial state $\mathbf{x}[0]$ may also be easily computed (Forcada and Carrasco, 1995; Bulsari and Saxén, 1995; Blair and Pollack, 1997), by initializing them accordingly (that is, $\partial x_i[0]/\partial x_j[0] = \delta_{ij}$).

most of which rely on weight perturbation schemes. Of those, two batch learning algorithms are worth mentioning:

- Alopex (Unnikrishnan and Venugopal, 1994) is a batch learning algorithm that biases random weight updates according to the observed correlation between previous updates of each learnable parameter and the change in the total error for the learning set. It does not need any knowledge about the network's particular structure; that is, it treats the network as a black box, and, indeed, it may be used to optimize parameters of systems other than neural networks; this makes it specially attractive when it comes to test a new architecture for which derivatives have not been derived yet. Alopex has only found limited use so far in connection with DTRNN (but see Forcada and Ñeco (1997) or Ñeco and Forcada (1997)).

- The algorithm by Cauwenberghs (1993) (see also (Cauwenberghs, 1996)) uses a related learning rule: the change effected by a random perturbation π of the weight vector $\mathbf{W}$ on the total error $E(\mathbf{W})$ is computed and weights are updated in the direction of the perturbation so that the new weight vector is $\mathbf{W} - \mu E(\mathbf{W} + \pi) - E(\mathbf{W}))\pi$, where μ acts as a learning rate.. Cauwenberghs (1993) shows that this algorithm performs gradient descent on average when the components of the weight perturbation vector are mutually uncorrelated with uniform auto-variance, with error decreasing in each epoch for small enough π and μ, and with a slowdown with respect to gradient descent proportional to the square root of the number of parameters.

3.4.3 Architecture-coupled methods

A number of learning algorithms for DTRNN are coupled to a particular architecture: for example, BPS (Gori et al., 1989) is a special algorithm used to train local feedback networks, that is, DTRNN in which the value of a state unit $x_i[t]$ is computed by using only its previous value $x_i[t-1]$ but not the rest of the state values $x_j[t-1], j \neq i$. Local-feedback DTRNN using threshold linear units and having a two-layer output network capable of performing any Boolean mapping have recently been shown (Frasconi et al., 1996) to be capable of recognizing only a subset of regular languages, and to be incapable of emulating all FSM (Kremer, 1999). A related algorithm is *focused backpropagation*(Mozer, 1989). Learning algorithms are also very simple when states are observable (such as in NARX networks, see section 3.2.3), because, during learning, the desired value for the state may be fed back instead of the actual value being computed by the DTRNN; this is usually called *teacher forcing*..

But sometimes not only learning algorithms are specialized on a particular architecture but it is also the case that the algorithm modifies the architecture during learning. One such algorithm is Fahlman's (Fahlman, 1991) *recurrent cascade correlation*, which is described in the following section.

Recurrent cascade correlation

Fahlman (1991) has recently proposed a learning algorithm that establishes a mechanism to grow a DTRNN during training by adding hidden state units which are trained separately so that their output does not affect the operation of the DTRNN. Training starts with an architecture without hidden state units,

$$y_i[t] = g\left(\sum_{j=1}^{n_U} W_{ij}^{yu} u_j[t] + W_i^y\right), i = 1 \dots n_Y, \tag{3.29}$$

and a pool of n_C candidate hidden units with local feedback which are connected to the inputs are trained to follow the residual error of the network:

$$x_i[t] = g\left(\sum_{j=1}^{n_U} W_{ij}^{xu} u_j[t] + W_{ii}^{xx} x_i[t-1] + W_i^x\right) \tag{3.30}$$

with $i = 1 \dots n_C$. Training adds the best candidate unit to the network in a process called *tenure*. If there are already n_H tenured hidden units, the state of candidate i is

$$x_i[t] = g\left(\sum_{j=1}^{n_U} W_{ij}^{xu} u_j[t] + W_{ii}^{xx} x_i[t-1] + \sum_{j=1}^{n_H} W_{ij}^{xx'} x_j[t] + W_i^x\right) \tag{3.31}$$

(the prime in $W_{ij}^{xx'}$ meaning that it weights state values at time t, not $t-1$ as usual). Tenure adds the best of the candidates to the network as a hidden unit labeled $n_H + 1$ (where n_H is the number of existing hidden units), its incoming weights are frozen and connections are established with the output units and subsequently trained. Therefore, hidden units form a lower-triangular structure in which each of the units receives feedback only from itself (local feedback)and the output is computed from the input and each of the hidden units:

$$y_i[t] = g\left(\sum_{j=1}^{n_U} W_{ij}^{yu} u_j[t] + \sum_{j=1}^{n_H} W_{ij}^{yx} x_j[t] + W_i^y\right), i = 1 \dots n_Y. \tag{3.32}$$

Recurrent cascade correlation networks have recently been shown to be incapable of recognizing certain classes of regular languages (see section 4.2.3).

3.5 Learning problems

When it comes to train a DTRNN to perform a certain sequence processing task, the first thing that should be checked is whether the DTRNN architecture chosen can actually represent or approximate the task that we want to learn. However, this is seldom possible, either because of our incomplete knowledge of the computational nature of the sequence processing task itself or because of

our lack of knowledge about the tasks that a given DTRNN architecture can actually perform. In most of the following, I will be assumed that the DTRNN architecture (including the representation used for inputs, the interpretation assigned to outputs and the number of neurons in each layer) has already been chosen and that further learning may only occur through adjustment of weights, biases and similar parameters. We will review some of the problems that may occur during the adjustment of these parameters.

Some of the problems may appear regardless of the kind of learning algorithm used, and others may be related to gradient-based algorithms.

Multiple minima: The error function for a given learning set is usually a function of a relatively large number of learnable parameters. For example, a rather small DTRNN, say, an Elman net with two inputs, two output units, and three state units has 21 weights, 5 biases and, in case we decide to adjust them, 3 initial state values. Assume we have already found a minimum in the error surface. Due to the structure of connections, choosing any of the 6 possible permutations of the 3 state neurons would yield exactly the same value for the error function. But, in addition to this, it is very likely that the 26-dimensional space of weights and biases is plagued with local minima, some of which may actually not correspond to the computational task we want to learn. Since it is not feasible for any learning algorithm to sample the whole 26-dimensional space, the possibility that it finds a suboptimal minimum of the error function is very high. This problem is especially important with local-search algorithms such as gradient descent: if the algorithm slowly modifies the learnable parameters to go downhill on the error surface, it may end up trapped in any local minimum. The problem of multiple minima is not a specific problem of DTRNN; it affects almost all neural-network architectures.

Long-term dependencies: The problem of long-term dependencies, or the problems when training DTRNN to perform tasks in which a late output depends on a very early input which has to be remembered, is more specific to DTRNN, because it is a sequence-processing problem; one of the most exhaustive studies of this problem has been done by Bengio et al. (1994) (see (Haykin, 1998, 773)). See section 3.6 for a discussion.

3.6 Papers

Four papers are featured in this chapter:

Jordan (1986): Jordan's paper, "Serial Order: A Parallel Distributed Processing Approach", describes how a particular recurrent neural network architecture, described in page 3.2.1, which almost everyone calls a Jordan net now, may be trained to generate sequences of vectors corresponding to various tasks. Inputs to Jordan's nets are constant during the generation of a whole sequence

and are aptly called the "plan" for that sequence: the net is taught to produce a different sequence for each plan.

State in Jordan nets is not hidden but fully observable (what Jordan (1986) calls *hidden units* are used as an intermediate step for the computation of the output, see eq. 3.12). Therefore, it is possible to train them using *teacher forcing* (see section 3.4.3). Jordan finds that teacher forcing favors fast learning.

Jordan also studies the behavior of trained DTRNN when their state is perturbed and observes that learned sequences are attractors for the DTRNN. For example, if the DTRNN has been trained to produce an oscillating output pattern, it is attracted towards the corresponding limit cycle of the network. When the vectors in the desired output sequence are not completely specified but only some of their components have target values, Jordan shows that the use of "don't care" conditions for the remaining coordinates favors the learning of "anticipation" behaviors: components not specified during learning tend to adopt values which approach the specified value for the component in the closest future vector of the sequence. The results are discussed in the context of the simulation of coarticulatory features of speech, that is, inter-phoneme influences. Finally, Jordan also studies the modelling of dual tasks, that is, tasks that are learned separately but have to be performed simultaneously.

Elman (1990): Elman's paper , "Finding structure in time" (`http://www.dlsi.ua.es/~mlf/nnafmc/papers/elman90finding.pdf`), introduces another widely-used recurrent architecture, the *simple recurrent net*, which everyone calls now an Elman net (see section 3.2.2); previous state $\mathbf{x}[t-1]$ is called *context* —in view of the fact that they try to encode information about the inputs seen so far, $\mathbf{u}[1]\dots\mathbf{u}[t-1]$— and current state $\mathbf{x}[t]$ is said to be stored in *hidden* units. Instead of using BPTT or RTRL, the networks are trained using simple backpropagation in each time step without considering the recurrent effects of each weight on the values of context units. Elman studies the performance of this network and the nature of the representations learned by the network when it is trained to perform four sequence prediction tasks:

- Predicting the next bit in a sequence in which every third bit is the exclusive or of the previous two, which are randomly chosen; the error of the trained network drops every three cycles, when the current bit may be predicted from past inputs.

- Predicting the next letter in a random sequence of the three syllables *ba*, *dii* and *guu* where letters are represented by binary vectors representing their articulatory (phonetic) features. The network learns to predict the vowels from the consonants and also the fact that a consonant follows the vowels, even if it is impossible to predict which one.

- Predicting the next letter in a sequence of concatenated words (without blanks) from a 15-word lexicon; letters are represented by random 5-bit vectors. As a result, prediction error gracefully falls inside words and

raises at the end of the word, where the letter starting the next word cannot be predicted: the network learns to predict word boundaries.

- Predicting the next word in a concatenated sequence of two- and three-word sentences; each words is represented by a randomly-assigned binary vector having a single bit on (*one-hot* encoding). Hierarchical clustering studies of the hidden unit activation patterns show that the net has developed representations of words that correspond to lexical classes (noun, verb) and subclasses (transitive verb, animate noun), etc. simply by learning the sequential layout of words.

In all cases, Elman (1990) nets learn the temporal structure present in the sequences of events they are trained to predict.

Pollack (1990): The paper by Pollack, "Recursive distributed representations" (`http://www.dlsi.ua.es/~mlf/nnafmc/papers/pollack90recursive.pdf`) introduces a new architecture, which is nowadays called recursive auto-associative memory (RAAM). When used to process sequences, the system is basically a set of two discrete-time recurrent neural networks, the encoder (or *compressor*) and the decoder (or *reconstructor*) . The encoder is trained to produce a different final state vector for each sequence, so that the trained decoder may trace back the steps and retrieve the sequences from the states; therefore, a distributed representation of sequences is achieved. But RAAMs are more general devices because they may be used not only to obtain distributed representations of sequences, but also of trees with a maximum valence.[13]

A RAAM Z, that is, a recursive auto-associative memory (Pollack, 1990) with valence V is a tree-storing device composed by two different subsystems:

- the *encoder* $E = (X, U, V, \mathbf{f})$, where $X = [S_0, S_1]^{n_X}$ is the *state space* of the RAAM, with S_0 and S_1 in $\mathbb{R}$, n_X the *order* (number of state units), $U = [S_0, S_1]^{n_U}$ the set of possible inputs with n_U the number of input signals; and $\mathbf{f} : X^V \times U \to X$ is the *encoding function*;

- the *decoder* $D = (X, U, V, \mathbf{h}, \mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_V)$ where $\mathbf{h} : X \to U$ is the output function and the V functions $\mathbf{d}_i : X \to X$, $i = 1, \ldots V$ are the *decoding* functions.

Encoding, decoding, and output functions are realized in RAAM by feedforward neural networks, usually single-layer feedforward neural networks with neurons whose outputs are in the interval $[S_0, S_1]$. The usual choices for S_0 and S_1 are $S_0 = 0$ and $S_1 = 1$ when the activation function of neurons is the logistic function $g_L(x) = 1/(1 + \exp(-x))$.

RAAMs may be used as tree-storing devices which store trees of maximum valence V as follows:

[13] The *valence* (also *arity*) of a tree is the maximum number of daughters that can be found in any of its nodes.

- A special value of X, which we will call $\mathbf{x}_0$ is used to represent the missing daughters of a node in an input tree when there are less than V daughters.

- Each possible node label σ_k in the set $\Sigma = \sigma_1, \sigma_2 \ldots, \sigma_{|\Sigma|}$ of tree node labels is assigned a value $\mathbf{u}_k \in U$.

- For each of the decoding functions $\mathbf{d}_i$, a special region of X, which will be called $X_\epsilon^{(i)}$, is defined such that when the output of the decoding function is in that region the RAAM is interpreted as designating a missing daughter (for a node having less than V daugthers). This is necessary for the RAAM to produce finite-sized trees as outputs (they have to end in nodes having no daughters)[14].

- Each possible output node label σ_m is also assigned a nonempty region $U_m \in U$ such that when the result of the output function is in U_m the RAAM is interpreted as outputting a node with label σ_m.

- The encoder E walks the input tree in a bottom-up fashion, computing a state value in X for each possible subtree from the state values corresponding to its daughter subtrees; that is, it *encodes* the tree as a vector in X.

- the decoder D generates the output tree in a top-down function, generating, from the state representation of each output node, suitable state representations in X for its daughter nodes and suitable labels in U; that is, it *decodes* the vector obtained by the encoder to produce a tree.

Pollack (1990) trained RAAMs to store the trees in a learning set.

It has to be noted that, in principle, RAAMs may be used to store trees even when labels are not taken from a finite alphabet of symbols but instead consist of arbitrary vectors in U, but Pollack (1990) emphasizes symbolic computations.

RAAMs have been used for various tasks, most of them related to language processing:

- for translating sentences from one language to another, by training RAAMs to represent the source and target sentences and then either by training a multilayer perceptron to obtain the RAAM representation for the target sentence from the RAAM representation of the source sentence (Chalmers, 1990) or by training the two RAAM so that the corresponding representations are identical (Chrisman, 1991).

- More recently, RAAM have been extended to RHAM (*recursive heteroassociative memories*) which learn to obtain representations of input trees that are directly decoded into a different output tree (Forcada and Ñeco, 1997).

[14] Stolcke and Wu (1992) added a special unit to the state pattern which is an indicator showing whether the pattern is a terminal representation. Another possibility would be to add a special output neuron.

- Kwasny and Kalman (1995) have used Elman (1990) nets to obtain, from a sentence, a RAAM representation of its parse tree.
- Sperduti (1994) has introduced *labeling RAAMs* which may be used to store directed labeled graphs (Sperduti, 1994; Sperduti and Starita, 1995; Sperduti, 1995).

Bengio et al. (1994): This paper `http://www.dlsi.ua.es/~mlf/nnafmc/papers/bengio94learning.pdf` discusses the problem of long-term dependencies, a problem which is specific to DTRNN-like sequence processing devices and may be formulated as follows: when the sequence processing task is such that the output after reading a relatively long sequence depends on details of the early items of the sequence, it may occur that learning algorithms are unable to acknowledge this dependency due to the fact that the actual output of the DTRNN at the current time is very insensitive to small variations in the early input, or, what is equivalent, to the small variations in the weights involved in the early processing of the event (even if the change in the early input is large); this is known as the problem of vanishing gradients (see also (Haykin, 1998, 773)). Small variations in weights are the *modus operandi* of most learning algorithms, in particular, but not exclusively, of gradient-descent algorithms. Bengio et al. (1994) prove that the vanishing of gradients is specially severe when we want the DTRNN to robustly store information about a very early effect. The paper also presents a series of experiments in which the performance of alternate DTRNN learning methods is evaluated for three simple single-input single-output problems having long-term dependencies; the experiments show a partial success of some of them.

Chapter 4

Computational capabilities of DTRNN

This chapter reviews the computational capabilities of discrete-time recurrent neural networks and features a number of relevant papers. Section 4.1 reviews some concepts of formal language theory. Section 4.2 studies under which conditions DTRNN behave as finite-state machines, and introduces two main groups of featured papers: those discussing the construction of FSM with, on the one hand, DTRNN which have threshold linear units, and, on the other hand, DTRNN which have sigmoid units. Section 4.3 discusses the comparative capacity of DTRNN and a cornerstone computational model, the Turing machine, and introduces a third group of featured papers dealing with this subject.

4.1 Overview of formal language theory: Languages, grammars and automata

This section reviews some of the basic results of the formal theory of languages and computation (for further information, the reader is referred to books on the subject such as Hopcroft and Ullman (1979), Salomaa (1973) or Lewis and Papadimitriou (1981)), and in particular, the relation between languages, grammars and automata.

4.1.1 Grammars and Chomsky's hierarchy.

Grammars

Grammars provide a way to define languages by giving a finite set of rules that describe how the valid strings may be constructed. A *grammar* G consists of: an alphabet Σ of *terminal symbols* or *terminals*, a finite set of *variables* V, a

set of *rewrite rules* P or *productions*, and a *start symbol* S (a variable):

$$G = (V, \Sigma, P, S).$$

Rewrite rules or productions consist of a *left-hand side* α and a *right-hand side* β: $\alpha \rightarrow \beta$. Their meaning is: replace α with β.

Left-hand sides α are strings over $V \cup \Sigma$, containing *at least one* variable from V: $\alpha \in (V \cup \Sigma)^* V (V \cup \Sigma)^*$. [1] Right-hand sides are strings over $V \cup \Sigma$: $\beta \in (V \cup \Sigma)^*$

The grammar generates strings in Σ^* by applying rewrite rules to the start symbol S until no variables are left. Each time a rule is applied, a new *sentential form* (string of variables from V and terminals from Σ) is produced. For each rule $\alpha \rightarrow \beta$, any occurence of a left-hand side α as a subscript of the sentential form may be substituted by β to form a new sentential form.

The language generated by the grammar, $L(G)$, is the set of all strings that may be generated in that way. Recursive rewrite rules, that is, those which have the same variable in both the left-hand and the right-hand side of a rule lead to infinite languages (if the grammar has no useless symbols).[2]

As an example, consider the following grammar,

$$G = (V, \Sigma, P, S)$$

$$V = \{S, A\}$$

$$\Sigma = \{\mathtt{a}, \mathtt{m}, \mathtt{o}, \mathtt{t}\}$$

$$P = \left\{ \begin{array}{llll} (1) & S & \rightarrow & \mathtt{to}A\mathtt{to} \\ (2) & A & \rightarrow & \mathtt{ma} \\ (3) & A & \rightarrow & \mathtt{ma}A \end{array} \right\}$$

which generates the language

$$L(G) = \{\mathtt{tomato}, \mathtt{tomamato}, \mathtt{tomamamato} \ldots\}.$$

The generation of the string `tomamamato` would proceed as follows:

$$S \Rightarrow_1 \mathtt{to}A\mathtt{to} \Rightarrow_3 \mathtt{toma}A\mathtt{to} \Rightarrow_3 \mathtt{tomama}A\mathtt{to} \Rightarrow_2 \mathtt{tomamamato}$$

where the subscripted arrows refer to the application of particular rules in G.

4.1.2 Chomsky's hierarchy of grammars

Grammars are usually classified according to a hierarchy established by Chomsky (1965) (see also (Hopcroft and Ullman, 1979) or Salomaa (1973, 15)), according to the form of their productions. Each of the levels in the hierarchy has a corresponding automaton class:

[1] Here, an abbreviated notation for the concatenation of languages, $L_1 L_2 = \{w_1 w_2 | w_1 \in L_1,\ w_2 \in L_2\}$, is used.

[2] This occurs also when a string containing the variable may be derived in more than one step from another string containing it, that is, when recursion is indirect.

- Type 0 or *unrestricted* (all possible grammars). Languages generated are *recognized* by *Turing machines* (automata that read from and write on an endless tape, which will be defined in section 4.3.1). *Recognizing* a string is outputting "yes" if the string belongs to the language.

- Type 1 or *context-sensitive*: β never shorter than α, $|\alpha| \leq |\beta|$ (except for $S \rightarrow \epsilon$). Languages generated are recognized by *linearly-bounded automata* (a subclass of Turing machines, see section 4.3.1).

- Type 2 or *context-free*: α is a single variable ($\alpha \in V$). The class of languages generated by type 2 grammars is exactly the class of languages recognized by *pushdown automata* (PDA, Hopcroft and Ullman (1979, 110), Salomaa (1973, 34)), which may be seen as finite-state machine which can push symbols into and pop symbols from an infinite stack. A (nondeterministic) *pushdown automaton* is a 7-tuple $M = (Q, \Sigma, \Phi, \delta, q_I, \phi_0, F)$ where:

  - Q is the finite set of states, with $q_I \in Q$ the initial state;
  - Σ is the finite input alphabet;
  - Φ is a finite set of symbols which may be stored in the stack, which is not initially empty, but instead contains a special symbol ϕ_I;
  - $F \subseteq Q$ is the subset of *accepting states*.
  - δ is the next-state or transition function of the automaton, which maps $Q \times \Sigma^* \times \Phi$ into finite subsets of $Q \times \Phi^*$; that is, when the PDA is in a state, reads in a string, pops a symbol from the stack, changes state, and pushes zero or more symbols into the stack, nondeterministically choosing the next state and the symbols pushed from a finite set of choices.

  The PDA is said to accept an input string from Σ^* when it is led to at least one accepting state in F. An alternate acceptance criterion prescribes an empty stack at the end of the processing.

- Type 3 or *regular*: $\alpha \in V$, β contains at most *one* variable at the right end. Languages generated are recognized by *deterministic finite-state automata* (see section 2.3.3).

4.2 Discrete-time recurrent neural networks behaving as finite-state machines

Discrete-time recurrent neural networks (DTRNN) may be constructed using either threshold linear units (TLU) or units showing a continuous response such as sigmoid units. DTRNN using TLU were actually the earliest models of finite-state machines (FSM), as has been discussed in chapter 2, but in recent times there have been interesting theoretical advances in the representation of FSM

in DTRNN; the corresponding papers are discussed in section 4.2.1. Sigmoid-based DTRNN have also recently been proven (by construction) to be able to behave as FSM; section 4.2.2 discusses the main issues involved and features three representative papers in the field.

4.2.1 DTRNN based on threshold units

The first four papers featured in this chapter (Alon et al., 1991; Goudreau et al., 1994; Kremer, 1995; Horne and Hush, 1996) deal with the implementation of finite-state machines such as deterministic finite-state automata (DFA, section 2.3.3) or Mealy machines (section 2.3.1) in DTRNN made up of threshold linear units. The first such construction was proposed by (Minsky, 1967) (see chapter 2), and used a number of TLU which grew linearly with the number of states $|Q|$ in the FSM. Two of the papers featured in this section (Alon et al., 1991; Horne and Hush, 1996) explore the possibility of encoding $|Q|$-state automata in less than $O(|Q|)$ units.[3]

Kremer (1995): This paper generalizes the result by Minsky (1967) by showing that TLU-based Elman (1990) nets may actually represent finite automata using $\Theta(|Q||\Sigma|)$ states,[4] but the construction by Kremer (1995) allows for input and output symbols to be represented by arbitrary patterns of '0' and '1' activations (instead of being represented by exclusive or *one-hot* patterns having precisely one '1'). This has the effect of reducing the number of learnable parameters in the input–state ($\mathbf{W}^{xu}$) and the state–output ($\mathbf{W}^{yx}$) weight matrices (see section 3.2.2).

Alon et al. (1991): The work by Alon et al. (1991) (`http://www.dlsi.ua.es/~mlf/nnafmc/papers/alon91efficient.pdf`) stems from the following consideration: if n_X TLUs may be found in 2^{n_X} different states, "one might wonder if an $|Q|$-state FSM could be implemented in a network with $O(\log|Q|)$ nodes". These authors set out to establish bounds on the smallest number of TLU needed to implement a binary Mealy machine, that is, a Mealy machine with binary input and output alphabets ($\Sigma = \Gamma = \{0, 1\}$), using a single-layer first-order DTRNN architecture which is basically a neural Moore machine (section 3.2.2) with threshold linear units. The results are far above the optimistic $O(\log|Q|)$. The main result presented by Alon et al. (1991) is that the number of units necessary to implement a $|Q|$-state Mealy machine in such an architecture is $\Omega((|Q|\log|Q|)^{1/3})$ (lower bound to the complexity) and $O((|Q|)^{3/4})$ (upper bound to the complexity) when no restrictions whatsoever are imposed

[3] As usual, $O(F(n))$, with $F : \mathbb{N} \to \mathbb{R}^+$, denotes the set of all functions $f : \mathbb{N} \to \mathbb{R}$ such that, for any n_0 there exists a positive constant K such that, for all $n > n_0$, $f(n) \leq KF(n)$ (upper bound). Similarly, a lower bound may be defined; $\Omega(F(n))$ denotes the set of all functions $f : \mathbb{N} \to \mathbb{R}$ such that, for any n_0 there exists a positive constant Q such that, for all $n > n_0$, $f(n) \geq QF(n)$.

[4] $\Theta(F(n))$, with $F : \mathbb{N} \to \mathbb{R}^+$, is the set of all functions $f : \mathbb{N} \to \mathbb{R}$ belonging both to $O(F(n))$ and $\Omega(F(n))$ (see the preceding footnote).

on the fan-in or the fan-out of the units or on the values of weights. To obtain the lower bound, they use a counting argument based on a lower bound on the number of "really different" Mealy machines of $|Q|$ states and a lower bound on the number of different Mealy machines that may be built using n_X or less TLU. The upper bound is obtained by construction but details are not given in the current paper. However, reasonable ("mild") restrictions on fan-ins, fan-outs and weights lead to equal upper ($O(|Q|)$) and lower ($\Omega(|Q|)$) bounds.

Horne and Hush (1996): The authors of this more recent paper (`http://www.dlsi.ua.es/~mlf/nnafmc/papers/horne96bounds.pdf`) try to improve these bounds by using a different DTRNN architecture, which, instead of using a single-layer first-order feedforward neural network for the next-state function, uses a *lower-triangular* network, that is, a network in which a unit labeled i receives inputs only from units with lower labels ($j < i$); lower triangular networks include layered feedforward networks as a special case. If no restrictions are imposed on weights, the lower and upper bounds are identical and the number of units is $\Theta(|Q|^{1/2})$ (the bound being better than the one obtained by Alon et al. (1991)). When thresholds and weights are restricted to the set $\{-1, 1\}$, the lower and upper bounds are again the same, but better: $\Theta((|Q| \log |Q|)^{1/2})$. When limits are imposed on the fan-in, the result by Alon et al. (1991) is recovered: $\Theta(|Q|)$.

Goudreau et al. (1994), These authors (`http://www.dlsi.ua.es/~mlf/nnafmc/papers/goudreau94first.pdf`) study whether simple single-layer DTRNN (neural Moore machines, see section 3.2.2) with threshold linear units (TLU) are capable of representing finite-state recognizers or DFA. Their study shows that,

- when the output $\mathbf{y}[t]$ of the neural Moore machine is simply a projection of the state $\mathbf{x}[t]$, first-order DTRNN are not capable of representing some DFA, whereas second-order DTRNN may represent any DFA, and
- when the output is computed by a single-layer feedforward neural network (as in Elman (1990) nets), then first-order DTRNN may represent any DFA provided that some of the states (not all of them) in the DFA are represented by more than one state unit in the DTRNN, which Goudreau et al. (1994) call *state-splitting* (see section 4.2.2).

Complete splitting would lead to $n_X = |Q||\Sigma|$ state units, as in the construction by Minsky (1967) , but the authors show an example where less units are used. With second-order networks, state-splitting is not necessary and $n_X = |Q|$ units are sufficient.

4.2.2 DTRNN based on sigmoid units

Much of the work performed by researchers in the contact area between neural networks and formal language and computation theory concerns the training of

sigmoid-based DTRNN to identify or approximate finite-state machines, and, in particular, regular language acceptors such as deterministic finite-state automata (DFA). This is the subject of chapter 5. Most of this work starts by assuming that a given DTRNN architecture is capable of performing the same computation as a finite-state machine (FSM). This section addresses the following question: "When does a DTRNN behave as a FSM?"

In a recent paper, Casey (1996) has shown that a DTRNN performing a robust DFA-like computation (a special case of FSM-like computation) must organize its state space in mutually disjoint, closed sets with nonempty interiors corresponding to the states of the DFA. These states can be taken to be all of the points such that if the DTRNN is initialized with any of them, the DTRNN will produce the same output as the DFA initialized in the corresponding state. This formulation, which is closely connected to what Pollack (1991) called a *dynamical recognizer*, has inspired the following definition. A DTRNN $N = (X, U, Y, \mathbf{f}, \mathbf{h}, \mathbf{x}_0)$ behaves as a FSM $M = (Q, \Sigma, \Gamma, \delta, \lambda, q_I)$ when the following conditions are held:

Partition of the state space: Each state $q_i \in Q$ is assigned a nonempty region $X_i \subseteq X$ such that the DTRNN N is said to be in state q_i at time t when $\mathbf{x}[t] \in X_i$. Accordingly, these regions must be disjoint: $X_i \cap X_j = \emptyset$ if $q_i \neq q_j$. Note that there may be regions of X that are not assigned to any state.

Representation of input symbols: Each possible input symbol $\sigma_k \in \Sigma$ is assigned a different vector $\mathbf{u}_k \in U$ (it would also be possible to assign a *region* $U_k \subseteq U$ to each symbol).

Interpretation of output: Each possible output symbol $\gamma_m \in \Gamma$ is assigned a nonempty region $Y_m \subseteq Y$ such that the DTRNN N is said to output symbol γ_m at time t when $\mathbf{y}[t] \in Y_m$. Analogously, these regions must be disjoint: $Y_m \cap Y_n = \emptyset$ if $\gamma_m \neq \gamma_n$. Note that there may be regions of Y that are not assigned to any output symbol.

Correctness of the initial state: The initial state of the DTRNN N, belongs to the region assigned to the initial state q_I, that is, $\mathbf{x}_0 \in X_I$.

Correctness of the next-state function: For any state q_j and symbol σ_k of M, the transitions performed by the DTRNN N from any point in the region of state space X_j assigned to state q_j when symbol σ_k is presented to the network must lead to points that belong to the region X_i assigned to $q_i = \delta(q_j, \sigma_k)$; formally, this may be expressed as

$$\mathbf{f}_k(X_j) \subseteq X_i \quad \forall q_j \in Q, \sigma_k \in \Sigma : \delta(q_j, \sigma_k) = q_i \tag{4.1}$$

where the shorthand notation

$$\mathbf{f}_k(A) = \{\mathbf{f}(\mathbf{x}, \mathbf{u}_k) : \mathbf{x} \in A\} \tag{4.2}$$

has been used.

Correctness of output: In the case of Mealy NSM, for any state q_j and symbol σ_k of M, the output produced by the DTRNN N from any point in the region of state space X_j assigned to state q_j when symbol σ_k is presented to the network belongs to the region Y_m assigned to $\gamma_m = \lambda(q_j, \sigma_k)$; formally, this may be expressed as

$$\mathbf{h}_k(X_j) \subseteq Y_m \quad \forall q_j \in Q, \sigma_k \in \Sigma : \lambda(q_j, \sigma_k) = \gamma_m \tag{4.3}$$

where the shorthand notation

$$\mathbf{h}_k(A) = \{\mathbf{h}(\mathbf{x}, \mathbf{u}_k) : \mathbf{x} \in A\} \tag{4.4}$$

has been used. In the case of Moore NSM, for any state q_j, the output produced by the DTRNN N from any point in the region the condition for the correctness of the output may be expressed as:

$$\mathbf{h}(X_j) \subseteq Y_m \quad \forall q_j \in Q : \lambda(q_j) = \gamma_m, \tag{4.5}$$

with $\mathbf{h}(A) = \{\mathbf{h}(\mathbf{x}) : \mathbf{x} \in A\}$.

Note that the regions $X_i \subseteq X$, $i = 1, \ldots, |Q|$ and $Y_m \subseteq Y$, $m = 1, \ldots, |\Gamma|$ may have a nondenumerable [5] number of points, because of being subsets of $\mathbb{R}^n$ for some n. However, for a finite input alphabet Σ, only a denumerable number of points in the state (X) and output (Y) spaces are actually visited by the net for the set of all possible finite-length input strings over Σ, denoted Σ^*, which is also denumerable.

Note also that DFA represent a special case: as said in section 2.3.3, deterministic finite-state automata may be seen as Moore or Mealy machines having an output alphabet $\Gamma = \{\mathcal{Y}, \mathcal{N}\}$ whose output is only examined after the last symbol of the input string is presented, and it is such that, for a Moore machine,

$$\lambda(q_i) = \begin{array}{l} \mathcal{Y} \text{ if } q_i \in F \\ \mathcal{N} \text{ otherwise,} \end{array} \tag{4.6}$$

and for a Mealy machine,

$$\lambda(q_i, \sigma_k) = \begin{array}{l} \mathcal{Y} \text{ if } \delta(q_i, \sigma_k) \in F \\ \mathcal{N} \text{ otherwise.} \end{array} \tag{4.7}$$

A region in output space Y would be assigned to each one of these two symbols: $Y_\mathcal{Y}$, $Y_\mathcal{N}$, such that $Y_\mathcal{Y} \cap Y_\mathcal{N} = \emptyset$, and the output of the NSM would only be examined after the whole string has been processed.

Recently, Šíma (1997) has shown that the behavior of any DTRNN using threshold activation functions may be stably emulated by another DTRNN using activation functions in a very general class which includes the sigmoid functions considered in this paper, and describes the conditions under which this emulation is correct for inputs of any length. This means that any of the constructions described in section 4.2.1 to encode FSM in DTRNN may be converted, using the method by Šíma (1997) into an equally-capable analog DTRNN.

[5] A set is *denumerable* (also *countable*) when there is a way to assign a unique natural number to each of the members of the set. Some infinite sets are nondenumerable (or *uncountable*); for example, real numbers form a nondenumerable set.

Some DTRNN architectures cannot implement all FSM

Not all DTRNN architectures are capable of representing all FSM. For example, as discussed in section 4.2.1, Elman (1990) neural nets can emulate any FSM using $n_X = |Q||\Sigma|$ state neurons using a step function (see Kremer (1995)). As will be seen, Elman nets using sigmoids over rational numbers (Alquézar and Sanfeliu, 1995) may also be used to implement FSM (see the paper by Alquézar and Sanfeliu (1995)) ; more recently, Carrasco et al. (2000) have shown that this is also the case for Elman nets using real sigmoids.

On the other hand, first-order DTRNN without an output layer which use as output a projection of the state vector, that is,

$$y_i[t] = x_i[t]; \;\; i = 1, \ldots, n_Y; \;\; n_Y < n_X \tag{4.8}$$

cannot emulate all FSM, analogously to the work by Goudreau et al. (1994) . These networks can however emulate DFA using a suitable end-of-string symbol and $|Q||\Sigma| + 1$ state neurons.

Finally, it is easy to show (using a suitable odd-parity counterexample in the way described by Goudreau et al. (1994)) that the networks by Robinson and Fallside (1991) networks (see section 3.2.1) cannot represent the output function of all Mealy machines unless a two-layer scheme as the following is used:

$$h_i(\mathbf{x}[t-1], \mathbf{u}[t]) = g\left(\sum_{j=1}^{n_Z} W_{ij}^{yz} z_j[t] + W_i^y\right) \tag{4.9}$$

with

$$z_i[t] = g\left(\sum_{j=1}^{n_X} W_{ij}^{zx} x_j[t-1] + \sum_{j=1}^{n_U} W_{ij}^{zu} u_j[t] + W_i^z\right) \tag{4.10}$$

($i = 1, \ldots, n_Z$), and n_Z the number of units in the layer before the actual output layer (the hidden layer of the output function). This may be called an *augmented Robinson-Fallside network*. The encoding of arbitrary FSM in augmented Robinson-Fallside networks is described by Carrasco et al. (2000).

Another interesting example is that of Fahlman's recurrent cascade correlation networks, which are constructed by an algorithm having the same name (see section 3.4.3). Kremer (1996b) —see the following section— has shown these networks cannot represent all FSM by defining suitable classes of nonrepresentable machines.

4.2.3 Featured papers

Alquézar and Sanfeliu (1995), These authors (`http://www.dlsi.ua.es/~mlf/nnafmc/papers/alquezar95algebraic.pdf`) show how arbitrary FSM may be represented in Elman nets under the condition that the inputs, the outputs, and the state values are all rational numbers and the sigmoid operates with rational arithmetic, and give a simple recipe to select the weights

of the network so that this occurs, which is derived from a representation of the next-state function of the FSM in terms of a system of linear equations; the construction byMinsky (1967) happens to be a special case of the proposed method. The construction needs a split-state representation of the states in the FSM for the reasons given by Goudreau et al. (1994) . Corresponding results for second-order DTRNN are also presented. The authors also indicate how the derived algebraic relations may be used to constrain gradient-descent algorithms to preserve prior knowledge inserted in the DTRNN in form of FSM transitions.

Omlin and Giles (1996b) These authors `http://www.dlsi.ua.es/~mlf/nnafmc/papers/omlin96stable.pdf`set out to prove whether there exists a way to choose weights in a DTRNN based on real sigmoids (in particular, the logistic function $g(x) = 1/(1 + \exp(-x))$), therefore allowing real ranges of outputs and state values, so that the DTRNN behaves as a deterministic finite-state automaton (see section 2.3.3). They propose a way to choose the weights of a second-order DTRNN that guarantees that the language accepted by the DTRNN and that accepted by the DFA are identical; all weights and biases are simple multiples of a single value H (H, $-H$, $-H/2$ and 0).[6] A careful worst-case analysis of the fixed points and bounds of repeated applications of the next-state function defines the actual value of H, which is always greater than 4 and grows roughly as $\log(n_X)$. The experimental values of H found by the authors seem however to be constant for a set of large random DFA.

Kremer (1996b): As has been discussed earlier in this chapter, certain DTRNN architectures may not be capable of representing all possible FSM. *Recurrent cascade correlation* (RCC) (see section 3.4.3) is both the name of a learning algorithm which constructs a DTRNN during learning and the name of the class of architectures generated by this algorithm. A number of papers have addressed the representational capabilities of RCC networks ((Giles et al., 1995), (Kremer, 1996a),; (Kremer, 1996b)) by defining a class of FSM that cannot be represented in RCC nets; the class grows in generality as we go from one paper to another. A general formulation is presented by Kremer (1996b) (`http://www.dlsi.ua.es/~mlf/nnafmc/papers/kremer96finite.pdf`), but the class of FSM that *can* be represented in this architecture still remains to be defined. According to Kremer (1996b), which uses a *reductio ad absurdum* proof, RCC nets cannot represent FSM which, as a response to input strings whose symbols repeat with a periodicity p, output strings have a periodicity ω such that $p \bmod \omega = 0$ (if p is even) or $2p \bmod \omega = 0$ (if p is odd). One such automaton is the odd parity automaton (see figure 2.3), whose output has a period 2 if its input is a constant "`11111...`" (period 1).

[6] In the same spirit, Carrasco et al. (2000) have recently proposed similar encodings for general Mealy and Moore machines on various first- and second-order DTRNN architectures.

4.3 Turing computability with discrete-time recurrent neural nets

4.3.1 Turing machines

A *Turing machine* ((Turing, 1936); Hopcroft and Ullman (1979, 147); Salomaa (1973, 36)) is an automaton that uses an endless tape as memory. Any finite procedure, algorithm, or computable function may always be reduced to a Turing machine (TM).

In each move, the machine reads a symbol from the tape. Depending on the symbol and the current state, the TM changes state, writes a new symbol, and moves right or left.

A TM may be defined as $M = (Q, \Sigma, \Lambda, \delta, q_I, B, F)$ where

- Q is a finite set of states;
- Σ is the *input alphabet*;
- Λ is the *tape alphabet*;
- $\delta : Q \times \Lambda \to Q \times \Lambda \times \{L, R\}$ is the *next move* function (R is "right", L is "left"; the function may not be defined for some arguments; there, the TM stops);
- $q_I \in Q$ is the *initial state*;
- $B \in \Lambda$ ($B \notin \Sigma$) is a special *blank symbol*;
- $F \subset Q$ is the set of *final states* (the machine stops when entering any $q \in F$).

Turing machines as language acceptors: The TM is started on a tape containing a string $w \in \Sigma^*$ at the beginning of the tape and blanks B after it. A TM *accepts* a string w when it enters a final state in F; if the string is not accepted, the TM may or may not stop. It may be shown (Hopcroft and Ullman (1979, 221); Salomaa (1973, 37)) that the class of languages accepted by TM is the same as the class of languages generated by unrestricted grammars (defined in section 4.1.2).

Turing machines as function computers: TM may compute partial functions mapping natural numbers into natural numbers (Hopcroft and Ullman, 1979, 151). A possible construction uses $\Sigma = \{1\}$ and zero (0) as the blank B. If the function computed by the TM is f, and the initial tape is $11\ldots10000\ldots$ with n ones, the result is $0000\ldots011\ldots10000\ldots$ with $f(n)+1$ ones if f is defined for n and with zero ones if undefined. Any recursively computable function mapping naturals into naturals may be simulated by a Turing machine.

The universal Turing machine: All TMs of the form

$$M = (Q, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \{q_2\})$$

may be encoded as strings over $\{0,1\}$ and written, followed by the string w to be processed, into a tape. There exists a special Turing machine called the *universal Turing machine* (UTM, Hopcroft and Ullman (1979, 181), Salomaa (1973, 117)) which will read this tape and simulate M on w.

Linearly-bounded automata

Linearly bounded automata (LBA) are a special class of nondeterministic[7] Turing machines which have two extra symbols in their input alphabet, say @ and $, which are called the left and right *endmarkers*; the LBA can neither overwrite these markers nor move left from @ or right from $; therefore, it uses only a limited amount of tape. LBA accept exactly Type 1 or context-sensitive languages (Hopcroft and Ullman (1979, 225); Salomaa (1973, 35))[8].

Turing computability with DTRNN

Discrete-time recurrent neural networks may be shown to be capable of performing the same computation as any Turing machine by showing that a DTRNN may simulate the universal Turing Machine. This is the subject of one of the papers featured in this section:

Siegelmann and Sontag (1991) (`http://www.dlsi.ua.es/~mlf/nnafmc/papers/siegelmann91turing.pdf`) show that a single-input single-output DTRNN can simulate any TM; in particular, they show that the UTM may always be encoded in a DTRNN with far less than 100 000 state units. Their architecture is a simple first-order DTRNN (a "neural Moore machine"):

$$x_i[t] = g_\sigma\left(\sum_{j=1}^{n_X} W_{ij}^{xx} x_j[t-1] + W_i^{xu} u[t] + W_i^x\right)$$

$$y[t] = x_1[t]$$

where the $x_i[t]$ are rational numbers, the $u[t]$ are either 0 or 1 and $g_\sigma(x)$ is 0 if $x < 0$, 1 if $x > 1$ and x otherwise.

The input tape is the sequence $u[1]u[2]\ldots$; the output tape is the sequence $y[1]y[2]\ldots$.

The proof by Siegelmann and Sontag (1991) stands on the following findings:

[7] Nondeterminism does not add any power to Turing machines.

[8] These machines are called *linearly-bounded automata* because a *deterministic* TM having its tape bounded by an amount which is a linear function of the length of the input would have the same computational power as them.

- A TM may always be simulated by a pushdown automaton with three *unary* stacks (counters) [indeed, only two are needed (Hopcroft and Ullman, 1979, 172)].
- A unary stack may be encoded as a fractional number (in binary): $q_s = 0.1111 = 15/16$ represents four items. Popping an item is the same as taking $q'_s = \sigma(2q_s - 1)$; pushing an item is the same as taking $q'_s = \sigma(1/2 + q_s/2)$.
- All stack operations and all state transitions triggered by states of the control unit and the symbols at the top of stacks may be computed in at most two time steps of the DTRNN.

4.4 Super-Turing capabilities of discrete-time recurrent neural nets

More recently, DTRNN have been shown to have super-Turing computational power. This is the subject of one of the papers featured in this document:

Siegelmann (1995): This author has shown that the above first-order DTRNN using *real* instead of *rational* state values have super-Turing computational power. The computational power of DTRNN is similar to that of *nonuniform Turing machines*, that is, nonrealizable TM that receive in their tape, in addition to the input w, another string $W(|w|)$ called "advice" (a function only of the length of w) to assist in the computation[9].

In particular, the computational class P/Poly refers to those nonuniform Turing machines in which both the length of the advice and the computational time are polynomially (not exponentially) long (relative to the length of w). DTRNN compute the super-Turing class P/Poly in polynomial time too.

[9]Nonuniform TMs are unrealizable because the length of w is not bounded, and thus, the number of possible advice strings is infinite and cannot be stored in finite memory previous to any computation.

Chapter 5

Grammatical inference with discrete-time recurrent neural networks

This chapter introduces the reader to a group of papers which deal with the inference of grammar rules using discrete-time recurrent neural networks (DTRNN). Section 5.1 defines the problem of grammatical inference. The use of DTRNN for grammatical inference is discussed in section 5.2. The fact that DTRNN capable of representing some tasks may not be able to learn them is the subject of section 5.3. Section 5.4 discusses the algorithms that may be used to extract finite-state machines from trained DTRNN. Finally, section 5.5 introduces the featured papers by dividing them in two main groups: papers dealing with the inference of finite-state machines and papers dealing with the inference of context-free grammars.

5.1 Grammatical inference (GI)

In chapter 4, the computational capabilities of discrete-time recurrent neural networks were discussed in terms of their equivalence or their parallelism to various classes of automata. In particular, automata such as deterministic finite-state automata and Turing machines may be seen as language acceptors. Chapter 4 also discussed a generative way of defining formal languages, grammars, and describes the relationships among languages, grammars and automata.

If a given application involves data sequences which can be represented as strings of symbols from a finite alphabet and a sequence processing task (chapter 3) that involves classification of these sequences into two or more classes or sets (languages) or recognition of those sequences that belong to a single class of interest, then it may be the case that language acceptors such as finite-state automata may be capable of performing the task or that grammars may be used

to express the rules that define the sequences belonging to one of the classes (languages).

Many sequence recognition/classification applications may be reduced to this formulation. In this case, it may be interesting to learn a sequence-processing task by inferring the rules of the grammar(s) or the structure of the accepting automaton (automata) from samples (learning sets) of classified sequences (strings).

Grammatical inference is the usual name given to the process of learning (inferring) a grammar from a set of sample strings, and, in view of the equivalences that may be established between grammars and automata, the task of learning an automaton from a set of sample strings may also be called grammatical inference.

Grammatical inference is usually formulated in terms of learning recognition or classification tasks from sets in which all strings are labeled as belonging to one or another class (language); however, tasks such as learning a finite-state machine that transduces (translates) strings from one language into strings from another language or learning a probabilistic finite-state machine that generates strings following a certain probability distribution may also be formulated as grammatical inference tasks.

5.2 Discrete-time recurrent neural networks for grammatical inference

This chapter is concerned with the use of discrete-time recurrent neural networks (DTRNN) for grammatical inference. DTRNN may be used as sequence processors in three main modes:

Neural acceptors/recognizers: DTRNN may be trained to accept strings belonging to a language and reject strings not belonging to it, by producing suitable labels after the whole string has been processed. In view of the computational equivalence between some DTRNN architectures and some finite-state machine (FSM) classes, it is reasonable to expect DTRNN to learn regular (finite-state) languages. A set of neural acceptors (separately or merged in a single DTRNN) may be used as a neural classifier.

Neural transducers/translators: If the output of the DTRNN is examined not only at the end but also after processing each one of the symbols in the input, then its output may be interpreted as a synchronous, sequential transduction (translation) for the input string. DTRNN may be easily trained to perform synchronous sequential transductions and also some asynchronous transductions.

Neural predictors: DTRNN may be trained to predict the next symbol of strings in a given language. The trained DTRNN, after reading string outputs a mixture of the possible successor symbols; in certain conditions (see e.g. Elman (1990)), the output of the DTRNN may be interpreted

as the probabilities of each of the possible successors in the language. In this last case, the DTRNN may be used as a probabilistic generator of strings.

When DTRNN are used for grammatical inference, the following have to be defined:

- A learning set. The learning set may contain: strings labeled as belonging or not to a language or as belonging to a class in a finite set of classes (recognition/classification task)[1]; a draw of unlabeled strings, possibly with repetitions, generated according to a given probability distribution (prediction/generation task); or pairs of strings (translation/transduction task).
- An encoding for input symbols as input signals for the DTRNN. This defines the number of input lines n_U of the DTRNN.
- An interpretation for outputs: as labels, probabilities for successor symbols or transduced symbols. This defines the number of output units n_Y of the DTRNN.
- A suitable DTRNN architecture, the number of state units n_X and the number of units in other hidden layers.
- Initial values for the learnable parameters of the DTRNN (weights, biases and initial states).
- A learning algorithm (including a suitable error function and a suitable stopping criterion) and a presentation scheme (the whole learning set may be presented from the beginning or a staged presentation may be devised).
- An extraction mechanism to extract an automaton or grammar rules from the weights of the DTRNN. This will be discussed in detail in section 5.4.

5.3 Representing and learning

One of the most interesting questions when using DTRNN for grammatical inference is expressed in the title of this section. Whereas in chapter 4 it was shown that certain DTRNN architectures may actually perform some symbolic string processing tasks because they behave like the corresponding automata, it remains to be seen whether the learning algorithms available are capable of finding the corresponding sets of weights by using examples of the task to be learned. This is because all of the learning algorithms implement a certain heuristic to search for the solution in weights space, but do not guarantee that the solution will be found, provided that it exists. Some of the problems have already been mentioned in chapter 3, such as the presence of local minima not

[1] For some recognition tasks, positive samples may be enough.

corresponding to the solution or the problem of long-term dependencies along the sequences to be processed. It may be said that each learning algorithm has its own inductive bias, that is, its preferences for certain solutions in weight space.

But even when the DTRNN appears to have learned the task from the examples, it may be the case that the internal representation achieved may not be easily interpretable in terms of grammar rules or transitions in an automaton. In fact, most learning algorithms do not force the DTRNN to acquire such a representation, and this makes grammatical inference with DTRNN a bit more difficult. However, as we will see, in recognition and transduction tasks this problem is surprisingly not frequent: hidden states $\mathbf{x}[t]$ cluster in certain regions of the state space X, and these clusters may be interpreted as automaton states or variables in a grammar.

5.3.1 Open questions on grammatical inference with DTRNN

Summarizing, here are some of the open questions remaining when training DTRNNs to perform string-processing tasks:

- How does one choose n_X, the number of state units in the DTRNN? This imposes an inductive bias (only automata representable with n_X state units can be learned, and maybe not all of them).
- Will the DTRNN exhibit a behavior that may easily be interpreted in terms of automaton transitions or grammar rules? There is no bias toward a symbolic internal representation: the number of available states in X is infinite.
- Will it learn? Even if a DTRNN can represent a FSM compatible with the learning set, learning algorithms do not guarantee a solution. *Learning* a task is harder than *programming* that task on a DTRNN.
- There is the problem of *multiple minima*: most algorithms may get trapped in undesirable local minima.
- If the task exhibits long-term dependencies along the strings, it may be very hard to learn (see section 3.5).

5.3.2 Stability and generalization

Stability: Some authors use the term stability to refer to the following property: that a DTRNN is exhibiting *stable* behavior (in the sense of being a neural language recognizer or a language transducer) when outputs are within the regions of output space assigned to the corresponding symbols for strings of any length, as discussed in section 4.2.

In general, *trained* DTRNN are "stable" only for strings up to a given length; this is due to the nature of the internal representation assumed. For example, if the DTRNN is trained to behave as a finite-state machine instability means

that the regions of state space visited by the DTRNN corresponding to the FSM to be inferred are not disjoint and merge as they grow with string length; as a consequence, they do not clearly map into the regions Y_m of output space assigned to the desired outputs.

As shown in chapter 4, DTRNN may be *constructed* so that they behave stably as FSM.

Generalization: When a *test set* has been set aside, one may check whether the behavior learned by the DTRNN from the *learning set* is consistent with the *test set*. This is called the *generalization test*.

Some learning algorithms (Giles et al., 1992) partition learning sets in a small *starting set* and a number of *test sets* (see (Giles et al., 1992)). Once the starting set is learned, a test set is used to check the DTRNN. If the test fails, the test is added to the learning set to be relearned. If, after some relearning runs, the remaining test sets are correctly classified without having to relearn, the learning algorithm terminates.

5.4 Automaton extraction algorithms

When discrete-time recurrent neural networks are used for grammatical inference purposes, and, in particular, to infer a finite-state machine, the grammatical inference task is not complete unless a true symbolic representation of the rules defining the language (grammar rules, finite-state machine transitions) are extracted from the dynamics of the DTRNN. Since the dynamics of the DTRNN is in a real vector space, all of the methods try to discover a partition of state space so that the dynamics of the DTRNN may be described in terms of a finite number of states.

Kolen (1994) has strongly criticised these methods on the grounds of their sensitivity to the initial conditions and the behavioral changes in the extracted description that may be induced simply by changing the way the continuous-state dynamics of the DTRNN of the DTRNN is observed: his main point is that looking for a finite-state description of the dynamics of the DTRNN may be incorrect because of the continuous state nature of the DTRNN. However, other authors such as Casey (1996) have proved that, under certain conditions, the behavior of a DTRNN may actually be described as that of a finite-state machine (see section 4.2), and other authors have shown that a DTRNN may actually be induced to behave as a FSM by suitably programming its weights ((Omlin and Giles, 1996a), (Carrasco et al., 2000); see also (Omlin and Giles, 1996b)) . These results do not completely invalidate the criticisms by Kolen (1994), which may be still applicable because most extraction methods assume, but cannot test, that the DTRNN is actually behaving as a FSM and then proceed to extract.

Three main types of automaton extraction algorithms will be discussed in this document: state-space partition methods, clustering methods, and methods based on Kohonen's self-organizing maps.

5.4.1 State-space partition methods

This method has been used, among others, by Giles et al. (1992) . The n_X-dimensional state space hypercube is divided in q^{n_X} equal hypercubes by dividing each of the edges of the state-space hypercube in q equal parts. The hypercube containing the initial state $\mathbf{x}[0]$ of the DTRNN is labeled as the initial state of the FSM, q_I and marked as visited. Then the DTRNN is alowed to process all possible input strings. If, after reading symbol σ_k the DTRNN performs a transition from a state $\mathbf{x}[t-1]$ in a hypercube labeled as q_j to a state $\mathbf{x}[t]$ in a new hypercube, then the new hypercube is given a new label q_n (a new state is created) and the transition $\delta(q_j, \sigma_k) = q_n$ is recorded. If the transition results in a state $\mathbf{x}[t]$ in an existing hypercube q_i, the transition $\delta(q_j, \sigma_k) = q_i$ is recorded and the DTRNN dynamics on that string is no pursued no longer. The resulting FSM may then be minimized. The partition parameter q is chosen to be the smallest one leading to a FSM which is compatible with the learning set.

A variation of this method was used by Blair and Pollack (1997) to study the nature of the representations achieved by DTRNN: instead of using the actual values of $\mathbf{x}[t]$ computed by the DTRNN, these authors used the centers of the hypercubes. The method of Blair and Pollack (1997) is used to determine whether the DTRNN actually shows a behavior that may be described in terms of a finite-state machine or a regular language; this is the only extraction method known that establishes (with desired accuracy) whether the DTRNN is behaving as a FSM or not before extracting a finite-state description of the behavior, thus addressing some of the concerns expressed by Kolen (1994).

5.4.2 Clustering methods

Finite-state machines may also be extracted from DTRNN by means of clustering methods. These methods rely on the following assumption: when the DTRNN behaves as a FSM the points it visits when reading strings form low-dimensional clusters in state space that correspond to the states of the FSM. Therefore, one may use a clustering method to discover this structure and then define the transitions of the FSM in terms of the transitions observed for these clusters. The first observation of this is reported by Cleeremans et al. (1989) for a simple grammar. Manolios and Fanelli (1994) start with n randomly distributed markers which are iteratively moved toward the closest network states until they reach the centroid of the sets of points they are closest to; then, they check whether the transitions between DTRNN states are compatible with clusters being interpreted as discrete states; if not, a new set of markers is generated and clustering starts again.

Gori et al. (1998) use a clustering method to extract simple approximate FSM descriptions from DTRNN trained on *noisy* learning sets, that is, learning sets generated by flipping the membership labels of a few strings in a *clean* learning set compatible with a small FSM, and find that the original FSM is sometimes recovered. Their method relies on the assumption that small DTRNN, tend to form clusters corresponding to a simple finite-state description of a ma-

jority of the strings in the learning set, because a FSM corresponding exactly to the learning set may be impossible to represent. They contend that approximate FSM learning may indeed be one promising application of DTRNN.

Some authors integrate clustering in the learning algorithm, as Das and Mozer (1998) do (see also (Das and Das, 1991) and (Das and Mozer, 1994)).

5.4.3 Using Kohonen's self-organizing maps

Kohonen's self-organizing feature maps (SOFM) may also be used to extract finite-state!behavior from the dynamics of a trained DTRNN. This has been done, among others, by Tiňo and Sajda (1995) . The neurons in a SOFM form a neuron field (NF); neurons are organized according to a predetermined topology and then these neurons are *topologically mapped* onto the state space of the DTRNN as follows: a position in state space is assigned to each one of the neurons in the NF in such a way that neighborhood is preserved: points that are close in DTRNN space are assigned to neurons in the NF that are close. Tiňo and Sajda (1995) use a star topology for the NF after assigning the points in DTRNN space to clusters, they determine intercluster transitions and determinize the transition diagram until transitions are compatible with a deterministic FSM. The resulting FSM is finally minimized.

5.5 Featured papers

This chapter features a selection of papers dealing with grammatical inference and DTRNN. The featured papers may be divided in two main groups: papers dealing with the inference of finite-state machines and regular grammars and papers dealing with the inference of context-free grammars or pushdown automata.

5.5.1 Inference of finite-state machines

Five of the featured papers deal with the inference of finite-state machines. The papers may further be divided in three groups, depending whether the DTRNN used are trained to predict the next symbol of a word (Cleeremans et al., 1989), to classify a string (word) as belonging or not to a language (Pollack, 1991; Giles et al., 1992; Manolios and Fanelli, 1994), or to translate a string over the input alphabet into a string over the output alphabet (Tiňo and Sajda, 1995). The papers also present a wide variety of DTRNN architectures as well as of training and extraction schemes.

Cleeremans et al. (1989): this paper describes how an Elman (1990) simple recurrent network may be trained to read a string and predict the next symbol. When a network is trained to predict the next symbol, a *one-hot*, exclusive, or local interpretation (one unit per symbol, high when the symbol is present

and low otherwise) is used for the alphabet, and a quadratic error function is used, it is the case that the actual outputs of the DTRNN after having seen a string are a good approximation to the frequencies observed for each of the successors of that string in the learning set. If the learning set is interpreted to be a representative draw from a given probability distribution over strings, then the outputs may actually be interpreted as probabilities, and the DTRNN may be used as a generator having a similar probability distribution.

However, Cleeremans et al. (1989) do not use the trained DTRNN as a generator but rather as an acceptor for the language (they explored two languages). A string is accepted if each of its successors is predicted with a probability higher than a threshold, which the authors set to 0.3[2]. For the simplest language, using 70,000 random strings, of which only 210 were grammatical, the network only accepted the grammatical ones. Also, the network accepted all string in a set of 20,000 random grammatical strings. The second language modelled long-term dependencies: the strings started and ended with the same symbol; if the probabilities of intervening strings were independent of that symbol, the network failed to learn the language; however, when the distribution of intervening strings depended even if very slightly on the initial symbol, then the DTRNN learned the language.

The authors also perform a hierarchical clustering of the observed values of the hidden units and observe clusters that correspond to the states in the automata defining the languages.

Pollack (1991): This paper (`http://www.dlsi.ua.es/~mlf/nnafmc/papers/pollack91induction.pdf`) deals with the training of a class of second-order DTRNN (see section 3.2.1) to behave as language recognizers (Pollack (1991) uses the name *dynamical recognizers*, and defines them in a way parallel to the definition of deterministic finite automata, see section 2.3.3). The DTRNN is trained to recognize the seven languages in Tomita (1982) using a gradient-descent algorithm. One of the main emphases of the paper is in the cognitive implications of this process. Pollack (1991) also shows that, as learning progresses, the DTRNN undergoes a sudden change similar to a phase transition. He also formulates a tentative hypothesis as to the classes of languages that may be recognized by a dynamical system such as a DTRNN and its relation to the shape of the area visited by the network as strings get longer and longer (the attractor) and the way it is cut by the decision function used to determine grammaticality. Pollack (1991) studies then the nature of the representations learned by the DTRNN, first by examining the labels given by the networks to all strings up to length 9 (to find that the labelings are not completely consistent with the languages), and then by looking at the state space of the DTRNN, either graphically or by studying its fractal dimension.

[2]This is presumably because in each state, and for the language studied by Cleeremans et al. (1989), there is a maximum of two possible successors, and 0.3 is a safe threshold below 0.5.

Giles et al. (1992): The architecture used by these authors is very similar to that used by Pollack (1991): a second-order DTRNN (see section 3.2.1) which is trained using the RTRL algorithm (Williams and Zipser, 1989c) (see section 3.4.1). The presentation scheme is one of the main innovations in this paper: training starts with a small random subset; if the DTRNN either learns to classify it perfectly or spends a maximum number of training epochs, the learning set is incremented with a small number of randomly chosen strings. When the network correctly classifies the whole learning set, then it is said to converge. A special symbol is used to signal the end of strings; this gives the network extra flexibility, and may be easily shown to be equivalent to adding a single-layer feedforward neural network as an output layer, as done by Blair and Pollack (1997) or Carrasco et al. (1996)(see section 3.2.2). These authors extract automata from DTRNN by dividing the state space hypercube in q^{nx} equally-sized regions. The extraction algorithm has been explained in more detail in section 5.4. One of the main results is that, in many cases, the deterministic finite automata extracted from the dynamics of the DTRNN exhibit better generalization than the DTRNN itself. This is related to the fact that the DTRNN may not be behaving as the corresponding FSM in the sense discussed in section 4.2, but instead it shows what some authors call *unstable* behavior (see section 5.3.2).

Manolios and Fanelli (1994) use Elman's first-order architecture ((Elman, 1990)), as Cleeremans et al. (1989) , but train it to classify strings in small learning sets as grammatical or ungrammatical —according to four of the grammars in Tomita (1982) — using backpropagation through time (Rumelhart et al., 1986) with batch updating (see section 3.4.1). Once the DTRNN has been trained, automata are extracted using a special clustering algorithm that has been described in section 5.4. One of the most interesting features of the paper is its graphical study of the learning process, and , in particular, that of the size of clusters as training progresses. The size of the clusters is proposed as an indicator of generalization ability.

Tiňo and Sajda (1995) (`http://www.dlsi.ua.es/~mlf/nnafmc/papers/tino95learning.pdf`) use a first-order DTRNN which is basically an augmented version of the recurrent error propagation network of Robinson and Fallside (1991), first-order DTRNN (see section 3.2.1), with an extra layer to compute the output, to learn the transduction tasks performed by Mealy machines (see section 2.3.1). The network is trained using an online algorithm similar to RTRL (see section 3.4.1); weights are updated after each symbol presentation (online learning). In addition to being one of the few papers dealing with transduction instead of recognition tasks, it introduces a new FSM extraction method based on Kohonen's self-organizing maps (see section 5.4.3; see also Haykin (1998, 408)).

Das and Mozer (1998) (`http://www.dlsi.ua.es/~mlf/nnafmc/papers/sreerupa98dynamic.pdf`) encourage a second-order DTRNN (similar to the ones used by Giles et al. (1992)) to adopt a finite-state like behavior by means of clustering methods, which may be unsupervised or supervised. In the first case, unsupervised clustering of the points of state space visited by the network is used after a certain number of training epochs and a new next-state function is constructed as follows: first, a next state candidate $\mathbf{x}'[t]$ is computed from $\mathbf{x}[t-1]$ using eq. 3.8; then, it is assigned to the corresponding cluster; finally, it is linearly combined with the corresponding centroid $\mathbf{c}[t]$ to obtain the next state: $\mathbf{x}[t] = (1-\alpha)\mathbf{x}'[t] + \alpha\mathbf{c}[t]$, with $\alpha \in [0,1]$ estimated from the current error. In the second (supervised) case, states are assumed to be ideally fixed points but actually corrupted by noise that follows a Gaussian distribution whose mean and variance is estimated for each state simultaneously to the weights of the DTRNN. The method assumes a known number of states and uses a temperature parameter to gradually shrink the Gaussians as the error improves. In the experiments, both the supervised and unsupervised approaches improve the results obtained without using any clustering; the supervised clustering method performs much better than the unsupervised one. The idea of using clustering to improve FSM learning by DTRNN had been previously reported by Das and Das (1991) and Das and Mozer (1994).

5.5.2 Inference of context-free grammars

Other authors have studied the ability of DTRNN-like architectures to learn context-free grammars from examples. In some of the cases, the DTRNN is augmented with an external stack which it can manipulate, in much the same way as a pushdown automaton (the recognizer for context-free grammars) is a finite-state machine with an external stack (Giles et al., 1990; Mozer and Das, 1993). Others rely on the ability of the DTRNN to organize their state space in a stack-like fashion.

Giles et al. (1990) use a second-order DTRNN of the kind used by Giles et al. (1992), but supplemented with an external analog (continous) stack, to learn successfully two simple context-free languages. A set of special output units, called *action* units, are used to control the stack. The stack is continuous, in that it may contain elements of variable thickness; this is so that gradient-based algorithms may be used.

Mozer and Das (1993) propose what could be considered as an asynchronous DTRNN-like architecture which effectively acts as a pushdown automata implementing a shift-reduce parsing strategy.[3] The implementation is based on the following modules:

[3] Also known as LR, *Left-to-right, Rightmost-derivation* parsing (Hopcroft and Ullman, 1979, 248).

- A stack which may contain input symbols (terminals) shifted onto it or nonterminal symbols (variables).
- A set of demon units that can read the top two stack symbols. Each demon unit reacts to a particular pair of symbols, pops both of them from the stack, and pushes a particular nonterminal. When no demon unit reacts, a new input symbol is pushed onto the stack.

Grammars induced are of the form $X \rightarrow YZ$ where Y and Z are either nonterminals or terminals. Both the stack (as in Giles et al. (1990)) and the demons are continuous to allow for partial demon activity and variable-thickness symbols in the stack. This in turn allows for the use of a gradient-based method to train the network; the network is trained to obtain a stack containing simply the start symbol after reading a grammatical string, and any other symbol after reading a nongrammatical string. Mozer and Das (1993) successfully train the network to learn four simple grammars from relatively small learning sets.

Zeng et al. (1994) describe a method —partially described earlier in (Zeng et al., 1993)— to use and train a second-order DTRNN such as the one used by Giles et al. (1992) , without and with an external stack, so that stable finite-state or pushdown automaton behavior are ensured. The method has two basic ingredients: (a) a discretization function

$$D(x) = \begin{cases} 0.8 & \text{if} \quad x > 0.5 \\ 0.2 & \text{otherwise} \end{cases} ,$$

which is applied after the sigmoid function when computing the new state $\mathbf{x}[t]$ of the DTRNN, and (b) a pseudo-gradient learning method, which may be intuitively described as follows: the RTRL formulas are written for the corresponding second-order DTRNN without the discretization function (as in (Giles et al., 1992)) but used with discretized states instead. The resulting algorithm is empirically investigated to characterize its learning behavior; the conclusion is that, even if it does not guarantee the reduction of the error, the algorithm is able to train the DTRNN to perform the task. One of the advantages of the discretization is that FSM extraction is trivial: each FSM state is represented by a single point in state space. Special error functions and learning strategies are used for the case in which the DTRNN manipulates an external stack for the recognition of a subset of context-free languages (the stack alphabet is taken to be the same as the input alphabet and transitions consuming no input are not allowed; unlike Giles et al. (1990), these authors use a *discrete* external stack).

Author Index

Index

List of abbreviations

BPTT: Backpropagation through time (section 3.4.1).

DFA: Deterministic finite-state automaton (section 2.3.3).

DTRNN: Discrete-time recurrent neural network (section 3.2).

EKF: Extended Kalman filter (section 3.4.1).

FFNN: Feedforward neural network (section 3.4.1).

FSM: Finite-state machine (section 2.3)

NARX: Nonlinear Auto-Regressive with eXogenous inputs (section 3.2.3).

NF: Neuron field (in Kohonen's self-organizing feature maps)

NSM: Neural state machines (section 3.2).

PDA: Pushdown automaton (section 4.1.2).

RAAM: Recursive auto-associative memory (section 3.6).

RCC: Recurrent cascade correlation (section 3.4.3).

RTRL: Real-time recurrent learning (section 3.4.1).

SOFM: Kohonen's Self-organizing feature map.

TDNN: Time-delay neural net (section 3.2.3).

TM: Turing machine (section 4.3.1).

TLU: Threshold linear unit (section 2.2).

TPE: Temporal propositional expression (section 2.1).

UTM: Universal Turing machine (section 4.3.1).

Bibliography

Adalı, T., Bakal, B., Sönmez, M. K., Fakory, R., and Tsaoi, C. O. (1997). Modeling nuclear reactor core dynamics with recurrent neural networks. *Neurocomputing*, 15(3-4):363–381.

Alon, N., Dewdney, A. K., and Ott, T. J. (1991). Efficient simulation of finite automata by neural nets. *Journal of the Association of Computing Machinery*, 38(2):495–514.

Alquézar, R. and Sanfeliu, A. (1995). An algebraic framework to represent finite state automata in single-layer recurrent neural networks. *Neural Computation*, 7(5):931–949.

Aussem, A., Murtagh, F., and Sarazin, M. (1995). Dynamical recurrent neural networks — towards environmental time series prediction. *International Journal of Neural Systems*, 6:145–170.

Baltersee, J. and Chambers, J. (1997). Non-linear adaptive prediction of speech with a pipelined recurrent neural network and a linearised recursive least squares algorithm. In *Proceedings of ECSAP'97, European Conference on Signal Analysis & Prediction*.

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

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

Box, G. E. P., Jenkins, G. M., and Reinsel, G. C. (1994). *Time series analysis: forecasting and control*. Prentice-Hall, Englewood Cliffs, NJ. 3rd. ed.

Bradley, M. J. and Mars, P. (1995). Application of recurrent neural networks to communication channel equalization. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 5, pages 3399–3402.

Bridle, J. S. (1990). Alphanets: A recurrent neural network architecture with a hidden Markov model interpretation. *Speech Communication*, 9:83–92.

Bullock, T. H. and Horridge, A. G. (1965). *Structure and Function in The Nervous System of Invertebrates*. W.H. Freeman and Co., New York, NY.

Bulsari, A. B. and Saxén, H. (1995). A recurrent network for modeling noisy temporal sequences. *Neurocomputing*, 7(1):29–40.

Burks, A. W. and Wang, H. (1957). The logic of automata. *Journal of the ACM*, 4:193–218 and 279–297.

Carrasco, R. C., Forcada, M. L., and Santamaría, L. (1996). Inferring stochastic regular grammars with recurrent neural networks. In Miclet, L. and de la Higuera, C., editors, *Grammatical Inference: Learning Syntax from Sentences*, pages 274–281, Berlin. Springer-Verlag. Proceedings of the Third International Colloquium on Grammatical Inference, Montpellier, France, 25-27 September 1996.

Carrasco, R. C., Forcada, M. L., Valdés-Muñoz, M. Á., and Ñeco, R. P. (2000). Stable encoding of finite-state machines in discrete-time recurrent neural nets with sigmoid units. *Neural Computation*, 12(9):2129–2174.

Casey, M. (1996). The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction. *Neural Computation*, 8(6):1135–1178.

Cauwenberghs, G. (1993). A fast-stochastic error-descent algorithm for supervised learning and optimization. In *Advances in Neural Information Processing Systems 5*, pages 244–251, San Mateo, CA. Morgan-Kaufmann.

Cauwenberghs, G. (1996). An analog VLSI recurrent neural network learning a continuous-time trajectory. *IEEE Transactions on Neural Networks*, 7(2):346–361.

Chalmers, D. J. (1990). Syntactic transformations on distributed representations. *Connection Science*, pages 53–62.

Chen, T.-B., Lin, K. H., and Soo, V.-W. (1997). Training recurrent neural networks to learn lexical encoding and thematic role assignment in parsing Mandarin Chinese sentences. *Neurocomputing*, 15(3):383–409.

Chen, W.-Y., Liao, Y.-F., and Chen, S.-H. (1995). Speech recognition with hierarchical recurrent neural networks. *Pattern Recognition*, 28(6):795–805.

Cheng, Y., Karjala, T. W., and Himmelblau, D. M. (1995). Identification of nonlinear dynamic processes with unknown and variable dead time using an internal recurrent neural network. *Ind. Eng. Chem. Res.*, 34:1735–1742.

Chiu, C.-C. and Shanblatt, M. A. (1995). Human-like dynamic programming neural networks for dynamic time warping speech recognition. *Int. J. Neural Syst.*, 6(1):79–89.

Chomsky, N. (1965). *Aspects of the Theory of Syntax*. MIT Press, Cambridge, MA.

Chovan, T., Catfolis, T., and Meert, K. (1994). Process control using recurrent neural networks. In *2nd IFAC Workshop on Computer Software Structures Integrating AI/KBS System in Process Control*.

Chovan, T., Catfolis, T., and Meert, K. (1996). Neural network architecture for process control based on the RTRL algorithm. *AIChE Journal*, 42(2):493–502.

Chrisman, L. (1991). Learning recursive distributed representations for holistic computation. *Connection Science*, 3(4):345–366.

Cid-Sueiro, J., Artes-Rodriguez, A., and Figueiras-Vidal, A. R. (1994). Recurrent radial basis function networks for optimal symbol-by-symbol equalization. *Signal Processing*, 40:53–63.

Cid-Sueiro, J. and Figueiras-Vidal, A. R. (1993). Recurrent radial basis function networks for optimal blind equalization. In *Neural Networks for Processing III: Proceedings of the 1993 IEEE-SP Workshop*, pages 562–571.

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

Clouse, D., Giles, C., Horne, B., and Cottrell, G. (1994). Learning large debruijn automata with feed-forward neural networks. Technical Report CS94-398, Computer Science and Engineering, University of California at San Diego, La Jolla, CA.

Clouse, D. S., Giles, C. L., Horne, B. G., and Cottrell, G. W. (1997a). Representation and induction of finite state machines using time-delay neural networks. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems*, volume 9, page 403. The MIT Press.

Clouse, D. S., Giles, C. L., Horne, B. G., and Cottrell, G. W. (1997b). Time-delay neural networks: Representation and induction of finite-state machines. *IEEE Transactions on Neural Networks*, 8(5):1065–1070.

Connor, J. T. and Martin, R. D. (1994). Recurrent neural networks and robust time series prediction. *IEEE Trans. Neural Networks*, 5(2):240–254.

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

Das, S. and Mozer, M. (1994). A unified gradient-descent/clustering architecture for finite state machine induction. In Cowan, J., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems 6*, pages 19–26. San Mateo, CA: Morgan Kaufmann.

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

Dertouzos, M. (1965). *Threshold Logic: A Synthesis Approach*. MIT Press, Cambridge, MA.

Draye, J., Pavisic, D., Cheron, G., and Libert, G. (1995). Adaptive time constants improve the prediction capability of recurrent neural networks. *Neural Processing Letters*, 2(3):12–16.

Dreider, J. F., Claridge, D. E., Curtiss, P., Dodier, R., Haberl, J. S., and Krarti, M. (1995). Building energy use prediction and system identification using recurrent neural networks. *Journal of Solar Energy Engineering*, 117:161–166.

Elman, J. (1991). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7(2/3):195–226.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

Fahlman, S. E. (1991). The recurrent cascade-correlation architecture. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 190–196. Morgan Kaufmann, Denver, CO.

Forcada, M. L. and Carrasco, R. C. (1995). Learning the initial state of a second-order recurrent neural network during regular-language inference. *Neural Computation*, 7(5):923–930.

Forcada, M. L. and Carrasco, R. C. (2001). *Simple stable encodings of finite-state machines in dynamic recurrent networks*, pages 103–127. IEEE Press.

Forcada, M. L. and Ñeco, R. P. (1997). Recursive hetero-associative memories for translation. In Mira, J., Moreno-Díaz, R., and Cabestany, J., editors, *Biological and Artificial Computation: From Neuroscience to Technology (Proceedings of the 1997 International Work-conference on Artificial and Natural Neural Networks)*, volume 1240 of *Lecture Notes in Computer Science*, pages 453–462, Berlin. Springer-Verlag.

Frasconi, P., Gori, M., Maggini, M., and Soda, G. (1996). Representation of finite-state automata in recurrent radial basis function networks. *Machine Learning*, 23:5–32.

Gilbert, E. N. (1954). Lattice theoretic properties of frontal switching functions. *Journal of Math. and Physics*, 33:57–67.

Giles, C., Sun, G., Chen, H., Lee, Y., and Chen, D. (1990). Higher order recurrent networks & grammatical inference. In Touretzky, D., editor, *Advances in Neural Information Processing Systems 2*, pages 380–387, San Mateo, CA. Morgan Kaufmann.

Giles, C. L., Chen, D., Sun, G. Z., Chen, H. H., Lee, Y. C., and Goudreau, M. W. (1995). Constructive learning of recurrent neural networks: limitations of recurrent cascade correlation and a simple solution. *IEEE Transactions on Neural Networks*, 6(4):829–836.

Giles, C. L., Miller, C. B., Chen, D., Chen, H. H., Sun, G. Z., and Lee, Y. C. (1992). Learning and extracted finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3):393–405.

Gori, M., Bengio, Y., and De Mori, R. (1989). BPS: A learning algorithm for capturing the dynamical nature of speech. In *Proceedings of the IEEE-IJCNN89*, Washington.

Gori, M., Maggini, M., Martinelli, E., and Soda, G. (1998). Inductive inference from noisy examples using the hybrid finite state filter. *IEEE Transactions on Neural Networks*, 9(3):571–575.

Goudreau, M., Giles, C., Chakradhar, S., and Chen, D. (1994). First-order vs. second-order single layer recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(3):511–513.

Haykin, S. (1998). *Neural Networks - A Comprehensive Foundation (2nd. ed.)*. Prentice-Hall, Upper Saddle River, NJ.

Haykin, S. and Li, L. (1995). Nonlinear adaptive prediction of nonstationary signals. *IEEE Transactions on Signal Processing*, 43(2):526–535.

Hebb, D. O. (1949). *The Organization of Behavior*. Wiley.

Hertz, J., Krogh, A., and Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley Publishing Company, Inc., Redwood City, CA.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to automata theory, languages, and computation*. Addison–Wesley, Reading, MA.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554.

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

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366.

Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, 148:574–591.

Hush, D. and Horne, B. (1993). Progress in supervised neural networks. *IEEE Signal Processing Magazine*, 10(1):8–39.

Ifeachor, E. C. and Jervis, B. W. (1994). *Digital Signal Processing: a practical approach*. Addison-Wesley, Wokingham, U.K.

Irving M. Copi, Calvin C. Elgot, J. B. W. (1958). Realization of events by logical nets. *Journal of the ACM*, 5(2):181–186.

Janacek, G. and Swift, L. (1993). *Time series: forecasting, simulation, applications*. Ellis Horwood, New York, NY.

Jordan, M. (1986). Serial order: A parallel distributed processing approach. Technical Report ICS Report 8604, Institute for Cognitive Science, University of California at San Diego, La Jolla, CA.

Kechriotis, G., Zervas, E., and Manolakos, E. S. (1994). Using recurrent neural networks for adaptive communication channel equalization. *IEEE Transactions on Neural Networks*, 5(2):267–278.

Kleene, S. (1956). Representation of events in nerve nets and finite automata. In Shannon, C. and McCarthy, J., editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, N.J.

Kohavi, Z. (1978). *Switching and Finite Automata Theory*. McGraw-Hill, Inc., New York, NY, second edition.

Kohonen, T. (1974). An adaptive associative memory principle. *IEEE Transactions on Computers*, C-23:444–445.

Kolen, J. F. (1994). Fool's gold: Extracting finite state machines from recurrent network dynamics. In Cowan, J. D., Tesauro, G., , and Alspector, J., editors, *Advances in Neural Information Processing Systems 6*, pages 501–508, San Mateo, CA. Morgan Kaufmann.

Kremer, S. C. (1995). On the computational power of Elman-style recurrent networks. *IEEE Transactions on Neural Networks*, 6(4):1000–1004.

Kremer, S. C. (1996a). Comments on "constructive learning of recurrent neural networks: limitations of recurrent cascade correlation and a simple solution". *IEEE Transactions on Neural Networks*, 7(4):1047–1051. includes a reply by Dong Chen and C. Lee Giles.

Kremer, S. C. (1996b). Finite state automata that recurrent cascade-correlation cannot represent. In Touretzky, D., Mozer, M., and Hasselmo, M., editors, *Advances in Neural Information Processing Systems 6*, Cambridge, Massachusetts. MIT Press.

Kremer, S. C. (1999). Identification of a specific limitation on local-feedback recurrent networks acting as mealy-moore machines. *IEEE Transactions on Neural Networks*, 10(2):433–438.

Kuhn, G., Watrous, R. L., and Ladendorf, B. (1990). Connected recognition with a recurrent network. *Speech Communication*, 9:41–48.

Kwasny, S. C. and Kalman, B. L. (1995). Tail-recursive distributed representations and simple recurrent networks. *Connection Science*, 7(1):61–80.

Lang, K. J., Waibel, A. H., and Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3:23–44.

Lawrence, S., Giles, C. L., and Fong, S. (1996). Can recurrent neural networks learn natural language grammars? In *Proceedings of ICNN'96*, pages 1853–1858.

Lettvin, J. Y., Maturana, H. R., McCulloch, W. S., and Pitts, W. (1959). What the frog's eye tells the frog's brain. *Proceedings of IRE*, 47:1940–1959.

Lewis, H. R. and Papadimitriou, C. H. (1981). *Elements of the theory of computation*. Prentice-Hall, Englewood Cliffs, N.J.

Li, C. J., Yan, L., and Chbat, N. W. (1995). Powell's method applied to learning neural control of three unknown dynamic systems. *Optimal Control Applications & Methods*, 16:251–262.

Lin, T., Horne, B. G., Tiňo, P., and Giles, C. L. (1996). Learning long-term dependencies in narx recurrent neural networks. *IEEE Transactions on Neural Networks*, 7(6):1329–1338.

Manolios, P. and Fanelli, R. (1994). First order recurrent neural networks and deterministic finite state automata. *Neural Computation*, 6(6):1154–1172.

Markov, A. A. (1958). On the inversion complexity of system of functions. *Journal of the ACM*, 5(4):331–334.

McCarthy, J. (1956). The inversion of functions defined by Turing machines. In Shannon, C. E. and McCarthy, J., editors, *Automata Studies*, pages 177–181. Princeton University Press, Princeton, N.J.

McClelland, J. L., Rumelhart, D. E., and the PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 2. MIT Press, Cambridge.

McCulloch, W. S. (1959). Agathe tyche: of nervous nets — the lucky reckoners. In *Mechanization of Thought Processes 2*, pages 611–634. H.M. Stationery Office, London, UK.

McCulloch, W. S. (1960). The reliability of biological systems. In *Self-Organizing Systems*, pages 264–281. Pergamon Press.

McCulloch, W. S. and Pitts, W. H. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133.

Minsky, M. (1956). Some universal elements for finite automata. In Shannon, C. E. and McCarthy, J., editors, *Automata Studies*, pages 117–128. Princeton University Press, Princeton, N.J.

Minsky, M. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., Englewood Cliffs, NJ. Ch.: Neural Networks. Automata Made up of Parts.

Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT Press, Cambridge, MA.

Minsky, M. L. (1959). Some methods of heuristic programming and artificial intelligence. In *Proc. Symposium on the Mechanization of Intelligence*, pages 3–36, London, UK. H.M. Stationery Office.

Mitra, S. K. and Kaiser, J. F., editors (1993). *Handbook for digital signal processing*. Wiley, New York, N.Y.

Moore, E. F. and Shannon, C. E. (1956). Reliable circuits using less reliable relays. *Journal of the Franklin Institute*, 262:191–208, 291–297.

Mozer, M. (1989). A focused backpropagation algorithm for temporal pattern processing. *Complex Systems*, 3(4):349–381.

Mozer, M. C. and Das, S. (1993). A connectionist chunker that induces the structure of context-free languages. In Hanson, S. J., Cowan, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*, San Mateo, CA. Morgan Kaufmann Publishers.

Narendra, K. S. and Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, 1:4–27.

Ñeco, R. P. and Forcada, M. L. (1997). Asynchronous translations with recurrent neural nets. In *Proceedings of the International Conference on Neural Networks ICNN'97*, volume 4, pages 2535–2540. Houston, Texas, June 8–12, 1997.

Nerrand, O., Roussel-Gagot, P., Urbani, D., Personnaz, L., and Dreyfus, G. (1994). Training recurrent neural networks: Why and how? an illustration in dynamical process modeling. *IEEE Transactions on Neural Networks*, 5(2):178–184.

Omlin, C. W. and Giles, C. L. (1996a). Constructing deterministic finite-state automata in recurrent neural networks. *Journal of the ACM*, 43(6):937–972.

Omlin, C. W. and Giles, C. L. (1996b). Stable encoding of large finite-state automata in recurrent neural networks with sigmoid discriminants. *Neural Computation*, 8:675–696.

Oppenheim, A. V. and Schafer, R. W. (1989). *Discrete-time signal processing*. Prentice-Hall, Englewood Cliffs, NJ.

Ortiz-Fuentes, J. D. and Forcada, M. L. (1997). A comparison between recurrent neural network architectures for digital equalization. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 4, pages 3281–3284.

Parberry, I. (1994). *Circuit Complexity and Neural Networks*. MIT Press, Cambridge, Mass.

Parisi, R., Claudio, E. D. D., Orlandi, G., and Rao, B. D. (1997). Fast adaptive digital equalization by recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2731–2739.

Pearlmutter, B. A. (1995). Gradient calculations for dynamic recurrent neural networks: a survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228.

Perrin, D. (1990). Finite automata. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. The MIT Press, Cambridge, MA.

Pineda, F. J. (1987). Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59(19):2229–2232.

Pollack, J. (1991). The induction of dynamical recognizers. *Machine Learning*, 7(2/3):227–252.

Pollack, J. B. (1990). Recursive distributed representations. *Artificial Intelligence*, 46:77–105.

Puskorius, G. V. and Feldkamp, L. A. (1994). Neurocontrol of nonlinear dynamical systems with kalman filter-trained recurrent networks. *IEEE Transactions on Neural Networks*, 5(2):279–297.

Qian, N. and Sejnowski, T. (1988). Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, 202:865–884.

Rashevsky, N. (1938). *Mathematical Biophysics*. Dover, Chicago. Revised edition, 1960.

Rashevsky, N. (1940). *Advances and Application of Mathematical Biology*. University of Chicago Press, Chicago.

Robinson, T. (1994). An application of recurrent nets to phone probability estimation. *IEEE Transactions on Neural Networks*, 5(2):298–305.

Robinson, T. and Fallside, F. (1991). A recurrent error propagation network speech recognition system. *Computer Speech and Language*, 5:259–274.

Rosenblatt, F. (1962). A comparison of several perceptron models. In *Self-Organizing Systems*. Spartan Books, Washington, DC.

Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning internal representations by error propagation. In *Parallel Distributed Processing*, chapter 8. MIT Press, Cambridge, MA.

Salomaa, A. (1973). *Formal Languages*. Academic Press, New York, NY.

Sejnowski, T. and Rosenberg, C. (1987). Parallel networks that learn to pronounce english text. *Complex Systems*, 1:145–168.

Shannon, C. (1949). The synthesis of two–terminal switching circuits. *Bell System Technical Journal*, 28:59–98.

Siegelmann, H., Horne, B., and Giles, C. (1996). Computational capabilities of recurrent NARX neural networks. *IEEE Trans. on Systems, Man and Cybernetics*, 26(6).

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

Siegelmann, H. T. (1995). Computation beyond the Turing limit. *Science*, 268:545–548.

Šíma, J. (1997). Analog stable simulation of discrete neural networks. *Neural Network World*, 7:679–686.

Siu, K.-Y., Roychowdhury, V., and Kailath, T. (1995). *Discrete Neural Computation. A Theoretical Foundation.* Prentice-Hall, Englewood Cliffs.

Sluijter, R., Wuppermann, F., Taori, R., and Kathmann, E. (1995). State of the art and trends in speech coding. *Philips Journal of Research*, 49(4):455–488.

Solomonoff, R. (1964). A formal theory of inductive inference. *Information and Control*, 7(1-22):224–254.

Sperduti, A. (1994). Labelling recursive auto-associative memory. *Connection Science*, 6(4):429–459.

Sperduti, A. (1995). Stability properties of the labeling recursive auto-associative memory. *IEEE Transactions on Neural Networks*, 6(6):1452–1460.

Sperduti, A. and Starita, A. (1995). A neural network model for associative access of structures. *International Journal of Neural Systems*, 6:189–194.

Stiles, B. W. and Ghosh, J. (1997). Habituation based neural networks for spatio-temporal classification. *Neurocomputing*, 15:273–307.

Stiles, B. W., Sandberg, I. W., and Ghosh, J. (1997). Complete memory structures for approximating nonlinear discrete-time mappings. *IEEE Trans. on Neural Networks*, 8(6):1–14.

Stolcke, A. and Wu, D. (1992). Tree matching with recursive distributed representations. Technical Report TR-92-025, International Computer Science Institute, Berkeley, CA.

Tiňo, P. and Sajda, J. (1995). Learning and extracting initial Mealy automata with a modular neural network model. *Neural Computation*, 7(4).

Tomita, M. (1982). Dynamic construction of finite-state automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Cognitive Science Conference*, pages 105–108, Ann Arbor, Mi.

Tsoi, A. C. and Back, A. (1997). Discrete time recurrent neural network architectures: a unifying review. *Neurocomputing*, 15:183–223.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265.

Unnikrishnan, K. P. and Venugopal, K. P. (1994). Alopex: a correlation-based learning algorithm for feedforward and recurrent neural networks. *Neural Computation*, 6(3):469–490.

von Neumann, J. (1956). Probabilistic logics and the synthesis of reliable organisms from unreliable components. In Shannon, C. and McCarthy, J., editors, *Automata Studies*, pages 43–98. Princeton University Press, Princeton.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. (1989). Phoneme recognition using time–delay neural networks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(3):328–339.

Wang, H. (1957). A variant to Turing's theory of computing machines. *Journal of the ACM*, 4(1).

Wang, J. and Wu, G. (1995). Recurrent neural networks for synthesizing linear control systems via pole placement. *International Journal of Systems Science*, 26(12):2369–2382.

Wang, J. and Wu, G. (1996). A multilayer recurrent neural network for on-line synthesis of minimum-norm linear feedback control systems via pole assignment. *Automatica*, 32(3):435–442.

Watrous, R. L. and Kuhn, G. M. (1992). Induction of finite-state languages using second-order recurrent networks. *Neural Computation*, 4(3):406–414.

Watrous, R. L., Ladendorf, B., and Kuhn, G. (1990). Complete gradient optimization of a recurrent network applied to /b/, /d/, /g/ discrimination. *Journal of the Acoustic Society of America*, 87:1301–1309.

Weigend, A. S. and Gershenfeld, N. A., editors (1993). *TIME SERIES PREDICTION: Forecasting the Future and Understanding the Past*. Addison-Wesley, Reading, MA. Proceedings of the NATO Advanced Research Workshop on Comparative Time Series Analysis held in Santa Fe, New Mexico, May 14-17, 1992.

Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Doctoral Dissertation, Applied Mathematics, Harvard University, Boston, MA.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proc. IEEE*, 78(10):1550–1560.

Williams, R. and Zipser, D. (1989a). Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1(1):87–111.

Williams, R. and Zipser, D. (1989b). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280.

Williams, R. J. (1992). Training recurrent networks using the extended kalman filter. In *Proceedings of the 1992 International Joint Conference on Neural Networks*, volume 4, pages 241–246.

Williams, R. J. and Zipser, D. (1989c). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280.

Williams, R. J. and Zipser, D. (1995). Gradient-based learning algorithms for recurrent networks and their computational complexity. In Chauvin, Y. and Rumelhart, D. E., editors, *Back-propagation: Theory, Architectures and Applications*, chapter 13, pages 433–486. Lawrence Erlbaum Publishers, Hillsdale, N.J.

Wu, L., Niranjan, M., and Fallside, F. (1994). Fully vector-quantised neural network-based code-excited nonlinear predictive speech coding. Technical report, Cambridge University Engineering Department, Cambridge CB2 1PZ, U.K. CUED/F-INFENG/TR94.

Zbikowski, R. and Dzielinski, A. (1995). Neural approximation: A control perspective. In Hunt, K., Irwin, G., and Warwick, K., editors, *Neural Network Engineering in Dynamic Control Systems*, chapter 1, pages 1–25.

Zeng, Z., Goodman, R., and Smyth, P. (1993). Learning finite state machines with self-clustering recurrent networks. *Neural Computation*, 5(6):976–990.

Zeng, Z., Goodman, R. M., and Smyth, P. (1994). Discrete recurrent neural networks for grammatical inference. *IEEE Transactions on Neural Networks*, 5(2):320–330.