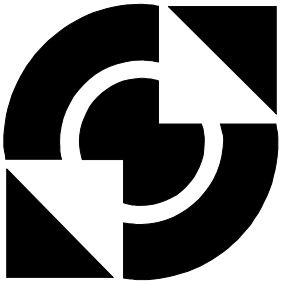
University of Twente

department of
Electrical Engineering



Analysis of Recurrent Neural Networks with Application to Speaker Independent Phoneme Recognition

Esko O. Dijk

M.Sc. Thesis

Supervisors: Prof. Dr.-Ing. O.E. Herrmann
Ir. L.P.J. Veelenturf
Dr. Ir. S.H. Gerez

Date: June 1999
Report number: S&S 023N99

Laboratory of Signals & Systems – Network theory
Department of Electrical Engineering
University of Twente
Enschede, The Netherlands

Author e-mail: esko@ieee.org

Summary

This report investigates how recurrent neural networks can be applied to the task of speaker independent phoneme recognition. Several recurrent neural network architectures found in literature are listed and categorized. A general modular description method is used to describe all the architectures found. The state-space neural network architecture and the Fully Recurrent Neural Network (FRNN) are investigated in more detail.

Training algorithms for recurrent neural networks are investigated and derived, especially the *Backpropagation Through Time* (BPTT) and *Real-time Recurrent Learning* (RTRL) algorithms. BPTT is found to be most suited for a phoneme recognition task. The state-space neural network together with training algorithms is implemented as a Matlab toolbox.

Some properties of recurrent neural networks are investigated, especially their ability to perform classification of sequences of data.

Several neural network methods are used on an artificial phoneme classification task. Mel-cepstrum feature extraction is used to preprocess the artificial phoneme sounds. In the subsequent classification the recurrent networks are found to outperform the non-recurrent architectures. A preliminary experiment on real speech signals is performed.

Acknowledgements

Bij deze wou ik mijn dagelijks begeleider Leo Veelenturf bedanken voor de begeleiding, het nalezen van de (soms omvangrijke) hoofdstukken en de vele nuttige discussies.
Ook wou ik de overige leden van mijn afstudeercommissie, Otto Herrmann en Sabih Gerez, bedanken voor de tussentijdse evaluaties van mijn werk en het nuttige commentaar.

Ook zou ik mijn collega-afstudeerders Sebastiaan, Wouter en Erik willen bedanken en natuurlijk de rest van de NT 'lunchploeg': Marco, Mark, Asker en Frits.

Enschede, Esko Dijk
1 juni 1999

Contents

(continued on the next page)

(continued on the next page)

Abbreviations

BFN	Block Feedback Network
BPTT	Backpropagation Through Time
BPTT-LM	Backpropagation Through Time combined with Levenberg-Marquardt
C	The 'C' programming language
FIR	Finite Impulse Response (neural network)
FRNN	Fully Recurrent Neural Network
FSM	Finite State Machine
k-NN	k-Nearest Neighbor classifier
LM	Levenberg-Marquardt
logsig	logistic sigmoid transfer function $f(x) = \frac{1}{1+e^{-x}}$
Matlab	The Matlab software package (see References/software)
MLP	Multi-Layer Perceptron
ModNet	Modular Network (toolbox)
NARX	Nonlinear AutoRegressive with eXogenous outputs
PRN	Partially Recurrent Network
RMLP	Recurrent Multi-Layer Perceptron
RNN	Recurrent Neural Network
RTRL	Real-Time Recurrent Learning
SISO	Single Input Single Output
SRN	Simple Recurrent Network(s)
tansig	tangential sigmoid transfer function $f(x) = \frac{2}{1+e^{-2x}} - 1$

Symbols and notation

(This list is not complete. Local symbols and notation will be introduced throughout the report. Here only definitions are listed that are not explicitly introduced.)

min(a,b)	Minimum of *a* and *b*
x **x**(n) **X**	Vectors and matrices are **bold**faced
$\delta^{kro}(a,b)$	Kronecker delta function: $\delta^{kro}(a,b) = \begin{cases} 1 & \text{if a = b} \\ 0 & \text{otherwise} \end{cases}$
$\delta^{kro}_{a,b}$ or $\delta_{a,b}$	Kronecker delta function (same)
z^{-1}	Delay element with delay 1

CHAPTER 1 INTRODUCTION

The standard feedforward neural network, or multilayer perceptron (MLP), is the best known member of the 'family' of many types of neural networks. Feedforward neural networks have been applied in tasks of prediction and classification of data for many years.

More recently a new class of neural networks, based upon feedforward neural networks, was introduced. These *dynamic neural networks*, or *neural networks for temporal processing* extend the feedforward networks with the capability of dynamic operation, which means that the neural network behavior depends not only on the current input (as in feedforward networks) but also on previous operations of the network.

Neural networks for temporal processing can be grouped in two classes. The first class, which is called *time-delay networks* here, is based on feedforward neural networks that have certain structures of delay elements added. These structures perform some temporal pre-processing of the input data before the data is presented to neurons in the network. The second class consists of *recurrent neural networks*, which have recurrent connections (neuron outputs are fed back into the network as additional inputs) as well as the structures of delay elements seen in time-delay networks. Time-delay networks do not have any recurrent connections.

This division in two classes, essentially non-recurrent and recurrent dynamic networks, is analogous to the division of discrete-time (sampled) filter networks in finite impulse response (FIR) and infinite impulse response (IIR) filters. FIR filters do not have recurrent connections and IIR filters always have recurrent connections.

1.1 Recurrent neural networks

Recurrent Neural Networks (RNN) are nonlinear or linear dynamic systems. They can be simulated in software on computers or implemented in hardware (analog or digital). A first property that can be used to distinguish RNN in two distinct groups is the representation of time in the system. We have:

- continuous-time systems
- discrete-time systems

A second property is the representation of signals in the system. The signals can be

- real-valued
- quantized

A system can be real-valued if implemented in analog hardware. All digital implementations use quantized values since values are stored in a finite number of bits. However, a digital implementation is often analyzed as if it were real-valued in case that the error introduced by the quantization is too small to be noticed (for example when floating point numbers are used).

The above properties of the system do not say much about the intended application of the RNN. All possible applications of RNN can be grouped into two broad categories. Recurrent neural networks can be used as

- associative memories
- sequence mapping systems

Recurrent neural networks used as sequence mapping systems

Recurrent neural networks used as sequence mapping systems are operated by supplying an input *sequence*, which consists of different input patterns at each time step (in case of a discrete-time system), or a time-varying input pattern over time (in case of a continuous-time system). At each time instant, an output is generated which depends on previous activity of the system and on the current input pattern. The entire output *sequence* generated over time is considered the result of the computation.

The class of sequence mapping systems is interesting for practical applications in sequence recognition, generation or prediction and it will be examined in the next chapter. Sequence mapping neural networks are nearly always implemented in software or clocked digital hardware (both have a discrete representation of time).

This report will focus on recurrent neural networks used as sequence mapping systems. Using the common ways of implementation these networks are discrete-time systems. Therefore, all treatment of recurrent neural networks in the next chapters will be restricted to discrete-time systems. Some examples of recurrent neural networks used as associative memories will be given now.

Recurrent neural networks used as associative memories

Recurrent neural networks used as associative memories are operated by applying a fixed input pattern (that does not change over time). Then the network is operated according to a set of equations describing the network dynamics. Internal signals and the network outputs will change over time. Under certain conditions (and waiting for a sufficient time interval), the network 'settles'. This means the system's output has converged to some *static pattern* which is considered the result of the computation performed by the system. This result is some association made by the system in response to the input, hence the name associative memories.

The difference with sequence mapping systems lies in supplying a static input to the network (not a sequence) and only using the final output values of the network as a result (and not the output sequence over time). So both input and output are static patterns whereas for the sequence mapping systems, both input and output are sequences.

These recurrent neural network architectures were proposed to create associative content-addressable memories. They were used in Artificial Intelligence research and they contributed to research about the way the (human) brain works. Associative memories are sometimes implemented in analog hardware, but generally for research purposes a software implementation is favored because it is more convenient and flexible.

Examples of these architectures are the Brain-State-in-a-Box neural network and the Hopfield network. The Hopfield network model was later on extended with neurons that operate in a stochastic manner (using theory from the field of statistical mechanics) which are called Boltzmann machines [Hertz e.a., 1991].

See [Patterson, 1996] for a short introduction to all the network architectures mentioned.

1.2 Applications of discrete-time recurrent neural networks

Sequence mapping recurrent neural networks can be used in a number of different ways. The most common uses or task types are:

1) *modeling the input-output behavior of a dynamic system*: In this case the network is trained with known examples of input-output behavior of a 'black-box' dynamic system. The trained network is then used to predict dynamic system output, given an input sequence.
2) *time-series prediction*: In this case the network is trained with an output data sequence of some 'black-box' process. The trained network is then used to predict the future (unknown) process outputs, given a sequence of process outputs up to the current time.
3) *sequence classification*: The network is trained with a data sequence that belongs to one class ω_i out of N possible classes. The target value for the network is a coded representation of the class ω_i. Then, the network can be used to assign a classification to a new data sequence.
4) *feature extraction:* The network is trained with a sequence of raw data vectors as input and a smaller feature sequence as a target. The network can learn to transform the 'low-level' raw data vectors into 'higher-level' features. These features can then be used as input to any subsequent classification procedure.

5) *modeling a finite state machine:* The network is trained with example sequences of input and output of a 'black box' finite state machine. The trained network is used to identify the state machine or predict its output, given an input sequence.

Most attention will be given to task 3 with the application to speech recognition (classification) in mind. Task 1 is briefly encountered in demonstrating training algorithms in the next chapter. Task 2 is mentioned in chapter 4 as an alternative approach to classification.

In principle, all recurrent neural networks treated in this chapter can be applied to all task types.

Classification tasks and regression tasks

The specific task can be divided in one of two global classes [Bishop, 1995] [Bengio, 1996]. In a *classification* task (task 3) a neural network has to assign new input data to one out of N discrete classes. Therefore the output of this neural network is in some way converted to a discrete output. In the tasks 1,2 and 4, the neural network continuous-valued outputs are directly used. These tasks are often referred to as *regression* tasks.

Task 5 can not always be categorized this way because it isn't classification and does not have to be regression, because often the neural network outputs are eventually discretized in Finite State Machine modeling.

1.3 Aim

The aim of this assignment is to make an inventarisation and analysis of recurrent neural networks, with an application to speaker independent phoneme recognition in mind.
More specifically:

1. Investigate recurrent neural network architectures and training algorithms ;
2. Investigate (both analytically and experimentally) how recurrent neural networks can be applied to classification of time-series of data, especially series of speech features ;
3. Look at existing neural network methods for phoneme recognition ;
4. Investigate how recurrent neural networks can be used for speaker independent phoneme recognition.

The reason that recurrent neural networks are investigated, is that these dynamic structures might be able to perform phoneme recognition in a better way than static neural networks. Some phoneme sounds have a dynamically changing frequency content over time (see [Janssen, 1998]). This suggests that a dynamic neural network, that can learn to identify the typical dynamic frequency pattern of a phoneme, can perform better recognition of a phoneme than a static neural network. A static network can only use the current information $\mathbf{u}(n)$ at a certain time n to make a classification. So the dynamic relation between different pieces of information $\mathbf{u}(.)$ at different moments in time can not be learned by a static network.

Structure of this report

The structure of this report is given here.

- Chapter 2 introduces several recurrent neural network architectures. These are grouped into three categories.
- Chapter 3 gives algorithms that can train these networks. The two basic training approaches Backpropagation Through Time and Real-time Recurrent Learning are investigated.
- Chapter 4 lists properties of recurrent networks and shows how recurrent networks can be used for classification of data sequences.
- Chapter 5 describes the experiments that were done with recurrent neural networks in a classification task. Several neural network structures are compared.
- Chapter 6 gives final conclusions and recommendations.

CHAPTER 2 RECURRENT NEURAL NETWORK ARCHITECTURES

2.1 Introduction

In this chapter, a number of Recurrent Neural Network (RNN) architectures are discussed. The discussion will be restricted to discrete-time systems, as pointed out in chapter 1. Training algorithms for these architectures, that can train the networks (using examples) so that they will compute a certain useful function of the input data, are discussed in the next chapter.

All architectures are based on the structure of standard feed-forward neural networks. Neurons, layers of neurons, or entire parts of multilayer perceptrons can be found in discrete-time RNN combined with some temporal processing (using time-delay elements) of data and recurrent connections (also called feedback connections).

Aim

This chapter presents the different recurrent neural network architectures, encountered in literature during this project. It aims to categorize the architectures which may facilitate the choice of an architecture at a later stage.

Overview of this chapter

In the first three sections, a number of discrete-time RNN architectures are presented. The architectures are discussed from two dynamic systems viewpoints: the state-space model (in section 2.2) and the input-output model (in section 2.3). The last section (2.4) presents modular architectures that are best described in yet another way.

The architectures treated in this chapter can be ordered hierarchically since some architectures are special cases of more general architectures. This hierarchy is visualized in figure 2.1. The numbers in brackets refer to the corresponding sections in this chapter.

The most general architectures are at the left, specific architectures are at the right. The accolades show what architectures are part of a more general architecture description.

Recurrent neural networks with continuous- or quantized values

All analysis of neural networks in this chapter is done under the assumption of continuous internal signal values (i.e. real numbers).

2.2 Architectures based on the State-space model

This section presents a number of classes of RNN architectures. All architectures can be best described using the state-space model from systems theory. This state-space model will be introduced in subsection 2.2.1. After that, the following architectures or classes of architectures are presented:

- Fully Recurrent Neural networks (FRNN); subsection 2.2.2
- Subsets of FRNN: Recurrent Neural Networks (RNN); subsection 2.2.3
- Partially Recurrent Networks (PRN); subsection 2.2.4
- Simple Recurrent Networks (SRN); subsection 2.2.5

These architectures emerge by applying constraints to the general state-space model. The architectures have been investigated and tested in applications by many researchers. In the following subsections, these specific constraints will be listed and the resulting architectures will be discussed. Each class is presented together with a quick look at some properties and examples of their application.

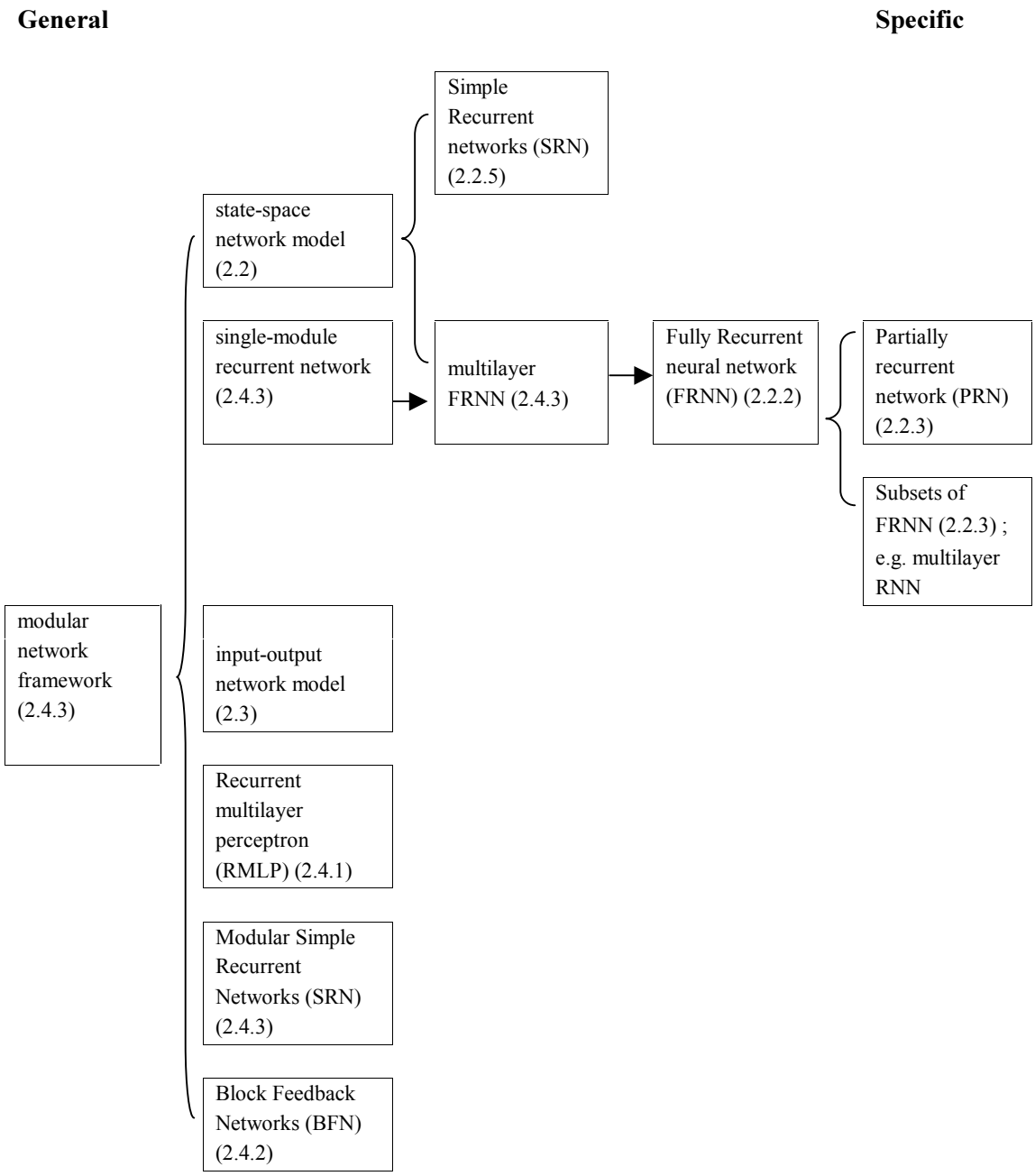


Figure 2.1; Recurrent neural network architectures hierarchy (numbers indicate sections)

2.2.1 General state-space model

The *first general state-space model* will be presented first. After this model, is the second general state-space model is introduced. The second model is actually a special case of the first model, so it is less general but it is still important enough to be presented as a separate model.

Because the first model is the most general, it is also referred to as the *general state-space model* in this report.

The first general state-space model

In systems theory, a general model to describe a discrete-time nonlinear system is a nonlinear state-space model. The system-input vector at time n is $\mathbf{u}(n)$, the output vector is $\mathbf{y}(n)$ and the state of the system is $\mathbf{x}(n)$. The general state-space model is given by:

$$\begin{aligned} \mathbf{x}(n+1) &= \mathbf{F}(\mathbf{x}(n), \mathbf{u}(n)) \\ \mathbf{y}(n) &= \mathbf{G}(\mathbf{x}(n), \mathbf{u}(n)) \end{aligned} \qquad (2.1\text{ a,b})$$

The following (loose) definition describes what the *state* is: The state of a system summarizes the past of the system insofar as relevant for its future behavior [Kwakernaak e.a., 1991].

It is generally assumed that any discrete-time dynamic system can be described by a suitable state-space model. From this it follows that all discrete-time RNN architectures can be described by a suitable state-space model.

But the state-space model does not have to be the default choice for describing a discrete-time system. Network architectures that are better described in another way are treated in section 2.3 on the input-output model and in section 2.4 on modular architectures.

In this section, neural network architectures are presented that can be best described using a state-space model. The (non)linear vector functions **F**(.) and **G**(.) are replaced by more specific functions, realized by neural networks. Each of these functions can of course also be realized by a group of several neural networks, in other words a modular neural network.

A general state-space neural network architecture realizing the equations 2.1 is shown in figure 2.2a.

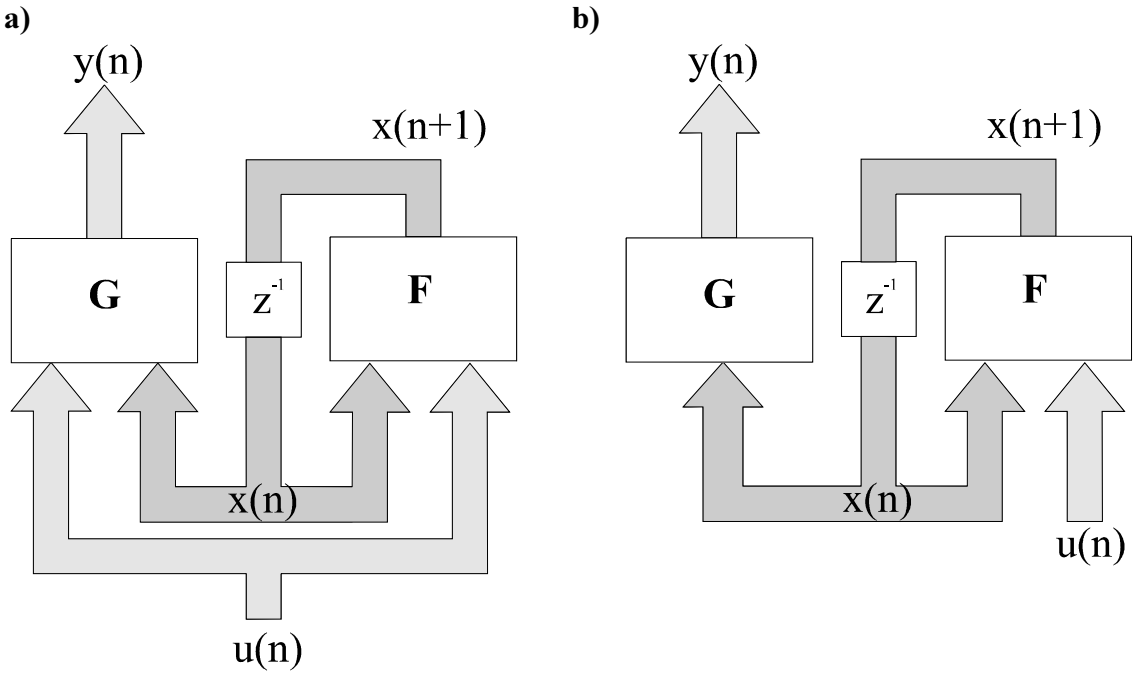


Figure 2.2; First (a) and second (b) state-space neural network architectures with neural networks realizing F(.) and G(.)

The second general state-space model

In the second general state-space model the system output is a function **G**(.) of the current state only. It is therefore a special case of the first general state-space model. It is given by:

$$\begin{aligned} \mathbf{x}(n+1) &= \mathbf{F}(\mathbf{x}(n), \mathbf{u}(n)) \\ \mathbf{y}(n) &= \mathbf{G}(\mathbf{x}(n)) \end{aligned} \qquad (2.2)$$

Some RNN architectures can be best described using this second model. A general state-space neural network architecture realizing the equations 2.2 is shown in figure 2.2b.

Control theory assumption

The limitation that the output is a function of the current state $\mathbf{x}(n)$ only, is a result of the common control theory assumption, that the output of a (continuous) dynamic system to be controlled or identified by the neural network cannot react instantaneously on the applied input

(for example assumed in [Wentink, 1996] or [Santini, 1995a]). Then the second state-space model adequately describes the dynamic system and the first state-space model is not used.

A modification to the second state-space model

Sometimes a modification to the second state-space model is made. In this model the output of the function F (the next state) is fed to function G directly without passing through a delay element:

$$\begin{aligned} \mathbf{x}(n+1) &= \mathbf{F}(\mathbf{x}(n), \mathbf{u}(n)) \\ \mathbf{y}'(n) &= \mathbf{G}(\mathbf{x}(n+1)) \end{aligned} \tag{2.3}$$

Obviously the dynamic properties of this model are the same as the equations 2.2 because the process equation is identical. The output sequence however is advanced by one time step, $\mathbf{y}'(n) = \mathbf{y}(n+1)$.

Properties of the general state-space models

The task of training this network to realize the correct function **F**(.) poses a special problem, because target values for the output of **F**(.) given the input are not known in most cases, since only the desired input-output behavior is known. This problem, that can be solved by error backpropagation algorithms, is examined in chapter 3 on training algorithms.

The first state-space network model can be viewed as a static neural network **G** extended with an **F** module that allows for dynamic operation.

Approximation of dynamic systems

As will be shown in section 4.1, the general state-space neural network architecture is capable of approximating any dynamic system.

Description using layers

The functions F and G as used in this subsection are yet unspecified. However, static networks normally have a layered architecture (as can be found in textbooks on neural networks). Each layer computes a function of the output of the previous layer. A general layered architecture with *i* layers is described by the following equations:

$$\begin{aligned} \mathbf{x}(n+1) &= \mathbf{F}(\mathbf{x}(n), \mathbf{u}(n)) = \mathbf{F}_{\mathrm{Li}}(...\mathbf{F}_{\mathrm{L3}}(\mathbf{F}_{\mathrm{L2}}(\mathbf{F}_{\mathrm{L1}}(\mathbf{x}(n), \mathbf{u}(n))))...) \\ \mathbf{y}(n) &= \mathbf{G}(\mathbf{x}(n), \mathbf{u}(n)) = \mathbf{G}_{\mathrm{Li}}(...\mathbf{G}_{\mathrm{L3}}(\mathbf{G}_{\mathrm{L2}}(\mathbf{G}_{\mathrm{L1}}(\mathbf{x}(n), \mathbf{u}(n))))...) \end{aligned} \tag{2.4}$$

where the functions $\mathbf{F}_{\mathbf{Lj}}$, $\mathbf{G}_{\mathbf{Lj}}$ denote the computations performed by layer *j* of the F and G network respectively. The layered description is defined here because it will be used later in introducing the state-space SRN (subsection 2.2.5).

Applications

In most applications the state-space neural network architectures as presented here, are not used. In system identification and control applications, the unknown target values for module **F** are the main reason to dismiss a general state-space neural network. See subsection 4.1.7 for a further discussion.

Other applications like classification are generally not viewed from a system theory viewpoint so the general state-space neural network is not used, but the 'subsets' of the state-space network (to be treated in the remainder of this section) are used.

2.2.2 Fully Recurrent Neural Network (FRNN)

The Fully Recurrent Neural Network (FRNN) is first described here in terms of individual neurons and their connections, as was done in [Williams e.a., 1989]. Then the FRNN is considered as a special case of the general state-space model and a convenient matrix notation of the network is given.

The name 'Fully Recurrent Neural Network' for this network type is proposed by [Kasper e.a., 1994]. Another name for this type of network is the 'Real-Time Recurrent Network'. This name will not be used further, because the name strongly implies that training is accomplished using

the Real-Time Recurrent Learning (RTRL) algorithm proposed for this network in [Williams e.a., 1989] which is not necessarily the case because other algorithms can be used.

In general a FRNN has N neurons, M external inputs and L external outputs. In figure 2.3 an example of a FRNN is given which has N=4 neurons, M=2 external inputs $u_1(n)$, $u_2(n)$ and L=2 external outputs $y_1(n)$, $y_2(n)$.

The network is called Fully Recurrent because the output of all neurons are recurrently connected (through N delay elements and N^2 weighted feedback connections) to all neurons in the network. The external network inputs are connected to the neurons by N*M feedforward connections *without* delay element. A bias (also called threshold) can be introduced for every neuron by applying a constant external input $u_1(n) = 1$ to the network.

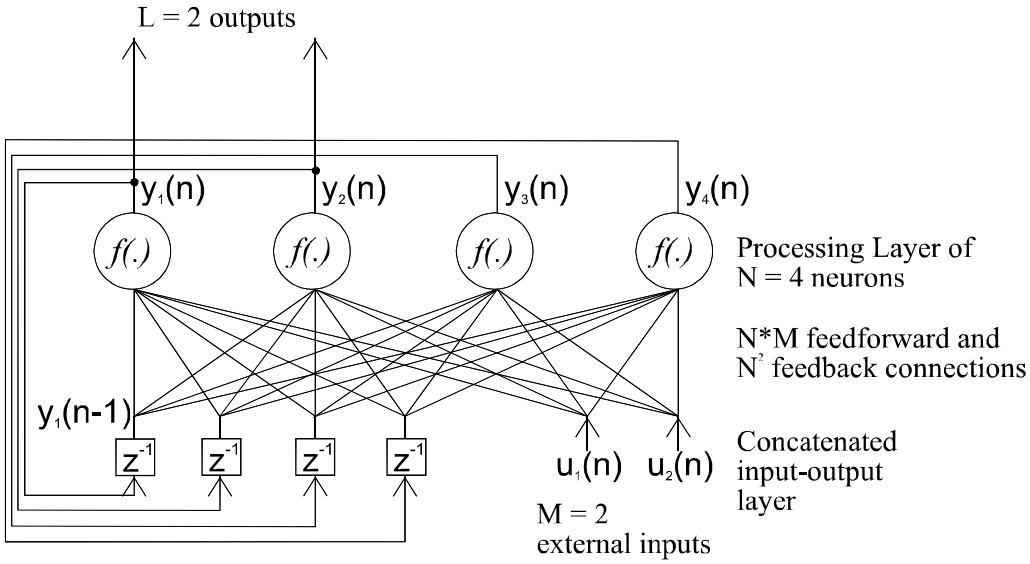


Figure 2.3; Example of a Fully Recurrent Neural Network (of type 1)

The definition of layers

For static neural networks, the number of layers in the neural network can be clearly defined as the number of neurons an input signal passes through before reaching the output. For the FRNN however the same definition is ambiguous, because signals applied at time n are fed back and reach the output at times n, n+1, and so on. The term layer therefore appears to be never used in literature in FRNN descriptions.

By redefining the concept of layer to: the *minimum* number of neurons an input signal passes through before reaching the output, a workable definition is obtained for the FRNN. Also this definition can be equally well used for subsets of the FRNN that will be described later on in section 2.2. It also makes much sense in describing the single module recurrent network (subsection 2.4.3). Now the network can be seen as a single layer feed-forward network, extended with delayed recurrent connections.

If the layer definition is not used, the (N-L) hidden neurons numbered $n > L$ can also be seen as a hidden layer because these are not connected as an external network output.

Two types of FRNN

The network as shown, having delay-less inputs, will be referred to as a *type 1 FRNN*. Often a slightly different FRNN appears in literature (e.g. [Haykin, 1994] and [Williams e.a., 1995]) that uses delayed input values instead. This type of network will be referred to as a *type 2 FRNN*. Both types will be introduced first. At the end of this subsection it will be shown that the two types have exactly the same dynamic behavior, so the use of either type in an application is an arbitrary choice.

Network dynamics for the type 1 FRNN

Define the external input vector $\mathbf{u}(n)$ that holds the inputs $u_i(n)$ applied at time n. The neuron outputs $y_i(n)$ are elements of the combined output/state vector $\mathbf{y}(n)$. To describe the network dynamics, the external input $\mathbf{u}(n)$ and the delayed output/state $\mathbf{y}(n\text{-}1)$ are first grouped together to form the (N+M)-by-1 extended input vector $\mathbf{z}(n)$:

$$\mathbf{z}(n) = \begin{bmatrix} \mathbf{y}(n-1) \\ \mathbf{u}(n) \end{bmatrix} \tag{2.5}$$

The following equations describe the network dynamics. First the extended inputs are multiplied by weights w_{ij} and summed up to the sums $s_i(n)$. The network outputs are computed by passing the sums through the neuron activation function $f(.)$.

$$\begin{aligned} s_i(n) &= \sum_j w_{ij} \cdot z_j(n) \\ y_i(n) &= f(s_i(n)) \end{aligned} \tag{2.6 a,b}$$

Network dynamics for the type 2 FRNN

The network with delayed external inputs has slightly different dynamic behavior. As can be seen in figure 2.4a and equation 2.7a below, the input is now delayed.

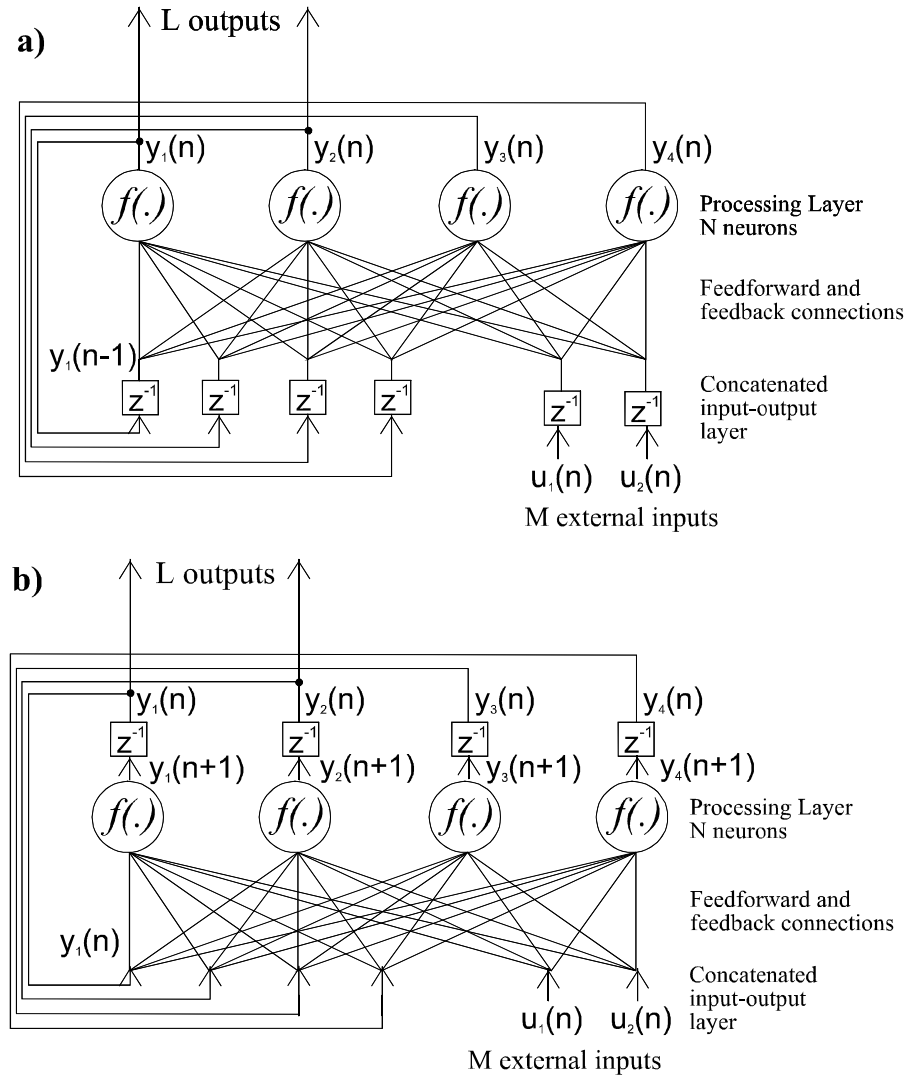


Figure 2.4; A type 2 FRNN presented in two different ways: a) with delayed external inputs ; b) with delayed neuron outputs

$$\begin{aligned} \mathbf{z}^*(n) &= \begin{bmatrix} \mathbf{y}(n-1) \\ \mathbf{u}(n-1) \end{bmatrix} \\ s_i^*(n) &= \sum_j w_{ij} \cdot z_j^*(n) \\ y_i(n) &= f(s_i^*(n)) \end{aligned} \qquad (2.7\text{ a,b,c})$$

The network extended input $\mathbf{z}^*(n)$ consists of the delayed vectors $\mathbf{y}(n\text{-}1)$ and $\mathbf{u}(n\text{-}1)$. These delays can be seen in figure 2.4a on both $\mathbf{y}(.)$ and $\mathbf{u}(.)$ This notation is not found in literature, probably because the definition of the extended input vector $\mathbf{z}(n)$ is a bit confusing: the time index n is shifted. So the extended input vector is commonly defined as

$$\mathbf{z}(n) = \begin{bmatrix} \mathbf{y}(n) \\ \mathbf{u}(n) \end{bmatrix} \qquad (2.8)$$

leading to the following equations for the network dynamics, which are equivalent to the equations 2.7:

$$\begin{aligned} \mathbf{z}(n) &= \begin{bmatrix} \mathbf{y}(n) \\ \mathbf{u}(n) \end{bmatrix} \\ s_i(n) &= \sum_j w_{ij} \cdot z_j(n) \\ y_i(n+1) &= f(s_i(n)) \end{aligned} \qquad (2.9\text{ a,b,c})$$

The delays have been moved from the inputs to the output of every neuron (the delay is now present in equation 2.9c). The resulting network structure is shown in figure 2.4b. The two networks in figure 2.4a,b are equivalent in input-output behavior, but the intermediate summing expression for $s_i(n)$ is not equal to $s_i^*(n)$ defined in equation 2.7b, but rather $s_i(n) = s_i^*(n+1)$.

State-space model and matrix notation of the type 1 network

A state-space description of the type 1 FRNN will be derived. To obtain the state-space model, the equations 2.6 for the network dynamics are converted to a matrix notation. Define the N-by-N weight matrix $\mathbf{W_x}$ containing all recurrent weights and the N-by-M weight matrix $\mathbf{W_u}$ containing all external input weights:

$$\mathbf{W_x} = \begin{bmatrix} w_{11} & w_{12} & .. & w_{1N} \\ w_{21} & w_{22} & & \cdots \\ \cdots & & & \cdots \\ w_{N1} & .. & .. & w_{NN} \end{bmatrix} \quad \mathbf{W_u} = \begin{bmatrix} w_{1;N+1} & w_{1;N+2} & .. & w_{1;N+M} \\ w_{2;N+1} & w_{2;N+2} & & \cdots \\ \cdots & & & \cdots \\ w_{N;N+1} & .. & .. & w_{N;N+M} \end{bmatrix} \qquad (2.10)$$

and the total network weight matrix $\mathbf{W}$ containing all the weights:

$$\mathbf{W} = \begin{bmatrix} \mathbf{W_x} & \mathbf{W_u} \end{bmatrix} \qquad (2.11)$$

Let the diagonal mapping $\mathbf{F}(.)$ (also referred to as the vector function $\mathbf{F}(.)$) denote all neuron activation functions f:

$$\mathbf{F}: \begin{bmatrix} s_1 \\ s_2 \\ \cdots \\ s_N \end{bmatrix} \rightarrow \begin{bmatrix} f(s_1) \\ f(s_2) \\ \cdots \\ f(s_N) \end{bmatrix} \qquad (2.12)$$

The network dynamics of equation 2.6 can now be written in an equivalent matrix notation:

$$\mathbf{y}(n) = \mathbf{F}(\mathbf{W} \cdot \mathbf{z}(n)) = \mathbf{F}(\mathbf{W}_x \cdot \mathbf{y}(n-1) + \mathbf{W}_u \cdot \mathbf{u}(n)) \qquad (2.13)$$

The equation above can be rewritten to show more clearly it is a special case of the general form of a state-space system (equations 2.1). For this purpose the following vectors of the state-space system S are defined: the state vector $\mathbf{x_S}(n)$, the input vector $\mathbf{u_S}(n)$, the output vector $\mathbf{y_S}(n)$ and the vector functions $\mathbf{F_S}(.)$ and $\mathbf{G_S}(.)$. If we define these vectors as:

$$\begin{aligned} \mathbf{u}_S(n) &= \mathbf{u}(n) \\ \mathbf{x}_S(n) &= \mathbf{y}(n-1) \\ \mathbf{y}_S(n) &= \begin{bmatrix} y_1(n) & y_2(n) & \dots & y_L(n) \end{bmatrix} \end{aligned} \tag{2.14}$$

the network can be written as a state-space model S:

$$\begin{aligned} \mathbf{x}_S(n+1) &= \mathbf{F}_S(\mathbf{x}_S(n), \mathbf{u}_S(n)) &&= \mathbf{F}(\mathbf{W}_x \cdot \mathbf{x}_S(n) + \mathbf{W}_u \cdot \mathbf{u}_S(n)) \\ \mathbf{y}_S(n) &= \mathbf{G}_S(\mathbf{x}_S(n), \mathbf{u}_S(n)) &&= \mathbf{C} \cdot \mathbf{F}(\mathbf{W}_x \cdot \mathbf{x}_S(n) + \mathbf{W}_u \cdot \mathbf{u}_S(n)) \end{aligned} \tag{2.15 a,b}$$

In these equations, matrix **C** is the L-by-N matrix

$$\mathbf{C} = \begin{bmatrix} \mathbf{I}_L & \mathbf{0}_{L;N-L} \end{bmatrix} \tag{2.16}$$

where $\mathbf{I_L}$ is the L-by-L unity matrix and $\mathbf{0_{L;N-L}}$ the L-by-(N-L) zero matrix. As a result of this definition of **C**, the output vector $\mathbf{y_S}(n)$ consists of the first L elements of the output/state-vector $\mathbf{y}(n)$. The state-space model S conforms to the first general form of the state-space model presented in the previous section.

Note that the function $\mathbf{F_S}$ appears 'duplicated' inside the function $\mathbf{G_S}$:

$$\begin{aligned} \mathbf{x}_S(n+1) &= \mathbf{F}_S(\mathbf{x}_S(n), \mathbf{u}_S(n)) \\ \mathbf{y}_S(n) &= \mathbf{C} \cdot \mathbf{F}_S(\mathbf{x}_S(n), \mathbf{u}_S(n)) = \mathbf{C} \cdot \mathbf{x}_S(n+1) \end{aligned} \tag{2.17}$$

so one could say the output is simple a function of the next state.

In figure 2.5 the example type 1 FRNN is again shown. All variables in the network are now shown in the state-space notation introduced above.

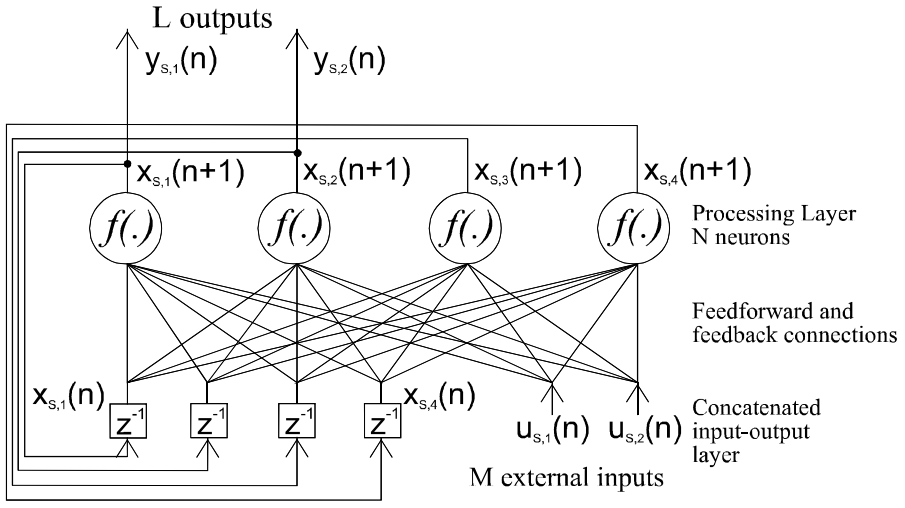


Figure 2.5; The type 1 example FRNN with all variables shown in state-space notation

State-space model and matrix notation of the type 2 network

The type 2 network can be written as a new state-space system T if the following variables are defined:

$$\begin{aligned} \mathbf{u}_T(n) &= \mathbf{u}(n) \\ \mathbf{x}_T(n) &= \mathbf{y}(n) \\ \mathbf{y}_T(n) &= \begin{bmatrix} y_1(n) & y_2(n) & \dots & y_L(n) \end{bmatrix} \end{aligned} \tag{2.18}$$

The different definition of the state $\mathbf{x_T}$ results from the delay in the neuron outputs (see equation 2.9c). This gives the following state-space system:

$$\begin{aligned} \mathbf{x}_T(n+1) &= \mathbf{F}_T(\mathbf{x}_T(n), \mathbf{u}_T(n)) &&= \mathbf{F}(\mathbf{W}_x \cdot \mathbf{x}_T(n) + \mathbf{W}_u \cdot \mathbf{u}_T(n)) \\ \mathbf{y}_T(n) &= \mathbf{G}_T(\mathbf{x}_T(n)) &&= \mathbf{C} \cdot \mathbf{x}_T(n) \end{aligned} \tag{2.19}$$

In this case the output only depends on the current state vector $\mathbf{x_T}(n)$. So the state-space model T conforms to the second form of general state-space model. In figure 2.6 the example type 2 FRNN from figure 2.4b is shown again. All variables in the network are now shown in the state-space notation introduced above.

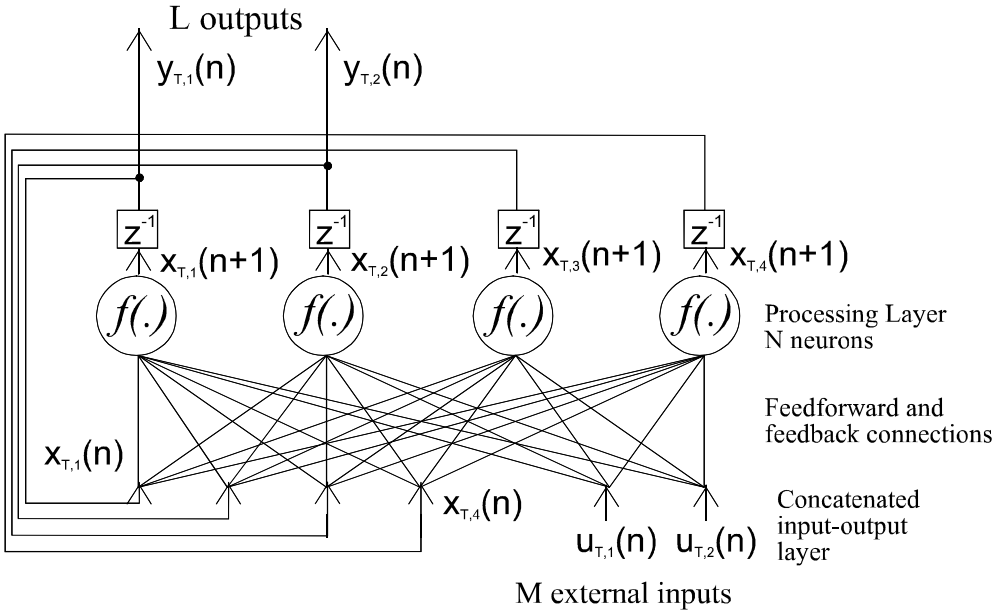


Figure 2.6; The type 2 example FRNN with all variables shown in state-space notation

Comparison of the two types of FRNN

The difference between the two types is illustrated in figure 2.7, where the computation of the two types of FRNN models in matrix notation is shown schematically. Both types have an identical dynamic behavior because the input and feedback structure of both models is exactly the same.

The only difference is in the output computation. In the type 1 FRNN, $\mathbf{x_S}(n+1)$ before the delay is used for this and in the type 2 FRNN, $\mathbf{x_T}(n)$ after the delay is used. For the outputs of both networks, this results in the relation $\mathbf{y_T}(n+1) = \mathbf{y_S}(n)$ which means the output of the type 2 FRNN is the same as the output of the type 1 FRNN, but delayed by one time step.

It can therefore be concluded that the choice between the two types is arbitrary because the output of the selected type can always be transformed (shifted in time by one time step) such that the output of the other type is obtained. Both types are encountered in literature on FRNN.

Applications of the FRNN

Identification and simulation of nonlinear systems

A FRNN can be used to identify or simulate a nonlinear dynamic system. It is shown in section 4.1 however, that a FRNN can not simulate all state-space systems.

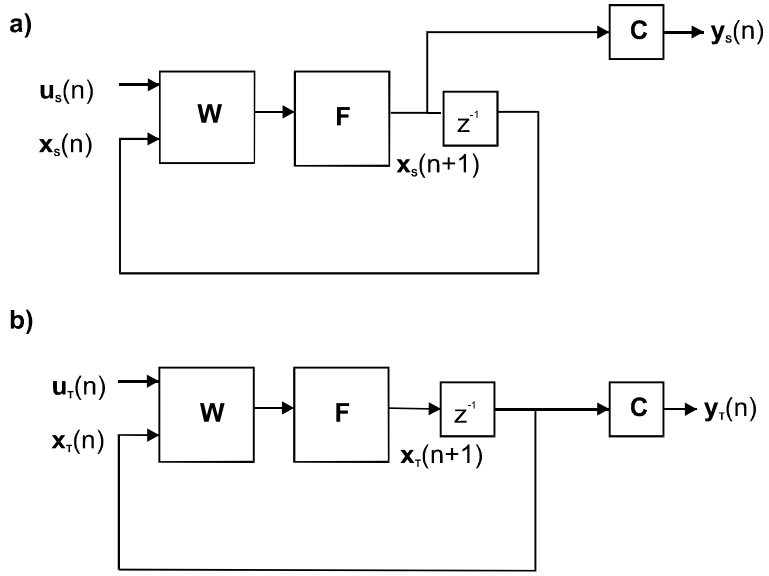


Figure 2.7; FRNN computation shown in 'matrix notation'; a) type 1 network ; b) type 2 network

Identification and simulation of linear systems
Using linear neuron transfer functions, the FRNN can be used to identify or simulate a linear state-space system. An example is not given here, because an example of linear state system simulation by the *partially recurrent neural network* will be given in subsection 2.2.3.

Speech recognition
FRNN were applied to speech recognition problems. See for example [Kasper e.a., 1994] and [Robinson, 1994].

Learning the behavior of Finite State Machines
FRNN can learn to behave like a Finite State Machine (FSM) by presenting examples of the output of the FSM to be identified. See also [Giles e.a., 1994] where a method is given to convert a representation of the FSM learned by the (continuous-valued) FRNN back to a discrete-valued representation.

2.2.3 Subsets of the FRNN: Recurrent Neural Networks (RNN)

Additional restrictions can be imposed on the FRNN architecture described in the previous subsection to create other (restricted) Recurrent Neural Network (RNN) architectures. This subsection will describe some of these restricted architectures. Because the FRNN can be written as a state-space model, all 'subsets' of FRNN are in many cases most conveniently written as state-space models.

The following categories of restrictions can be used (individually or in a combination):

1) forcing certain weights to zero (called removing or *pruning* the weight)
2) forcing weights to non-zero value (called *fixing* the weight or making the weight non-learnable)
3) forcing weights to be equal to other weights (called *sharing* of weights) or approximately equal to other weights (called *soft sharing* of weights)

These restrictions will be looked at in this subsection. Note that the three restrictions listed are fairly general and can be applied to other neural networks architecture than the FRNN, for example to the standard feedforward network.

All three restrictions have a property in common: the number of free parameters of the network is reduced when compared to a non-modified FRNN. Reasons for doing so will be given now. More reasons for applying restrictions will be given in the category descriptions.

Reduction of the number of free parameters

The training of a neural network is in fact a procedure that tries to estimate the parameters (weights) of the network such that an error measure is minimized. Reducing the number of parameters to be estimated may simplify training.

Another good reason for reducing the number of free parameters is to reduce training algorithm overhead, which often grows quickly for an increasing number of weights N_W. (Some analysis of computational overhead will be done in subsections 3.3.5 and 3.4.3.)

1. Forcing weights to zero (pruning)

Different RNN architectures can be created by removing connections from the FRNN. An example of a RNN that is not fully recurrent anymore is shown in figure 2.8a. This structure looks quite different from the FRNN architecture. However, it must be noted that all connections have a time-delay of 1, even the feed-forward connections between neurons. These delay elements are shown in the figure. All RNN structures having this property are in fact subsets of the FRNN and can be written in the default form, as is shown in figure 2.8b. In the figure the connections that are removed (pruned) from the general FRNN architecture are shown as dashed lines (so these connections are not actually present anymore).

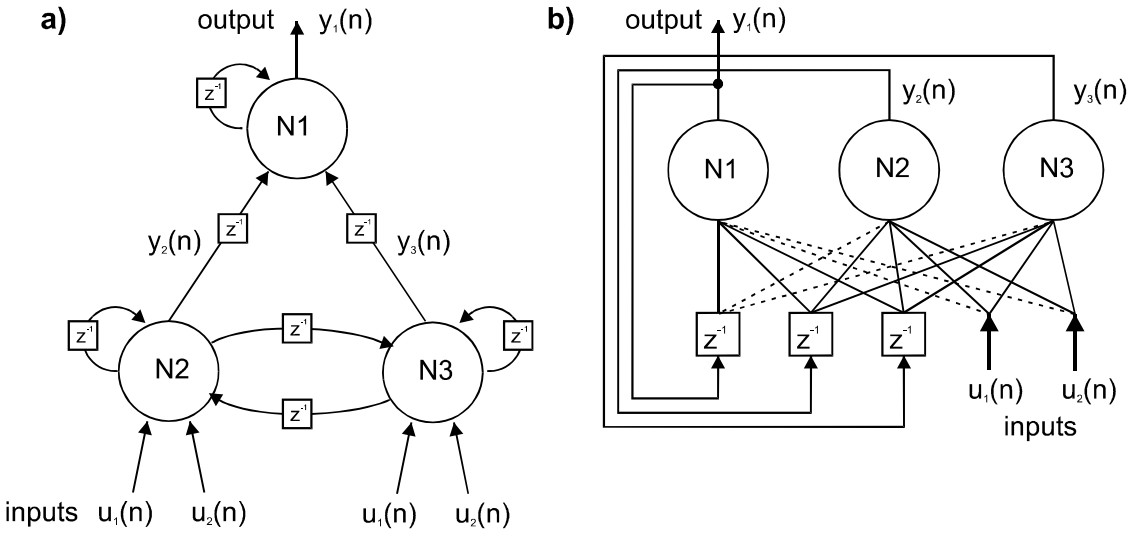


Figure 2.8; One example RNN presented as (a) a two-layer neural network with delays and recurrent connections ; and (b) as a FRNN with four removed connections

Specific reasons for pruning connections are:

- weights can be pruned to make the network computation and training algorithm computation more local in space. An architecture with for example only self-recurrent connections can be implemented efficiently on parallel computing devices
- less connections often simplify and therefore speed up the training algorithm used for the network

Sparse network

The term *sparse network* is used for a large network that has most connections pruned. Sparse networks try to combine the advantages of many neurons with the advantages of a relatively small number of weights. The Simple Recurrent Networks (SRN) in subsection 2.2.5 are sparse networks, for example.

Properties: Effective number of layers

The FRNN was called a one layer network, using the new layer definition in subsection 2.2.2. However, the RNN in figure 2.8a (and b) is now called a two layer network. In making this statement the new layer definition is used again.

One notable difference between this example RNN and any standard two-layer MLP is that the RNN has delay between external inputs and external outputs. In the example, the minimum

delay of an input signal to the output is D=1 time step. A consequence of this inherent delay D is that the training target for this RNN should also be delayed such, that a training example input $\mathbf{x}(n)$ at time n has no relation with the target $\mathbf{T}(m)$ that has to be learned, for all time $m < n$+D, simply because $\mathbf{x}(n)$ can't have any influence on the output for such a time m.

It should be noted that any effective number of layers can be created this way, by taking a large enough FRNN and pruning specific connections. For an example of a three-layer structure see [Koizumi e.a., 1996].

Applications to speech recognition

Many types of constrained RNN have been used in speech recognition (for example [Bengio, 1996] and [Koizumi e.a., 1996]). In their experiments, specific architectures were pre-selected and their performance was tested. In [Kasper e.a., 1994] it was argued that these kind of restrictions often seem arbitrarily chosen and may limit the network performance in an application. Kasper therefore used a FRNN (with no restrictions) for a speech recognition task.

The partially recurrent network

By pruning weights a well-known architecture, the partially recurrent network, is obtained. This network type will be discussed separately in subsection 2.2.4.

2. Forcing weights to non-zero constant values (fixing)

Some reasons for forcing weights to constant values, or *fixing* weights, are:

- A-priori knowledge about the problem can be coded into the network using fixed weights. This knowledge can't be 'unlearned' by the training algorithm in a direct way because the fixed connections can't be changed in the training process.
- Fixing certain weights can simplify or speed-up the training algorithm. But any neuron having a fixed connection can still use the information, that comes in through the fixed connection, in its computation. Some Simple Recurrent Networks (SRN) architectures also make use of fixed weights for this reason (see subsection 2.2.5).

An example of a network class using fixed connections is the K-L network [Frasconi e.a., 1995]. It contains a 'K' (knowledge) sub-network in which fixed connections are used to code a-priori knowledge, and an 'L' (learnable) sub-network that is trained to 'fine-tune' the rather coarse a-priori information. Some experiments on speech recognition were done [Frasconi e.a., 1995] with the K-L network. The total network is made up of two FRNN (the K and the L network) and a feedforward network that combines the output of both.

3. Sharing weights

Sharing weights (setting weights to be always equal to other weights) reduces the number of free parameters like pruning and fixing. Shared weights are still learnable, unlike fixed weights. In [Bengio, 1996] and [Bishop, 1995] weight sharing is further discussed and soft weight sharing (weights are approximately equal to other weights) is introduced.

In fact, a FRNN can in many cases be considered a large static feedforward neural network with shared weights across layers. This can be seen by unfolding the network in time (see section 3.4).

Another reason why weight sharing is used (in RNN used as associative memories, see chapter 1) is to guarantee stability of the dynamic system [Patterson, 1996].

Discussion

All the restrictions listed in this subsection are a form of application of very general a-priori knowledge about the problem to be solved. By applying restrictions, the number of different possible networks (which means, all networks that could be the result of a training procedure) is reduced, which can simplify the training problem. It is thereby assumed that an 'adequate solution' network is still among the reduced collection of possible networks.

The more correct a-priori knowledge about the task can be incorporated into the 'candidate' neural network structure, the better will be the expected performance of the neural network, applied to the task. The restrictions posed on the FRNN can therefore be a sensible application of a-priori knowledge.

The restrictions mentioned in this subsection are in fact 'hardcoded' into the network architecture. But it is also possible to dynamically impose or remove restrictions on the network parameters while the network is being trained. The restrictions to be applied are then prescribed by the training algorithm (adaptive structures were not investigated in this project).

2.2.4 Partially recurrent network (PRN)

The output vector $\mathbf{y}(n)$ of the FRNN consists of the first L elements of the state vector $\mathbf{x}(n)$, as was shown in figure 2.3. So the output signals are a 'subset' of state signals. In a general state-space description this is certainly not the case, the output is determined by a separate calculation (the output equation) which is some function of the external input and the state.

To obtain a network that effectively has separate state and output units (analogous to a state-space system that has separate process and output equations), the feedback connections from all L output neurons $y_i(n)$ with i=1…L are removed. An example of the *partially recurrent neural network* (PRN) [Robinson e.a., 1991], also named the *simplified recurrent neural network* [Janssen, 1998], that results is shown in figure 2.9. The name 'partially recurrent neural network' will be used in this report to avoid confusion in the terms simple/simplified recurrent networks in the next subsection.

This particular case of a constrained FRNN results from the FRNN by pruning certain weights (see subsection 2.2.3). It is discussed in more detail here, because it was found in speech recognition applications [Robinson, 1991] [Chen e.a., 1996] where it was used instead of a FRNN.

Example of a partially recurrent network

The example network has L=2 outputs , M=2 inputs and N=4 neurons from which L output neurons compute $y_i(n)$, and N-L state neurons compute $x_i(n+1)$. In this architecture, there is a clear separation between the L output neurons realizing the output equation **G**(.) and N-L neurons computing the process equation **F**(.) of the state-space model. It should be noted that both functions **F** and **G** are computed by a single layer of neurons.

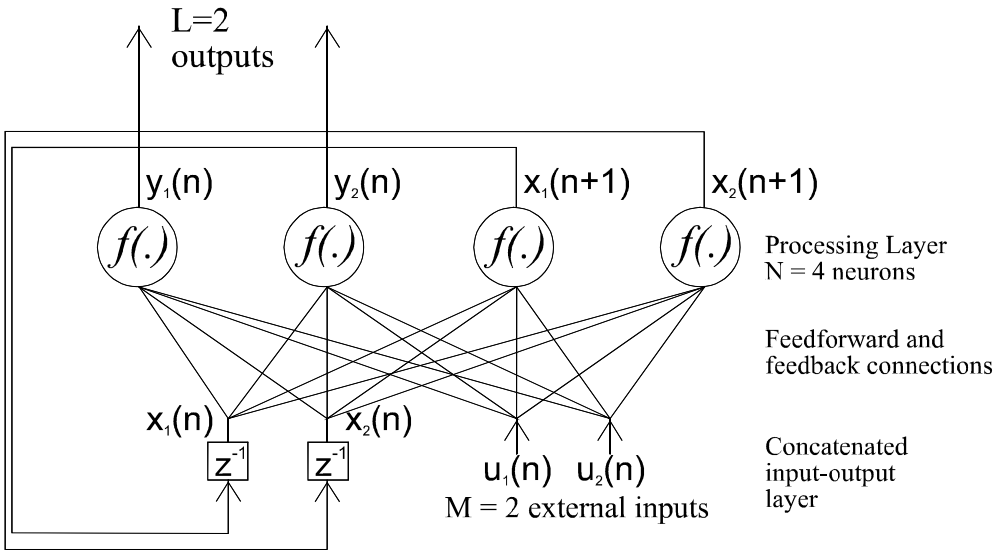


Figure 2.9; Example of a partially recurrent neural network

State-space model and matrix notation of the partially recurrent network

To obtain a state-space model of the partially recurrent network, a first approach would be to take the state-space model of the FRNN and just set the weights corresponding to the removed connections to zero.

There is however a more elegant way to represent the partially recurrent network. Because the L external outputs are not fed back anymore as inputs to the neurons, the values of these L

neurons outputs should not be part of the state vector $\mathbf{x}(n)$. Only neuron outputs that are fed back into the network, and thus summarize information of the past, should be part of the state.

Therefore a new state-space model for the partially recurrent network is now derived in which some variables of the system are redefined. The state vector $\mathbf{x}(n)$ is redefined to a state vector of size (N-L) and vectors $\mathbf{u}(n)$ and $\mathbf{y}(n)$ remain unchanged. The two weight matrices $\mathbf{W}_x$ and $\mathbf{W}_u$ (defined in equation 2.10) and the vector function $\mathbf{F}(.)$ (equation 2.12) are split up into upper parts (subscript U) and lower parts (subscript L) as follows:

$$\mathbf{W}_{x,U} = \begin{bmatrix} w_{1;L+1} & w_{1;L+2} & \cdots & w_{1N} \\ w_{2;L+1} & w_{2;L+2} & & \cdots \\ \cdots & & \cdots & \cdots \\ w_{L;L+1} & \cdots & \cdots & w_{LN} \end{bmatrix} \qquad \mathbf{W}_{x,L} = \begin{bmatrix} w_{L+1;L+1} & w_{L+1;L+2} & \cdots & w_{L+1;N} \\ w_{L+2;L+1} & w_{L+2;L+2} & & \cdots \\ \cdots & & \cdots & \cdots \\ w_{N;L+1} & \cdots & \cdots & w_{NN} \end{bmatrix} \tag{2.20}$$

$$\mathbf{W}_{u,U} = \begin{bmatrix} w_{1;N+1} & w_{1;N+2} & \cdots & w_{1;N+M} \\ w_{2;N+1} & w_{2;N+2} & & \cdots \\ \cdots & & \cdots & \cdots \\ w_{L;N+1} & \cdots & \cdots & w_{L;N+M} \end{bmatrix} \qquad \mathbf{W}_{u,L} = \begin{bmatrix} w_{L+1;N+1} & w_{L+1;N+2} & \cdots & w_{L+1;N+M} \\ w_{L+2;N+1} & w_{L+2;N+2} & & \cdots \\ \cdots & & \cdots & \cdots \\ w_{N;N+1} & \cdots & \cdots & w_{N;N+M} \end{bmatrix} \tag{2.21}$$

$$\mathbf{F}_U : \begin{bmatrix} S_1 \\ \cdots \\ S_L \end{bmatrix} \to \begin{bmatrix} f(S_1) \\ \cdots \\ f(S_L) \end{bmatrix} \qquad \mathbf{F}_L : \begin{bmatrix} S_{L+1} \\ \cdots \\ S_N \end{bmatrix} \to \begin{bmatrix} f(S_{L+1}) \\ \cdots \\ f(S_N) \end{bmatrix} \tag{2.22}$$

These matrices fit into the original weight matrices (defined in equation 2.10) in the following way:

$$\mathbf{W}_x = \begin{bmatrix} \mathbf{0}_{L;L} & \mathbf{W}_{x,U} \\ \mathbf{0}_{N-L;L} & \mathbf{W}_{x,L} \end{bmatrix} \qquad \mathbf{W}_u = \begin{bmatrix} \mathbf{W}_{u,U} \\ \mathbf{W}_{u,L} \end{bmatrix} \qquad \mathbf{F}(.) = \begin{bmatrix} \mathbf{F}_U(.) & \mathbf{0}_{L;N-L} \\ \mathbf{0}_{N-L;L} & \mathbf{F}_L(.) \end{bmatrix} \tag{2.23}$$

such that $\mathbf{W}_{x,U}$ is L-by-(N-L) ; $\mathbf{W}_{x,L}$ is (N-L)-by-(N-L) ; $\mathbf{W}_{u,U}$ is L-by-M ; $\mathbf{W}_{u,L}$ is (N-L)-by-M ; $\mathbf{F}_U(.)$ is L-by-L ; $\mathbf{F}_L(.)$ is (N-L)-by-(N-L) and $\mathbf{0}_{a;b}$ is the a-by-b zero matrix. In the matrix $\mathbf{W}_x$ above it can be seen which connections are effectively set to zero. The following state-space description results:

$$\begin{aligned} \mathbf{x}_S(n+1) &= \mathbf{F}_S(\mathbf{x}_S(n), \mathbf{u}_S(n)) &&= \mathbf{F}_L(\mathbf{W}_{x,L} \cdot \mathbf{x}_S(n) + \mathbf{W}_{u,L} \cdot \mathbf{u}_S(n)) \\ \mathbf{y}_S(n) &= \mathbf{G}_S(\mathbf{x}_S(n), \mathbf{u}_S(n)) &&= \mathbf{F}_U(\mathbf{W}_{x,U} \cdot \mathbf{x}_S(n) + \mathbf{W}_{u,U} \cdot \mathbf{u}_S(n)) \end{aligned} \tag{2.24}$$

Application: the linear network case

A partially recurrent network can be used to simulate any N_L-dimensional linear state system L of the form:

$$\begin{aligned} \mathbf{x}_L(n+1) &= \mathbf{A}_L \mathbf{x}_L(n) + \mathbf{B}_L \mathbf{u}_L(n) \\ \mathbf{y}_L(n) &= \mathbf{C}_L \mathbf{x}_L(n) + \mathbf{D}_L \mathbf{u}_L(n) \end{aligned} \tag{2.25}$$

when the neuron transfer functions are linear. In this case, the network structure of the partially recurrent network exactly matches the linear state-space system equations. The linear N_L-dimensional system L of equation 2.25 can be simulated by a partially recurrent network by choosing $\mathbf{W}_{x,L}=\mathbf{A}$; $\mathbf{W}_{u,L}=\mathbf{B}$; $\mathbf{W}_{x,U}=\mathbf{C}$; $\mathbf{W}_{u,U}=\mathbf{D}$, and $N=N_L+L$ linear neurons in the equations 2.24 for the PRN.

For pure linear system applications however, a neural network perspective is almost never used. This is sensible because well-known and more effective methods already exist for linear systems. [Haykin, 1998].

Application of the partially recurrent network structure to speech recognition

A nonlinear partially recurrent network was used in [Robinson e.a., 1991] for speech recognition. The reason for not using an FRNN was reduction in the number of weights and the argument that the output and state do not have to coincide (as is the case in the FRNN).

2.2.5 Simple Recurrent Networks (SRN)

Simple Recurrent Networks (SRN) are a category of neural networks that have far simpler recurrent connections, compared to the FRNN. This category consists of networks that have only 'one-to-one' (or 'simple') recurrent connections. This means that a certain neuron output is fed back only to the input of one neuron (possibly itself), not to all neurons. Figure 2.10 shows four examples of SRN architectures. Only the recurrent connections have a time delay, the forward connections are instantaneous. Thin arrows represent one-to-one connections and the broad arrows represent fully interconnected layers.

By convention [Hertz e.a., 1991], the network layers that use the delayed recurrent connections in their computation, are called *context layers* because they use the state of the network which represents a certain context (of previous network activation).

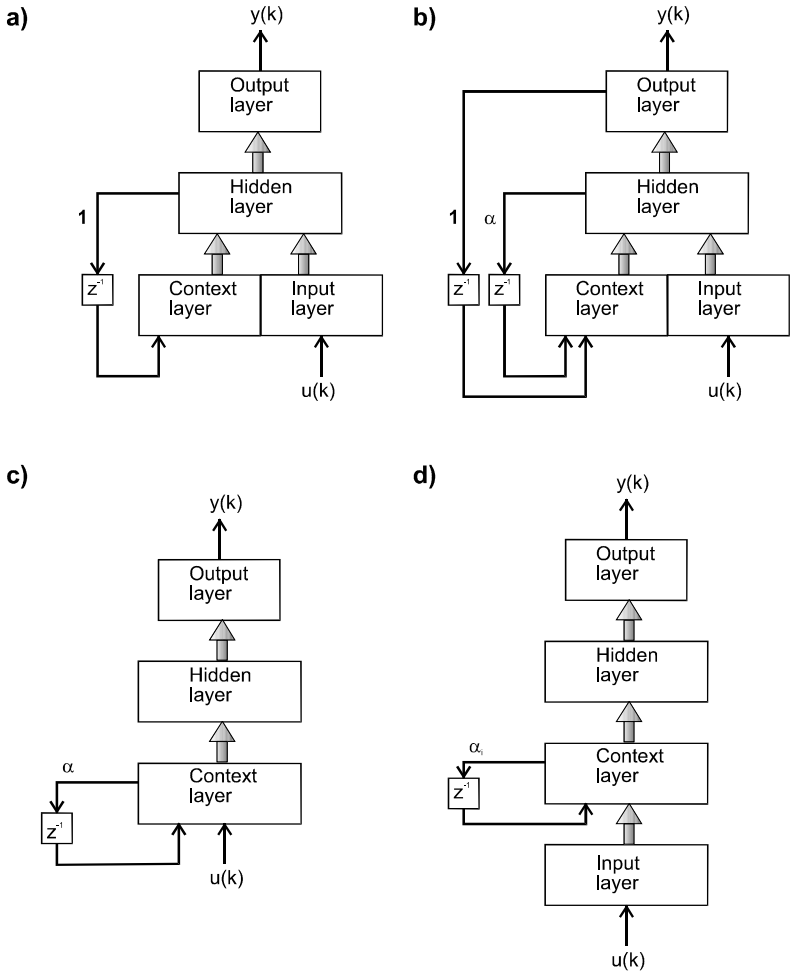


Figure 2.10: Four examples of SRN architectures. Thin arrows represent 'one-to-one' recurrent connections between layers. Broad arrows represent fully connected layers

Layer definition

Again the new layer definition of subsection 2.2.2 gives the number of layers in SRN structures. For the examples the number of layers is 3, 3, 3, and 4 for a, b, c, d respectively. In the figure each layer is drawn on its own vertical position.

SRN Extensions

There are many possible extensions of these architectures, not shown in figure 2.10. For example the use of multiple sets of context units as noted in [Hertz e.a., 1991] or different time delays for different context layers.

The state-space SRN

The general state-space neural network (subsection 2.2.1) can be straightforwardly modified such that a SRN structure is obtained. This new structure will be called the *state-space SRN* in this report. The assumption from subsection 2.2.1 is used that the function realized by the neural network can be decomposed into layers. Then, the function **F**(.) in the first general state-space model can be decomposed as follows:

$$\mathbf{x}_S(n+1) = \mathbf{F}_S(\mathbf{x}_S(n), \mathbf{u}_S(n)) = \mathbf{F}_{Li}(...\mathbf{F}_{L3}(\mathbf{F}_{L2}(\mathbf{F}_{L1}(\mathbf{x}_S(n), \mathbf{u}_S(n))))...) \tag{2.26}$$

where i is the number of layers and $\mathbf{F}_{Lj}$ is the calculation performed by layer j. The state-space SRN is now obtained by simply taking the following first layer function:

$$\mathbf{F}_{L1}(\mathbf{x}_S(n), \mathbf{u}_S(n)) = \mathbf{F}(\mathbf{W}_{x,DIAG} \cdot \mathbf{x}_S(n) + \mathbf{W}_u \cdot \mathbf{u}_S(n)) \tag{2.27}$$

where **F** describes the layer 1 neuron transfer functions, $\mathbf{W}_u$ is a normal N_1-by-N_S fully filled input weight matrix and $\mathbf{W}_{x,DIAG}$ is a *diagonal* N_S-by-N_S recurrent weight matrix. The total state-space SRN is described by:

$$\begin{aligned} \mathbf{x}_S(n+1) &= \mathbf{F}_{Li}(...\mathbf{F}_{L3}(\mathbf{F}_{L2}(\mathbf{F}(\mathbf{W}_{x,DIAG} \cdot \mathbf{x}_S(n) + \mathbf{W}_u \cdot \mathbf{u}_S(n))))...) \\ \mathbf{y}_S(n) &= \mathbf{G}_S(\mathbf{x}_S(n), \mathbf{u}_S(n)) \end{aligned} \tag{2.28}$$

The requirement of a diagonal recurrent weight matrix corresponds to first taking N_S input neurons in the first layer of the F-network, and second, allowing only recurrent connections from each output neuron to *one* other layer 1 input neuron. This last property makes it a SRN.

Properties

The SRN architectures can be divided into two substantially different classes, with the historical development of SRN in mind. The first class has fixed recurrent connections (not learnable), while the second class has learnable recurrent weights. The properties of these two classes will be briefly looked at.

1. Fixed recurrent weights

The first class of SRN has fixed recurrent weights. They are set to a fixed value $\alpha \leq 1$. When the Simple recurrent networks in figure 2.10a,b are used together with fixed recurrent connections, the resulting architectures are respectively the Elman network and the Jordan network [Hertz e.a., 1991]. These are historically the first discrete-time RNN architectures used as sequence mapping systems.

An advantage of this class of networks, according to [Hertz e.a., 1991], is that the fixed feedback weights don't have to be updated by a training algorithm. Hertz suggests that standard backpropagation training algorithms for MLP are not able to update the recurrent weights. But if the signals on the recurrent connections are just considered to be additional inputs, then the recurrent weights *are* learnable by standard backpropagation training.

So it is not clear why fixed recurrent weights were used for these first discrete-time RNN architectures if it is possible to let the recurrent weights be trained using a standard backpropagation algorithm.

A possible reason for this could be that using standard backpropagation training for a SRN is an ad-hoc method, i.e. it does not result by mathematically deriving an algorithm from an expression for the network error but it is rather an intuitive extension of existing training algorithms for static neural networks.

2. Learnable recurrent weights

The second class of SRN architectures has learnable feedback connections. As stated above these feedback weights could be trained ad-hoc by using the standard backpropagation training. It is also possible to derive a training algorithm. Architecture-specific training algorithms to update the feedback weights α_i were derived in literature [Hertz, 1991].

Some SRN are state-space neural networks
Some SRN are special cases of the state-space neural network architecture. They can be fitted into the state-space neural network description when the delay elements used all have unit delay. In Appendix A.1 this is shown for two example SRN.

Applications
SRN architectures were among the first discrete-time RNN architectures that have been developed. They have been used for applications of sequence recognition (e.g. speech recognition), sequence generation and simulating Finite State Machines [Hertz e.a., 1991].

Discussion
Soon after SRN were used first as sequence mapping neural networks, more complicated and powerful fully connected networks (like the FRNN) were proposed and research focused on these architectures.

Some aspects of SRN are however still widely used in research, for example the simple one-to-one recurrent connections and the delayed feedback from more than one other layer (e.g. in [Bengio, 1996]). These aspects can be introduced into complex networks with multiple context layers and multiple groups of delay elements (that may each have a different time-delay). These complex networks are best described as modular networks, which will be introduced in section 2.4.

Such a specific network architecture can perform better on a given problem than more general architectures like the FRNN: The restrictions imposed limits the number of free parameters and can therefore make it easier to train the network to find an adequate solution. Restrictions can also simplify the training algorithm and thereby speed up the training process (see subsection 2.2.3).

An alternative implementation: feedback of neuron activation values
In some SRN architectures that have been used the neuron activation value (the linear weighted sum of inputs) is fed back through the recurrent connections, instead of the neuron output value [Hertz e.a., 1991]. So in these structures the feedback neurons that establish the state of the network are linear, while the function of input and state to the output is nonlinear. This way the context units accumulate a linear weighted moving average of past input values.

As an example, the updating rule for the i neurons in the context layer of the SRN of figure 2.10c can be [Hertz e.a., 1991]:

$$S_i(n+1) = \alpha \cdot S_i(n) + u_i(n) \tag{2.29}$$

where at time n, $S_i(n)$ is the activation sum of neuron i in the context layer and $u_i(n)$ is the external input applied. The nonlinear activation function $f(.)$ is not used in this rule, but it is used in the connection to the next layer:

$$y_{i,context}(n) = f(S_i(n)) \tag{2.30}$$

This approach is explained in much more detail in [Mozer, 1995] where the corresponding algorithm is called *focussed backpropagation*. See also subsection 4.1.2 where a possible reason for using linear feedback is discussed.

2.3 Architectures based on the input-output system model

There exists another general model for nonlinear dynamic systems, that does not make use of the concept of state to model the system's memory of past events. This kind of model is called an input-output recurrent model since it can be expressed entirely as a functional relationship between present and past system inputs and outputs.

A single-input single-output (SISO) recurrent neural network based on the input-output recurrent model is shown in figure 2.11. The general input-output recurrent model structure is referred to as the class of *nonlinear autoregressive with exogenous input* (NARX) models.

Sometimes the same model is named NARMA or NARMAX, referring to the inputs as 'moving average' [Haykin, 1998].

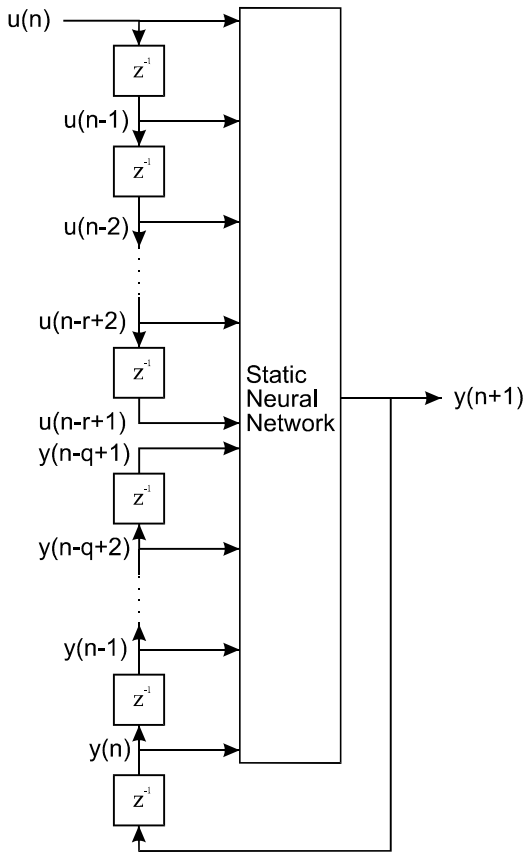


Figure 2.11; Recurrent neural network based on the NARX model

Besides the current input u(n), a number of r-1 past input values are kept in a delay line memory. Instead of having an internal state that summarizes information of the past, a number of q previous outputs are remembered, or regressed. This way a summary of *all* information of the past can be kept, because for each time n the output y(n) depends on (so it can 'remember') at least one y(m) for which: n-q+1 ≤ m < n. This implies that information from the initial time n_0 up to the present time n is used to form the present output y(n). The network dynamics are described by the following equation:

$$y(n+1) = F(y(n),...,y(n-q+1),u(n),...,u(n-r+1)) \tag{2.31}$$

The SISO NARX model described in this subsection has scalar signals u(n) and y(n) but these could of course be replaced by vectors **u**(n) and **y**(n) and a network realizing a vector function **F**(.) to obtain a general M-input L-output NARX neural network.

For the static neural network inside the NARX network structure, all types of static neural networks can be used. These specific choices and the choice of the variables q and r form the special cases of this network class.

Properties

The network that computes the output y(n+1) is a standard MLP. An advantage of the NARX model is therefore that it can be learned using standard algorithms for MLP. There is no problem of state-outputs for which target values are unknown, as is the case in the state-space model. This also allows the use of the *teacher forcing* technique during training on all outputs (this technique will be discussed in subsection 3.6.2). The parameters q and r have to be chosen in advance, similar to choosing the number of state neurons in a state-space network.

A disadvantage of the NARX model is that the number of inputs to the static network can become large. This number of inputs is the size of the *extended input vector* $\mathbf{z}(n)$ that holds all network input:

$$\mathbf{z}(n) = [(y(n) \ \ldots \ y(n-q+1) \ u(n) \ \ldots \ u(n-r+1)]^T \tag{2.32}$$

whose size is Size($\mathbf{z}$) = (q*L + r*M) with L,M the number of external outputs and the number of external inputs, respectively. Size($\mathbf{z}$) increases with q, r in steps of at least *min*(L,M). For state-space networks, on the other hand, Size($\mathbf{z}$) = (2*L + 2*S) with S the state size. The size can be incremented in minimum step sizes of 2 (by incrementing S by 1) which is generally smaller than *min*(L,M).

A SISO state-space neural network has an equivalent NARX network

It is shown in [Haykin, 1998] that any recurrent network described by state-space equations can be simulated by an equivalent input-output network model, if

1) the recurrent network is locally *observable* (subsection 4.1.3)
2) the network is SISO (single input, single output)

This important relation was not thoroughly investigated because the equations of a state-space network in [Haykin, 1998] at first sight do not seem the most general case:

$$\begin{aligned} \mathbf{x}(n+1) &= \mathbf{F}(\mathbf{x}(n), \mathbf{u}(n)) \\ \mathbf{y}(n) &= \mathbf{C} \cdot \mathbf{x}(n) \end{aligned} \tag{2.33}$$

The multiplication by $\mathbf{C}$ is equivalent to the operation performed in a linear neural network layer. In deriving this and other properties, Haykin uses even the assumption that $\mathbf{F}(.)$ is computed by a single layer of neurons. This seems like a highly restricted model and not a general state-space model. It could however be that a sufficiently large enough class of state-space models is captured by the above equation, but time did not permit an investigation of this. In [Haykin, 1998] the model was just used without further comment.

More properties of NARX networks and their relationship with state-space model networks will be given in chapter 4.

Applications

The equivalence property given above shows that an equivalent NARX network could be used instead of a state-space network in the SISO case. The NARX network model can be used in any of the applications listed in this chapter, but no application to speech recognition has been found in literature. The model was studied in the context of finite state machine identification [Haykin, 1998].

2.4 Modular recurrent neural network architectures

Some neural network architectures can be best described as modular architectures. The definition of a modular architecture as used in this report is: an architecture that consists of several static neural networks, that are interconnected in a specific way. There is, in most cases, not a clear boundary between a modular network and a single neural network because the total modular architecture can be looked at as a single neural network, and some existing single networks can also be described as modular networks. It is rather a convenient way of describing complex neural networks.

In this section the category of modular *recurrent* neural network architectures is looked at, modular architectures that all have one or more internal feedback connections.

The modular recurrent neural network architectures were not introduced in previous sections, because they do not fit very well in the state-space system description or the NARX description. Formally they can be described as a state-space system (like any dynamic system) but this could result in a very complicated and unnecessarily large state-space system description.

In this section three classes of modular recurrent neural network architectures are presented:

- Recurrent Multi-layer Perceptron (RMLP); subsection 2.4.1
- Block Feedback Networks (BFN) framework; subsection 2.4.2
- General modular neural network framework; subsection 2.4.3

The first model (RMLP) is a rather specific one and it is included as an example of a modular architecture. Undoubtedly, many more such architectures are proposed in literature and they cannot all be listed here. Another example is the Pipelined Recurrent Neural Network found in [Haykin, 1998] and applied to speech prediction in [Baltersee e.a., 1998].

The second model is far more general and was meant to provide a structured way to describe a large class of recurrent neural networks and their training algorithms. The third model attempts to do the same and it turns out that this model is the most general one: it incorporates the first two as special cases, so in this section the attention will be mainly focussed on the third model, the general modular network framework.

2.4.1 Recurrent Multilayer Perceptrons (RMLP)

An extension of the regular MLP has been proposed by Puskorias e.a. (see [Haykin, 1998]) which adds self-feedback connections for each layer of the standard MLP. The resulting Recurrent Multilayer Perceptron (RMLP) structure with N layers is shown in figure 2.12.

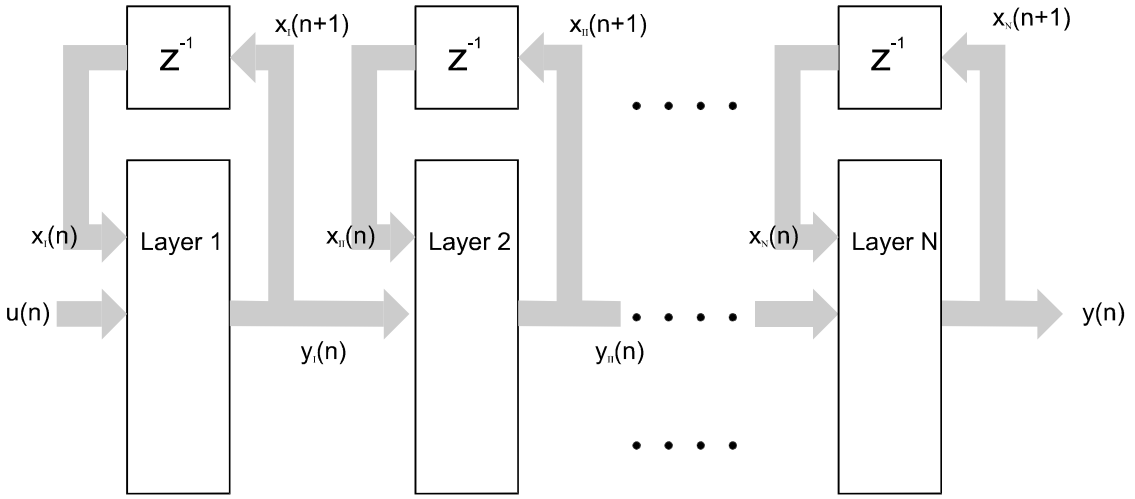


Figure 2.12; Recurrent multi-layer perceptron (RMLP) architecture with N layers

Each layer is a standard MLP layer. The layer outputs are fed forward to the inputs of the next layer and the delayed layer outputs are fed back into the layer itself. So the layer output of time n-1 for a certain layer acts as the state variable at time n for this layer. The global state of the network consists of all layer states $\mathbf{x}_i(n)$ together.

Effectively, this type of network can have both a very large total state vector and a relatively small number of parameters because the neurons in the network are not fully interconnected. There are no recurrent interconnections across layers. All recurrent connections are local (1-layer-to-itself).

Properties

Comparing one layer of the RMLP structure with the FRNN architecture (see figure 2.3) it is clear that one RMLP layer is in fact a type 1 FRNN network with N neurons and L=N outputs for each layer. The RMLP can thus be written as a cascade of multiple FRNN networks.

Applications

In [Haykin, 1998] the RMLP is applied to a chaotic time-series prediction problem. In general, the RMLP can be used in any of the applications listed in this chapter. An application to speech recognition was not found in literature.

2.4.2 Block Feedback Networks (BFN)

A framework for describing recurrent neural networks that has been introduced by [Santini e.a., 1995b] provides a systematic way for modular design of networks of high complexity. This class of networks is called Block Feedback Neural Networks (BFN), referring to the *blocks* that can be connected to each other using a number of elementary connections. The term *feedback* is used because one of the elementary connections is a feedback connection, thus enabling the construction of recurrent neural networks. The network that results from the construction can in turn be considered a 'block' and it can be used again as a basic building block for further construction of progressively more complex networks.

So a recursive, modular way of designing networks is provided. The training algorithm for any BFN is based on backpropagation training for MLP and the backpropagation through time (BPTT) algorithm for recurrent networks. It is recursively constructed along with the network structure. So the BFN framework introduces a class of (infinitely many) recurrent networks, which can be trained using a correspondingly constructed backpropagation algorithm.

The BFN framework

The framework which describes the BFN and its training algorithms is called the *BFN framework* in this report. This framework is not fully introduced in this report, because it turns out the modular network framework in subsection 2.4.3 is an easier and more general way of describing a large class of recurrent networks, which includes all BFN networks. For a full description of the BFN framework see [Santini e.a., 1995a,b,c].

A very short presentation of the BFN framework will be given now. The basic unit is a single neural network layer, an example of which is shown in figure 2.13a. The corresponding matrix notation is shown in figure 2.13b. **A** is a 6-by-3 matrix and **F**(.) is a 6-by-6 diagonal mapping containing the neuron activation functions (of the form of equation 2.12). One such layer is defined as a *single layer block* N.

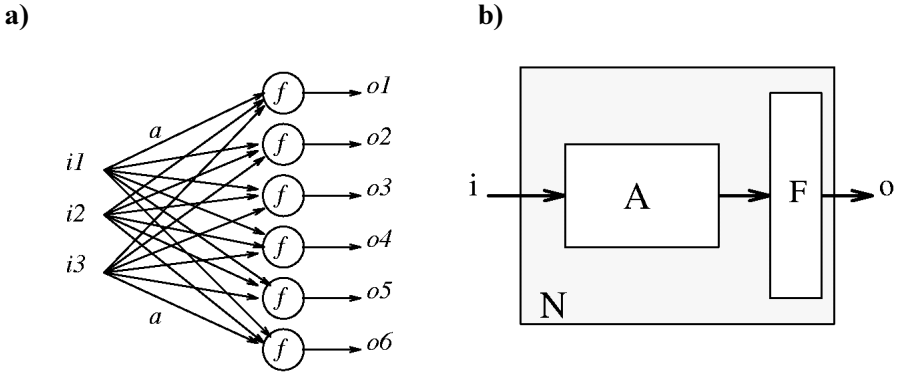


Figure 2.13; a) example of a single network layer ; b) the layer in BFN block notation as a block N

This block computes the function:

$$\mathbf{o}(n) = \mathbf{F}(\mathbf{A} \cdot \mathbf{i}(n)) \tag{2.34}$$

Single layer blocks can be connected together using the four elementary connections shown in figure 2.14. They are called the cascade, the sum, the split and the feedback connection. Each of these connections consists of one or two embedded BFN blocks (these are called N1 and N2 in the figure) and one connection layer (which has the structure of a single-layer block). This connection layer consists of the weight matrices **A** and **B**, and the vector function **F**(.). Each of the four elementary connections itself is defined as a block and can therefore be used as the embedded block of yet another elementary connection.

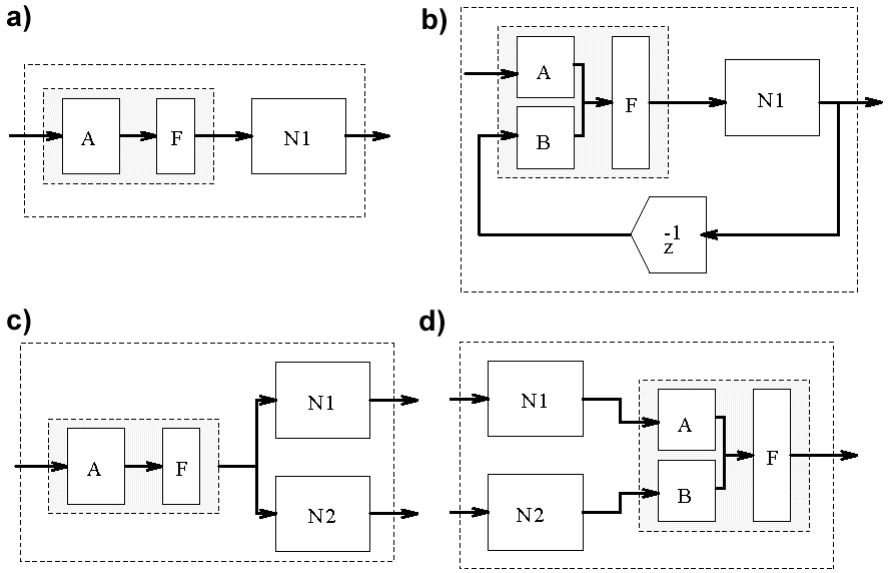


Figure 2.14; The four elementary BFN connections: the cascade (a), feedback (b), split (c) and sum (d) connection.

Applications
BFN networks can be used in the same applications as other recurrent neural networks. A specific application of the algorithmic construction of a training algorithm, is the automatic construction of a training algorithm for given network structures. Many alternatives of a network architecture could be tested and compared automatically this way.

One of the possible advantages of the BFN framework suggested by [Santini e.a., 1995b,c] is that sub-classes of BFN networks may be compared, instead of comparing just individual network architectures. The systematic way of construction suggests that theories might be developed that apply to subclasses or the entire class of BFN networks, instead of applying only to one architecture.

Application of the BFN framework to describe recurrent networks
Some of the recurrent neural network architectures described in this report can be constructed (along with a suitable BPTT training algorithm) using the BFN framework. Examples of such BFN constructed networks can be found in [Santini, 1995a,b,c].

Discussion
The BFN framework is not used further because it is superseded by the modular network framework. This will be shown in subsection 2.4.3. BFN has however a very distinct feature: networks are defined in a recurrent manner, whereas the modular network framework uses only one level of hierarchy (a flat hierarchy). The implications of this feature were not further investigated, but it is conceivable that it allows theorems to be proven that would not be provable using a flat hierarchy description.

2.4.3 General modular network framework

In [Bengio, 1996] a general modular network framework is introduced. This framework is similar to the BFN framework: it can be used to describe many different *modular networks* which are built of neural network modules linked together. Each module is a static feedforward neural network and the links between the modules can incorporate delay elements. In this subsection the modular network framework is introduced. The purpose of this framework is:

- to describe many types of recurrent neural networks as modular networks
- to derive a training algorithm for a network that is described as a modular network

The use of the framework for developing such a training algorithm, that can do *joint training* of the individual network modules, will be discussed in section 3.8. Here only the structural

description method of the framework is introduced and used to describe neural network architectures.

First the definitions will be given that are needed to describe modular networks. These are followed by an example to clarify the definitions.

Definitions

Define a modular network structure having N_{mod} modules that are interconnected by *links*. Each network module $i = 1 \ldots N_{mod}$ has L_i outputs and is *static* or *memoryless*. In other words, this requirement means that each module has no internal delay elements. So, it cannot be a recurrent network, for example.

Links are numbered $l \in [1,\ldots,N_{links}]$ where N_{links} is the total number of links in the modular network.

Define a link l as the connection of the output of module a to the input of module b. Because the link comes from module a this module is called the *predecessor* module and is denoted by $p(l)$ with $p(l) = a$. The link goes to module b which is called the successor module, denoted by $s(l)$ with $s(l) = b$. Each link can have its own delay value, denoted $d(l)$, implemented by delay registers.

So a link l goes from a predecessor module $p(l)$ to a successor module $s(l)$ with a delay $d(l)$. Each module i computes a function $\mathbf{F}_i$:

$$\mathbf{y}_i(n) = \mathbf{F}_i(\boldsymbol{\theta}_i, \mathbf{Z}_i(n), \mathbf{u}_i(n)) \tag{2.35}$$

where $\mathbf{y_i} = (y_{i1}, y_{i2}, \ldots, y_{i;Li})$ is the module-output vector of module i consisting of L_i scalar outputs and $\mathbf{F_i}(.)$ is the vector function computing the network output, depending on the network parameter vector (of module i) θ_i containing scalar parameters $\theta_i = \{\theta_{ij}\}$. The external input vector is $\mathbf{u_i}(n)$. The vector $\mathbf{Z_i}(n)$ holds all module-output vectors $\mathbf{y}_{p(l)}(n - d(l))$ of modules $p(l)$ at times n-$d(l)$ for all links l that go to module i.

To define $\mathbf{Z_i}(n)$ more clearly, first the set S_i is defined that holds all links that have module i as a successor (all links, that are going to module i):

$$S_i = \{ l \,|\, s(l) = i \} \tag{2.36}$$

The number of elements of this set S_i is the number of links going to module i:

$$N_L(i) = \sum_{l=1}^{N_{links}} \delta^{kro}(s(l), i) \tag{2.37}$$

Each element j of S_i will be denoted $S_i\{j\}$. Then, with the following definition:

$$z_{ij}(n) = \mathbf{y}_{p(S_i(j))}(n - d(S_i\{j\})) \tag{2.38}$$

we can write down the vector $\mathbf{Z_i}(n)$ that contains $N_l(i)$ vectors $\mathbf{z_{ij}}(n)$:

$$\mathbf{Z}_i(n) = \begin{bmatrix} \mathbf{z}_{i1}(n) \\ \mathbf{z}_{i1}(n) \\ \ldots \\ \mathbf{z}_{i;N_L(i)}(n) \end{bmatrix} = \begin{bmatrix} \mathbf{y}_{p(S_i\{1\})}(n - d(S_i\{1\})) \\ \mathbf{y}_{p(S_i\{2\})}(n - d(S_i\{2\})) \\ \ldots \\ \mathbf{y}_{p(S_i\{N_{links}(i)\})}(n - d(S_i\{N_L(i)\})) \end{bmatrix} \tag{2.39}$$

So $\mathbf{Z_i}(n)$ contains all delayed outputs coming from other modules and possibly module i itself, going to the input of module i. The delay $d(l)$ can be different for each link and can be zero.

There is a restriction, which is needed to obtain a computable algorithm without algebraic loops: for every closed path of links there must be at least one delay $d(l) > 0$ in one of the links l that make up the path. A special case of this requirement is that a recurrent link that goes from a module to the module itself must be zero.

An example of a modular network

In figure 2.15, an example modular network, having two modules and three links, is given. The variables just introduced can all be found in the figure. Note that delay d(1) is allowed to be zero (in fact, arbitrary values of d(1) would give rise to the same network dynamics in this example) whereas delays d(2) and d(3) must be greater than zero because otherwise algebraic loops will be present in the network equations.

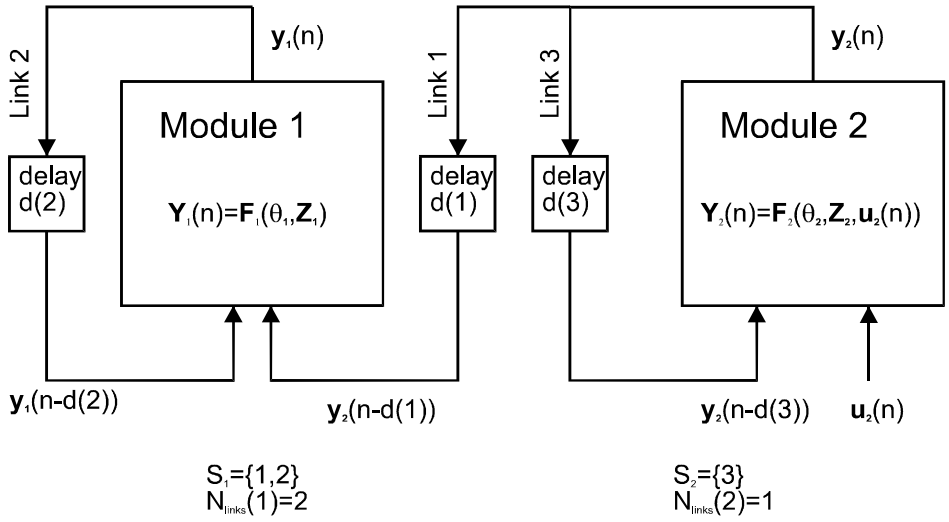


Figure 2.15; Example of a modular network. All variables as defined in the modular network description are shown.

The vectors $\mathbf{Z}_i(n)$ for the example are given by:

$$\begin{aligned} \mathbf{Z}_1 &= \begin{bmatrix} \mathbf{z}_{11}(n) \\ \mathbf{z}_{12}(n) \end{bmatrix} = \begin{bmatrix} \mathbf{y}_2(n-d(1)) \\ \mathbf{y}_1(n-d(2)) \end{bmatrix} \\ \mathbf{Z}_2 &= \begin{bmatrix} \mathbf{z}_{21}(n) \end{bmatrix} = \begin{bmatrix} \mathbf{y}_2(n-d(3)) \end{bmatrix} \end{aligned} \tag{2.40}$$

Application of the framework

Many recurrent network architectures can be described with the modular network framework. Two special cases of this framework are the FRNN and the state-space neural network architecture. Both will be shown to be specific instances of modular networks in the remainder of this subsection.

Other architectures that can be described by the modular network framework are the NARX network architecture (using 1 module, r external inputs and q recurrent links with increasing delay value) and the RMLP. This is not shown further in this report.

Special case 1: the FRNN

The Fully Recurrent Neural Network is a special case of the modular network framework. The FRNN can be specified as a single-module network having one link that connects the module output vector to its input. The module is a single layer static feedforward network. The FRNN is shown as a modular network in figure 2.16.

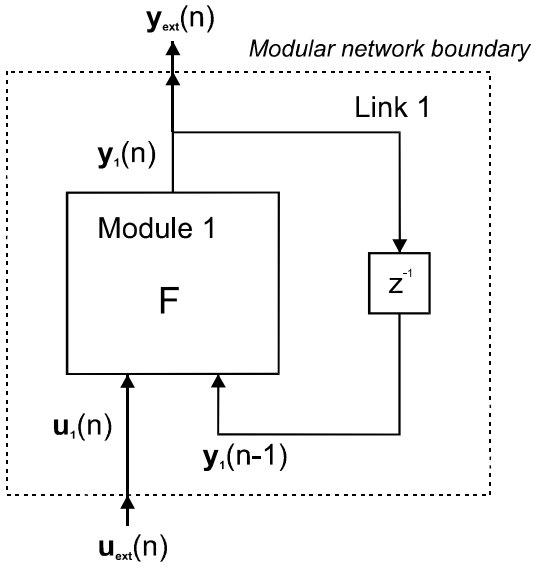


Figure 2.16; FRNN presented as a single module network with one delayed recurrent link

The static feedforward network module 1 has N neurons (so N outputs), M+N inputs, and as parameters the weights $\theta_1=\mathbf{w_1}=\{w_{ij}\}$. Here the weights w_{ij} and N, M are the same as defined for the FRNN in subsection 2.2.2 and equation 2.6.

The module performs the computation:

$$\mathbf{y}_1(n) = \mathbf{F}_1(\boldsymbol{\theta}_1, \mathbf{Y}_1(n), \mathbf{u}_1(n)) \tag{2.41}$$

Because the output $\mathbf{y_1}(n)$ is taken as the external output, we have $\mathbf{y_1}(n) = \mathbf{y_{ext}}(n)$.

It may seem that all N neurons are always connected as external outputs, which does not conform to the general FRNN architecture, where a number of neurons L ≤N are taken as external outputs. However, this problem is solved by the specific definitions of the errors on each neuron $e_l(n)$ as will be given later in section 3.1 in equations 3.1 / 3.2. Effectively, by only defining target values for the L first neurons and not for the other neurons the situation of L < N external outputs is created. For more information see section 3.1

Now the entire network function can be described by:

$$\mathbf{y}_{\text{ext}}(n) = \mathbf{F}(\mathbf{W}, \mathbf{y}_1(n-1), \mathbf{u}_{\text{ext}}(n)) \tag{2.42}$$

where **F**(.) is chosen identical to the corresponding function in the FRNN (equation 2.13) and the parameters are the weight matrix **W** as defined in equation 2.10c for the FRNN. It is now identical to the FRNN.

General single-module recurrent network and multilayer FRNN

Instead of allowing only one-layer modules in the single-module structure (figure 2.16), every type of static neural network could possibly be allowed. In allowing this, an extension of the FRNN to a general single-module recurrent network is obtained.

When module 1 is restricted to a multilayer perceptron, the *multilayer FRNN architecture* is obtained.

Only if targets are defined for all outputs, this structure can also be seen as a special case of the input-output network description (subsection 2.3) with the delay buffer configuration q= 1 and r = 1.

Special case 2: the general state-space network architecture

The general state-space neural network architecture, introduced in subsection 2.2.1, can be described using the modular network framework. The network architecture in a two-module description is shown in figure 2.17.

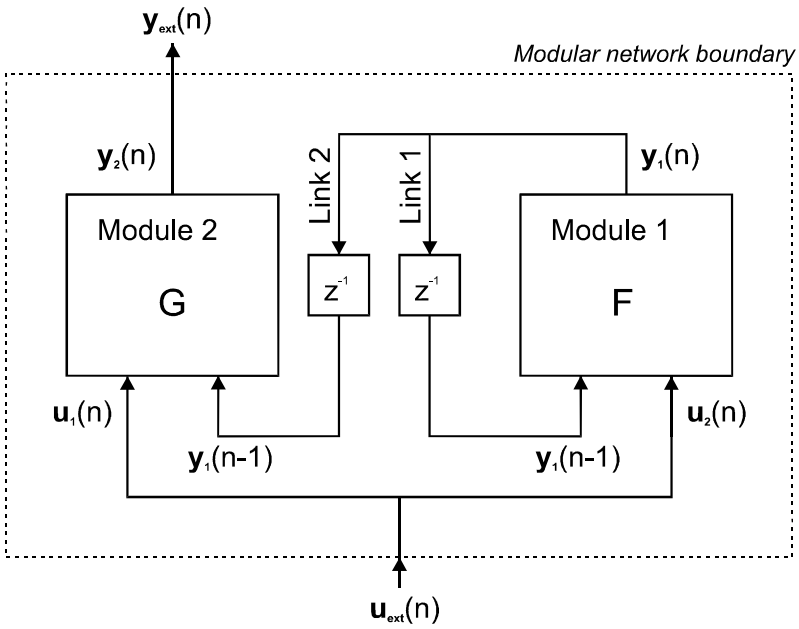


Figure 2.17; General state-space network is a two-module network with two links

In this case the functions **F** and **G** are each computed by a single neural network. It is also possible to let several separated static network modules compute **F** or **G**. This gives different modular structures than the one given here. But in that case it is also possible to *merge* the separate static networks back into one network so that the two-module description is again obtained.

Equations of the modular state-space network

The equations for the two-module state-space network are now given. There is one self-recurrent link (link 1) and a link (link 2) from module 1 to module 2. The external input $\mathbf{u}_{ext}(n)$ is fed to both modules. The modules compute the following functions:

$$\begin{aligned} \mathbf{y}_1(n) &= \mathbf{F}_1(\boldsymbol{\theta}_1, \mathbf{Y}_1(n), \mathbf{u}_1(n)) \\ \mathbf{y}_2(n) &= \mathbf{F}_2(\boldsymbol{\theta}_2, \mathbf{Y}_2(n), \mathbf{u}_2(n)) \end{aligned} \tag{2.43}$$

Using the configuration as shown in figure 2.17 the network equations can be rewritten to:

$$\begin{aligned} \mathbf{y}_1(n) &= \mathbf{F}(\mathbf{W}_1, \mathbf{y}_1(n-1), \mathbf{u}_{ext}(n)) \\ \mathbf{y}_{ext}(n) &= \mathbf{G}(\mathbf{W}_2, \mathbf{y}_1(n-1), \mathbf{u}_{ext}(n)) \end{aligned} \tag{2.44}$$

where $\mathbf{W}_1$ and $\mathbf{W}_2$ are the weights of module 1 and 2 respectively. When the state $\mathbf{x}(n)$ is defined $\mathbf{x}(n) = \mathbf{y}_1(n\text{-}1)$ the general state-space description of equation 2.1 is obtained:

$$\begin{aligned} \mathbf{x}(n+1) &= \mathbf{F}(\mathbf{W}_1, \mathbf{x}(n), \mathbf{u}_{ext}(n)) \\ \mathbf{y}_{ext}(n) &= \mathbf{G}(\mathbf{W}_2, \mathbf{x}(n), \mathbf{u}_{ext}(n)) \end{aligned} \tag{2.45}$$

where the functions **F**(.) and **G**(.) are realized by static neural networks.

Properties and discussion

The following goals of the modular network framework were mentioned:

- to describe many types of recurrent neural networks as modular networks
- to derive a training algorithm for a network that is described as a modular network

These goals are also realized by the BFN framework. Still, the modular network framework supersedes the BFN framework that was introduced in the previous subsection. The reasons for this are:

- it can describe more network architectures than BFN. An example case, the two-layer RMLP is shown in Appendix A.2. It can not be described by the BFN framework.
- static network modules can be any type of static neural network. BFN on the other hand prescribes standard single- or multilayer perceptrons.

Applications
The modular network framework is used in this report for derivation of training algorithms. It can also be used for automatic derivation and automatic calculation of training algorithms, given a modular network structure. Training algorithms for modular networks are the topic of section 3.8.

In [Bengio, 1996] the application of *joint training* of neural networks is suggested. This means for example that a collection of neural networks are first trained separately until no performance increase can be achieved anymore. Then the collection of networks is jointly trained (using an algorithm derived with the modular network framework) so that an additional performance increase can be realized. In this example it is assumed that targets are known for all modules.

2.5 Conclusions

Many types of recurrent neural networks can be constructed. A number of these architectures have been listed in this chapter, but naturally this can not include all types encountered in literature.

It was found that the general modular network framework is a convenient way to describe a large class of architectures. It is able to describe all architectures mentioned in this chapter. Based on the results in this chapter a hierarchical ordering of network architectures is possible. This ordering is shown in figure 2.1. The most general architectures are listed left. Architectures that can be conveniently described as a special case of another architecture, are listed at the right of it.

An architecture that was not found in literature but is introduced in this report, is the single-module recurrent network (or multilayer FRNN). This structure contains the FRNN as a special case.

We can conclude that implementing the general modular network framework seems the best option. This gives the option to simulate all recurrent network architectures so that they can be experimentally tested.

However, at the beginning of this project it was estimated that implementing the general framework would be too much work. Therefore, the state-space network was chosen as the second-best architecture. The state-space network architecture is promising because:

- it was found that it can in principle simulate any state-space system. The FRNN can not do this. (This will be shown in subsections 4.1.4 and 4.1.5).
- it contains architectures like the FRNN, some Simple Recurrent Networks (SRN), partially recurrent network and the multi-layer RNN as special cases. In literature it was found that these architectures are often used for speech recognition tasks.

The conclusion that can be drawn is that the state-space architecture should be further investigated. But before final conclusions are drawn, neural network training algorithms should be investigated.

Neural network structures are only useful if they can be trained. In the next chapter training algorithms for recurrent networks will be investigated. The focus will be on training algorithms for the most promising structures that were listed in this chapter.

CHAPTER 3 TRAINING ALGORITHMS FOR RECURRENT NEURAL NETWORKS

A training algorithm is a procedure that adapts the free parameters of a neural network in response to the behavior of the network embedded in its environment. The goal of the adaptations is to improve the neural network performance for the given task. Most training algorithms for neural networks adapt the network parameters in such a way that a certain error measure (also called cost function) is minimized. Alternatively, the *negative* error measure or a *performance measure* can be maximized.

Aim

In this chapter different training algorithms and strategies are investigated. Based on the conclusions of the previous chapter, the focus will be on learning algorithms for state-space networks and for modular networks in general. As done for network structures, a categorization of training algorithms will be made. Finally, comparing learning algorithms should lead to a decision on what algorithms are to be implemented.

Chapter outline

In this chapter on training algorithms, first

- error measures are presented (section 3.1) from which training algorithms may be derived ;
- training algorithms are then classified into a number of categories, depending on their properties (section 3.2) ;
- in sections 3.3 and 3.4 two training algorithms, Backpropagation Through Time (BPTT) and Real-time Recurrent Learning (RTRL), for training FRNN are derived. Restricting the discussion to algorithms for FRNN first allows a detailed derivation ;
- in section 3.5 the derivation of both training algorithms will be extended from FRNN to modular networks ;
- on the two basic training algorithms numerous variations are possible, some will be given in section 3.6 ;
- references to other training approaches will be made in section 3.7 ;
- the chapter conclusions, with a discussion and comparison of training algorithms, are in section 3.8.

The sections on training algorithms in this chapter will be augmented by small-scale experiments that demonstrate the workings of the algorithms. These experiments were performed using the ModNet toolbox for Matlab. (The toolbox is introduced in subsection 3.8.2.)

3.1 Error measures

Error measures can be postulated because they seem intuitively right for the task, because they lead to good performance of the neural network for the task, or because they lead to a convenient training algorithm. An example is the Sum Squared Error measure, which always has been a default choice of error measure in neural network research. It will be introduced first in subsection 3.1.1.

Error measures can also be obtained by accepting some general ‘induction principle’ and deriving from this an error measure that conforms to this principle. In subsection 3.1.2 some induction principles are briefly mentioned. In subsection 3.1.3 an additional error measure is mentioned that can be derived from an induction principle appropriate for classification.

3.1.1 The Sum Squared Error measure

A standard error measure which is used for training neural networks is the Sum Squared Error measure (SSE):

$$E(n) = \frac{1}{2}\sum_{i=1}^{L} e_i^2(n) \tag{3.1}$$

The index i runs over all L network outputs and the error signal $e_i(n)$ is defined for all i as the difference between the target value $d_i(n)$ and the actual external network output $y_i(n)$:

$$e_i(n) = \begin{cases} d_i(n) - y_i(n) & \text{for i} \in \text{D} \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

where the set D holds all outputs $i \in$ D that have a target defined. If no target is defined the error is zero. Note that it is possible to make D time-dependent, D(n), which means targets can be defined on specific time instants only. This situation is called *partial supervision* and will be introduced later in subsection 4.3.3.

Using this error measure, big differences between output and targets are 'punished' more than small differences. At the same time, the squaring of the error will ensure the error is always a positive value and that a derivative of the error measure with respect to the network weights w_j exists for every $e_i(n)$.

The coefficient ½ in the error measure is used to simplify the subsequent derivation of a training algorithm.

Error measures for dynamic neural networks

Error measures for a static neural network use only the errors at the outputs of the network at the current time n. These are called *instantaneous error measures* and are denoted E(n). In dynamic neural networks, the network performance is usually considered over a relevant time interval instead of a single time n. The network error measure over a time interval $[n_0, n_1]$ can be defined as the sum of the instantaneous error measures at each of the times in the interval:

$$E_{TOTAL}(n_0, n_1) = \sum_{n=n_0}^{n_1} E(n) \tag{3.3}$$

Using equation 3.3 an instantaneous error measure for a static network can be converted into a total error measure for use in dynamic neural networks. Note that in this report $E(n_0,n_1)$ is used as a short notation of $E_{TOTAL}(n_0,n_1)$.

Sum Squared Error measure for dynamic neural networks

Combing equations 3.1 and 3.3, the SSE for dynamic neural networks for an interval $[n_0,n_1]$ is given by

$$E_{TOTAL}(n_0, n_1) = \frac{1}{2}\sum_{n=n_0}^{n_1}\sum_{i=1}^{L} e_i^2(n) \tag{3.4}$$

3.1.2 Induction principles

The maximum likelihood model for neural networks

A probabilistic principle can be used as an induction principle. This probabilistic principle and resulting error measures are given in [Rumelhart e.a., 1996] and [Bishop, 1995] for the case of a static multilayer perceptron. This probabilistic principle is actually the *maximum likelihood* principle, according to [Bishop, 1995]. It states that the neural network has to be found, which is the most likely explanation of the observed data sequence (the 'data' means the examples, including both input patterns and their targets).

Under the assumption of independence of observed examples and a certain probabilistic distribution of the targets $d_k(n)$, and interpreting the network outputs $y_k(n)$ as the mean of $d_k(n)$ (in other words a prediction of the target $d_k(n)$), some error measures can be derived. Even the Sum Squared Error measure that was postulated in the previous subsection, may be derived using the maximum likelihood model. See [Rumelhart e.a., 1996] or [Bishop, 1995] for the derivation of error measures using maximum likelihood.

Other induction principles

Discussions on other induction principles can be found in [Smolensky e.a., 1996] and [Bishop, 1995]. Examples of information-theoretic approaches are [Rissanen, 1996] the *Minimum Description Length* (MDL) principle and the *Maximum Mutual Information* (MMI) or *Infomax* principle. Principles rooted in statistics are described in [Bishop, 1995], e.g. the *Bayesian inference* approach. Another principle that can be found in Support Vector Machine literature is *Structural Risk Minimalization* (SRM) [Haykin, 1998]. This principle seeks to minimize the upper bound on the network generalization error, as opposed to minimizing the error on a training set which is the common goal in neural network training.

3.1.3 The cross-entropy error measure

Another well known error measure is the *cross-entropy error* measure that can be used when the neural network outputs are interpreted as probabilities. They are used for speech recognition in [Robinson e.a., 1991]. The probability interpretation of outputs can be very useful for reasons explained in [Bishop, 1995].

Because of time limitations the lengthy derivation of the cross-entropy error measure, which gives insight into the motivation behind choosing an error measure for classification tasks, can not be given here. For more details on this and other error measures [Bishop, 1995] is advised (sections 6.7-6.9). In [Rumelhart e.a., 1996] and [Smolensky e.a., 1996] this topic is also treated.

3.2 Categorization of training algorithms

Different error measures lead to different algorithms to minimize these error measures. Training algorithms can be classified into categories depending on certain distinguishing properties of those algorithms. Four main categories of algorithms that can be distinguished are:

1. *gradient-based algorithms.* The gradient of the equation of the error measure with respect to all network weights is calculated and the result is used to perform *gradient descent*. This means that the error measure is minimized in steps, by adapting the weight parameters proportional to the negative gradient vector.
2. *second-order gradient-based algorithms.* In second-order methods, not only the first derivatives of the error measure are used, but also the second-order derivatives of the error measure.
3. *stochastic algorithms*. Stochastic weight updates are made but the stochastic process is directed in such a way, that on average the error measure becomes smaller over time. A gradient of the error measure is not needed so an expression for the gradient doesn't have to exist.
4. *hybrid algorithms*. Gradient-based algorithms are sometimes combined with stochastic elements, which may capture the advantages of both approaches.

This list is probably not complete and other classifications are possible. In this report, only gradient-based algorithms will be discussed in detail (as they are the default choice for training neural networks). The four categories are further discussed below.

1. Gradient-based algorithms

Gradient-based training algorithms compute an exact or approximate gradient of the error measure with respect to the free parameters of the network (the weight vector $\mathbf{w}$). The gradient

of E with respect to a variable **w** is denoted $\nabla_w E$. Different classes of gradient algorithms can be distinguished for recurrent networks, based on the way of gradient computation and the way of operation (continuous or epochwise updating of weights) [Williams e.a., 1995]. Table 3.1 lists these classes. These are a number of very common classes of training algorithms which include all of the algorithms in this report, but it is very likely that not all existing algorithms can be classified in this way.

a	b	c	d
Type of algorithm	**Operation**	**Gradient computation performed by the algorithm**	**Computed at which time(s)**
exact gradient computation algorithm	continuous	$\nabla_w E_{TOTAL}(n_0,n)$	at each time n
"	epochwise	$\nabla_w E_{TOTAL}(n_0,n_1)$	at end-time n_1
real-time gradient computation	continuous	$\nabla_w E(n)$	at each time n
"	epochwise	$\nabla_w E(n)$	at each time $n = n_0 \ldots n_1$; but weight update only at $n=n_1$
approximate gradient computation	continuous	approximation to $\nabla_w E_{TOTAL}(n_0,n)$	at each time n
"	epochwise	approximation to $\nabla_w E_{TOTAL}(n_0,n_1)$	at end-time n_1

Table 3.1: Classes of gradient based algorithms for recurrent networks

The interval $[n_0,n]$ is the interval of time of operation in the case of continuous operation and $[n_0,n_1]$ is the interval of one epoch in case of epochwise operation. Columns a and b describe the algorithm type. Column c shows what gradient computation this algorithm performs and column d shows when this calculation is done.

Using the gradient to adapt the weight vector as in equation 3.5 below, gradient descent is performed such that a local or global minimum of the error measure E is found. The parameter η is the *learning rate*:

$$\Delta \mathbf{w} = -\eta \nabla_{\mathbf{w}} E$$
$$\mathbf{w}_{\mathbf{New}} = \mathbf{w}_{\mathbf{Old}} + \Delta \mathbf{w} \qquad (3.5\ a,b)$$

Particular cases of these equations, using for E the Sum Squared Error measure and selecting as a neural network the FRNN, will be given in sections 3.3 and 3.4.

Approximate gradient algorithms are related to hybrid algorithms

The approximate gradient algorithms perform weight updates that deviate from the true gradient. These deviations are stochastic in nature, for example depending on the order of presentation of training examples to the network. Therefore the approximate algorithms can also be considered to be hybrid algorithms. The approximate gradient algorithms can therefore be called *stochastic gradient algorithms*.

Gradient-based training algorithms for the FRNN and RNN

Depending on the type of neural network, choice of error measure E, computation of the gradient and the way of operation, different training algorithms result. The following algorithms for the Fully Recurrent Neural Network and the subsets of the FRNN are investigated in this chapter:

1. Backpropagation Through Time (BPTT)
2. Real-time Recurrent Learning (RTRL)

These are the two best known algorithms for training recurrent networks. The basic form of these algorithms will be derived in this chapter.

Gradient-based training algorithms for modular networks
The method of gradient-based joint training of modular networks (the modular network framework was introduced in subsection 2.4.3) is discussed in section 3.5. The two main approaches BPPT en RTRL also apply to modular networks.

2. Second-order gradient-based algorithms
Normal gradient-based algorithms actually perform a first-order (linear) approximation of the error measure function in the local neighborhood of the parameter vector **w**. Second-order algorithms make a closer approximation of the error measure function by performing a second-order (quadratic) local approximation of the cost function around **w**. The second-order derivatives are needed to make this closer approximation [Haykin, 1998].

In many algorithms this computation is not performed exactly, but an estimation of the second order derivatives is made to speed up the process or to overcome potential problems associated with an exact computation [Haykin, 1998].

Well-known second-order algorithms are *Newton's method*, the *quasi-Newton* method and the *conjugate-gradient* method. These can all be found in [Haykin, 1998]. The *Levenberg-Marquardt* algorithm is another powerful second-order training algorithm [Bishop, 1995]. Second order methods are known to be computationally much more efficient than standard gradient descent for many neural network learning tasks. Note that a single iteration of a second-order method requires more calculations, but usually far less iterations are required for the total learning task, thereby providing an overall saving of computations. A comparison done in [Peelen, 1999] shows the dramatic improvement using second order methods for a classification task.

3. Stochastic algorithms
Stochastic algorithms make use of random weight updates. These updates are not completely random, but the parameters of a stochastic process (which is used to generate the random updates) are dictated by the training algorithm.

An error measure is needed to test the quality of the parameters (weights) found and to select a parameter set with the highest quality (the lowest error measure) out of the several parameter combinations tested. The goal of the algorithm is to minimize the error measure. The advantage of stochastic algorithms is compared to gradient algorithms that the gradient of the error measure is not computed, so an expression for this gradient does not even have to exist.

Stochastic algorithms have been used to train neural networks or to optimize the structure of networks. Examples of error measures where an expression for the gradient is certainly not known, are mostly found outside the domain of neural networks. For example:

- The error measure is a value that is (subjectively) assigned by a human. In this case the error measure itself is stochastic.
- The error measure results from a complex simulation.
- The error measure results from operations performed in 'the real world'. It can depend, for example, on the behavior of a robot interacting with its environment.

An example of a stochastic algorithm is a genetic algorithm. In (for example) [McDonnel, 1994] genetic algorithms are used for training recurrent neural networks.

4. Hybrid algorithms
In hybrid algorithms a gradient algorithm is combined with elements of stochastic algorithms. The approximate gradient algorithms were already mentioned as being examples of a hybrid algorithm.

Another example of a hybrid algorithm would be an algorithm that alternates between a gradient algorithm and a stochastic algorithm, switching to the stochastic mode when the

gradient algorithm gets stuck in a local minimum and switching back to a gradient mode again after some time.

3.3 Backpropagation Through Time (BPTT) algorithm for FRNN

The Backpropagation Through Time (BPTT) algorithm is an algorithm that performs an exact computation of the gradient of the error measure for use in the weight adaptation. In this section the BPTT algorithm will be derived for a (type 1) FRNN using a Sum Squared Error measure.

Methods of derivation of the algorithm
There are two different methods to develop the BPTT algorithm. Both are shown in this report:

- derivation by unfolding the network in time, which also gives intuitive insight in how the algorithm works.
- a formal derivation of the algorithm using the *ordered derivative* notation.

The 'unfolding in time' approach will be explained first in subsection 3.3.1 because the unfolding procedure gives a better understanding of how the algorithm works. This approach is only briefly introduced in this subsection and the rest of the derivation can be found in appendix B.1. The second approach will be given in subsection 3.3.3, after the ordered derivative notation has been introduced (in subsection 3.3.2).

3.3.1 Derivation of the BPTT algorithm by unfolding the network in time

As an error measure to be minimized, the Sum Squared Error for a sequence (subsection 3.1.1) denoted $E(n_0,n)$ is used:

$$E(n_0,n) = \sum_{m=n_0}^{n} \sum_{i=1}^{L} e_i^2(m) \tag{3.6}$$

The error measure is calculated for one example sequence that runs in the time interval $[n_0,n]$. Time n can be the current time (in this case, future values of $e_i(m)$, $m > n$, are still unknown) or the end time of the example sequence (in this case, $n=n_1$).

Using equation 3.5a the error measure may be minimized by gradient descent:

$$\Delta w_{ij} = -\eta \frac{\partial E(n_0,n)}{\partial w_{ij}} \tag{3.7}$$

Before this can be further developed, the network equations are first given and the network has to be unfolded in time. The network dynamics of equations 3.8 are used to describe the FRNN. The matrix notation is not used because the algorithm will be developed in scalar notation. The equations are repeated here for convenience.

$$\begin{aligned} z_i(n) &= \begin{cases} y_i(n-1) & 1 \le i \le N \\ u_{i-N}(n) & N+1 \le i \le N+M \end{cases} \\ s_i(n) &= \sum_{j=1}^{N+M} w_{ij} \cdot z_j(n) \\ y_i(n) &= f(s_i(n)) \end{aligned} \tag{3.8 a,b,c}$$

Note that initial values $y_i(n_0-1)$ have to be known to be able to compute the initial extended input vector $\mathbf{z}(n_0) = \{z_i(n_0)\}$. The example FRNN of subsection 2.2.2 is repeated in figure 3.1 for convenience.

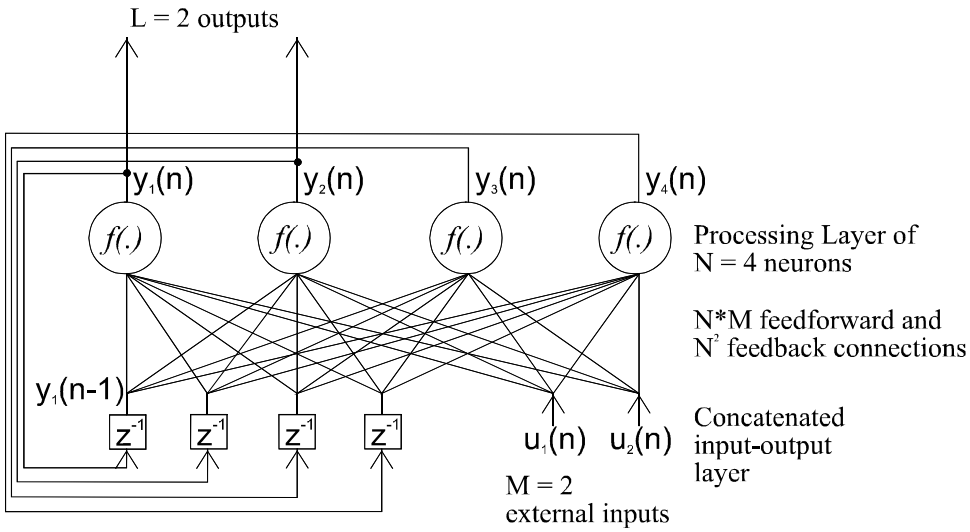


Figure 3.1; Example FRNN with four neurons and two external inputs

In this approach, the recurrent network is *unfolded in time* into an equivalent static feedforward network. For a certain initial time n_0 and current time n, the Fully Recurrent (single-layer) network N_R with N neurons is unfolded into a feedforward network N_R* which has a layer of N neurons for every time step in the interval $[n_0,n]$. So each neuron in N_R has a copy in each layer of N_R* and each weight w_{ij} in N_R that connects unit j to unit i through a delay, has a copy $w_{ij}(m)$ in N_R* that connects unit j in layer (m-1) to unit i in the next layer m. Each input weight w_{ij} (with $1 \le i \le N$ and $N+1 \le j \le N+M$) that connects input $u_{(j-N)}(.)$ to neuron i has a copy $w_{ij}(m)$ that connects input $u_{(j-N)}(m)$ to neuron i in layer m.

As an example of unfolding, the unfolded equivalent of the example FRNN of figure 3.1 is presented for four time-steps (in the interval $[n_0,n]$ =[0,3]) in figure 3.2. The labels for the sets of duplicated weights $w_{ij}(m)$ are shown on the left.

The unfolding of the network solves the problem of unknown target values for the hidden neurons N3 and N4 because at time n=3, the only known error terms $e_1(3)$ and $e_2(3)$ can be backpropagated through neurons N1 and N2 in layer 3 to neurons N3 and N4 in layer 2 with the backpropagation algorithm described in this section.

The network dynamics of the *unfolded* FRNN are now given by the equations:

$$
\begin{aligned}
z_i(n) &= \begin{cases} y_i(n-1) & 1 \le i \le N \\ u_{i-N}(n) & N+1 \le i \le N+M \end{cases} \\
s_i(n) &= \sum_{j=1}^{N+M} w_{ij}(n) \cdot z_j(n) \qquad \text{(3.9 a,b,c)} \\
y_i(n) &= f(s_i(n))
\end{aligned}
$$

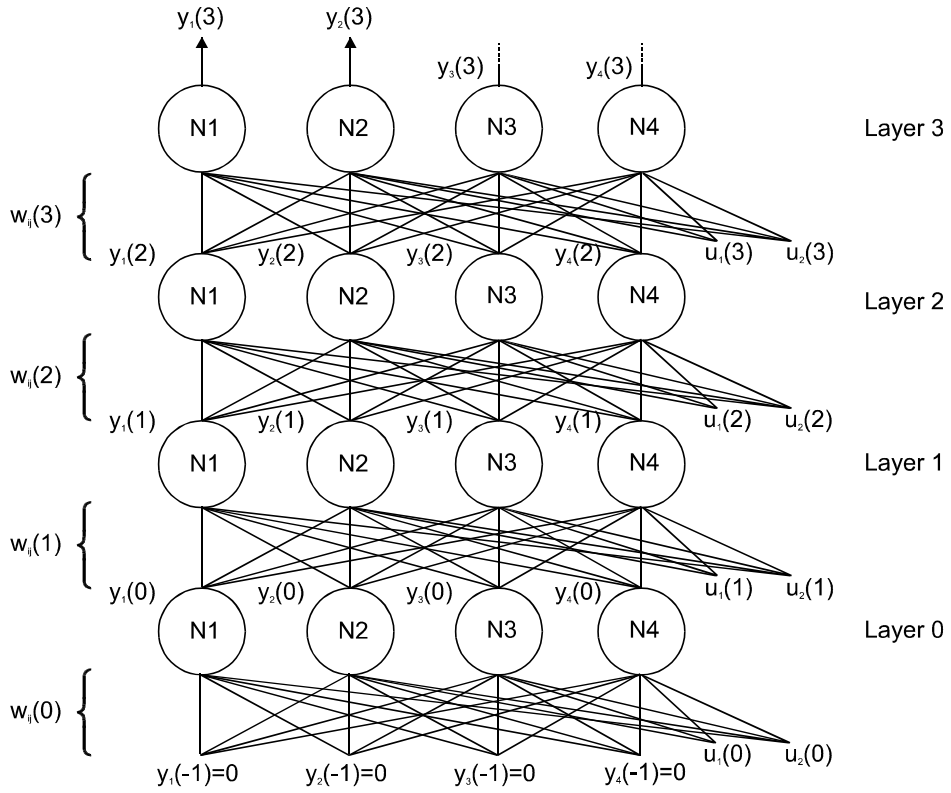


Figure 3.2; Unfolded type 1 example FRNN for the time interval [0,3].

The only difference with the standard FRNN dynamics (equations 3.8) is that the weights w_{ij} are now denoted as multiple copies $w_{ij}(m)$. Because they are just copies, $w_{ij}(m) = w_{ij}$ holds for all times m. But this approach does allow different partial derivatives with respect to the error measure, so the following situation is very well possible:

$$\frac{\partial E(n_0,n)}{\partial w_{ij}(m)} \neq \frac{\partial E(n_0,n)}{\partial w_{ij}(l)} \quad \text{for m} \neq \text{l} \tag{3.10}$$

With the unfolded network and duplicated weights, the gradient computation of equation 3.7 can be decomposed as follows [Williams e.a., 1995] into multiple partial derivatives.

$$\frac{\partial E(n_0,n)}{\partial w_{ij}} = \sum_{m=n_0}^{n} \frac{\partial E(n_0,n)}{\partial w_{ij}(m)} \frac{\partial w_{ij}(m)}{\partial w_{ij}} = \sum_{m=n_0}^{n} \frac{\partial E(n_0,n)}{\partial w_{ij}(m)} \tag{3.11}$$

where $\partial w_{ij}(m)/\partial w_{ij} = 1$ because a change in the weights w_{ij} of the recurrent network N_R implies an equal change in each copy $w_{ij}(m)$ in network N_R^*. The last equation implies that the gradient of the recurrent network can be obtained by calculating the partial derivatives of the error measure with respect to all weights $w_{ij}(m)$, $m \in [n_0,n]$ in the unfolded equivalent static network N_R^* and add them up.

The algorithm is obtained by further developing the expressions for the partial derivatives $\partial E(n_0,n)/\partial w_{ij}(m)$. This is shown in appendix B.1. For a similar derivation of BPTT for the type 2 FRNN, by unfolding the network in time, see [Williams e.a., 1995].

3.3.2 Introduction to the ordered derivative

To proceed with the formal derivation of the BPTT algorithm (that does not use unfolding in time) the ordered derivative notation has to be introduced. The ordered derivative was first introduced by Werbos (referred to in [Santini, e.a., 1995b]). A formal definition of the ordered derivative is given in [Santini e.a., 1995b]. In this report the full definition is not shown because it would occupy several pages. The chain rules for ordered derivatives that result from the

ordered derivative definition will however be given and used. The chain rules are followed by an example to clarify the concept.

Definitions

Let q_j with j=1…n denote the variables of a system and $L(q_1,\ldots q_n)$ be a scalar function of these variables. The variables are *ordered* such that each variable q_j can depend on (can be a function of) the variables $q_1\ldots q_{j-1}$ but does not depend on $q_{j+1}\ldots q_n$. So we have for each j:

$$q_j = f_j(q_1, q_2, ..., q_{j-1}) \tag{3.12}$$

Define a set D_j that holds all indices k for which q_k depends on q_j:

$$D_j = \left\{ k \mid q_k = f_k(..., q_j) \right\} \tag{3.13}$$

with the definition of ordered variables just given, this simply becomes:

$$D_j = \{j+1, j+2, ..., n\} \tag{3.14}$$

These definitions will be used in the remainder of this subsection.

As stated before the exact definition is not included in this report, so a less formal 'definition' will be given here. The ordered derivative is actually defined as a partial derivative in the ordered-variable system described above. It is denoted by:

$$\frac{\partial^+ u}{\partial v}$$

In the ordered-variable system, the ordered derivative has the general form $\partial^+X/\partial q_j$ (where X can be any q_k or the scalar function L). This term describes all influence (both direct and indirect, through the system equations) of q_j upon X. There is an important aspect to the notation: when ordered derivative notation is used, the default partial derivative notation $\partial X/\partial q_j$ is assigned *a different meaning*. This notation is then used to denote all *direct* influence on q_j upon X, not including the indirect influence through other system variables. Therefore it will be called *direct derivative* in this section for convenience.

Another way to state the difference between derivatives, is the following:

- the ordered derivative $\partial^+X/\partial q_j$ is computed by first substituting all equations that depend on q_j into the expression X and then computing the partial derivative.
- the direct derivative $\partial X/\partial q_j$ is obtained by calculating the partial derivative of X *without* substituting any equations. In effect, the system equations are decoupled before the calculation.

Chain rules for ordered derivatives

For computing ordered derivatives two chain rules can be derived [Santini e.a., 1995b]. The first chain rule for ordered derivatives is given by:

$$\frac{\partial^+ L}{\partial q_j} = \frac{\partial L}{\partial q_j} + \sum_{k \in D_j} \frac{\partial^+ L}{\partial q_k} \frac{\partial q_k}{\partial q_j} \tag{3.15}$$

This states that L directly depends on q_j (the left) term and indirectly, through all variables $q_k=f_k(\ldots,q_j,\ldots)$ that are a direct function of q_j.

The second chain rule is given by:

$$\frac{\partial^+ L}{\partial q_j} = \frac{\partial L}{\partial q_j} + \sum_{k \in D_j} \frac{\partial L}{\partial q_k} \frac{\partial^+ q_k}{\partial q_j} \tag{3.16}$$

Again the overall influence of q_j upon L is divided into direct influence (left) and indirect influence through all variables q_k that are both directly and indirectly influenced by q.

The first chain rule will be used in the derivation of the BPTT algorithm. The second chain rule will be used in the derivation of the RTRL algorithm for modular networks.

Example of the use of the ordered derivative
In the following example [Bengio, 1996] of the ordered derivative, the following system is defined:

$$\begin{aligned} y &= f(x,z) = x^2 - 2z \\ x &= g(u,z) = \log u + z^2 \end{aligned} \tag{3.17}$$

In this system z has a direct influence on y but also an indirect influence through x.

1. The standard partial derivative can be calculated (using the chain rule) as
$$\frac{\partial y}{\partial z} = -2 + 2x\frac{\partial x}{\partial z} = -2 + (2x)(2z) = -2 + 4xz$$
2. the ordered derivative $\partial^+ y/\partial z$ is calculated (using its first chain rule) as
$$\frac{\partial^+ y}{\partial z} = \frac{\partial y}{\partial z} + \frac{\partial^+ y}{\partial x}\frac{\partial x}{\partial z} = -2 + (2x)(2z) = -2 + 4xz \tag{3.18}$$

To perform the computation the first chain rule (equation 3.15) was used two times. The first use of the rule was to decompose $\partial^+ y/\partial z$ as shown and the second (implicit) usage was in calculating

$$\frac{\partial^+ y}{\partial x} = \frac{\partial y}{\partial x} + 0 = \frac{\partial y}{\partial x} = 2x \tag{3.19}$$

which holds because y does not indirectly depend on x, only directly.

Discussion
The result of both methods in the example above is the same. The difference lies not in the result, but in the notation. Using the standard partial derivative there is no notation for the term "-2" in the result and it is written down directly. Using the ordered derivative method, this term is denoted $\partial y/\partial z$ and represents only the direct influence of z on y (without any substitutions).

On the other hand, the term $\partial^+ y/\partial z$ denotes all influence (both direct and indirect through substitutions) of z on y. The ordered derivative approach allows easier derivation of algorithms in some cases, for example in deriving the BPTT algorithm. This was also noted in [Haykin, 1998].

Criticism on the ordered derivative
The ordered derivative is sometimes introduced in literature as a 'new' mathematical operation not equal to the standard partial derivative. The formal definition of the ordered derivative (given in [Santini e.a., 1995b]) also does not use the standard partial derivative but a separate definition. This viewpoint is taken in [Santini e.a., 1995abc] and [Bengio, 1996].

Opposed to this stands the viewpoint that the ordered derivative in fact equals the standard partial derivative so it is nothing new, and the only new thing is the definition of the direct derivative. This viewpoint is taken in [Baldi, 1995] and in this report.

Therefore the sole use of the ordered derivative can be criticized, but it is a fact that this notation allows a very clear derivation of algorithms and seems to be approved upon by several authors [Haykin, 1998] [Bengio, 1996] [Williams e.a., 1995]. The final results (i.e. the algorithm) of ordered derivative derivations are of course equal to those using the standard partial derivative.

Alternatives to the ordered derivative notation
In literature, the ordered derivative notation is not always explicitly used. There are at least three alternatives to this notation. The first is to always use standard partial derivative notation and stating in the text if a particular derivative denotes 'all influence' like the normal partial derivative or just 'direct influence'. This was done in [Baldi, 1995]. The second alternative would be to simply introduce a very clear different notation (e.g. $\partial^* u/\partial v$) for the direct derivatives, but surprisingly this was not found in literature. Yet another approach is followed in [Williams e.a., 1995] and in Appendix B.1 in this report where a new set of variables is defined

that create a decoupled version of the original system equations. These new variables are used (together with the standard partial derivative) in denoting direct derivatives.

3.3.3 Derivation of the BPTT algorithm using the ordered derivative

The BPTT algorithm is derived now using the ordered derivative notation. Again the gradient descent procedure (equation 3.7) is used. The gradient is computed using the ordered derivative to account for all influence of the weights on the error measure:

$$\Delta w_{ij} = -\eta \frac{\partial^+ E(n_0, n)}{\partial w_{ij}} \tag{3.20}$$

The weights w_{ij} are *not* denoted with a time index (like $w_{ij}(m)$) because 'multiple copies' of weights are not made. The values of w_{ij} are always the 'latest' values so they could be denoted as $w_{ij}(n)$, but this is easily confused with the 'multiple copies' notation so they will be just denoted by w_{ij}.

From here, the first chain rule for ordered derivatives (equation 3.15) will be used a few times during the derivation. Because the parameters w_{ij} of neuron i can only affect the error measure indirectly through the sums $s_i(.)$ and thus the outputs $y_i(.)$ of neuron i, the first chain rule can be applied twice as follows:

$$\begin{aligned}
\frac{\partial^+ E(n_0, n)}{\partial w_{ij}} &= \frac{\partial E(n_0, n)}{\partial w_{ij}} + \sum_{m=n_0}^{n} \frac{\partial^+ E(n_0, n)}{\partial s_i(m)} \frac{\partial s_i(m)}{\partial w_{ij}} = \\
&= 0 + \sum_{m=n_0}^{n} \left(\frac{\partial E(n_0, n)}{\partial s_i(m)} + \frac{\partial^+ E(n_0, n)}{\partial y_i(m)} \frac{\partial y_i(m)}{\partial s_i(m)} \right) \frac{\partial s_i(m)}{\partial w_{ij}} = \\
&= \sum_{m=n_0}^{n} \frac{\partial^+ E(n_0, n)}{\partial y_i(m)} \frac{\partial y_i(m)}{\partial s_i(m)} \frac{\partial s_i(m)}{\partial w_{ij}}
\end{aligned} \tag{3.21}$$

The direct derivative terms $\partial E / \partial w_{ij}$ and $\partial E / \partial s_i(m)$ are zero because the variables w_{ij} and s_i are not directly present in the error measure E.

For the first term in the above equation the first chain rule can be applied twice again. Because the outputs $y_i(m)$ influence E(.) not only directly but also indirectly, through the 'next' sums $s_l(m+1)$ and outputs $y_l(m+1)$, applying the first chain rule twice yields:

$$\begin{aligned}
\frac{\partial^+ E(n_0, n)}{\partial y_i(m)} &= \frac{\partial E(n_0, n)}{\partial y_i(m)} + \sum_{l=1}^{N} \frac{\partial^+ E(n_0, n)}{\partial s_l(m+1)} \frac{\partial s_l(m+1)}{\partial y_i(m)} = \\
&= \frac{\partial E(n_0, n)}{\partial y_i(m)} + \sum_{l=1}^{N} \left(\frac{\partial E(n_0, n)}{\partial s_l(m+1)} + \frac{\partial^+ E(n_0, n)}{\partial y_l(m+1)} \frac{\partial y_l(m+1)}{\partial s_l(m+1)} \right) \frac{\partial s_l(m+1)}{\partial y_i(m)} = \\
&= \frac{\partial E(n_0, n)}{\partial y_i(m)} + \sum_{l=1}^{N} \left(0 + \frac{\partial^+ E(n_0, n)}{\partial y_l(m+1)} \frac{\partial y_l(m+1)}{\partial s_l(m+1)} \right) \frac{\partial s_l(m+1)}{\partial y_i(m)} = \\
&= \frac{\partial E(n_0, n)}{\partial y_i(m)} + \sum_{l=1}^{N} \frac{\partial^+ E(n_0, n)}{\partial y_l(m+1)} \frac{\partial y_l(m+1)}{\partial s_l(m+1)} \frac{\partial s_l(m+1)}{\partial y_i(m)}
\end{aligned} \tag{3.22}$$

This expression exists for $n_0 \le m \le n$ only if an additional requirement is made that will be given soon in equation 3.25. Using the definition of the error measure 3.6, selecting a neuron transfer function f for all neurons and using equations 3.9a,b to obtain the weight w_{li}:

$$\frac{\partial E(n_0,n)}{\partial y_i(m)} = \begin{cases} -e_i(m) & \text{for } n_0 \le m \le n \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial y_l(m)}{\partial s_l(m)} = f'(s_l(m)) \qquad \text{for all m} \qquad \text{(3.23 a,b,c)}$$

$$\frac{\partial s_l(m+1)}{\partial y_i(m)} = w_{li} \qquad \text{for all m}$$

we now obtain for $n_0 \le m \le (n\text{-}1)$:

$$\frac{\partial^+ E(n_0,n)}{\partial y_i(m)} = -e_i(m) + \sum_{l=1}^{N} \frac{\partial^+ E(n_0,n)}{\partial y_l(m+1)} f'(s_l(m)) \cdot w_{li} \qquad (3.24)$$

Because the outputs at times $m \ge n+1$ do not influence the cost function $E(n_0,n)$, the following condition arises:

$$\frac{\partial^+ E(n_0,n)}{\partial y_i(n+1)} = 0 \qquad (3.25)$$

Finally, using this condition a recursive equation for $\partial^+E(n_0,n)/\partial y_i(m)$ is obtained:

$$\frac{\partial^+ E(n_0,n)}{\partial y_i(m)} = \begin{cases} -e_i(m) & \text{for m = n} \\ -e_i(m) + \sum_{l=1}^{N} \frac{\partial^+ E(n_0,n)}{\partial y_l(m+1)} f'(s_l(m)) \cdot w_{li} & \text{for m < n} \end{cases} \qquad (3.26)$$

The recursively obtained terms are used in equation 3.21, which gives the required weight update.

The following definitions are now made for a compact notation of the algorithm:

$$\varepsilon_i(m) = -\frac{\partial^+ E(n_0,n)}{\partial y_i(m)}$$

$$\delta_i(m) = -\frac{\partial^+ E(n_0,n)}{\partial s_i(m)} \qquad (3.27)$$

This gives for equation 3.26:

$$\varepsilon_i(m) = \begin{cases} e_i(m) & \text{m = n} \\ e_i(m) + \sum_{l=1}^{N} w_{li}\delta_l(m+1) & \text{m < n} \end{cases} \qquad (3.28)$$

$$\delta_i(m) = \varepsilon_i(m) \cdot f'(s_i(m))$$

Using equations 3.20, 3.21 and substituting the following partial derivatives:

$$\frac{\partial s_i(m)}{\partial w_{ij}} = z_j(m) \qquad (3.29)$$

the weight adaptation at time n can now be written down:

$$\begin{aligned} \Delta w_{ij} &= -\eta \frac{\partial^+ E(n_0,n)}{\partial w_{ij}} = -\eta \sum_{m=n_0}^{n} \frac{\partial^+ E(n_0,n)}{\partial y_i(m)} \frac{\partial y_i(m)}{\partial s_i(m)} \frac{\partial s_i(m)}{\partial w_{ij}} = \\ &= -\eta \sum_{m=n_0}^{n} -\delta_i(m) z_j(m) = \eta \sum_{m=n_0}^{n} \delta_i(m) z_j(m) \end{aligned} \qquad (3.30)$$

The values of $\delta_i(.)$ required for the adaptation can be recursively computed using equations 3.28 in a so-called single *backward pass*. The following *forward pass* is the use of equation 3.30 to calculate the weight updates. The BPTT algorithm is named after the backpropagation of gradients according to equation 3.28. To obtain $z_j(n_0)$, initial conditions of the delay elements are set zero, $y_j(n_0\text{-}1) = 0$.

The same equations were obtained in appendix B.1 by unfolding the recurrent network in time and proceeding the derivation with the unfolded network.

The BPPT algorithm

Now the BPTT algorithm can be summarized as follows:

1. set initial time $n = n_0$
2. calculate the N neuron output values for time n using the network equations 3.8
3. recursively calculate $\varepsilon_i(m)$ then $\delta_i(m)$ with equations 3.23 backwards in time starting with $m = n$ back to $m = n_0$.
4. calculate for all i,j the weight updates according to equation 3.24
5. update the weights w_{ij}
6. increase time n to n+1 and go back to step 2

3.3.4 Variations on the BPTT algorithm

From the general BPTT algorithm description above, some variations can be developed. The epochwise BPTT, real-time BPTT and truncated BPTT are mentioned below. Some more BPTT variations can be found in [Williams e.a., 1995]. The reason for using these variations is often to reduce computations as will become clear in the next subsection.

Epochwise BPTT

If we take the BPTT algorithm as derived above and perform the weight update only at the end of each training example sequence (which is called an *epoch*), the epochwise BPTT algorithm is obtained. An epoch E has a length L_E and is presented to the network during the time interval $[n_{E0},n_{E1}]$ where $L_E = (n_{E1} - n_{E0}+1)$. All epoch intervals are subsets of the network operation interval, so $[n_{E0},n_{E1}] \subset [n_0,n]$ for all E. After weights are updated at the end of an epoch, the new weights are used as starting weights for the next epoch.

The epochwise BPTT differs from the standard BPTT algorithm in the following aspects:

- The error measure used is the error over one epoch $E(n_{E0},n_{E1})$
- The BPTT algorithm to calculate weight updates is only run once per epoch (at the end time n_{E1}) and therefore the weights are only updated once per epoch. For all other times $n_{E0}\ldots(n_{E1}\text{-}1)$ only the network equations 3.9 are calculated, not the training algorithm (i.e. keep the network running but do not adapt any weights).

Truncated BPTT

An example of an approximate gradient training algorithm is *truncated BPTT*. To simplify the required computation, the backward-through-time propagation of information (as defined in equation 3.23) is truncated after a certain number of h time steps has been taken back.

The algorithm is denoted BPTT(h) where h is the number of prior time steps of which information is used. The equations 3.23, 3.24 are used together with the truncation that can be expressed as

$$\varepsilon_i(m) = 0 \qquad \text{for } m < n - h \tag{3.31}$$

It will be shown in the next subsection that this truncation saves computation time.

Real-time BPTT

A real-time BPTT algorithm can be obtained by using $E(n)$, the error at time n only, as the error measure and deriving the algorithm in the same way as was done above for the epochwise BPTT. See [Williams e.a., 1995] for more information on this algorithm.

3.3.5 Computational complexity of BPTT algorithms

The computational complexity can be measured in the required number of mathematical operations (time complexity) and the amount of numbers that have to be stored (space complexity). The computational complexity of the BPTT algorithm will be assessed in this subsection by examining the equations. A similar comparison is done [Williams e.a., 1995].

The operations required for calculating the network equations 3.9 (not applying a training algorithm) are not counted, as these requirements are low and independent of the training algorithm. Also the few operations for calculating network error $e_i(n)$ from target and output, and the number of transfer function evaluations are not counted.

The resources required by the BPTT algorithm are listed in table 3.2. For each equation (column a) of the algorithm the number of multiplications and additions are counted (columns b and c). The sum of both at the bottom row is the total number of operations. The amount of required storage (space) is listed in column d. The dominating terms (for number of neurons N >> 1) are printed in bold. The value N_w is defined as the total number of weights N_w = (N+M)*N. The number of time steps the network is running is $h = n\text{-}n_0+1$ for the interval $[n_0,n]$.

a	b	c	d
Equation	**Multiplications**	**Additions**	**Storage**
3.23b	2*N*h [1)]	-	N*h values $s_i(m)$
3.23a	**N*N*h**	**(N-1+1)*N*h**	N*h values $e_i(m)$
3.24	**(h-1+1)*N_w**	**(h-2)*N_w**	N*h values $\delta_i(m)$
Totals:	[2N*N + 2*N_w +2*N] * h - 2 N_w		3*N*h
Order of magnitude:	O[(N^2 + N(N+M))h]		O[N*h]

[1)] assumed a sigmoid transfer function: $f(x) = (1+e^{-x})^{-1} \Rightarrow f'(x) = f(x)\cdot[1-f(x)]$

Table 3.2: Computational resources required for the BPTT algorithm

In general, the total number of operations is of the order O((N^2 + N(N+M)) *h*) and the storage is of the order O(N**h*). The number of operations and storage required grow linear with the time of operation *h* of the network.

When targets are only defined for a fraction F_T of all time steps the requirements scale down with F_T because the algorithm is not calculated when no target is defined (zero error requires no adaptation of weights). Then the average number of operations per time step is of order O([N^2 + N(N+M)] * *h* * F_T).

Because the ever-growing requirements over time (as *h* becomes large) the unmodified BPTT algorithm is almost never used in practice. Instead truncated or epochwise BPTT is used.

The results can also be applied to the different BPTT algorithms listed in subsection 3.3.4, epochwise BPTT and truncated BPTT. This will be done now.

Epochwise BPTT

Taking the results from above, with *h* the epoch length $h = (n_{E1}\text{-}n_{E0}+1)$ the requirements for epochwise BPTT are obtained. Because the weight adaptation is only computed once at the end of an epoch (i.e. once per *h* time steps), the order of magnitude of the mean number of operations per time step is

$$O\left(N^2 \, h/h + N(N+M) \, h / h \right) = O\left(N^2 + N(N+M) \right)$$

If the epoch length is variable, taking for *h* the mean epoch length will result in the same expression for the mean number of operations.

Truncated BPTT

The results from standard BPTT can be used to calculate the requirements for truncated BPTT(*h*). The value of *h* now represents the number of prior time steps that are used and *not* anymore the entire period of operation of the network. All information before time (*n*-*h*) is not considered anymore so no calculations have to be done with this information and no storage is required for it. Effectively, a *semi-initial time* n_0' can be defined that is always set at *h* time steps back ($n_0' = n - h$) of current time *n*. As a consequence $E(n_0',n)$ is minimized. The requirements are then the same as for standard BPTT but using the semi-initial time n_0' instead of the initial time n_0.

So the number of operations for BPTT(*h*) is of the order $O([N^2 + N(N+M)] * h * F_T)$ per time step and the required storage is O(N*h). The parameter *h* is however fixed in BPTT(*h*) and can be chosen small for a fast training algorithm or larger to minimize E over a larger interval $[n_0',n]$.

3.3.6 Example

In this experiment, an electrical circuit is identified by a linear partially recurrent neural network (PRN). The training data consists of samples of two input signals applied to the circuit and the resulting output signal.

The circuit to be identified

An RC circuit with two inputs (voltage sources e_1 and e_2), one output (the voltage $y = x_1-x_2$), two capacitors and three resistors is used. The schematic is shown in figure 2.8.

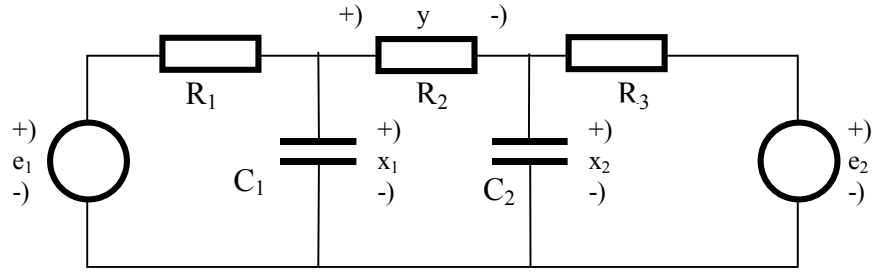


Figure 3.3; The RC circuit to be identified

The sampled circuit can be described as a discrete-time linear state-space system with a two-dimensional state vector:

$$\mathbf{x}(n+1) = \begin{bmatrix} 1-\frac{\Delta}{R_1C_1}-\frac{\Delta}{R_2C_1} & \frac{\Delta}{R_2C_1} \\ \frac{\Delta}{R_2C_2} & 1-\frac{\Delta}{R_3C_2}-\frac{\Delta}{R_2C_2} \end{bmatrix} \cdot \begin{bmatrix} x_1(n) \\ x_2(n) \end{bmatrix} + \begin{bmatrix} \frac{\Delta}{R_1C_1} & 0 \\ 0 & \frac{\Delta}{R_3C_2} \end{bmatrix} \cdot \begin{bmatrix} e_1(n) \\ e_2(n) \end{bmatrix} \quad (3.32)$$

$$y(n) = \begin{bmatrix} 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} x_1(n) \\ x_2(n) \end{bmatrix} + \begin{bmatrix} 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} e_1(n) \\ e_2(n) \end{bmatrix}$$

The component values are chosen as follows: R_1=3100(Ω), R_2=2320(Ω), R_3=1300(Ω), C_1=8(μF), C_2=2(μF). The sampling interval Δ = 1(ms).

The input to the circuit is a pattern of step signals. In figure 2.2 the two inputs $e_1(t)$, $e_2(t)$ and the resulting circuit output y(t) are plotted (140 samples).

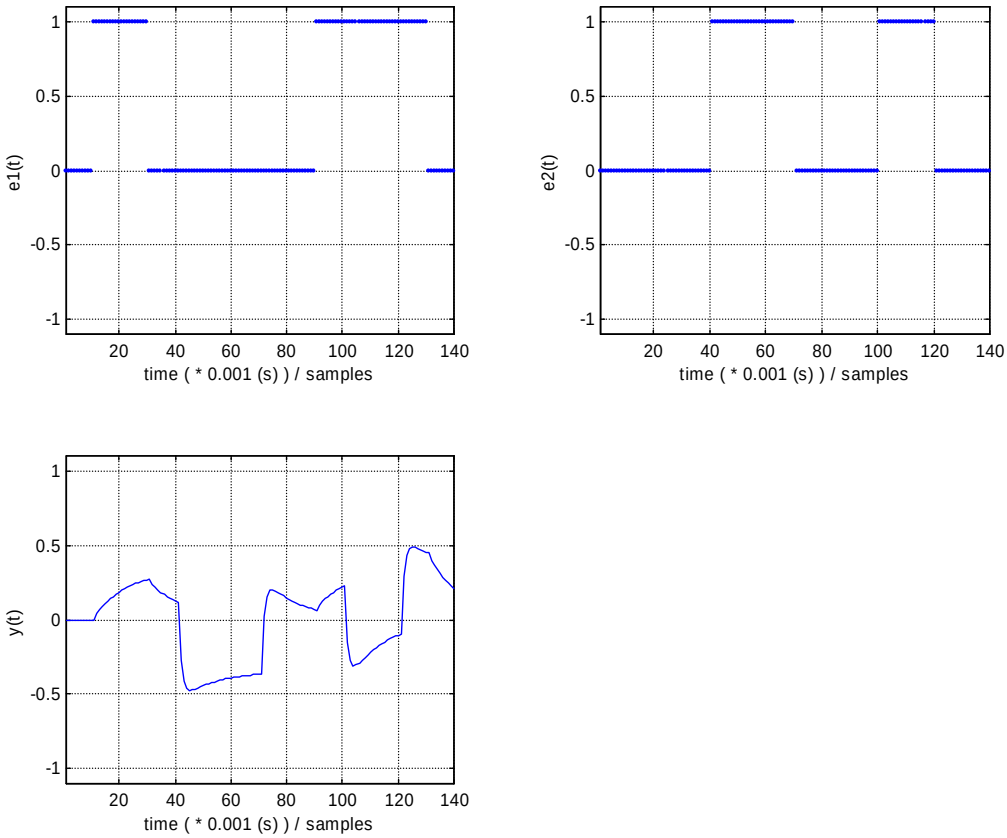


Figure 3.4: Input signals and circuit output y(t) (140 samples)

Choice of the neural network and training procedure

The neural network used is a Partially Recurrent Network (PRN) with 2 linear state neurons and one linear output neuron. The neurons do not have a bias. This choice is the 'ideal' structure because it matches exactly the state-space representation of the sampled RC circuit. The initial state of the neural network is set to be a vector of zeros and the weights are randomly initialised. The neural network was trained using the epochwise BPTT algorithm. The learning rate was chosen lr = 0.005 and the SSE performance goal 0.01 (this corresponds to a Mean Squared Error (MSE) of $0.01/140 = 7.1 \cdot 10^{-5}$).

Results of training the network with the epochwise BPTT algorithm

In figure 2.3 a series of several results is given. Each one of the eight graphs shows the output of the neural network after the network has been trained for the specified number of epochs. The target is shown as a dotted line, for comparison. It can be seen that the neural network output converges towards the target as the number of training epochs increases. The performance goal was eventually met in epoch 4674. Note that the error could be made smaller by more training.

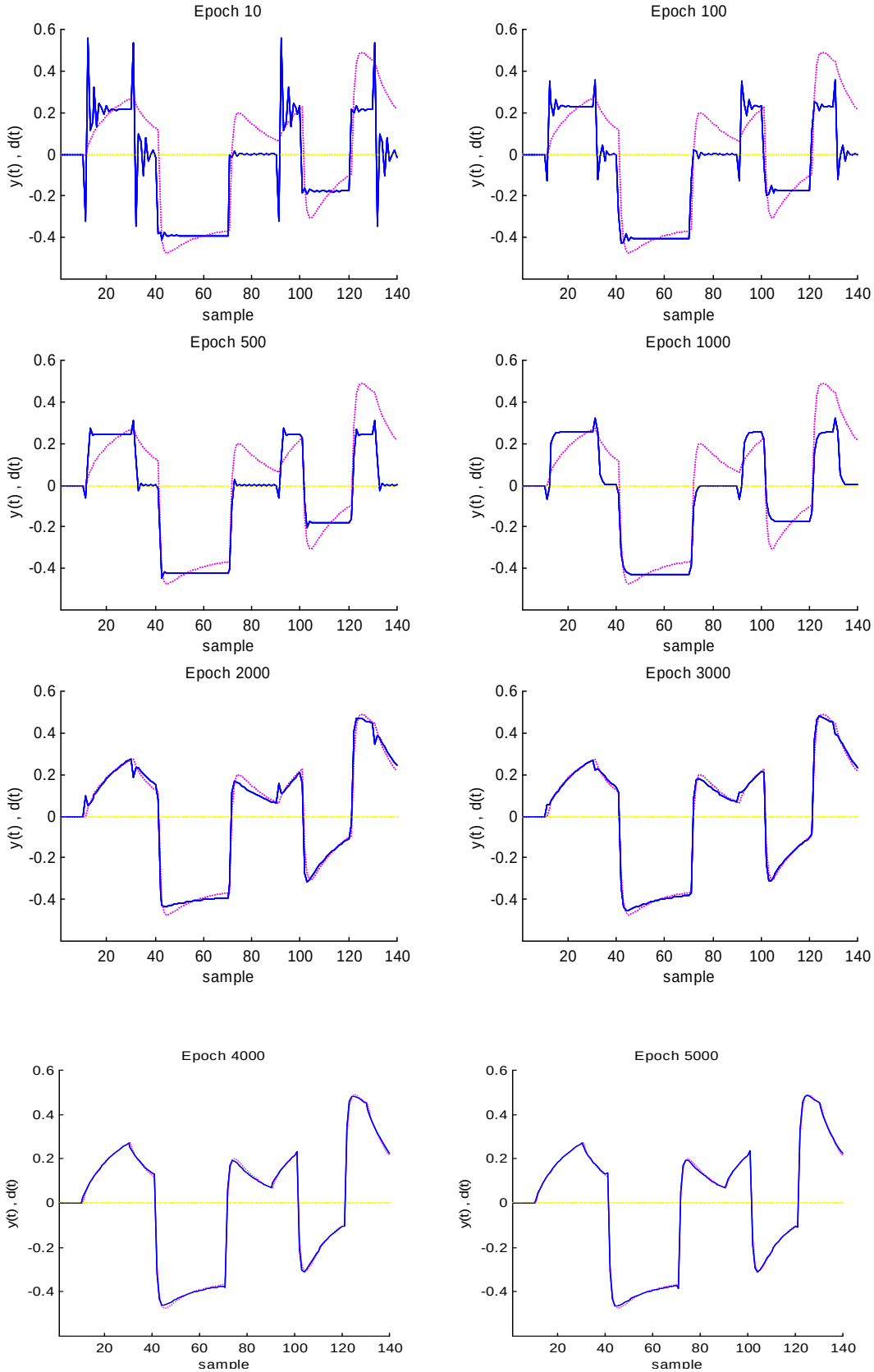


Figure 3.5; Neural network output (solid) and target (dotted) for several epochs of training with the epochwise BPTT algorithm.

In figure 2.9, Bode diagrams (amplitude and phase response) are given for both inputs (called U1=e_1 and U2=e_2 in the figure) of the final neural network (solid line) as well as the response of the RC circuit (dotted line). The plots were made using the Matlab *bode* function.

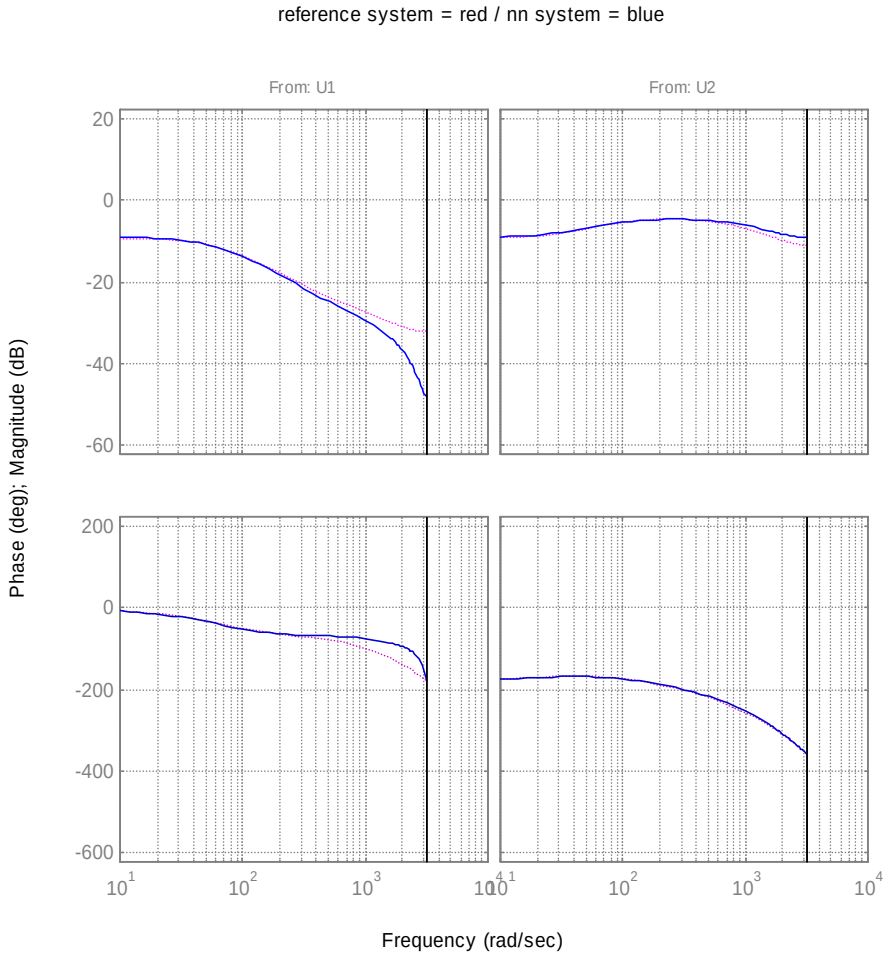


Figure 3.6; Bode diagrams for the neural network (solid) and the RC circuit (dotted) for both inputs U1/U2 ; magnitude response (upper two figures) and phase response (lower two figures).

The Bode diagrams give magnitude and phase response for each input separately. This figure shows the neural network has approximately learned the correct response. Errors are most clearly visible in this logarithmic plot for input U1 at high frequencies for two reasons:

- the attenuation is large there (>30 dB) so an error of order –30(dB) does only contribute to the (squared) SSE in the order of –60(dB).
- the input step-signals used have more low-frequency content than high-frequency content, so it is expected the frequency response deviation will be least at the lower frequencies.

Conclusions

It was shown in this experiment that the BPTT algorithm works for a simple linear learning task. The convergence of neural network weights towards an acceptable solution shows that the implementation of the algorithm is likely to be correct.

No comparison of the relative performance of the algorithm with other linear system identification methods is intended.

3.4 Real-time Recurrent Learning (RTRL) algorithm for FRNN

A real-time training algorithm for recurrent networks known as Real-time Recurrent Learning (RTRL) was derived by several authors [Williams e.a., 1995]. In this subsection the RTRL algorithm is derived for the type 1 FRNN and the Sum Squared Error as defined in equation 3.4. First the algorithm is completely developed using the standard partial derivative notation in subsection 3.4.1. A second approach follows in subsection 3.4.2, in which the ordered derivative notation is used to derive the same algorithm. In subsection 3.4.3 the computational complexity of RTRL is determined.

3.4.1 Derivation of the RTRL algorithm

The RTRL algorithm starts with a simplification of the error measure. Instead of using the true error measure $E_{TOTAL}(n_0,n)$ an approximation is made and only the instantaneous error measure $E(n)$ (equation 3.1) is used for calculating weight updates at each time instant n of the continually running network. Therefore RTRL can be categorized as a real-time algorithm. In standard RTRL the calculated updates for the weights are applied immediately after the calculation at each time n. But an *epochwise* RTRL algorithm can also be used, in which weight updates are only applied at the end of an epoch. First the standard RTRL algorithm is developed, then the epochwise RTRL algorithm is given at the end of this subsection.

So at every time n, the weights are adapted according to

$$\Delta w_{kl}(n) = -\eta \frac{\partial E(n)}{\partial w_{kl}} \tag{3.33}$$

where η is the learning rate. With the sum squared error measure substituted we have:

$$\frac{\partial E(n)}{\partial w_{kl}} = -\sum_{i=1}^{L} e_i(n) \frac{\partial y_i(n)}{\partial w_{kl}} \tag{3.34}$$

Substituting equation 3.8c of the FRNN in the partial derivative:

$$\frac{\partial y_i(n)}{\partial w_{kl}} = \frac{\partial y_i(n)}{\partial s_i(n)} \frac{\partial s_i(n)}{\partial w_{kl}} = f'(s_i(n)) \frac{\partial s_i(n)}{\partial w_{kl}} \tag{3.35}$$

The partial derivative of the neuron activation sums $s_i(n)$, with equations 3.8 substituted, yields the following results:

$$\begin{aligned} \frac{\partial s_i(n)}{\partial w_{kl}} &= \frac{\partial (\sum_{j=1}^{N+M} w_{ij} z_j(n))}{\partial w_{kl}} = \sum_{j=1}^{N+M} \left[w_{ij} \frac{\partial z_j(n)}{\partial w_{kl}} + z_j(n) \frac{\partial w_{ij}}{\partial w_{kl}} \right] = \\ &= \sum_{j=1}^{N+M} \left[w_{ij} \frac{\partial z_j(n)}{\partial w_{kl}} \right] + \delta_{ik} z_l(n) \end{aligned} \tag{3.36}$$

where δ_{ik} is the Kronecker delta function. A further simplification results because the network inputs do not depend on the weights ($\partial u_j / \partial w_{kl} = 0$):

$$\begin{aligned}\frac{\partial s_i(n)}{\partial w_{kl}} &= \sum_{j=1}^{N+M}\left[w_{ij}\frac{\partial z_j(n)}{\partial w_{kl}}\right] + \delta_{ik} z_l(n) = \\ &= \sum_{j=1}^{N}\left[w_{ij}\frac{\partial z_j(n)}{\partial w_{kl}}\right] + \sum_{j=N+1}^{N+M}\left[w_{ij}\frac{\partial z_j(n)}{\partial w_{kl}}\right] + \delta_{ik} z_l(n) = \\ &= \sum_{j=1}^{N}\left[w_{ij}\frac{\partial y_j(n-1)}{\partial w_{kl}}\right] + \sum_{j=N+1}^{N+M}\left[w_{ij}\frac{\partial u_{j-N}(n)}{\partial w_{kl}}\right] + \delta_{ik} z_l(n) \\ &= \sum_{j=1}^{N}\left[w_{ij}\frac{\partial y_j(n-1)}{\partial w_{kl}}\right] + 0 + \delta_{ik} z_l(n)\end{aligned} \tag{3.37}$$

The above equation can now be substituted into equation 3.35:

$$\frac{\partial y_i(n)}{\partial w_{kl}} = f'(s_i(n))\cdot\left[\sum_{j=1}^{N} w_{ij}\frac{\partial y_j(n-1)}{\partial w_{kl}} + \delta_{ik} z_l(n)\right] \tag{3.38}$$

To be able to compute the values $z_i(n_0)$, initial values for $y_i(n_0\text{-}1)$ must be set. These will also be set to zero:

$$y_i(n_0 - 1) = 0 \tag{3.39}$$

To be able to recursively compute equation 3.38 starting from time n=n_0, the partial derivatives $\partial y_j(n_0\text{-}1)/\partial w_{kl}$ must be known. Because the values $y_j(n_0\text{-}1)$ are fixed according to equation 3.39 these values can not be influence by the weights. So the 'initial' partial derivatives are zero:

$$\frac{\partial y_j(n_0 - 1)}{\partial w_{kl}} = 0 \tag{3.40}$$

For ease of notation of the algorithm we first define the following variable:

$$\pi_{kl}^{i}(n) = \frac{\partial y_i(n)}{\partial w_{kl}} \tag{3.41}$$

Real-time Recurrent Training algorithm

1. initialize as in equations 3.40, 3.39 and set initial time n=n_0
2. calculate the N neuron output values for time n using equations 3.8
3. calculate $\pi_{kl}^{i}(n)$ for all i, k and l using the following equation

$$\pi_{kl}^{i}(n) = f'(s_i(n))\cdot\left[\sum_{j=1}^{N} w_{ij}\pi_{kl}^{j}(n-1) + \delta_{ik} z_l(n)\right] \tag{3.42}$$

   which is equation 3.38 rewritten with the new definition of $\pi_{kl}^{i}(.)$.

4. use the values $\pi_{kl}^{i}(n)$ and the present error $e_i(n)$ to calculate the weight update:

$$\Delta w_{kl}(n) = \eta\sum_{i=1}^{L} e_i(n)\pi_{kl}^{i}(n) \tag{3.43}$$

   (obtained by substituting equations 3.34 and the definition $\pi_{kl}^{i}(.)$ into equation 3.33.)

5. Update all weights w_{kl}:

$$w_{kl}(n+1) = w_{kl}(n) + \Delta w_{kl}(n) \tag{3.44}$$

6. Increase time n to n+1 and go back to step 2

Epochwise Real-time Recurrent Training algorithm
A small change can be made to the RTRL algorithm that results in an epochwise algorithm, that updates the weights only after a full example sequence (epoch) in the interval $[n_{E0},n_{E1}]$ has been presented to the network. This weight update at the end of an example is the sum of individual calculated weight updates $\Delta w_{kl}(n)$ at each time n (that were not yet applied to w_{kl}):

$$\Delta w_{kl} = \sum_{n=n_{E0}}^{n_{E1}} \Delta w_{kl}(n) \tag{3.45}$$

This algorithm yields different results than standard RTRL. The continuous updating of weights in standard RTRL implies that different weights are used for each time step in the calculation of the algorithm. The epochwise RTRL algorithm keeps using the same weight values throughout the epoch so numerical results will differ.

In the literature reviewed, no proofs were found whether the standard RTRL algorithm is the fastest or not. No claims were found about the convergence of both methods (will they eventually converge to the same result, or not).

Intuitively one would say the more frequent weight updating in standard RTRL allows faster movement through the parameter space (i.e. weight space) and therefore convergence to a good enough result in less training epochs.

3.4.2 Derivation of the RTRL algorithm using the ordered derivative

The RTRL algorithm can be derived using the ordered derivative. This method was also used in [Bengio, 1996] but the derivation given in this report is a bit more clear and shows exactly when chain rules are applied.

The reason for doing the derivation again with the ordered derivative is to show the difference between the two approaches. Because this derivation is only for the specific case of the FRNN, it was found to be a useful preparation for the more general form of the RTRL algorithm for modular networks which will be treated in section 3.8.

The full derivation is given in Appendix B.2. The derivation starts with the weight update, which can be calculated using the gradient just as in equation 3.33:

$$\Delta w_{kl}(n) = -\eta \frac{\partial^{+} E(n)}{\partial w_{kl}} \tag{3.46}$$

The ordered derivative notation is now used. This means both direct and indirect influence of w_{kl} on the error measure is taken into account.

3.4.3 Computational complexity of RTRL

The computational complexity of RTRL will be considered for a fully connected network. (A constrained architecture will usually require less computations.) First define the total number of weights $N_w = (N+M)*N$. As can be seen in equation 3.42 a number of $N_\pi = N_w*N = (N+M)*N*N$ values $\pi_{kl}^{i}(.)$ of the previous time-step need to be stored. For this a storage space of $S=N_\pi$ values is required.

The required number of operations and storage space is listed for the RTRL algorithm in table 3.3. The dominating terms (for N>>1) are printed in bold.

If we ignore the smallest terms in the total sums (N>>1), it follows that the total number of operations per time step is of order $O(N^3 (N+M))$ and the required storage is of order $O(N^2 (N+M))$. There is no saving in computational resources, if the epochwise RTRL algorithm is used.

a	b	c	d
Equation	**Multiplications**	**Additions**	**Storage**
3.42	(1+N+1/N) * N_π = **(N+1) * N_π** + N_w	**(1+N-1) * N_π**	**N_π**
3.43	(L+1) * N_w	(L-1) * N_w	M values $e_i(m)$
Totals:	**(2N+1)*N_π** +(2*L+1)*N_w		**N_π**+M
Order of magnitude:	O[N^3(N+M)]		O[N^2(N+M)]

Table 3.3: Resources required for the RTRL algorithm

3.4.4 Example

The same experiment as performed in subsection 3.3.6 with the epochwise BPTT algorithm is repeated here for the RTRL algorithm.

Standard RTRL algorithm
With the RTRL algorithm the SSE goal was met after 786 epochs of training, which is much faster than with epochwise BPTT or epochwise RTRL (see below). The weight updating at every time step in standard RTRL, opposed to only once per epoch in epochwise algorithms, clearly leads to improved performance for this task. The learning rate was set lr = 0.01. The training record is shown in figure 3.7.

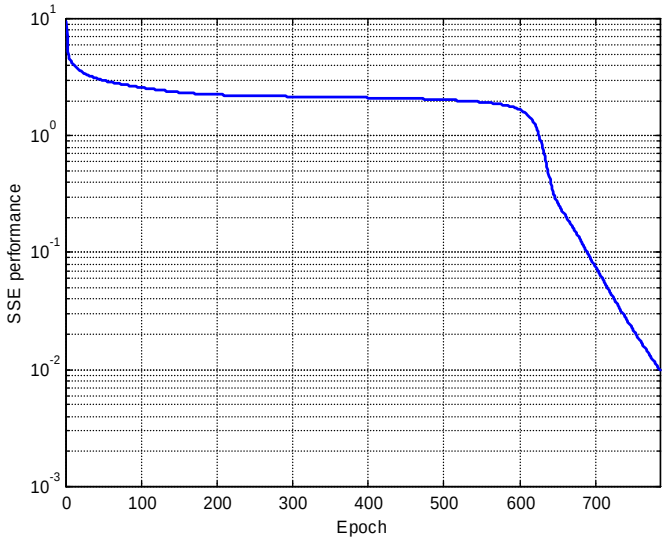


Figure 3.7: Sum Squared Error performance over the 786 epochs of training

These results are not compared to standard BPTT because this algorithm was not implemented.

Epochwise RTRL algorithm
When the epochwise RTRL algorithm is used, the outcomes of the experiment are *identical* to epochwise BPTT. (See subsection 3.3.6 for these results.)
This implies that for this task and using linear networks, the following must hold:

$$\frac{\partial E(n_0,n)}{\partial w_{ij}} = \sum_{m=n_0}^{n} \frac{\partial E(m)}{\partial w_{ij}} \tag{3.47}$$

for all weights w_{ij}. This relation states that the gradient calculations for epochwise BPPT (left) have the same value for all weights as the gradient calculations for epochwise RTRL (right).

This is not the case in general. By using nonlinear neuron transfer functions and running the experiment again it was found the property only holds for linear neurons.

3.5 Joint training algorithm for general modular network

Modular neural networks were presented in subsection 2.4.3. In this subsection it will be shown how these modular networks can be trained.

In [Bengio, 1996] it is shown how the static modules contained in a modular neural network architecture can be trained together to minimize a global error measure. To show why it can be advantageous to do a joint training of network modules instead of training each module separately, the following example situation is introduced.

Why use joint training (1)
As an example consider a two-network architecture where network 2 is cascaded after network 1. The training procedure uses the gradient of network 1 $\partial E_1/\partial \mathbf{w}_1$ and for network 2 $\partial E_2/\partial \mathbf{w}_2$. When both E_1 and E_2 are minimized (then $\partial E_1/\partial \mathbf{w}_1$ and $\partial E_1/\partial \mathbf{w}_2$ are zero) the training algorithm stops, but the error measure could perhaps be made smaller by evaluating $\partial E_2/\partial \mathbf{w}_1$ which may be non-zero. So the separate training of modules may yield sub-optimal performance.

It is assumed in this example that targets are known for both modules and that module 1 output is fed to module 2 to 'aid' module 2 in its task (e.g. classification).

Why use joint training (2)
Joint training can also be applied if not all modules of the network have targets. An example is the state-space network (in modular description) that only has targets defined for module 2 (the output function) and not for module 1 (the state function). In the derivation of the joint training algorithms it will be made clear that all modules can be trained even without targets for all modules.

BPTT and RTRL joint training algorithms
The derivation of the joint training algorithm (as given in [Bengio, 1996]) is given in subsection 3.5.1. It resembles the derivation of the BPTT algorithm for FRNN so it is called the *BPTT joint training algorithm*. Inspired on the RTRL algorithm for FRNN, another type of joint training algorithm may be derived. This *RTRL joint training algorithm* will be derived in subsection 3.5.3.

In both subsections two special cases of modular networks are treated: the state-space network and the FRNN. This shows that the BPTT algorithm for FRNN is in fact a special case of the BPTT joint training algorithm. The RTRL algorithm for FRNN is a special case of the RTRL joint training algorithm.

Alternative derivation of the BPTT and RTRL algorithms
In [Baldi, 1995] the BPTT and RTRL algorithms are derived in another way without using the modular description for neural networks. The model used by Baldi intends to be an even more general description than the modular network framework, since it includes continuous-time neural networks. The derivation shows the RTRL algorithm is a straightforward solving procedure (using numerical integration) for a dynamic system trajectory-following problem. The BPTT algorithm corresponds to a solving procedure for the *adjoint* dynamic system for the same problem. Unfortunately, there was no time left to study this approach and include it here. This is no real problem, since only time-discrete networks are considered in this chapter and the modular network framework is sufficient for this purpose.

Definitions
The definitions of a modular network as given in subsection 2.4.3 for l , $p(l)$, $s(l)$, $d(l)$, $\mathbf{y}_i(n)$, $y_{im}(n)$, and L_i are used throughout this section.

3.5.1 The BPTT joint training algorithm

The gradient of the cost function

$$E = E(n_0, n_1) \tag{3.48}$$

for a certain training sequence p in time interval $[n_0,n_1]$ can be used to adjust all weights *j* in module *i*, θ_{ij} , as follows:

$$\Delta\theta_{ij} = -\eta \frac{\partial^+ E(n_0, n_1)}{\partial \theta_{ij}} \tag{3.49}$$

where $\partial^+ \cdot / \partial \cdot$ denotes the *ordered derivative* (see section 3.3). Applying the first chain rule for ordered derivatives is the next step:

$$\frac{\partial^+ E}{\partial \theta_{ij}} = 0 + \sum_{n=n_0}^{n_1} \sum_{m=1}^{L_i} \frac{\partial^+ E}{\partial y_{im}(n)} \frac{\partial y_{im}(n)}{\partial \theta_{ij}} \tag{3.50}$$

The first term is zero because the error measure E is only influenced by the parameter θ_{ij} indirectly through all variables $y_{im}(n)$ (over all outputs m=1…L_i and over all times n=n_0…n_1). In this equation the second term depends on the definition of the module $\mathbf{F_i}(.)$. The first factor can be decomposed again using the first chain rule:

$$\frac{\partial^+ E}{\partial y_{im}(n)} = \frac{\partial E}{\partial y_{im}(n)} + \sum_{l \in P_i} \sum_{q=1}^{L_{s(l)}} \frac{\partial^+ E}{\partial y_{s(l);q}(n + d(l))} \frac{\partial y_{s(l);q}(n + d(l))}{\partial y_{im}(n)} \tag{3.51}$$

where the set P_i = { l | p(l)=i } denotes all links that come from module *i*. The first direct derivative term is the direct influence of $y_{im}(n)$ on the error measure. It is zero when there is no *supervision* (no target values are defined) on a module. It can be seen in the second term that any indirect influence of $y_{im}(n)$ on the error measure is exercised through the 'future' output vectors $\mathbf{y}_{s(l)}(n+d(l))$ of all modules s(l) that are connected as a successor of the module *i*.

The equation is similar to the one found for the BPTT algorithm for FRNN (equation 3.22) but more general. The procedure, of recursively computing equation 3.51 in a backward pass and computing equation 3.50 in a forward pass, is essentially the same as in the BPTT algorithm for FRNN.

By taking a specific modular network architecture the equations can be further developed and a BPTT-like training algorithm results to train the parameters of all network modules. This will be done next, for the two special cases of the modular network framework from subsection 2.4.3: the FRNN and the state-space network architecture.

Special case 1: the FRNN

The BPTT training algorithm for the Fully Recurrent Neural Network can be derived as a special case of the BPTT equations of joint training for modular networks.

For convenience, the description of a FRNN in terms of the modular network framework is repeated here (it was also given in subsection 2.4.3). The FRNN is defined as a single-module network having one link that connects the module output to its input. See figure 3.8.

In the general case, the module can contain any *n*-layer neural network. By constraining the module to a single-layer network, the special case of a FRNN is obtained.

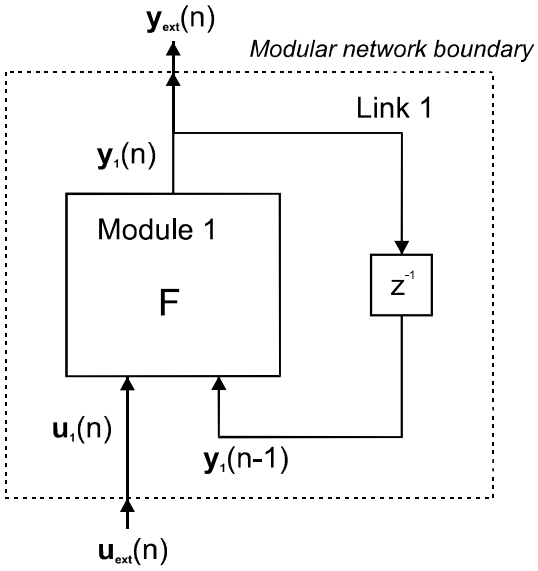


Figure 3.8; FRNN presented as a single module network with one delayed recurrent connection

The single-layer static feedforward network module 1 has N neurons which are all considered external outputs, M+N external inputs and as parameters the weights $\theta_1=\mathbf{w_1}=\{w_{1jk}\}$. This static network does the following computation for every neuron *j*:

$$\mathbf{y}_1(n) = \mathbf{F}_1(\mathbf{w}_1 \cdot \mathbf{z}_1(n))$$
$$\text{with } \mathbf{z}_1(n) = \begin{bmatrix} \mathbf{y}_1(n-1) \\ \mathbf{u}_1(n) \end{bmatrix} \tag{3.52}$$

and in scalar notation:

$$y_{1j}(n) = f\left(\sum_{k=1}^{N+M} w_{1jk} z_{1k}(n) \right) \tag{3.53}$$

where $z_{1k}(.)$, the inputs of module 1, are the elements of the input vector $\mathbf{z_1}(n)$.

There is only one module so *i*=1. The number of outputs for this module is L_1=N. Equation 3.50 is used with the weights w_{1jk} substituted as the parameters θ_{1j} and using L_1=N:

$$\frac{\partial^+ E}{\partial w_{1jk}} = \sum_{n=n_0}^{n_1} \sum_{m=1}^{N} \frac{\partial^+ E}{\partial y_{1m}(n)} \frac{\partial y_{1m}(n)}{\partial w_{1jk}} \tag{3.54}$$

because the last term is only non-zero when *m*=*j* the sum over *m* disappears:

$$\frac{\partial^+ E}{\partial w_{1jk}} = \sum_{n=n_0}^{n_1} \frac{\partial^+ E}{\partial y_{1j}(n)} \frac{\partial y_{1j}(n)}{\partial w_{1jk}} \tag{3.55}$$

Next equation 3.51 is used for writing out the first ordered derivative term $\partial^+E/\partial y_{1j}(n)$ in the equation above. For the only link *l*=1, both the predecessor and successor is module 1, s(1)=p(1)=1, and the delay in the link is one, d(1)=1.

$$\frac{\partial^+ E}{\partial y_{1m}(n)} = \frac{\partial E}{\partial y_{1m}(n)} + \sum_{l=1}\sum_{q=1}^{N} \frac{\partial^+ E}{\partial y_{s(l);q}(n+d(l))} \frac{\partial y_{s(l);q}(n+d(l))}{\partial y_{1m}(n)} =$$
$$= \frac{\partial E}{\partial y_{1m}(n)} + \sum_{q=1}^{N} \frac{\partial^+ E}{\partial y_{s(1);q}(n+1)} \frac{\partial y_{s(1);q}(n+1)}{\partial y_{1m}(n)} = \tag{3.56}$$
$$= \frac{\partial E}{\partial y_{1m}(n)} + \sum_{q=1}^{N} \frac{\partial^+ E}{\partial y_{1q}(n+1)} \frac{\partial y_{1q}(n+1)}{\partial y_{1m}(n)}$$

Because there is only one module, there is no need to give the module 1 variables the explicit subscript "1". So new variable names $y_m(n) \equiv y_{1n}(n)$ and $w_{jk} \equiv w_{1jk}$ are defined, that will be used for the final notation of the obtained algorithm. With the new definitions, equation 3.55 becomes:

$$\frac{\partial^+ E}{\partial w_{jk}} = \sum_{n=n_0}^{n_1} \frac{\partial^+ E}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial w_{jk}} \tag{3.57}$$

which is equal to the BPTT equation 3.21. With the new definitions, equation 3.56 can be written as:

$$\frac{\partial^+ E}{\partial y_m(n)} = \frac{\partial E}{\partial y_m(n)} + \sum_{q=1}^{N} \frac{\partial^+ E}{\partial y_q(n+1)} \frac{\partial y_q(n+1)}{\partial y_m(n)} \tag{3.58}$$

This equation is equal to the BPTT equation 3.22 for FRNN. From here on, the rest of the derivation is identical to that of the BPTT algorithm for FRNN if the Sum Squared Error measure is used. Therefore, it will not be shown here again. The derivation could also be continued using a different error measure.

Special case 2: the general state-space network architecture

As shown in subsection 2.4.3, the general state-space network architecture can be described as a modular network. Using the general equations for joint training of modular networks the BPTT training algorithm will be derived for the state-space network.

For convenience, the modular network description is shown again in figure 3.9.

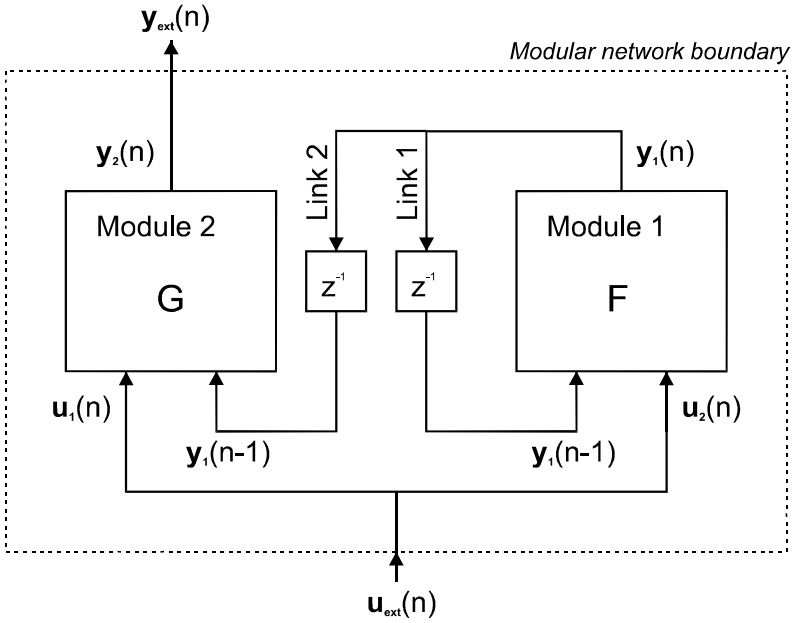


Figure 3.9; General state-space network shown as a two-module network with two links

The ordered derivative used for the parameter adaptations is given by equation 3.50 (repeated here for convenience):

$$\frac{\partial^+ E}{\partial \theta_{ij}} = \sum_{n=n_0}^{n_1} \sum_{m=1}^{L_i} \frac{\partial^+ E}{\partial y_{im}(n)} \frac{\partial y_{im}(n)}{\partial \theta_{ij}} \tag{3.59}$$

For module 1 equation 3.51 is developed with link 1 the self-recurrent link (p(1)=1; s(1)=1; d(1)=1) and link 2 the link to the second module (p(2)=1; s(2)=2; d(2)=1):

$$\begin{aligned} &\frac{\partial^+ E}{\partial y_{im}(n)} = \frac{\partial E}{\partial y_{im}(n)} + \sum_{l \in P} \sum_{q=1}^{L_{s(l)}} \frac{\partial^+ E}{\partial y_{s(l);q}(n+d(l))} \frac{\partial y_{s(l);q}(n+d(l))}{\partial y_{im}(n)} \Leftrightarrow \\ &\frac{\partial^+ E}{\partial y_{1m}(n)} = \frac{\partial E}{\partial y_{1m}(n)} + \sum_{q=1}^{L_1} \frac{\partial^+ E}{\partial y_{1q}(n+1)} \frac{\partial y_{1q}(n+1)}{\partial y_{1m}(n)} + \sum_{q=1}^{L_2} \frac{\partial^+ E}{\partial y_{2q}(n+1)} \frac{\partial y_{2q}(n+1)}{\partial y_{1m}(n)} \end{aligned} \tag{3.60}$$

For module 2 equation 3.51 is also developed. Since this module has no links going from it:

$$\frac{\partial^+ E}{\partial y_{2m}(n)} = \frac{\partial E}{\partial y_{2m}(n)} + 0 = \frac{\partial E}{\partial y_{2m}(n)} \tag{3.61}$$

Substituting equation 3.61 into equation 3.60, and using the fact that there is no supervision on module 1 ($\partial E / \partial y_{1m}(n) = 0$), the expression for module 1 becomes:

$$\frac{\partial^+ E}{\partial y_{1m}(n)} = \sum_{q=1}^{L_1} \frac{\partial^+ E}{\partial y_{1q}(n+1)} \frac{\partial y_{1q}(n+1)}{\partial y_{1m}(n)} + \sum_{q=1}^{L_2} \frac{\partial E}{\partial y_{2q}(n+1)} \frac{\partial y_{2q}(n+1)}{\partial y_{1m}(n)} \tag{3.62}$$

This equation can be computed recursively in a single backward pass. Further development of the partial derivative terms $\partial y..(n+1)/\partial y_{1m}(n)$ is possible when the static network module structures are known in more detail. The instantaneous error terms $\partial E/\partial y_{2q}(n)$ can be developed further when a specific error measure is chosen.

3.5.2 Example: Hénon's system

In this example a state-space network is used to identify the nonlinear Hénon's system.

Hénon's equations

The discrete-time nonlinear system is described by the following set of equations known as Hénon's equations [Janssen, 1998].

$$\begin{aligned} x(n+1) &= 1 - a \cdot x^2(n) + y(n) \\ y(n+1) &= b \cdot x(n) \end{aligned} \tag{3.63}$$

Although not given in the state-space form, they can be rewritten to a state-space system. The parameters are usually chosen a=1.4 and b=0.3. The system then generates a chaotic signal shown in figure 3.10 for the initial conditions x(0)=0, y(0)=0.

Experimental setup

The Hénon's time series was used in [Janssen, 1998] to show the capability of a Time-Delay Neural Network to approximate an unknown nonlinear dynamic system. In that experiment, 5000 samples of the Hénon time series were used to train a network containing 48 weights.

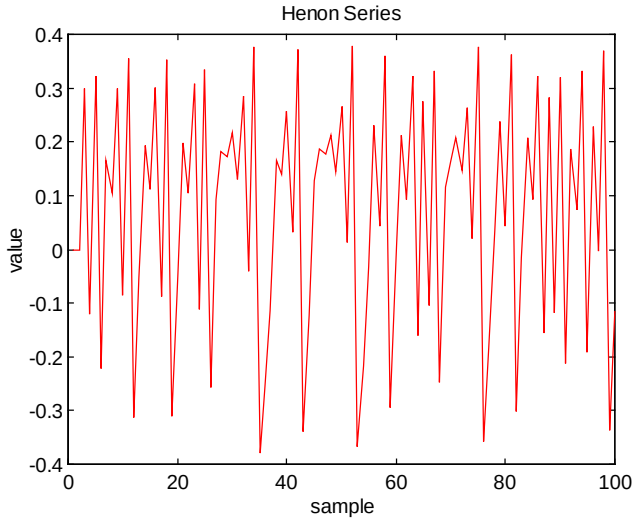


Figure 3.10: Hénon's time series (First 100 samples)

The experiment is repeated now for a nonlinear recurrent state-space network. The goal of the experiment is to train a neural network to predict the next sample y(n+1), when supplied with only the current sample y(n). Past information is summarized in the state of the neural network.
The following neural network structure (named 'N1') was used first:

- module 1: 2-layer network ; 5 nonlinear hidden tansig (tangential sigmoid transfer function) neurons ; 3 linear output (state) neurons
- module 2: 2-layer network ; 5 nonlinear hidden tansig neurons ; 1 linear output neuron

The total number of weights is 84. A much smaller training set than used in [Janssen, 1998] was used, 300 samples. The epochwise BPTT algorithm (the *bptt_epoch* algorithm) was used combined with adaptive learning rate but no momentum learning (the *traingda* training function). The SSE goal was set to 0.1. (This implies a Mean Squared Error of 0.1/300 = $3{,}3*10^{-4}$.)
In a second run, a smaller neural network 'N2' was chosen:

- module 1: 2-layer network ; 4 nonlinear hidden tansig neurons ; 2 linear output (state) neurons
- module 2: 2-layer network ; 4 nonlinear hidden tansig neurons ; 1 linear output neuron

The total number of weights was 47 in this case, one less than the FIR network in [Janssen, 1998].

Results of training network N1
After 9000 epochs of training network N1, the SSE performance goal was met. To get a visual impression of the system the neural network has identified, two plots are shown in figure 3.4 that show the relationship between the current sample x(n) and the previous sample x(n-1) for both the original Hénon's system and the trained neural network. This type of plot visualizes the so-called *attractor* of the system. It can be seen the neural network attractor plot resembles the Hénon attractor plot.

In figure 2.12, the Hénon time series and the neural network prediction are shown for the 51 samples 4000-4050.

These samples were not used to train the network. They constitute an independent test set. It can be seen the network can predict the Hénon samples with good accuracy. The Mean Squared Error for this subsection is $5*10^{-4}$, counting from sample 4002. (The first two are not counted because two samples must be known in the system equation 3.63 to be able to make a prediction). The MSE is slightly higher than the MSE for the training sequence.

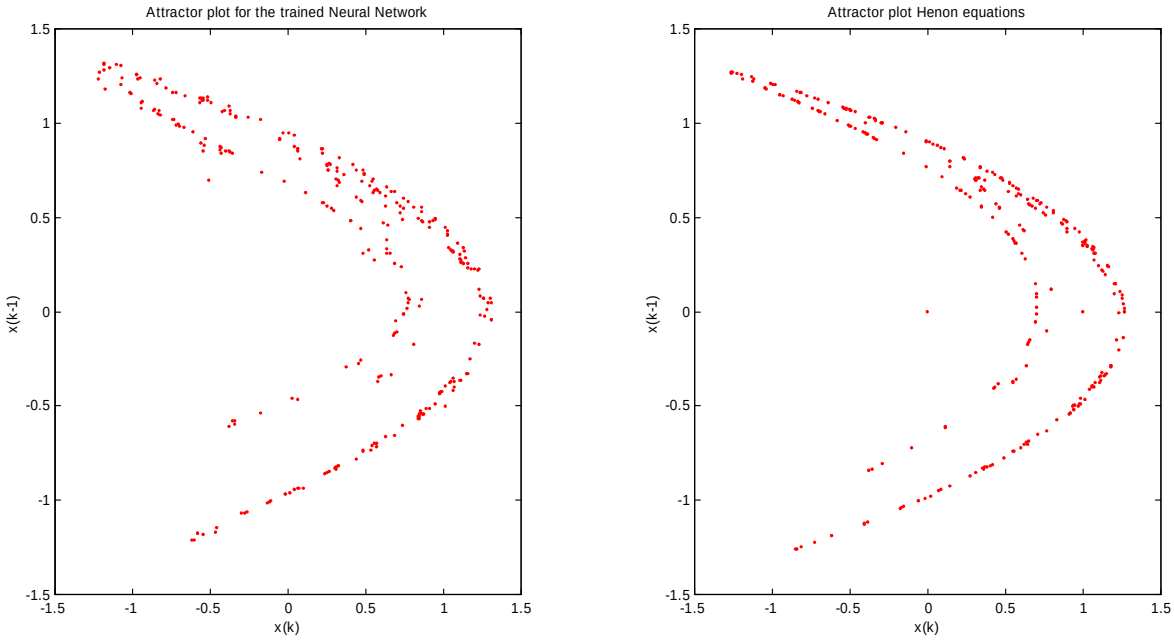


Figure 3.11; The neural network attractor plot (left) and the Hénon attractor plot (right)

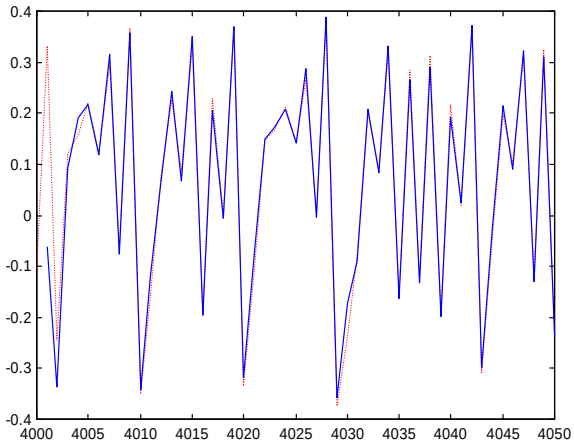


Figure 3.12; The Hénon time series (dotted) and its neural network prediction (solid) for the test samples 4000-4050.

Results of training network N2

The training was repeated with the (smaller) network N2. After 10000 training epochs, the SSE goal was still not met. Training was stopped and the SSE did come close to the goal, at SSE = 0.18. So the MSE for the test sequence (samples 4002-4050) was a bit higher, MSE = $7*10^{-4}$.

Conclusions

In this experiment, the state-space neural networks trained with the BPTT algorithm were able to find a representation of (to identify) the nonlinear Hénon's system. The training set used was relatively small compared to the number of weights in the networks. Note that a comparison of these results with other ones (e.g. [Janssen, 1998]) for the Hénon problem is not intended. For this, a comparable experimental setup should be used and several runs per setup.

It can be concluded that the BPTT algorithm works for the two-module two-layer state-space neural network used.

3.5.3 The RTRL joint training algorithm

Another algorithm for joint training of modular networks can be obtained inspired on the RTRL algorithm for FRNN. This RTRL joint training algorithm will be presented here. The ordered derivative notation is used because it allows easy notation of the algorithm as was found in Appendix B.2. An instantaneous error measure E(n) is used as was done previously for the RTRL algorithm for FRNN. The weight updates for parameters j in all modules i are then given by:

$$\Delta\theta_{ij} = -\eta \frac{\partial^+ E(n)}{\partial \theta_{ij}} \tag{3.64}$$

In the derivation of the BPTT joint training algorithm the first chain rule for ordered derivatives was used to develop the weight update expression. For deriving the RTRL joint training algorithm the second chain rule for ordered derivatives is used:

$$\frac{\partial^+ E(n)}{\partial \theta_{ij}} = 0 + \sum_{k=1}^{N_{\text{mod}}} \sum_{m=1}^{L_k} \frac{\partial E(n)}{\partial y_{km}(n)} \frac{\partial^+ y_{km}(n)}{\partial \theta_{ij}} = \sum_{k \in D'} \sum_{m=1}^{L_k} \frac{\partial E(n)}{\partial y_{km}(n)} \frac{\partial^+ y_{km}(n)}{\partial \theta_{ij}} \tag{3.65}$$

The set D' holds all modules whose outputs have targets defined (these outputs are the ones used in calculating the error measure). All modules k that are not in D' have $\partial E(n)/\partial y_{km}(n) = 0$ for all m. Now for each module k in D' the first partial derivative $\partial E(n)/\partial y_{km}(n)$ can be obtained directly from the error measure. The last term is developed further using the second chain rule. Because the current outputs at time n are influenced indirectly (through the links l) by all outputs of all modules at times n-d(l) that have a link going to module i, we have:

$$\begin{aligned} \frac{\partial^+ y_{km}(n)}{\partial \theta_{ij}} &= \frac{\partial y_{km}(n)}{\partial \theta_{ij}} + \sum_{l \in S_k} \sum_{q=1}^{L_{p(l)}} \frac{\partial y_{km}(n)}{\partial y_{p(l);q}(n-d(l))} \frac{\partial^+ y_{p(l);q}(n-d(l))}{\partial \theta_{ij}} = \\ &= \delta_{ik} \frac{\partial y_{im}(n)}{\partial \theta_{ij}} + \sum_{l \in S_k} \sum_{q=1}^{L_{p(l)}} \frac{\partial y_{km}(n)}{\partial y_{p(l);q}(n-d(l))} \frac{\partial^+ y_{p(l);q}(n-d(l))}{\partial \theta_{ij}} \end{aligned} \tag{3.66}$$

The set $S_k = \{ l \mid s(l) = k \}$ denotes all links that go to module k, and δ_{ik} is the Kronecker delta function. The ordered derivative term can be computed recursively using the above equation, by saving all values

$$\frac{\partial^+ y_{p(l);q}(n-d(l))}{\partial \theta_{ij}} \tag{3.67}$$

of the 'previous' time steps n-d(l). The first two partial derivative terms in equation 3.66 can be developed if the structure of the static network modules is known in more detail.

Special case: an RTRL algorithm for the general state-space network architecture

The general equations for the RTRL joint training algorithm are now developed for the special case of a two-module state-space neural network. The Sum Squared Error measure of equation 3.1 is used:

$$E(n) = \frac{1}{2} \sum_{i=1}^{L} e_i^2(n) \tag{3.68}$$

with $e_i(n)$ as defined in equation 3.2 but now with $d_{2i}(n)=d_i(n)$ (i.e. the targets on module 2 are the targets on the external output) and $y_i(n)=y_{2i}(n)$ (i.e. the module 2 outputs are the external outputs). So we have:

$$e_i(n) = \begin{cases} d_{2i}(n) - y_{2i}(n) & \text{for i} \in \text{D} \\ 0 & \text{otherwise} \end{cases} \tag{3.69}$$

Equation 3.65 now yields:

$$\frac{\partial^+ E(n)}{\partial \theta_{ij}} = \sum_{k\in D'}\sum_{m=1}^{L_k} \frac{\partial E(n)}{\partial y_{km}(n)} \frac{\partial^+ y_{km}(n)}{\partial \theta_{ij}} = \sum_{m=1}^{L_2} \frac{\partial E(n)}{\partial y_{2m}(n)} \frac{\partial^+ y_{2m}(n)}{\partial \theta_{ij}} = \\ = \sum_{m\in D} e_m(n) \frac{\partial^+ y_{2m}(n)}{\partial \theta_{ij}} \tag{3.70}$$

where D′ = {2} because of supervision on module 2 only.

Module 1 equations

For module 1 (i=1) equation 3.66 is developed with link l=1 the self-recurrent link (p(1)=1; s(1)=1; d(1)=1) and link 2 the link to the second module (p(2)=1; s(2)=2; d(2)=1):

$$\frac{\partial^+ y_{2m}(n)}{\partial \theta_{1j}} = 0 + \sum_{q=1}^{L_1} \frac{\partial y_{2m}(n)}{\partial y_{1q}(n-1)} \frac{\partial^+ y_{1q}(n-1)}{\partial \theta_{1j}} \tag{3.71}$$

$$\frac{\partial^+ y_{1m}(n)}{\partial \theta_{1j}} = \frac{\partial y_{1m}(n)}{\partial \theta_{1j}} + \sum_{q=1}^{L_1} \frac{\partial y_{1m}(n)}{\partial y_{1q}(n-1)} \frac{\partial^+ y_{1q}(n-1)}{\partial \theta_{1j}} \tag{3.72}$$

Here the second equation can be computed recursively and the first equation can be computed using the result of the second equation.

Module 2 equations

For module 2 (i=2) equation 3.66 yields with link 2 the link to this module (p(2)=1 ; s(2)=2 ; d(2) = 1):

$$\frac{\partial^+ y_{2m}(n)}{\partial \theta_{2j}} = \frac{\partial y_{2m}(n)}{\partial \theta_{2j}} + \sum_{q=1}^{L_1} \frac{\partial y_{2m}(n)}{\partial y_{1q}(n-1)} \frac{\partial^+ y_{1q}(n-1)}{\partial \theta_{2j}} \tag{3.73}$$

$$\frac{\partial^+ y_{1m}(n)}{\partial \theta_{2j}} = 0 + \sum_{q=1}^{L_1} \frac{\partial y_{1m}(n)}{\partial y_{1q}(n-1)} \frac{\partial^+ y_{1q}(n-1)}{\partial \theta_{2j}} = 0 \tag{3.74}$$

The last equation must be zero, which is known beforehand, because the parameters of module 2 cannot influence in any way the outputs of module 1 (There is no link from module 2 to module 1). Of course the same result is obtained when the last equation is actually calculated and an initial condition of zero is taken for the ordered derivative term. This is not shown further here.

The above equation simplifies equation 3.73 to:

$$\frac{\partial^+ y_{2m}(n)}{\partial \theta_{2j}} = \frac{\partial y_{2m}(n)}{\partial \theta_{2j}} \tag{3.75}$$

Final notation

To write the algorithm down in a convenient way, the following variable is defined:

$$\pi_{kl}^{ij}(n) \equiv \frac{\partial^+ y_{ij}(n)}{\partial \theta_{kl}} \tag{3.76}$$

Now the above equations 3.71, 3.72 and 3.75 respectively are rewritten using this definition, which yields the RTRL algorithm for a general state-space network architecture:

$$\pi_{1j}^{2m}(n) = \sum_{q=1}^{L_1} \frac{\partial y_{2m}(n)}{\partial y_{1q}(n-1)} \pi_{1j}^{1q}(n-1) \tag{3.77}$$

$$\pi_{1j}^{1m}(n) = \frac{\partial y_{1m}(n)}{\partial \theta_{1j}} + \sum_{q=1}^{L_1} \frac{\partial y_{1m}(n)}{\partial y_{1q}(n-1)} \pi_{1j}^{1q}(n-1) \tag{3.78}$$

$$\pi_{2j}^{2m}(n) = \frac{\partial y_{2m}(n)}{\partial \theta_{2j}} \tag{3.79}$$

Finally the update of parameter *j* for module *i* is calculated using equations 3.64/3.70. Using the newly defined variable π this can be written as:

$$\Delta\theta_{ij}(n) = \eta \sum_{q \in D} e_q(n) \pi_{ij}^{2q}(n) \tag{3.80}$$

The RTRL algorithm for state-space networks is hereby derived. For specific choices of neural network modules, the terms ∂y../∂y.. and ∂y../ ∂θ.. can be further developed.

3.5.4 Example: Hénon's system

The identification task of the Hénon system (subsection 3.5.2) was repeated for the RTRL algorithm. The learning rate was chosen lr = 0.1 initially. The learning rate was steadily decreased by multiplying with a factor 0.995 each epoch. This learning rate decreasing schedule was to ensure fast training initially and to have smaller weight adaptations later on, when the finer details of the training data had to be learned.

The performance goal SSE = 0.1 was met at epoch 210, much faster than with epochwise BPTT. The epochwise RTRL algorithm was not tried because of long simulation running times.

3.6 Extensions of training algorithms

There are many interesting extensions that can be applied to the standard training algorithms. In this section some of these are presented. First it is shown (subsection 3.6.1) how the epochwise BPTT algorithm can be easily combined with training algorithms for static networks using the *virtual targets* description. In subsection 3.6.2 the technique of *teacher forcing* is introduced, which is a simple modification often used during recurrent network training.

3.6.1 Using virtual targets in the BPTT training algorithm

A useful property of the BPTT training algorithm is that it allows the calculation of *virtual targets* for certain neurons *k* in the network that do not have a real target $d_k(n)$. This will be shown first and then the application of virtual targets in a training procedure will be explained. The terms *virtual error* and *virtual target* can also be found in [Williams, 1995]. There, the terms are introduced for more insight into the BPTT algorithm.

Definitions of the modular neural network BPTT training algorithm

The equations of the (very general) modular network architecture together with BPTT joint training from section 3.8 are used. When the following variable is defined:

$$\varepsilon_{im}(n) = -\frac{\partial^+ E}{\partial y_{im}(n)} \tag{3.81}$$

and when equations 3.49 and 3.50 are combined, the weight update for weight *j* of a module *i* can be written as:

$$\Delta\theta_{ij} = -\eta \left[\sum_{n=n_0}^{n_1} \sum_{m=1}^{L_i} \frac{\partial^+ E}{\partial y_{im}(n)} \frac{\partial y_{im}(n)}{\partial \theta_{ij}} \right] = \eta \sum_{n=n_0}^{n_1} \sum_{m=1}^{L_i} \varepsilon_{im}(n) \frac{\partial y_{im}(n)}{\partial \theta_{ij}} \tag{3.82}$$

The term $\varepsilon_{im}(n)$ will be called the *virtual error* because it is an error term that in general is not equal to the instantaneous error on the outputs:

$$\varepsilon_{im}(n) \neq \frac{\partial E(n)}{\partial y_{im}(n)} \tag{3.83}$$

This follows from equation 3.51. A virtual error can exist even if the 'real' instantaneous error is zero. To show the relation between the BPTT training algorithm and the backpropagation training algorithm for static neural networks, the equations for the static neural network training gradient descent algorithm with backpropagation will be given first.

Definitions of the static neural network training algorithm

In this text it is assumed that static networks *i* inside a modular network are trained *as if* they are just separate static neural networks. So each static neural network is a module *i* of a modular neural network. For all weights θ_{ij} of this static network the weight update can be calculated, when a pattern p is presented to the network and gradient descent training with backpropagation training [Veelenturf, 1995] is used. Each pattern p is in this case a single vector and *not* a sequence (as in the recurrent network). The error measure for a static network is calculated per pattern so it is denoted E_p. The weight update after presenting pattern p can be calculated as follows:

$$\Delta\theta_{ij}(p) = -\eta \frac{\partial E_p}{\partial \theta_{ij}} = -\eta \left[\sum_{m=1}^{L} \frac{\partial E_p}{\partial y_{im}(p)} \frac{\partial y_{im}(p)}{\partial \theta_{ij}} \right] \tag{3.84}$$

when the error measure used is the SSE:

$$E_p = (d_{im}(p) - y_{im}(p))^2 = e_{im}^2(p) \tag{3.85}$$

where $y_{im}(p)$ are the network outputs, $d_{im}(p)$ the targets on the outputs, and $e_{im}(p)$ the network output errors, all for a certain pattern p. Now equation 3.84 can be written as:

$$\Delta\theta_{ij}(p) = -\eta \sum_{m=1}^{L} -(d_{im}(p) - y_{im}(p)) \frac{\partial y_{im}(p)}{\partial \theta_{ij}} = \eta \sum_{m=1}^{L} e_{im}(p) \frac{\partial y_{im}(p)}{\partial \theta_{ij}} \tag{3.86}$$

When batch-mode training is used [Veelenturf, 1995], the weight updates are applied only after all N_p patterns p have been presented. So this weight update is the sum of individual pattern updates:

$$\Delta\theta_{ij} = \sum_{p=1}^{N_p} \Delta\theta_{ij}(p) \tag{3.87}$$

Using virtual targets in the BPTT training algorithm

Now it will be shown that the following training procedures are equivalent:

- training the modular network with epochwise BPTT
- training each static module with standard gradient descent backpropagation ; using virtual errors instead of the instantaneous errors

The first procedure was given in section 3.5.1. The latter procedure is now explained. The values of the virtual errors $\varepsilon_{im}(n)$ are needed. These can be obtained by the BPTT algorithm by first simulating the network for the current sequence and second, calculating the virtual error terms (equation 3.81) by the BPTT backward pass of equation 3.51.

The inputs $u_{im}(.)$, $Y_{im}(.)$ and the outputs $y_{im}(.)$ of the modular recurrent network over time interval $[n_0,n_1]$ (the current sequence) are then converted to a set T containing $N_P = n_1-n_0+1$ training patterns as follows:

$$\begin{aligned} y_{im}(p) &= y_{im}(n') \\ u_{im}(p) &= u_{im}(n') \\ Y_{im}(p) &= Y_{im}(n') \\ e_{im}(p) &= \varepsilon_{im}(n') \end{aligned} \tag{3.88}$$

in other words, by taking p = n' with n' = n-n_0+1 and n=[n_0,n_1]. Now this set is used in a standard gradient descent training procedure for the static network modules, by using the set T and the static network training algorithm of equation 3.87. This yields:

$$\begin{aligned} \Delta\theta_{ij} &= \sum_{p=1}^{N_P} \Delta\theta_{ij}(p) = \sum_{p=1}^{N_P} \eta \sum_{m=1}^{L} e_{im}(p) \frac{\partial y_{im}(p)}{\partial \theta_{ij}} = \\ &= \eta \sum_{n=n_0}^{n_1} \sum_{m=1}^{L} \varepsilon_{im}(n) \frac{\partial y_{im}(n)}{\partial \theta_{ij}} \end{aligned} \tag{3.89}$$

The last expression is equal to the BPTT training equation 3.82. Therefore a standard gradient descent training procedure using the 'training' set T is equivalent to applying epochwise BPTT over epoch [n_0,n_1].

The set T can not reasonably be called a training set, because a training set only supplies inputs and targets. To achieve a proper training set, the set T is converted to a real training set T′ by using the following conversion instead of equations 3.88:

$$\begin{aligned} u_{im}(p) &= u_{im}(n') \\ Y_{im}(p) &= Y_{im}(n') \\ d_{im}(p) &= \varepsilon_{im}(n') + y_{im}(n') \end{aligned} \tag{3.90}$$

The targets $d_{im}(p)$ are calculated using the virtual error ε, so these are called 'virtual targets'. The set T′ provides the input patterns and targets for use in the static neural network training procedure (equation 3.87).

Applications of the virtual targets BPTT algorithm

The reason for calculating virtual targets, is that any existing neural network training algorithm for static neural networks can be invoked whenever input and target values are known. And they are known, if the above virtual targets procedure is followed. This way different existing algorithms for static neural networks can be re-used in combination with the BPTT algorithm.

The virtual targets algorithm was in this report only used to numerically verify the gradient calculation routines of the ModNet state-space neural network toolbox against existing Matlab routines for static networks (see appendix E).

Disadvantages

A disadvantage of the virtual targets procedure is that most neural network sums, output values and gradients are calculated twice: one time in the BPTT procedure for obtaining the virtual errors and the second time in the existing standard gradient descent training algorithm. The values calculated in the BPTT algorithm could of course be passed to the existing static gradient descent algorithm, but most of these algorithms only accept only inputs and targets and they calculate all other needed variables again.

But this disadvantage can be easily avoided by slight modifications of the existing algorithm code. This approach was used successfully for creating the Levenberg-Marquardt algorithm in the ModNet toolbox, see appendix E.

Another disadvantage is that virtual errors/targets are only locally valid. Locally valid means: valid only for weight vectors that lie in a small neighborhood around the weight vector θ_i. Here θ_i is the weight vector used when the virtual error was calculated (using equation 3.81).

If the weights change (this happens after a weight update) the virtual error will generally be different. Therefore virtual targets can not be used to repeatedly train network modules for

several epochs, because the thereby accumulated weight changes are too large for the virtual targets to remain valid.

3.6.2 Teacher forcing

The technique of *teacher forcing* has appeared several times in research on recurrent neural networks (according to [Williams e.a., 1995]). Teacher forcing is to replace, during training, the actual output $y_k(n)$ of a neuron by the teacher signal $d_k(n)$ which is the target for output k. One effect of forcing the output $y_k(n)$ to the target value, is that the situation of 'correct performance of the network on all earlier time steps' is artificially created, even if the network performance was in fact not correct on earlier time steps. This may lead to a better training performance on the current time step, so that in total a network can be trained quicker.

Note that this form of teacher forcing can only be applied to recurrent network outputs that have targets. Examples are the FRNN where teacher forcing can be applied to outputs 1…L only and the NARX network architectures where teacher forcing can be applied to all outputs. Teacher forcing can not be applied to state-space networks, because then none of the recurrent network outputs have targets. Teacher forcing is explained in much more detail in [Williams e.a., 1995] together with the modifications of training algorithms that are needed.

It should be noted that teacher forcing is a term used in neural network research whereas the same kind of setup is called *open loop* control (or -identification) in control theory.

Weighted teacher forcing

Weighted teacher forcing [Weigend, 1996] is a modified form of teacher forcing. The network outputs $y_k(n)$ are not fully replaced by the targets $d_k(n)$ but partially, depending on a variable λ:

$$y_{k,forced}(n) = \lambda \cdot d_k(n) + (1-\lambda) \cdot y_k(n) \tag{3.91}$$

The variable λ can be fixed or it can be varied (even during training) to emphasize learning of correct short-term behavior (for λ close to 1) or to emphasize learning of correct long-term behavior (for λ close to 0).

3.7 Other training algorithms

In this subsection some training algorithms will be mentioned that were not discussed previously in this chapter. References will be given to literature about these algorithms. Because the number of training algorithms to be found in literature is large, they can not all be discussed in this report.

3.7.1 Matlab/Elman backpropagation algorithm

A simple training algorithm called 'Elman backpropagation' is used in the Matlab neural network toolbox to train recurrent neural networks.

In the Elman algorithm the recurrent neural network is effectively considered to be a static neural network wherever possible (static but *not* unfolded in time). The inputs arriving from delay registers are simply considered external inputs. This is standard backpropagation widely used for static neural networks.

But not all recurrent networks can be trained with this approach. For example neurons that do not have targets defined (i.e. state neurons) cannot be trained using the standard backpropagation approach.

Elman backpropagation solves this problem by backpropagating the error from a neuron that does have a target, 'through' the delay registers, to the neuron that does not have a target. This process is visualized in figure 3.13. The four-neuron example FRNN is shown, but for visibility not all connections are shown except the connections through which the error of neuron 1, $e_1(n)$,

is backpropagated to neuron 3 that does not have a target defined. The backpropagation of the error is shown with five arrows.

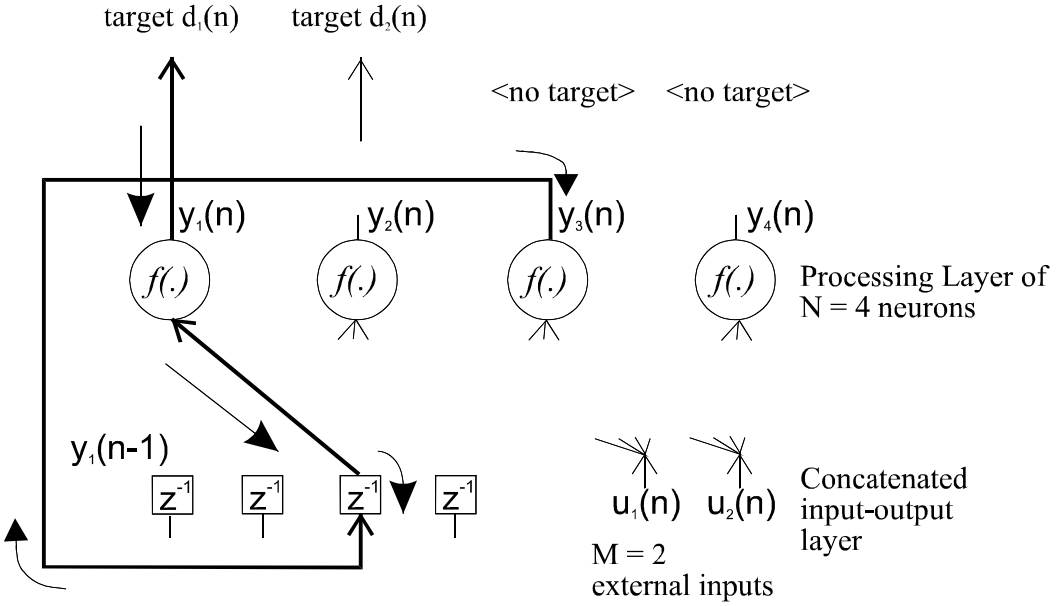


Figure 3.13: Backpropagation of error of neuron 1 (with a target) to neuron 3 (without target)

This algorithm is further described in Appendix B.3 for the case of a FRNN. It is shown there, the Elman algorithm can be described as a 'truncated' form of the epochwise RTRL algorithm for FRNN.

3.7.2 A hybrid BPTT/RTRL algorithm

An interesting BPTT/RTRL hybrid training algorithm is described in [Williams e.a., 1995] that is computationally efficient and combines some elements of both approaches. The hybrid algorithm consists of the following:

- a parameter *h* defines the mix between BPTT and RTRL
- the RTRL gradients are updated every *h* time steps (instead of each time step)
- this update is computed using computationally efficient truncated BPTT(*h*) over *h* time steps.

The benefits are:

- the number of (less efficient) RTRL computations is reduced, compared to standard RTRL
- the required computations does not grow with the time of network operation, as in standard BPTT
- no partitioning in epochs is needed, as in epochwise BPTT.

See [Williams e.a., 1995].

3.7.3 Kalman filter

In [Haykin, 1998] an unusual approach to training recurrent neural networks is presented. In this approach, Kalman filter theory is used to determine the weights of the network. First, the nonlinear neural network is linearized around the current weight vector. Now linear Kalman filter theory can be applied to this system, to provide optimal iterative estimates of the best weight vector.

Standard gradient descent does not use information obtained in the past to 'estimate' the next weight vector but only the last calculated gradient. Some properties of this Kalman filter approach to neural networks are:

1) the first-order linearization of the nonlinear network is done using the neural network gradients. This way, the Kalman filter algorithm can use any other gradient-based algorithm for its first calculation step.
2) it requires more computations than a normal gradient-based algorithm because the Kalman filter equations have to be calculated after the gradients [Haykin, 1998].
3) performance is reported to be better than normal gradient-based algorithms [Haykin, 1998].

BPS algorithm
Backpropagation for Sequences (BPS, see [Bengio, 1996]) is an example of an algorithm for SRN structures. It is a special case of the BPTT algorithm that only applies to the specific case of SRN with self-recurrent context layers. Because of this constraint the algorithm can however be computed quickly.

3.8 Conclusions

In this section some aspects of training algorithms are compared. As no benchmarking experiments (comparing performance on specific problems) were performed the conclusion on training algorithms will be based on theoretical grounds.

3.8.1 Comparing the BPTT and RTRL training algorithms

The fundamental difference between the BPTT and RTRL algorithms
The fundamental difference between the (epochwise) BPTT and RTRL algorithms can be best expressed in the following way. While the BPTT algorithm will try to perform the minimalization of the error over a sequence:

$$\min_{\mathbf{w}}\left[\sum_{n} E(n)\right] \tag{3.92}$$

the RTRL algorithm will try to perform the minimalization of individual error terms of a sequence:

$$\sum_{n} \min_{\mathbf{w}}\left[E(n)\right] \tag{3.93}$$

with the intention that 3.93 will approximate 3.92. Obviously the first expression is always less than or equal to the second expression. Epochwise BPTT is therefore expected to be at least as good or better in performance than epochwise RTRL.

Differences in resources required for FRNN
From the analysis of computational complexity of BPTT and RTRL performed in this chapter for FRNN, it follows that RTRL requires much more computations for 'large' N because the requirements scale up with N^4 as opposed to $N^2 * h$ for BPTT.

For highly parallel implementations of the algorithm, the number of required computations is much less an issue and RTRL may be a good choice then even for large N. For simulations on serial computers, BPTT quickly becomes the algorithm of choice as the following example will make clear.

Example
In a possible phoneme recognition problem, a speech signal short time spectrum is determined at 10 (ms) intervals and phoneme duration is less than 160 (ms)[1] for about 95% of phonemes. It follows $h = 0.160/10^{-3} = 16$ previous spectral samples matter in the classification of a phoneme. Therefore truncated BPTT(h) (h=16) can be used. This algorithm is quicker than RTRL for N $\geq$ 4 neurons. This number of neurons will be likely, because N=3 is not considered a very big network.

Differences in resources required for other networks
For the many other networks than FRNN, no exact resource calculations were done. The FRNN was taken as a 'typical' case because it can be often found in literature.

In [Baldi, 1995] the BPTT and RTRL algorithms are derived for a very general dynamic system description of neural networks. The computational requirements are also listed and turn out to be typically of order $O(N^2)$ for epochwise BPTT and $O(N^4)$ for RTRL in dense networks (i.e. fully connected), where N is the size of the state vector. These results are the same as for the FRNN.

Differences in application
As its name implies the RTRL algorithm is best suited for real-time applications where online training is needed. The reason for this is that the algorithm always adapts to minimize the error $E(n)$ on the latest sample that is available. This way, the system will always be adapted best to the latest data which is desired in a problem with non-stationary data sources. For this type of real-time applications a RTRL trained nonlinear neural network can provide a nonlinear alternative to a linear adaptive filter. Of course truncated BPTT and real-time BPTT can also serve for this purpose.

Epochwise BPTT is most suited for applications with offline training (for example speech recognition) when data is available offline, partitioned into separate epochs. The data source is now supposed to be stationary. (If the data source wouldn't be stationary, then collecting data for future uses would be of no use at all.)

RTRL adapts to the latest data
From the above remarks on (standard) RTRL it can be concluded that RTRL always adapts to the latest data available. During the first experiments with RTRL it was indeed found that neural network trained with RTRL performed well on the last part of a training sequence and already started to 'forget' the weight updates for the first part of a training sequence which was presented earlier. As a consequence performance on the first part of a sequence was worse than performance on the last part.

Epochwise BPTT is identical to epochwise RTRL for linear neural networks
It was experimentally found in subsection 3.4.4 that for the case of a linear neural network, the epochwise BPTT algorithm yields the same numerical results as the epochwise RTRL algorithm. It was already stated that the following relation must hold for linear networks:

$$\frac{\partial E(n_0, n)}{\partial w_{ij}} = \sum_{m=n_0}^{n} \frac{\partial E(m)}{\partial w_{ij}} \tag{3.94}$$

It would be interesting to find out analytically why this holds for linear networks, because it is not the case in general.

3.8.2 Matlab implementation of BPTT and RTRL algorithms for state-space networks

In chapter 2 it was concluded the state-space neural network architecture should be further investigated. The BPTT and RTRL training algorithms were derived for this network. It was

[1] Average phoneme length was measured over 30 sentences in the Timit database using the Timit Tools toolbox (see Appendix D.2)

noted that the state-space architecture describes several other network architectures when certain restrictions are imposed.

For these reasons the state-space network was chosen to be implemented in Matlab. Both the BPTT algorithm (epochwise) and the RTRL algorithm (continuous and epochwise) are implemented. It was not very difficult to add the BPTT algorithm, once the RTRL algorithm training functions were implemented.

All functions are in a toolbox which is called the *ModNet* toolbox (the name refers to modular network). Details about the implementation of the algorithms in the ModNet toolbox can be found in Appendix E.

The modular network description of the state-space network (see subsection 2.4.3) was used as a guideline behind the design of the ModNet code (e.g. in the function hierarchy and some variable names). These choices even allow the code to be straightforwardly extended, such that all architectures of the general modular network framework can be trained and simulated.

The extension to a general implementation of the modular network framework was however not opted for, because it would have been very slow when written in Matlab code. Instead the code was optimized for the state-space architecture alone. A general modular toolbox would be feasible if the code had been written in a faster language (like C). This was found by comparing the speed to that of the FIR toolbox written in C.

Also as an additional test a very small program was created that simulates a FRNN in C. It ran 1000 times faster than the corresponding Matlab code.

Previously it was assumed a general modular network toolbox would be very complicated to code in Matlab. As it seems now, such an implementation would be feasible in Matlab but the resulting code would be too slow to be of much use.

CHAPTER 4 PROPERTIES AND CLASSIFICATION CAPABILITIES OF RECURRENT NEURAL NETWORKS

Some general properties of recurrent networks are considered first in section 4.1. The classification capabilities of recurrent neural networks are examined in section 4.2. Section 4.3 looks at some of the choices that can be made in a classification system. The conclusions of this chapter will be summarized in section 4.4.

4.1 General properties

The issues that will be discussed:

- stability (4.1.1) ;
- the vanishing gradients effect in recurrent network training (4.1.2) ;
- controllability and observability (4.1.3) ;
- approximation properties of state-space networks (4.1.4) ;
- approximation properties of the FRNN (4.1.5) ;
- relation between recurrent networks and Turing machines (4.1.6);
- state-space model versus input-output model (4.1.7).

4.1.1 Stability of recurrent neural networks

As recurrent networks are dynamic systems, there is the chance of instability of these systems. Some remarks are made here on the stability of neural networks and references are given to appropriate literature.

BIBO and BIBS stability for linear state-space networks

The issue of stability is first examined for the case of a linear state-space network architecture. When the system output **y** remains bounded for all bounded inputs **u** the system is said to be Bounded Input Bounded Output (BIBO) stable. When the state **x** is bounded for every bounded input **u** the system is Bounded Input Bounded State (BIBS) stable.

Linear state-space neural networks obey the following general linear system equations:

$$\begin{aligned} \mathbf{x}_L(n+1) &= \mathbf{F}(\mathbf{x}_L(n),\mathbf{u}_L(n)) = \mathbf{A}_L\mathbf{x}_L(n) + \mathbf{B}_L\mathbf{u}_L(n) \\ \mathbf{y}_L(n) &= \mathbf{G}(\mathbf{x}_L(n),\mathbf{u}_L(n)) = \mathbf{C}_L\mathbf{x}_L(n) + \mathbf{D}_L\mathbf{u}_L(n) \end{aligned} \qquad (4.1\ a,b)$$

Here **G(x,u)** is bounded only when both inputs **x** and **u** are bounded. We assumed the input **u** is bounded so the state **x** remains a possible cause of instability. To guarantee BIBO stability of the linear system a sufficient condition is that the system is BIBS stable. In this case the state **x** is bounded (i.e. **F**(**x**,**u**) is bounded).

A sufficient condition for BIBS and thus BIBO stability for linear systems is that all eigenvalues λ_i of the matrix A have $|\lambda_i| < 1$ [Kwakernaak e.a., 1991]. This can be easily assessed.

BIBO and BIBS stability for nonlinear state-space networks

Most neural networks used in applications are nonlinear. In these cases BIBO/BIBS stability is not an issue because in any practical neural network there is always a layer of neurons that has bounded nonlinear transfer functions in both the **F** and **G** neural networks. This guarantees boundedness of **F** and **G** for all input.

All configurations for which **G**(**x**,**u**) is a bounded function of the inputs are always BIBO stable, because the output is bounded for *any* input. This category includes all neural network architectures with at least one layer of neurons having bounded neuron activation functions inside the **G** function. Note that all common nonlinear neuron activation functions are bounded.

All configurations for which **F**(**x**,**u**) is a bounded function of the inputs are always BIBS stable, because the state is then bounded for *any* input. Again, all neural network architectures with at least one layer of neurons having bounded neuron activation functions in the **F** function are BIBS stable.

So BIBO and BIBS stability in a nonlinear network can be easily guaranteed. The problem is that other types of instability (typical for nonlinear systems) can still occur. An example of instability is a system that starts an endless oscillation that does not stop, whatever input is applied.

Stability of nonlinear networks

For nonlinear networks, other definitions of stability are needed, which assess the local stability of the autonomous dynamic system around an equilibrium state. The equilibrium state can be *uniformly stable* (also called: *stable*), *convergent, asymptotically stable* (both stable and convergent), *globally asymptotically stable, unstable* (if it is not stable). The exact definitions can be found in [Haykin, 1998].

The stability of an equilibrium state can be proven for all linear systems and some nonlinear systems. Mathematical tools for this are [Haykin, 1998]:

- *linearization of the nonlinear system*. A nonlinear system can be linearized, then the stability can be assessed by calculating the **A** matrix of the linearized system (the Jacobian). The linearization is only valid locally, so only *local stability* can be assessed.
- *The first and second methods of Lyapunov.* These methods can prove global stability of a dynamic system in some cases.

How to ensure recurrent network stability

The above methods can be used to assess the stability of recurrent neural networks [Haykin, 1998]. Lyapunov is used in [Lewis e.a., 1999] to guarantee stable neural network systems which are always based on the input-output system model. The linearization approach is applied to a state-space network in [Zammareno e.a., 1998] but only proves local stability.

For dynamic Hopfield neural networks, application of the Lyapunov methods is known to be possible when the weight matrix is chosen to be symmetric thereby ensuring global stability of the network [Hertz e.a., 1991]. This symmetry condition is a severe constraint on the neural network structure.

So methods that guarantee global stability of the network require extensive modifications or restrictions on the network structure.

Stability in neural networks for classification

The stability requirement is of vital importance when neural networks are used in dynamic system control problems. In classification applications however, stability issues are often ignored because it is very unlikely that an unstable neural network is doing a good classification job. Certainly, during training each iteration produces a slightly different neural network and it seems reasonable that many unstable networks are among those. But since the training algorithm will seek a network that is adequately classifying, networks that become unstable given the training data will not likely be the final result of training.

However, there is the danger that a network is stable given the training data, but becomes unstable when new input data is supplied. This possibility does not seem to be considered in literature on classification experiments using recurrent networks. It is also not further investigated in this report because this danger was recognized at a late stage of the project.

Therefore, more research on recurrent network stability may be needed.

4.1.2 Vanishing gradients problem in recurrent network training

In training recurrent neural networks a problem called the *vanishing gradients problem* can occur. It is cited as an important restriction on training of recurrent networks [Bengio, 1993, 1994, 1996] [Haykin, 1998]. The problem occurs in training tasks where a recurrent neural

network should be trained with backpropagation (e.g. BPTT) to produce a desired response at the current time, depending on input data that was applied in the distant past.

In these kind of tasks a recurrent network should be able to store information for extended periods of time without losing it. This is defined in [Bengio, 1994] as *robust latching* of information in the presence of noise on the network inputs.

It is shown by Bengio that in such a setting, either:

- the network does not robustly latch information in presence of noise on the input ; or
- the network is unable to learn long-term dependencies (i.e. relations between current targets and distant past inputs)

Measures against the vanishing gradients problem

In [Lin e.a., 1998] it is shown the problems regarding the learning of long-time dependencies with recurrent networks can be alleviated by introducing recurrent neural networks that have a higher order of embedded memory. A simple example is adding delay elements to a network that delay by 10 time steps (with the original network having only single time step delays). The same solution was noted in [Hertz e.a., 1991].

The use of linear feedback neurons [Mozer, 1995] may be another way to overcome the vanishing gradients problem. More possible solutions, that make changes to the training strategy rather than the network structure, are given in [Haykin, 1998] and [Bengio, 1993, 1994].

The approaches mentioned were not further investigated.

Conclusions

In this report not much attention is paid to the vanishing gradients problem (only references to relevant literature are given). It was not thoroughly studied, because the type of neural network learning task that requires the latching of information over long (or even arbitrary) time spans is mostly found in the field of Finite State Machine (FSM) simulation and identification.

In a task like speech recognition the time span of input-data/target dependencies is relatively short.

4.1.3 Controllability and Observability

The definitions of *controllability* and *observability* of a dynamic system can be found in most books on dynamic systems e.g. [Lewis e.a., 1999] and also in [Haykin, 1998]. For this reason, this subject will not be elaborated here. These concepts are also not used further in this report.

4.1.4 Approximation properties of the state-space network model

The state-space neural network can approximate any state-space system

The following theorem on the function approximation properties of static MLP will be used.

> *Universal Approximation Theorem* [Veelenturf, 1995]
> A two-layer MLP with sigmoid transfer function for the neurons in the first (hidden) layer and one linear output neuron in the second layer can approximate any continuous function f: $R^n \rightarrow R$ in any domain with any given accuracy.

This theorem can be extended to vector functions f: $R^n \rightarrow R^m$ by using m linear output neurons and a 'larger number' of hidden neurons.

Now these two-layer perceptrons are used to realize the functions **F**(.) and **G**(.) in the general state-space network architecture. Now it can then be concluded that any state-space system can in principle be approximated arbitrarily well by a state-space neural network architecture with two-layer perceptrons to compute the functions **F** and **G**.

4.1.5 Approximation properties of the FRNN

The FRNN is not capable of simulating all state-space systems

It will be shown in this subsection that the FRNN is not capable of simulating all state-space systems. The type 1 FRNN written as a state-space model has the output equation 2.15b which is repeated here:

$$\mathbf{y}_S(n) = \mathbf{G}_S(\mathbf{x}_S(n), \mathbf{u}_S(n)) = \mathbf{C} \cdot \mathbf{F}(\mathbf{W}_x \cdot \mathbf{x}_S(n) + \mathbf{W}_u \cdot \mathbf{u}_S(n)) \quad (4.2)$$

with C given in equation 2.16. The above equation shows the output function $\mathbf{G}_S$ is computed by a single layer of neurons.

An important fact that is known for single-layer neural networks is that they are not able to approximate any continuous function as the two-layer neural network can (see subsection 4.1.4).

Proof

This inability is often demonstrated using the so-called XOR problem in which a neural network has to learn the (logical, or binary) XOR function. It turns out a single layer network can not realize this function.

The XOR function is not a continuous function but a binary function. Since continuous functions were assumed for F and G in all state-space models a continuous function is needed as a counterexample. So a similar 'continuous XOR' function will be used here:

$$y_{XOR}(x_1, x_2) = x_1(1 - x_2) + x_2(1 - x_1) = x_1 + x_2 - 2x_1x_2 \quad (4.3)$$

that has identical behavior to the binary XOR function for inputs $(x_1, x_2) = (0,0)$, (0,1), (1,0), (1,1) but is continuous.

Now the dynamic neural network learning task consists of learning the following function of input u(n):

$$y_{XOR}\{u(n), u(n-1)\} = u(n)\{1 - u(n-1)\} + u(n-1)\{1 - u(n)\} \quad (4.4)$$

We first assume there is a FRNN and a fixed set of weights $\mathbf{W}_x$, $\mathbf{W}_u$ such that the function y_{XOR} is realized. This leads to a contradiction as will be shown later.

The output is a scalar so a single output neuron is taken. This realizes the function:

$$y(n) = f(\mathbf{W}_x^{\mathbf{row1}} \cdot \mathbf{x}_S(n) + W_u^{row1} \cdot u(n)) \quad (4.5)$$

with scalar $\mathbf{u}_S(n)$=u(n) , the scalar $W_u' = W_u^{row1}$ is the first row of $\mathbf{W}_u$ and $\mathbf{W}_x^{\mathbf{row1}}$ the first row of $\mathbf{W}_x$. For $\mathbf{C}$ in equation 4.2 the vector $\mathbf{C}$ = [1 0 … 0] is taken to create a single output system. The transfer function f(.) can be a linear or (logistic or tangential) sigmoid function, all of which are monotonically increasing functions ($f'(x) > 0$).

All dynamic behavior that is needed for this task is to somehow 'remember' the input u(n-1) in the state vector $\mathbf{x}_S(n)$. So we suppose this information is extracted out of the state in the following way:

$$\mathbf{W}_x^{\mathbf{row1}} \cdot \mathbf{x}_S(n) = W_x' \cdot u(n-1) \quad (4.6)$$

so the output neuron function becomes:

$$y(n) = f(W_u' \cdot u(n) + W_x' \cdot u(n-1)) \quad (4.7)$$

Requirements

We see that for a correct realization of the task 4.4 the following four requirements should be met:

1. $f(0) = 0$ is required for $y_{XOR}(0,0)=0$
2. $f(W_u' + W_x') = 0$ is required for $y_{XOR}(1,1)=0$
3. $f(W_x') = 1$ is required for $y_{XOR}(0,1)=1$
4. $f(W_u') = 1$ is required for $y_{XOR}(1,0)=1$

Result 1

The equalities 3,4 combined with $f(.)$ monotonically increasing gives $W_u' = W_x'$. Equalities 1,3,4 together with $f(.)$ monotonically increasing gives $W_u', W_x' > 0$.

Result 2

Equations 1,2 combined with $f(.)$ monotonically increasing gives $W_u' = W_x' = 0$. This contradicts result 1.

Conclusion

So there exists no set of weights such that the FRNN realizes the function y_{XOR}. Therefore, the FRNN is not able to simulate all state-space systems. It was shown here for the type 1 FRNN. For type 2 a similar proof can be constructed.

4.1.6 Recurrent neural networks and Turing machines

An issue that appears a lot in literature (so it should be an important or interesting topic) is the relation between recurrent neural networks and the concept of the Turing machine. There are theorems that describe this relation. They generally state the following:

> All Turing machines may be simulated by recurrent neural networks with sigmoid neuron activation functions.

Specifically this can be stated for the FRNN, NARX and state-space networks. See [Haykin, 1998] for the above results.

This research is of use when recurrent networks are used for Finite State Machine simulation tasks. In this report these results are not used further.

4.1.7 State-space model versus input-output model

System identification and control

In [Wentink, 1996] the state-space model is favored over the input-output model at first (for nonlinear system identification) because it is more general. But in case the target values for the neural network realizing the function **F**(.) in the state-space equations 2.1 are unknown (that is, when desired values for the state variables are not known), this approach is quickly dismissed. The same argumentation is used in [Haykin, 1998]. The same argumentation is implicitly present in [Lewis e.a., 1999], because the state-space model is only used there in case the state of the dynamic system is fully measurable and target states are known.

From chapter 3 on training algorithms it follows however that these unknown target values can be estimated by backpropagating the error from the network outputs. Whether such a procedure will result in a state-space description that is better than an input-output description, is not known in general and will surely depend on the application. Nevertheless, the state-space approach should not be too quickly dismissed for the reason that target states are unknown. There may be other good reasons of course to dismiss state-space networks, for example in control applications, only networks are used that are guaranteed to be stable [Lewis e.a., 1999] because stability is one of the main requirements in controller design, for any control system used in practice.

An example dynamic system is shown in [Wentink, 1996] that can only be fully described by a state-space system. The input-output model is not able to describe the system (in fact, two alternating input-output models are needed to describe the system). Using an input-output neural

network to identify this system seems a problem. In the experiments by Wentink the performance of a single input-output neural network on the task is however still acceptable.

Classification
For classification tasks the dynamic systems perspective is often not used, so other criteria guide the selection of an architecture (e.g. experimental performance). [Santini, 1995b] uses a state-space description for sequence classification neural networks that is discussed in subsection 4.2.2.

4.2 Maximum A-Posteriori classification capabilities of recurrent neural networks

In this subsection it is discussed if recurrent networks are capable of Maximum A-Posteriori (MAP) classification. First the concept of MAP classification is introduced for static neural networks (subsection 4.2.1). Then MAP classification of sequences is looked at in the context of recurrent neural networks (in subsection 4.2.2).

4.2.1 Maximum A-Posteriori (MAP) classification

Definitions
The concept of the MAP classifier is only briefly introduced here because it is thoroughly treated in many textbooks. See [Bishop, 1995] and [van der Heijden, 1995] for more information on probability and classification concepts.

A key assumption for a classifier network is that it should not just learn the training set, but should be able to *generalize* the classification. This means the classifier also performs well for new data which was not in the training set. It is stressed in [Bishop, 1995] that the goal in network training then becomes to model the *underlying generator* that produced the training data. When the classification problem can be described within a probabilistic framework, the underlying generator of the data can be modeled by the joint probability $p(\omega_i, \mathbf{u})$. This can be written as the product of a *class-conditional probability* and a *prior probability*:

$$p(\omega_i, \mathbf{u}) = p(\mathbf{u} \mid \omega_i) p(\omega_i) \tag{4.8}$$

where $\omega_i \in \Omega$ are the K classes and $\mathbf{u}$ is the measurement vector. By using Bayes' theorem

$$p(\omega_i \mid \mathbf{u}) = \frac{p(\mathbf{u} \mid \omega_i) p(\omega_i)}{p(\mathbf{u})} \tag{4.9}$$

the *posterior probability* $p(\omega_i \mid \mathbf{u})$ can be expressed as a function of the prior and the class-conditional probability.

The MAP classifier is defined as the Bayesian classifier (that minimizes the risk) using an uniform cost function. It can be derived [van der Heijden, 1995] that the expression for the MAP classifier then becomes:

$$\hat{\omega}(\mathbf{x}) = \arg\max_{\omega_i \in \Omega} \{p(\omega_i \mid \mathbf{u})\} = \arg\max_{\omega_i \in \Omega} \{p(\mathbf{u} \mid \omega_i) p(\omega_i)\} \tag{4.10}$$

This classifier selects the class i that maximizes the posterior probability $p(\omega_i \mid \mathbf{u})$. In case the prior probabilities are unknown they are chosen equal ($p(\omega_i)=1/K$) and the *maximum likelihood* classifier results:

$$\hat{\omega}(\mathbf{u}) = \arg\max_{\omega_i \in \Omega} \{p(\mathbf{u} \mid \omega_i)\} \tag{4.11}$$

MAP classification in static neural networks
In [Bishop, 1995] it is shown a static neural network with K outputs, trained with input vectors $\mathbf{u}$ and appropriate 0/1 targets that encode the class i of the data $\mathbf{u}$, can learn to approximate the

function $p(\omega_i|\mathbf{u})$ which underlies the training data. The neural network can then be used for classification, so this neural network is a MAP classifier of the data $\mathbf{u}$.

MAP classification of sequences

The definition of MAP classification of sequences is now given. In this case of sequence classification, not a single measurement vector $\mathbf{u}(n)$ is available but instead $\mathbf{u}(n)$ and a sequence of k past measurements $\mathbf{u}(n),\ldots,\mathbf{u}(n\text{-}k)$ at current time n. These measurements are all assumed to belong to a single class ω_i. The posterior probability of this sequence of class ω_i is

$$p(\omega_i|\mathbf{u}(n),...,\mathbf{u}(n-k)) = p(\omega_i|\mathbf{U}_k(n)) \tag{4.12}$$

where the $\mathbf{U}_k(n)$ is defined as a single vector that holds the sequence. From the static MAP classification case the MAP classifier for sequences follows directly:

$$\hat{\omega}(\mathbf{u}) = \arg\max_{\omega_i \in \Omega}\{p(\omega_i \mid \mathbf{U}_k(n))\} = \arg\max_{\omega_i \in \Omega}\{p(\omega_i \mid \mathbf{u}(n),...,\mathbf{u}(n-k))\} \tag{4.13}$$

Such a classifier, that uses an appropriate sequence of measurements to classify, is often seen as the goal of a sequence classification system with recurrent neural networks.

The difference with a static MAP classification system is that subsequent measurements are not assumed to be independent (as is the assumption in a static classifier), on the contrary, they are highly correlated because the entire sequence is of one and the same class.

Therefore a classifier that looks at the entire sequence can possibly outperform one that looks only at individual samples.

Variable sequence length

An interesting property of recurrent networks is that they can be used to classify sequences of *variable* lengths. In other words, the variable k in the above equation does not have to be chosen beforehand but may vary depending on the data or the optimal value of k can be learned from the training data by the neural network.

If a static network or time-delay network is used for the same classification task, the variable k would have to be explicitly chosen by the designer beforehand.

4.2.2 MAP Classification of sequences in theory by recurrent neural networks

MAP classification with the BPTT algorithm

Using just the fact that the BPTT algorithm will minimize the error over an entire training sequence $E(n_0,n)$ it may be possible to prove that a BPTT trained network eventually becomes a MAP classifier of sequences, under certain conditions.

This idea emerged at a late stage of this project, so it could not be further investigated.

Proof of MAP classification capability

A proof given in [Santini e.a., 1995a] shows that recurrent neural networks (in fact, a state-space neural network) is capable of MAP classification of sequences. This proof is analyzed in Appendix C.1. The conclusion is that this proof is not of much value in practice.

MAP Classification by iterative updating of an estimate

In [Petridis e.a., 1996] and again in [Plataniotis e.a., 1997] it is shown how MAP Classification of a sequence can be computed iteratively using the *partition algorithm*. Previous MAP estimates are kept and used together with new input data to do a new (and better) MAP estimate. After k input samples have been processed, the result of the algorithm is the MAP estimate of the entire sequence.

For recurrent neural networks, it is also often argued that they can perform classification of a sequence by accumulating evidence (of the presumed class of the input sequence) over time. This strongly resembles the formation of an estimate, which is iteratively improved as more input data becomes available, as in the partition algorithm.

So it is possible that a recurrent neural network may learn to operate in a way that resembles the partition algorithm.

A disadvantage of the partition algorithm is that it requires for each class an *optimal estimator* for given data. This forces the classification system to be a classification-by-prediction system (see subsection 4.3.1 for this subject), which is not the case in general for a classification system.

Other theoretical results on MAP classification of sequences

In literature not much theoretical results apart from the partition algorithm were found on sequence classification. Most current texts on classification treat the case of 'static' classification, that is with the assumption of uncorrelated data patterns. This choice is made in order to obtain useful mathematical results. Sequence classification appears to be difficult subject that is more than a simple extension of existing theorems.

4.3 Choices in a recurrent neural network classification system

Several choices have to be made in a classification system using recurrent neural networks. First classification by prediction is mentioned (subsection 4.3.1), a very different approach to sequence classification. Next several options for modularity of a classification system are looked at (subsection 4.3.2). The choice of target values for neural network outputs is introduced in subsection 4.3.3. Several types of targets can be used.

4.3.1 Classification by prediction

A classification system can be built by using several prediction modules. For each class, such a predictor is trained. A predictor module is any system, for example a neural network, that is trained to predict the signal of a specific class. The prediction is one-sample-ahead prediction of the data sequence based on knowledge or a model of the specific class that the predictor is associated with. For each class k there is a predictor that tries to predict the signal for the class:

$$\hat{u}_k(n+1) = f_k(u(0),...,u(n);\omega = \omega_k) \tag{4.14}$$

During operation a decision module monitors the performance of all prediction modules. The estimated class is the one for which the prediction network makes the best predictions. Such a system is used in [Petridis, 1996], [Plataniotis e.a., 1997] and [Janssen, 1998].

This approach is rather different than training neural networks with targets that directly encode class information. A problem is that not all signals are predictable, for example some noise-like phonemes [Janssen, 1998].

Discriminant versus non-discriminant training

A notable difference is that classification by prediction implies *non-discriminant* training, whereas training with class targets is *discriminant*. While training predictor module k, only example sequences of one class ω_k are shown to the predictor and it learns to predict data for that class. But a predictor module is not explicitly trained to reject data of another class (i.e. to make bad predictions for data of another class). In other words, it cannot discriminate between classes and will just attempt to predict as good as possible. In training neural networks with class targets, both examples of one class ω_k and counter-examples (of all other classes) are shown to the network. The network can explicitly learn to recognize a certain class and reject other classes, so the network learns to be discriminant. In [Bengio, 1996] the importance of discriminant training is emphasized, although not in the context of classification-by-prediction systems.

Classification by prediction is not used in experiments in this report because it is a rather unconventional way of classifying.

4.3.2 Modularity of the classification system

A classification system can be made using just one neural network, or several in a modular classification system. The assumption from now on is the common choice, that an N-class classification system has N outputs, one for each class, in total. Each output is a measure of the probability that the current data belongs to the associated class. An exception is a two-class classification task that can be handle by just one output (N=1).
The two extreme cases for such a N-class classification system are:

- N neural network modules, one for each class (figure 4.1a)
- a single neural network with N class outputs (figure 4.1b)

Intermediate cases are also possible, for example:

- $N_M=N/N_O$ neural network modules, each capable of classifying N_O classes (figure 4.1c)
- 5 neural network modules, four N/6-class modules and one N/3-class module.

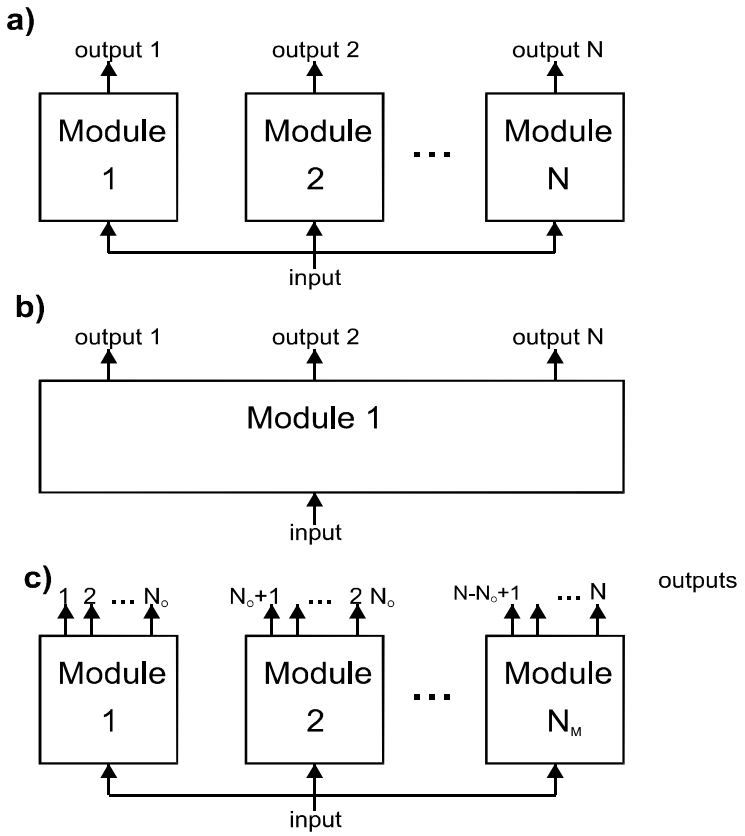


Figure 4.1; Three different modular classification systems. a) N modules with 1 output ; b) Single module with N class outputs ; c) N_M modules each with N_O outputs

The described modular structures all have no hierarchy (i.e. at most one module is present between input and any of the outputs). Hierarchical systems are possible, see 'ensembles' at the end of this subsection.

Small number of modules versus large number
For the following discussion, two types of systems are considered. The first type is highly modular (like in figure 4.1a) and has a large number of 'small' modules. The second type has a small number of 'large' modules.
First type
Classification systems with a large number of small modules have the advantage that the correct behavior for a single module is learned easier and quicker because the module is small. This is

the case because small modules have much less complex structures (with fewer degrees of freedom). This is known as the *divide and conquer* principle. A difficult problem is being split up into several more manageable sub-problems.

Second type

Classification systems with a smaller number of large modules have the advantage of *shared knowledge*. For example, suppose that a large module with a group of neurons within it, has learned to calculate a really useful 'high-level' feature of the input data. This feature that is important for correct classification can then be put to use in the calculations for several class probabilities within the module because the network module has several outputs. Effectively, network resources can be shared, resulting in less weights for the total system. In a highly modular system (the first type) each module would have to 'find out' this useful feature extraction function for itself so it would be duplicated many times in each module.

Obviously, the disadvantages of the two systems are the opposites: the lack of shared knowledge (for the first type) and the complexity of the learning task (for the second type). It depends on the application what kind of modularity gives best performance.

It is even possible to create a system that shares advantages of both approaches: special 'feature extraction' modules can be introduced whose output goes to all classification modules. This way knowledge is shared, while a highly modular architecture can still be maintained. This approach is mentioned in [Bengio, 1996] ('modularization') in the context of a speech recognition application. It requires however more application of prior phonetic knowledge than the other approaches mentioned.

Ensembles of neural networks

The benefits of modular networks are used in many neural network architectures. These architectures are often called *ensembles* of neural networks or neural network *committees* (see [Haykin, 1998] chapter 7). The corresponding training algorithms often do automatic construction and simultaneous training of an ensemble. See for example [Fritsch, 1998] for a hierarchical neural network ensemble approach applied to speech recognition. A hierarchical network growing approach called *Neural Tree Networks* is presented in [Patterson, 1996].

The simplest ensemble/committee method for neural network classifiers is described in [Bishop, 1995]. The committee consists simply of N identical neural network structures (but each trained with different initial random weights) whose class predictions are averaged. It is proven that classification error will not increase and may be decreased by selecting a higher N. Because of its simplicity this method can be readily tested.

Describing ensemble methods further is outside the scope of this report although the approach seems appealing to handle complex problems.

4.3.3 Selection of targets

Usually a neural network system for classification has N outputs, each for one of the N classes. During training, targets can be selected in a number of ways.

One-zero targets

Targets can be chosen as follows: during training an output *k* has a target one if a data sequence belongs to that class ($\omega=\omega_k$) and a target of zero otherwise. These are called one-zero targets or binary targets.

Double threshold targets

Double threshold targets [Veelenturf, 1995] for outputs are a region close to one, $d(n) \geq (1-\alpha)$, if a data sequence belongs to that class and a region close to zero, $d(n) \leq \alpha$, otherwise. Here α is chosen small. These targets are 'easier to reach' by the neuron outputs because these usually have a limited output range <0,1>. By using targets just inside this interval, the output neurons are not driven into their saturation region (where the output function derivative is close to zero) during training.

Although this will not be explicitly mentioned, the double threshold targets can be combined with any of the following methods of target selection.

Smooth targets

Suppose a data sequence is available for training that consists of class 1 data for the left half. The next half is data of class 2 (see figure 4.2a). The one-zero targets $d_1(n)$ and $d_2(n)$ for the two network outputs are given in figure 4.2b. One can argue the abrupt step in the targets is a difficult function to learn for the neural network with its smooth nonlinear functions. Or, in other words: the training algorithm will put a lot of 'effort' into approximating the step function even though that is absolutely not required for obtaining a good classifier. To solve this problem a smooth target can be used (see figure 4.2c) [Kasper e.a., 1994]. In this example, it is a linear interpolation between the desired targets. Other functions could be used.

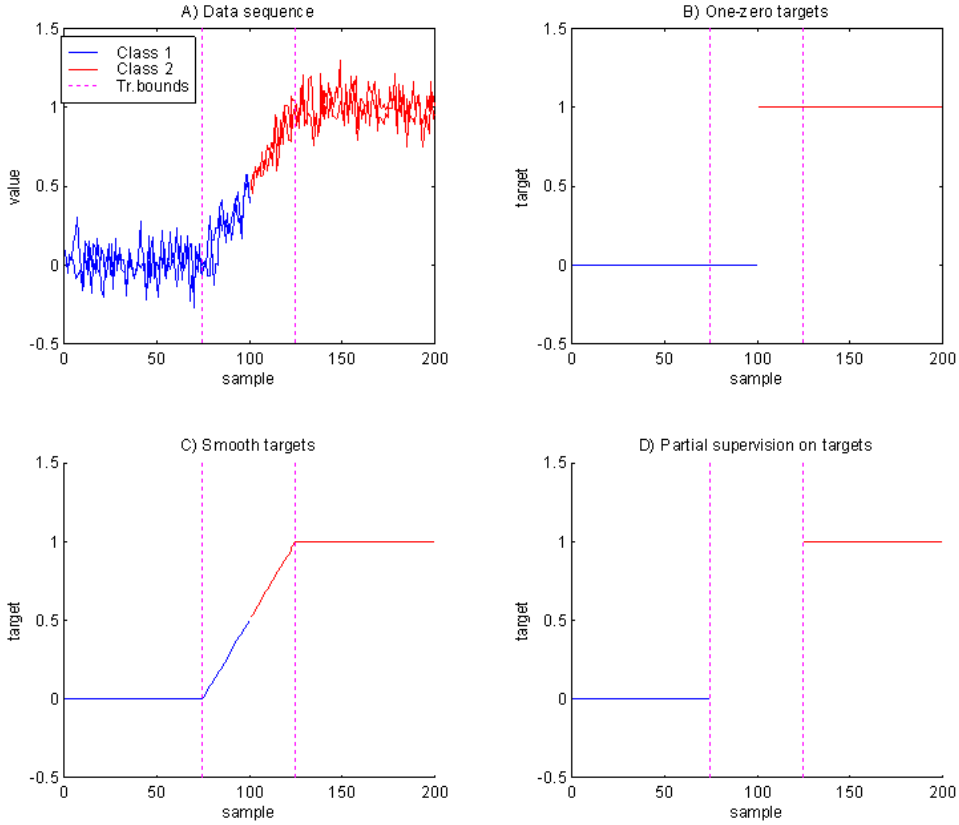


Figure 4.2; a) data sequence with ; b) one-zero targets and ; c) smooth targets; d) partial supervision

Partial supervision: supervision does not have to be everywhere

Another solution to the just-mentioned problem of abrupt steps in the targets is to remove supervision in certain intervals. In figure 4.2d this is shown. It is assumed that a transition from one class to another cannot be defined exactly (as is often the case in speech recognition), so a target is not given in this transition-interval. This procedure emphasizes that detecting the exact class-boundary is not the goal, but rather detecting if a certain class 'occurred' in the sequence (as is the case in speech recognition).

This procedure of partial supervision (see also [Bengio, 1996]) can easily be included in the training algorithms given in this report by setting the instantaneous error $e(n) = 0$ for all times n where no supervision is desired.

End-of-sequence supervision

Yet another option for targets based on partial supervision is only using supervision at the end of a sequence [Bengio e.a., 1993]. This kind of target selection can be used in sequence classification tasks. What it means is, that most of the time no targets are given (no supervision) but only at the end of a sequence the class target is given.

This approach seems logical, because only at the end of it can the full sequence be classified. The network is trained to first process the whole sequence and then classify it. So a wrong classification halfway the sequence is not 'punished', as long as the classification at the end of a sequence is still correct.

Weighted supervision

In the ModNet toolbox implementation, partial supervision is generalized to *weighted supervision* which means an error weighting coefficient $\alpha_i(n)$ can be given to each network output at each time instant. The coefficient can be close to 1 to stress the relative importance of the target, or close to zero if the current target is not very important. Values in between represent intermediate cases.

As special cases of weighted supervision are included

- normal supervision: if all supervision coefficients are one, then supervision is everywhere
- partial supervision: when the error weighting coefficients are sometimes zero and sometimes one. Zero values correspond to no supervision at that time/output.

Delayed targets

A commonly applied procedure in recurrent network classification tasks is to *delay* the target by 1, 2, or more time steps. Effectively, a delayed target allows the use of 'future' input data by the network. This is the reason that targets are delayed in some neural classification systems.

In speech recognition for example, input data u(n+1) does contain information about the phoneme class at time n (which is coded into t(n)).

Suppose a target t(n) is delayed by amount d to a new target sequence t′(n):

$$t'(n) = t(n-d) \tag{4.15}$$

The network uses u(n_0),…u(n) to calculate y(n) which should approximate the target t′(n). So u(n_0),…u(n) can be used by the network to approximate t′(n) = t(n-d). It can be seen that 'future' inputs u(n-d+1),…u(n) can be used by the network to approximate t(n-d).

A good example is the type 2 FRNN, where the minimum delay of a path between an input and an output is one. The target should then be delayed by d = 1 if there is a relation to be learned between input u(n) and target t(n).

4.4 Conclusions

Problems in using recurrent networks: stability and vanishing gradients problem

It was found that stability of recurrent networks is not guaranteed without extensive measures (in fact global stability can only be guaranteed by making very specific architectural choices). In spite of the danger of instability of recurrent networks, the topic of stability is usually not considered for recurrent networks used in classification tasks. Further research on this is recommended.

The vanishing gradients problem is an important problem in training recurrent networks. However this issue is mostly considered for Finite State Machine learning tasks where information has to be retained by the neural network over arbitrary large intervals. So for the more local task of phoneme classification this issue was not considered.

Approximation properties

It was shown that the FRNN architecture can not simulate all state-space systems because it has only one layer. The general state-space network architecture, that allows more layers, is a better choice.

State-space model versus input-output model

The conclusion on comparing the state-space model with the input-output model is that the latter is often chosen because a state-space model is considered not trainable. This report shows it can however be done. Which model is best, will depend on the task at hand. For classification purposes often the model is chosen that performs best at the task.

Classification capabilities of recurrent networks

It turned out difficult to actually prove that a recurrent network can do Maximum A-Posteriori sequence classification. (A proof was given but this one is of no practical value.)

A big advantage of recurrent networks over static networks is their ability to process (and classify) sequences of variable length.

Modularity of the classification system

There are several choices for modularity of a classification system. It can be concluded that one big neural network could in principle handle any classification task but in practice the training of such a network is increasingly difficult. To obtain 'trainable' networks a modular approach or ensemble approach is advised. An optimal trade-off in modularity can not be given beforehand, it should be obtained by combining experiments with the specific task at hand and a-priori knowledge about the task.

Selection of targets for classification

A number of options for target selection were considered. Especially the use of *partial supervision* is promising because it can describe very well the uncertainty in phoneme sound classification (i.e. the lack of an exactly defined separation between two phoneme sounds).

A generalized form of partial supervision was introduced in this chapter, weighted supervision, which is used in the ModNet toolbox (see subsection 3.8.2).

CHAPTER 5 PHONEME RECOGNITION EXPERIMENTS

5.1 Introduction

In this chapter several neural network structures and algorithms are applied to a classification problem: phoneme recognition.

The phoneme recognition experiment is first carried out with artificially created phoneme sounds to make it easier to spot possible errors in the system and gain experience with a classification task that is much easier than classifying real human speech. These experiments will be referred to as the ASP (Artificial SPeech) experiments.

The artificial phoneme sounds are first introduced in section 2.3. To classify phonemes characteristic features have to be extracted out of the phoneme sounds. The feature extraction process (or preprocessing) is treated in section 2.2.1. A postprocessor is part of most neural network classification systems. Some simple postprocessors are introduced in section 2.4.1. The procedure that was followed in the ASP experiments is explained in section 3.6.1. Results for the different classification methods are listed in section 2.2.4. Some preliminary experiments were done on real speech sounds in section 3.8.1. Conclusions about the experiments follow in section 3.1.3.

5.2 Phoneme data

The classification task was to classify a signal that contains artificial phoneme sounds. These phonemes can belong to one of three distinct classes. The artificial phoneme classes were inspired on spectrograms of the following real phonemes[2] (all vowels): Phoneme 1 is /ay/ as in the word 'bye' ; phoneme 2 is /ey/ as in 'bay' ; phoneme 3 is /iy/ as in 'few'.

These phonemes are all *non-stationary* or *transitional* phonemes. This means that the frequency content of the sound is not fixed but changes over time according to a general pattern.

Table 3.1 shows the frequency contents of the three artificial phonemes. Each phoneme consists of three sine waves (f1, f2 and f3) having varying frequency over time. The index t_{rel} is the relative time index of the phoneme with range [0,1]. A relative time index is defined because phonemes may have different lengths.

The table is interpreted as follows: for each relative time listed, the frequency values listed (for f1, f2 ,f3) are approximately the frequencies that will be present in the artificial phoneme at that moment. A blank entry means that no frequency is defined and the actual frequency of the sine wave will approximately lie between the preceding value and the next value.

The frequency values stated are however not completely fixed, because each generated phoneme would then be equal to all other generated phonemes. Small random deviations in the frequency values are therefore introduced. The values for the deviations are drawn from a normal random distribution (μ=0, σ=20).

Three spectrograms for three realizations of artificial phonemes (one realization per class) are shown in figure 2.8. The changes over time of the frequency components specified in the table can be recognized in the spectrograms. The characteristic 'bending' shape of the spectral components over time is a result of an interpolation process: the given frequency points are fitted with a cubic spline interpolation (with the default spline functions as provided by Matlab).

[2] found on the CSLU website (http://cslu.cse.ogi.edu) or [Rabiner e.a., 1993]

Phoneme 1	t_{rel}				
freq. comp.	**0**	**0.45**	**0.5**	**0.6**	**1.0**
f1	1000			1200	1200
f2	2200		2400		2700
f3	4400	4400			4400

Phoneme 2	t_{rel}				
freq. comp.	**0**	**0.45**	**0.5**	**0.6**	**1.0**
f1	1000				1000
f2	2900				3100
f3	4100		4300		4400

Phoneme 3	t_{rel}				
freq. comp.	**0**	**0.45**	**0.5**	**0.6**	**1.0**
f1	1000			1200	1200
f2	2700		2400		2200
f3	4400	4400			4400

Table 5.1: Frequency contents of three artificial phoneme classes (frequencies in Hertz)

This process leads to different phonemes for each class that 'look like' each other but are different for each realization.

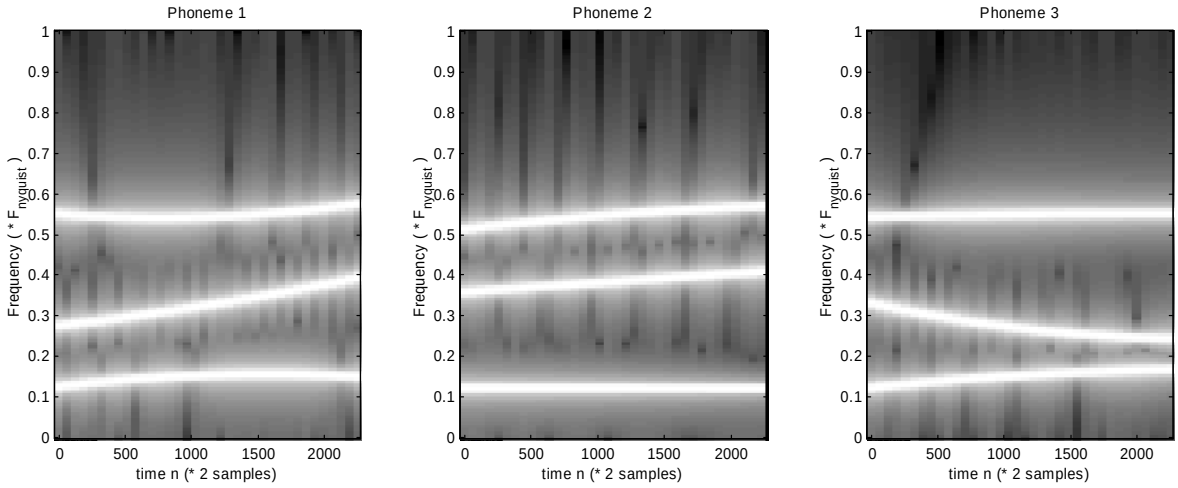


Figure 5.1: Logarithmic spectrograms of three artificial phonemes (one of each class). The presence of a signal is shown in white.

Further information

The duration of each phoneme was set $T_{phone} = 0.3$ (s) and sampling frequency $F_S = 16$(kHz) was used.

Phoneme class 2 is the most static one (the smallest maximum change in frequency). The left parts of phoneme 2 and 3 may cause confusion to a classifier because the frequency content is similar. The same holds for the right parts of phoneme 1 and 2 and the middle parts of all three phoneme classes.

5.3 Feature extraction

Before a speech signal can be classified, the sampled speech first has to be processed to extract characteristics of the speech that are important in the recognition process. Such a procedure is called *feature extraction* or *pre-processing*. In most cases the feature extraction process also significantly reduces the amount of data.

Many different feature extraction procedures have been used in speech recognition. During years of research, the *cepstrum* feature extraction method turned out to be one of the best performing methods [Rabiner e.a., 1993].

More recent research on feature extraction points to other methods that yield comparable or better performance in speech recognition. However, the cepstrum is still a standard that has not yet been replaced. Therefore a feature extraction procedure based on this method is used and it is described in this section.

First the enframing of a speech signal is described (subsection 2.2.2), a procedure that yields speech fragments that are suitable for further processing. In subsection 4.1.7 the basic cepstrum feature extraction method is introduced. In subsection 2.2.3 some more refinements of the cepstrum method are described. Cepstrum values are normalized which is discussed in subsection 2.4.2. Finally in subsection 5.3.5 the feature extractor is described that will be used in the artificial speech recognition experiments in this chapter.

In this chapter references will be made to VoiceBox [Brookes, 1998]. This Matlab toolbox will be used in the experiments to perform part of the feature extraction process that is described in this chapter.

Note that far more information than is presented here can be found in the literature (references are made) and this text is meant to summarize the choices made for the feature extraction process in the ASP experiments.

5.3.1 Enframing of a speech signal

In most feature extraction methods a spectrum of the sampled speech signal is calculated at some stage, for example by using the Discrete Fourier Transform (DFT). Using the spectrum, the frequency components present in a speech signal can be obtained. Because the DFT does not provide information on the temporal location of frequency components, it is not sensible to use it right away on a complete speech signal, a sentence for example.

To obtain temporal information the DFT analysis should be performed on small fragments of speech. The fragments are taken small enough to be considered quasi-stationary. The fragmenting procedure is often called *enframing* or *blocking* of the speech signal and can be done in a number of ways. See [Rabiner e.a., 1993] for more information.

The enframing procedure chosen for the ASP experiments takes frames of 16(ms) of speech (N samples) that overlap both the previous and the next frame by N/2 samples (8(ms)), a default choice as given by VoiceBox. Overlapping frames are used to obtain a smoother spectral representation of the speech. The enframing procedure used is shown in figure 2.9.

Interestingly, the choice of 16(ms) frames (N=256 at F_S=16(kHz)) was also found to be the best performing frame length in the phoneme recognition experiments in [ten Hove, 1996].

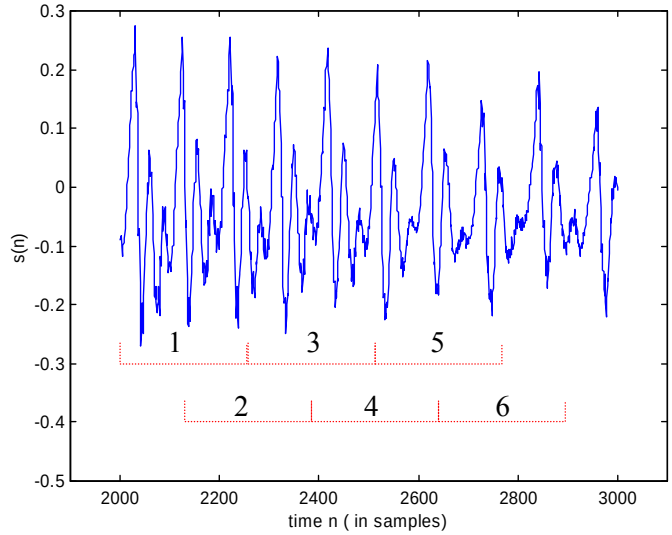


Figure 5.2: Enframing a speech signal s(n) with N/2 overlapping frames (first 6 frames shown below the signal)

5.3.2 The cepstrum

The cepstrum will be introduced here briefly. This text is a summary of the information that can be found in other books and reports [Rabiner e.a., 1993], [Nakagawa, 1995], [ten Hove, 1996], [Lokerse, 1995].

The name 'cepstrum' is derived from 'spectrum'. The cepstrum consists of individual coefficients that are called the *cepstral* coefficients. The cepstrum is calculated from a speech signal fragment s(n) using the procedure shown in figure 2.2.

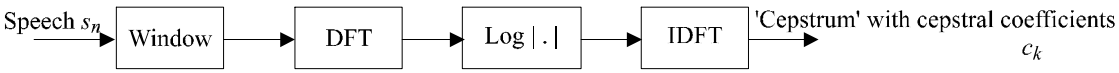


Figure 5.3: The procedure of the cepstrum calculation

These steps are now described in more detail.

Step 1: Windowing

The *windowing* procedure simply multiplies the speech fragment s(n) by a suitably chosen window function w(n):

$$s_w(n) = s(n) \cdot w(n) \tag{5.1}$$

to suppress boundary effects in the Discrete Fourier Transform (DFT) procedure that follows next.

Step 2: Logarithmic magnitude DFT

The DFT is performed to obtain the spectrum of the speech fragment:

$$S(k) = \sum_{n=0}^{N-1} w(n)s(n)e^{-j\left(\frac{2\pi nk}{N}\right)} \tag{5.2}$$

The logarithm of S(k) is then taken to account for the logarithmic nature of human loudness perception. The logarithmic magnitude spectrum Y(k) is obtained:

$$Y(k) = \log|S(k)| \tag{5.3}$$

Step 3: IDFT
The inverse DFT (IDFT) of Y(k) is calculated to obtain the cepstral coefficients (the cepstrum). The cepstrum can be seen as the spectrum of the logarithmic speech spectrum. To see why this is done, a model of human speech production is first introduced.

The source-filter model for speech production
The production of speech can be captured in a model called the source-filter model [Rabiner e.a., 1993] [ten Hove, 1996]. The model is shown in figure 2.3. The first block in this model is the source, that models the sound generated by air flowing from the lungs through the vocal chords. The second block models the passing of the sound through the vocal tract, that filters the sound. The vocal tract filter strongly influences the actual phoneme sound produced

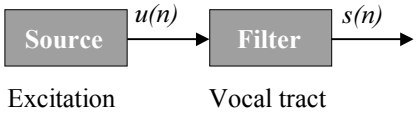


Figure 5.4: The source-filter model for speech production

Since the vocal tract cannot change shape very quickly the filter transfer function is varying relatively slowly. The speech wave can be expressed as a convolution of the filter impulse response h(n) with the source signal u(n):

$$s(n) = h(n) * u(n) \tag{5.4}$$

Meaning of the cepstrum
To show what meaning should be attributed to the cepstrum, the steps of the cepstrum processing are performed on the speech s(n) from equation 2.1. The DFT operation F(.) on the speech yields:

$$F(s(n)) = F(h(n) * u(n)) = F(h(n)) * F(u(n)) \tag{5.5}$$

The log magnitude operation yields:

$$\log | F(s(n)) | = \log | F(h(n)) * F(u(n)) | = \log | F(h(n)) | + \log | F(u(n)) | \tag{5.6}$$

We see the log spectra of the relatively broadband vocal tract filter h(n) and of the relatively narrowband source signal u(n) are added in the above equation. Because the IDFT of this quantity is the cepstrum and the IDFT is a linear operation, the speech cepstrum is the filter cepstrum added to the source cepstrum. It is explained in [Nakagawa e.a., 1995] that the source cepstrum can be removed by applying *cepstrum windowing*, which is in fact using a limited number of lower cepstral coefficients and discarding the remaining higher coefficients. This leaves only a cepstral representation of the vocal tract filter, which is related to the actual phoneme produced by the speaker, which is what we would like to know.

Spectral envelope
The cepstrum can be converted back to the log spectrum by applying the DFT to cancel the last IDFT operation. By applying cepstrum windowing first an approximation of the log spectrum y(n) called the *spectral envelope* is obtained. By discarding more and more of the higher cepstrum coefficients a progressively 'coarser' approximation of the log spectrum is obtained when the remaining cepstral coefficients are transformed back with the DFT. See [ten Hove, 1996] where different spectral envelopes are graphically compared, for several cepstral window sizes.

Using the Discrete Cosine Transform (DCT) in the cepstrum calculation
In the VoiceBox software a Discrete Cosine Transform (DCT) is used instead of the IDFT in the last step of the cepstrum calculation. Essentially this leads to the same numerical result (when applied on the real logarithmic speech spectrum) as the IDFT. The cepstral coefficients calculated by the DCT were found to only differ by a scaling factor, so the choice of either transform will have no influence on recognition performance. The mathematics behind the DCT approach was therefore not further investigated.

Insensitivity of the cepstrum to pitch of speech
Since the cepstrum is the spectrum of the log-magnitude speech spectrum, the 0^{th} cepstral coefficient represents the offset (DC value) of the log-magnitude speech spectrum. This 0^{th} coefficient is often not used in a subsequent speech recognition system and thus the information on the global pitch of a fragment of speech is discarded. This can be safely done because the pitch does not aid phoneme recognition in a speaker independent recognition setting.

However, the other cepstral coefficients do give information on absolute frequency *differences*. It is possible that using only relative frequency differences gives a better set of features for speech recognition, than using these absolute frequency differences. This would however require a different method of feature extraction so it is not further investigated here.

5.3.3 Extensions of the cepstrum

The mel-cepstrum
Studies have shown the human ear is more sensitive and accurate at lower frequencies of the hearing range. A procedure called Melscaling transforms the linear frequency axis of a speech signal to the nonlinear Melscale [Rabiner e.a., 1993] which corresponds to the scale of human sound perception.
The mel transformation is given by Fant's equation [ten Hove, 1996]:

$$TM = (\frac{1000}{\log 2}) \cdot \log(f/1000 + 1)\,[\text{Mel}] \qquad (5.7)$$

with f the linear frequency scale in (Hertz). Details on the practical implementation of the Melscaling transform can be found in [Rabiner e.a., 1993]. Often a non-uniformly scaled (melscale spaced) filterbank is used to reduce a complete logarithmic spectrum of a speech fragment to a smaller number of coefficients and at the same time perform the melscale transformation. The cepstrum of melscaled speech is called the *mel-cepstrum*.

The filterbank approach
A lot can be said about the use of *filterbanks* in the feature extraction process. More information can be found in [Rabiner e.a., 1993]. The filterbank approach can be used in a stand-alone manner to extract features, or it can be combined with the mel-cepstrum feature extraction procedure. The latter is done in VoiceBox.

Here, only an example of a triangular melscaled filterbank will be given. See figure 2.7 for a graphical representation of the filterbank with $N_{FB} = 5$ separate banks.

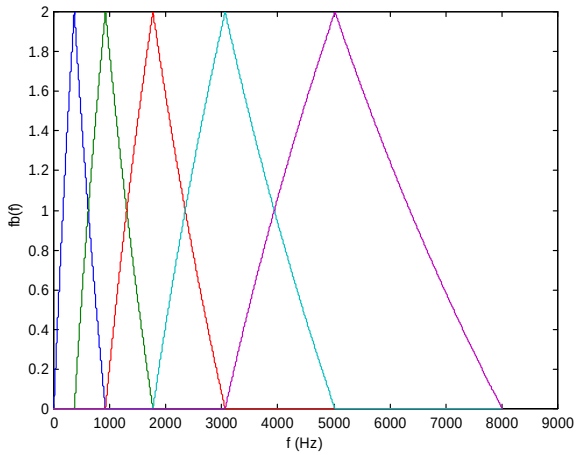


Figure 5.5: Graphical representation of a triangular melscaled filterbank

Each one of the five triangles is the transfer function $F_{FB}(f)$ of a filter. Together these five filters span the whole frequency range from 0 (Hz) up to the Nyquist frequency $F_S/2$ = 8000 (Hz) (sampling frequency F_S = 16 (kHz)).

After passing the full speech spectrum through each filter, the magnitude of the filter output is calculated. For each filter, these magnitude values are the only information that is kept. Therefore the full speech spectrum is reduced to just 5 coefficients for the example filterbank given above. This new spectral description is a crude approximation of the original spectrum in which spectral details are lost, but the global structure of the spectrum can be preserved if the number of filters in the filterbank is high enough.

The logarithmic melscaling operation (of equation 2.9) is visible in the width of the filters: these widths are not equal, but the filter width increases with higher frequency. The space between filter center frequencies also increases with higher frequency. As a consequence the filterbank coefficients are more 'sensitive' in the lower frequency range, just as the human ear is. This kind of setup, with varying filter width or center-to-center frequency distance, is called a non-uniform filterbank.

The filterbank method resembles a common model of the processing of sound by the human ear. This model of human hearing assumes the existence of a 'bank' of frequency-sensitive sound receptors in the ear. Frequencies that are very close trigger the same sound receptor so these sounds are perceived as having identical frequencies.

In VoiceBox, the filterbank method is used to filter the magnitude spectra of speech fragments. The default setting is a triangular melscaled filterbank with $N_{FB} = 29$ filters.

Psycho-acoustic processing

In [Foks, 1997] the effect of further psycho-acoustic processing of the speech data is measured. The argument used here is that sounds that are inaudible for the human ear, are best removed before attempting speech recognition because these sounds are effectively noise. A small performance improvement in speech recognition is reported, when using psycho-acoustic processing.

Interestingly, the implementation of a basic psycho-acoustic processor is very simple because existing MPEG encoder/decoder software (for example the mpeg layer 3 format) can be used. This allows for a quick experimental verification of the effect of psycho-acoustic processing.

Delta and delta-delta cepstrum features

It is very common in practical speech recognizers that the basic (mel-)cepstrum feature set is extended with the so-called *delta* and *delta-delta* cepstral coefficients [Rabiner e.a., 1993]. The delta coefficients represent the change of the cepstral coefficients over time, simply by calculating an approximation to the derivative (which is the difference or delta of the coefficients over time). The 'time' axis is in this case a discrete number that point to the successive frames (see subsection 2.2.2 about frames) of speech. In a similar way, the delta-delta coefficients represent the change of the delta coefficients so it can be viewed as the 'acceleration' of the cepstral coefficients over time.

These coefficients describe the transitional nature of speech sound, so delta and delta-delta features may improve classification of transitional phonemes.

In practice the delta coefficients, when calculated by the difference of the cepstrum coefficients between successive frames, are very 'noisy' so often a different computation is used. Rabiner suggests an estimation of the derivative which is a first-order polynomial fit (a least-squares estimate) over a finite length interval which can be described by the following equation:

$$\Delta c_m(n) = \mu \sum_{k=-K}^{K} k \cdot c_m(n+k) \tag{5.8}$$

Where $c_m(n)$, $\Delta c_m(n)$ are the m^{th} cepstral coefficient at times n and the estimated delta coefficient, respectively. μ is a normalization coefficient. The finite interval over which the estimate is calculated has length (2K+1). A value of K = 3 is suggested by Rabiner. VoiceBox was checked and was found to use exactly the same method with K = 4. To obtain the delta-delta coefficients the same fitting procedure can be applied again but this time on the delta coefficients. This is done in the VoiceBox software with K=1.

The Wepstrum
Instead of using Fourier transforms, various wavelet transforms can be incorporated in the cepstrum approach. This approach is described in [ten Hove, 1996] where the method has been named *wepstrum*. A wavelet approach seems very useful because time and frequency resolutions are not fixed as in the Fourier transform, but the resolution trade-off can be varied among different frequency ranges. The wepstrum outperformed the standard mel-cepstrum in a preliminary experiment [ten Hove, 1996].

Other extensions
A practical speech recognition system will have many more extensions to the feature extraction process. What additional processing is used, mostly depends on operation environment of the system. Examples are:

- different feature extraction procedures for different categories of phoneme sounds
- background noise reduction (e.g. speech recognition in a noisy environment)
- transmission channel noise and distortion reduction (e.g. speech over a telephone line)
- adapting the feature extraction mechanism or classification system to the current speaker

On these topics, relevant literature exists. These issues are not addressed in this report.

5.3.4 Normalization of features (to zero mean and unity variance)

It is a very common procedure that input features are *normalized* before using them in a neural network based classification procedure. Without normalizing it is possible that different coefficients in a feature vector have a widely varying magnitude of values. The differences can sometimes be several orders of magnitude.

This poses several problems, for example in neural network training, because some neural network weights should be initialized with larger values (to handle input coefficients with small magnitude) and other weights should be initialized with small values (to handle inputs with large magnitude).

It is easier to normalize the feature set itself in advance, so that all coefficients have a magnitude of the order unity. Then, all neural network weights can be safely initialized with values of order unity. The normalization procedure used in the ASP experiments is the following linear transformation of the coefficients $x_i(n)$:

$$x_i^{norm}(n) = \frac{x_i(n) - \overline{x_i}}{\sigma_i} \tag{5.9}$$

where $\mathbf{x}(n)$ are the feature vectors (patterns) $n = 1 \ldots N$ and the mean and standard deviation over the coefficients are given by:

$$\begin{aligned} \overline{x_i}(n) &= \frac{1}{N}\sum_{n=1}^{N} x_i(n) \\ \sigma_i^2 &= \frac{1}{N-1}\sum_{n=1}^{N}(x_i(n) - \overline{x_i})^2 \end{aligned} \tag{5.10}$$

The result is a new set of normalized patterns $x_i^{norm}(n)$ with for each coefficient zero mean and unity variance over the whole feature set.

This process is described in more detail in [Bishop, 1995]. In this normalization transform each coefficient is treated as independent from other coefficients. There are other approaches that also take into account correlations between coefficients, for example *whitening* of input vectors [Bishop, 1995] [van der Heijden, 1995].

Normalizing new features
The linear transformation of equation 2.25 is used as a part of the preprocessor, to process all new speech features that become available when using the speech recognition system in

practice. The new features will be scaled into approximately the same range as the original feature set.

5.3.5 Description of the mel-cepstrum feature extractor

The mel-cepstrum feature extractor from the VoiceBox software will be used in the ASP experiments. The complete procedure is summarized here.

The mel-cepstrum feature extraction procedure

- *Enframing the speech signal:* Frames of N=256 samples, 16(ms) at F_S = 16(kHz), are taken from the speech signal. The frames overlap by N/2 samples, 8(ms).
- *Windowing of frames:* A 256-point Hamming window is applied to each frame.
- *absolute DFT:* to obtain the magnitude spectrum of each frame
- *Filtering each spectrum*: A melscaled triangular filterbank with 29 filters is used to obtain a smoother spectrum (described by 29 coefficients) and at the same time transform the linear frequency scale to the melscale.
- *logarithm*: the melscaled filtered magnitude spectrum is converted to a logarithmic spectrum.
- *IDFT*: the logarithmic magnitude spectrum is converted to 29 cepstral coefficients using the IDFT.
- *cepstrum windowing*: the 0^{th} cepstral coefficient is discarded, then the following N_C cepstral coefficients are kept and the rest is discarded. The (VoiceBox default) value N_C=12 is taken so 29-12 = 17 coefficients are discarded.
- *normalization:* normalization by a linear transformation is used as described in subsection 2.4.2.

All steps except normalization are performed in the *melcepst* function in VoiceBox. Note that delta and delta-delta cepstral coefficients are not included in the feature set in the ASP experiments, only 12 cepstral coefficients.

5.4 Postprocessing

The term postprocessing is used in this report for any further processing that may be performed on the output of the neural networks in the classification system. At this stage the local classification of speech frame(s) is already done but the postprocessor can do more, for example:

- combine several phoneme classifications to one word classification
- reject certain classified frames that seem improbable, compared to classification results of nearby frames.

The final classification decision is always based on the output of the postprocessor. Some options for the postprocessor are discussed now.

No postprocessing

If no postprocessor is used, the classification decision is directly based on neural network outputs.

Simple moving average postprocessor

A simple postprocessor is the smoothing operation (implemented as a moving average filter). It calculates the mean of a number of (2K+1) neural network classification results (each result obtained by classifying a single frame) to obtain a final classification. This procedure can be described by:

$$y_i^{final}(n) = \frac{1}{2K+1} \sum_{k=-K}^{K} y_i(n+k) \qquad (5.11)$$

where $y_i(n)$ are the neural network outputs at time/frame n.

This kind of filter can improve classification performance because spurious misclassifications of a single frame or a few frames are removed. The use of this filter effectively implements the prior knowledge that the minimum phoneme duration is about M = (2K+1) frames.

Advanced postprocessors

In a complete phoneme-based speech recognition system the ultimate goal is not to detect only phonemes but complete words and sentences. In this case the pre-processor together with the phoneme classification system is only the front end of the total system. The postprocessor (or the *backend* of the recognition system) then includes language models (e.g. a pronunciation dictionary and grammar rules), models to detect common mistakes made by the phoneme classifier and perhaps more. These models extract the most probable sentence out of a sequence of detected phoneme sounds.

To implement a complete recognition system is not within the scope of this report. The phoneme recognition front-end can be evaluated without a backend speech recognition. In this case the postprocessor can be kept very simple: none, or the moving average filter.

The most used postprocessor is however a bit more complicated. It is the phoneme Insertion/Deletion/Substitution error calculation method as used in [Robinson, 1994].

5.5 Procedure

In this section the procedure for recognition experiments with artificial phonemes will be described. A number of different neural networks and one other method will be used as a classifier on the same classification task.

The following steps were taken in the ASP experiments. For each classification method:

1. a classifier is selected (subsection 3.6.2)
2. the classifier is initialized (subsection 3.1.2)
3. the classifier is trained with a training set of 15 phonemes, five of each class (subsection 3.1.1)
4. during training, a validation set of 3 phonemes is used, one of each class (subsection 2)
5. the training process is repeated several times for the same classifier (subsection 2.2)
6. when training is done, the performance of the classifier on an independent test set of 18 phonemes (6 of each class) is measured (subsection 2.2.5).

These steps will be clarified now in subsequent subsections.

5.5.1 Selection of a classifier (1)

The selection of a classifier starts with the selection of the method (or neural network type) to use for the task. To be able to compare the performance of the recurrent neural networks described in this report to other methods, several methods were used.

After the method selection, there remain more choices: for all neural network methods in this report, the network structure must be chosen by the user before training. This choice is often made by trial and error and maybe some insight or experience with similar classification tasks. There are neural network approaches that modify the network structure as part of the training process (i.e. the neural network is ‘grown’ or ‘downsized’ as prescribed by an algorithm) so arbitrarily choosing a structure is avoided. As these structure adaptation approaches were not investigated, a choice has to be made by hand.

In all experiments the trial and error approach was used to be able to compare the performance of a large number of different structures. The differences in final performance may reveal what type of structure is best for this classification task.

Methods of classification
The methods used are listed in Table 5.2.

a	b	c	d	e	f
Method No.	**Method**	**Number of layers**	**Toolbox / m-file**	**Training algorithm**	**Initiali -zation**
1	Multilayer Perceptron (MLP, static neural network)	2	NNet	LM	NW
2	State-space neural network	2 (in both modules)	ModNet	BPTT-LM	NW
3	FIR neural network (category: time delay neural network)	2	FIR	Temporal Backprop.	random
4	K-Nearest Neighbor classification (k-NN)	-	nnk_classifier.m	*(no training needed)*	
5	Fully Recurrent Neural Network (FRNN)	1 (by definition)	ModNet	BPTT-LM	NW
6	Multilayer Perceptron (MLP)	3	NNet	LM	NW
7	State-space neural network	2 (in both modules)	NNet	Matlab/ Elman	NW

Table 5.2: Classification methods

The methods (column b) are numbered (column a) for reference purposes. All these methods except the K-Nearest Neighbor classifier are neural networks. The number of layers (column c) of the neural network is chosen beforehand for each method.
All methods except the K-Nearest Neighbor classifier need training so they use a training algorithm (column e). The Matlab implementation of the algorithms (both training and simulation) is in a toolbox or m-file (column d). More information about these toolboxes can be found in the references [Matlab, 1997[S]] [Dijk, 1999[S]] [Janssen, 1998[S]] and appendices. The initialization method (column f) is further explained in subsection 3.1.2.

Choices in network structure for each method
For each method the architectural choices are different. These choices are summarized in table 3.3. For each method (column a) the parameters listed in column b have to be chosen. The meaning of the parameters is given in column c. The values that have been tried for each parameter (in all possible combinations) are in column d. The total number of different structures caused by trying parameter combinations is given in column e.

Repeated training is used (see subsection 2.2): the number of repeated runs with the same structure is in column f. The total number of training runs (which is the total number of structures times the repeat value) is listed in column g.
Occasionally the experiments were time-consuming and the last few runs were aborted (in methods 2 and 5). For method 2 not all structures were tried, just 32 of the total of 48.

a	b	c	d	e	f	g
Method No.	**Structural parameters**	**meaning**	**values for parameters**	**total number of structures**	**repeat training**	**total number of training runs**
1	N_1	The number of hidden neurons (in layer 1).	1…25	25	10	250
2	N_{1M1}	The number of hidden neurons (in layer 1 of module 1).	2,3	48 (only 32 tried)	10	320 (only 316 run)
	N_{1M2}	The number of hidden neurons (in layer 1 of module 2).	3…8			
	N_S	The size of the state vector (is by definition equal to the number of output neurons in layer 2 of module 1)	1…4			
3	N_1	The number of hidden neurons (in layer 1).	2,…,7	150	10	1500
	T_1	The number of delay taps for the layer 1 inputs.	2,…,6			
	T_2	The number of delay taps for the layer 2 inputs.	2,…,6			
4	k	The number of Nearest Neighbors	1, 3, 5, 7, …, 35	18	-	18
5	N	The number of neurons (minimum 3: external outputs, one for each class)	3,4,5,6	4	10	40 (only 38 run)
6	N_1	The number of hidden neurons in layer 1	2,…,10	81	10	810
	N_2	The number of hidden neurons in layer 2	2,…,10			
7	N_{1M1}	*same as method 2*	2	4	10	40
	N_{1M2}		6,7			
	N_S		2,3			

Table 5.3: Listing of structural parameters for each method, values tried, and training runs

Choice of neural network modularity

For the ASP experiments the number of phoneme classes is three. The following configurations of the neural network in the classification system can be used. Each configuration has a total of three outputs, one for each class:

- single neural network with 3 outputs
- 2 neural networks, network A has 1 output and network B has 2 outputs
- 3 neural networks with each 1 output

For a further discussion on modularity, see subsection 4.3.2. Because the ASP classification task has relatively low complexity, the single neural network configuration is chosen for the ASP experiments. The required number of output neurons per network is therefore three.

A motivation for each method will be given now.

Methods 1 and 6: MLP (two-layer and three-layer)
A 2-layer Perceptron can approximate any function (subsection 4.1.4) with sigmoid transfer functions for the hidden neurons and linear output neurons. For the current classification task however, only output values of 0 to 1 are required. Therefore the linear output neurons can be replaced by sigmoid neurons that have a 0 to 1 range.

This choice is also motivated by the results in [Bishop, 1995] subsection 3.1.3 on logistic discrimination. These state, that a sigmoid activation output function allows the outputs of the network to be interpreted as posterior probabilities in classification. To be fully correct, actually a *softmax* function should be used (see Bishop, or subsection 3.1.3). This will not be done because:

- the softmax function is not fully supported by the Matlab NNet toolbox and not supported by the FIR toolbox.
- interpreting outputs as probabilities is not really needed in the experiments. Only the maximum value of outputs is evaluated (see subsection 2.2.5) so the number itself is not used.

The number of hidden neurons is varied from very small (1) to large (25).

A 3-layer Perceptron is also used, to see if any performance increase results from adding an extra layer.

Methods 2 and 7: state-space neural network
In the state-space network module 1 computes the state function and should be able to approximate any function. This requires linear output neurons and sigmoid hidden neurons. Module 2 gives the final classification output so for the reasons given above sigmoid output and hidden neurons are chosen.

Small values were chosen for the parameters, in order to limit network complexity and simulation time. Method 2 uses BPTT-LM training and method 7 the Matlab Elman training algorithm.

Method 3: FIR
The FIR neural network toolbox [Janssen, 1998[S]] is used. Because the FIR toolbox implementation is fast (C code) a large number of structures could be tried. For the same reasons as in method 1, sigmoid output neurons and hidden neurons are chosen. The hidden neurons are tangential sigmoid (tansig) neurons.

Method 4: k-NN
The k-Nearest Neighbor (k-NN) classifier [van der Heijden, 1995] is a non-parametric classification method that directly compares new input data to all training data stored in memory. No training is needed. The Euclidean distance measure is used to compare feature vectors. An odd number k is required for k-NN. Usually k is not very large (<20) for fast classification, so k higher than 35 has not been tested. The classifier has been implemented in Matlab in *nnk_classifier.m*.

Method 5: FRNN
The FRNN requires only the choice of number of neurons and transfer functions. Because again sigmoid outputs were desired, all neurons were chosen sigmoid neurons. Because the first 3 neurons functioned as output neurons (one for each class), N=3 neurons is the minimum number in the network.

5.5.2 Initialization of a classifier (2)

Weight initialization
Neural network parameters (weights) are usually initialized with small random values before training. This procedure is used for method 3 because it is default in the FIR neural network toolbox [Janssen, 1998[S]].

The default initialization used in Matlab is the semi-random Nguyen-Widrow (NW) initialization method. The advantages over purely random weights and biases are according to the Matlab documentation [Matlab, 1997^S]:

- Few neurons are wasted (since all the neurons are in the input space).
- Training works faster (since each area of the input space has neurons).

The NW method is for all neural networks except FIR. The initialization method is listed in Table 5.2 column f.

State initialization
All recurrent neural networks have one or more state variables, that have to be assigned initial values. These values are the contents of the delay registers before a training epoch or before the network is simulated with input data. State initialization is required for the recurrent networks, methods 2, 5 and 7. In the experiments a state initialization function is used that draws weight values very close to zero from a normal distribution ($\mu = 0$, $\sigma = 0.1$). This choice was made rather arbitrarily.

5.5.3 Training of a neural network (3)

When a neural network structure is selected and initialized, training can start. The issue that has to be resolved now is when to stop training. A neural network can perform badly on a task if it is either trained too little (underfitting of the training data) or trained too much (overfitting of the training data). The decision to stop is made by the algorithm if one of the following stopping criteria is fulfilled:

1. The performance goal (the goal value for the error measure on the training set) is met ;
2. A preset maximum number of epochs is reached ;
3. The magnitude of the gradient reaches a preset minimum value ;
4. The magnitude of either the gradient, or the error measure, becomes infinite ;
5. (only in the Levenberg-Marquardt algorithm:) The maximum value of parameter μ is reached.
6. Validation stop.

These stopping criteria are available for all Matlab neural network implementations, except for the FIR neural network [Janssen, 1998^S]. For the FIR network only criterium 2 was originally implemented, but criterium 5 was added to the FIR toolbox (as a new function *firbp2_val.m*).

For more information on these stopping criteria the Matlab documentation can be consulted. The validation stop concept (criterion 6) is important so it will be described in more detail in subsection 2.

Training parameters
Neural network training algorithms are controlled by training parameters. These parameters are important because they influence the training process (for example the duration in number of training steps), but in practice the choice is often to use a convenient set of default values, that has proven to produce good results for a large class of training problems. The choice made can first be verified on some small scale 'trial and error' experiments.

In Table 3.2 the defaults as given by Matlab are shown for two training algorithms. The basic gradient descent backpropagation algorithm (called *traingd* in Matlab) has only a learning rate parameter. The Levenberg-Marquardt (LM) algorithm (called *trainlm*) has four parameters.

traingd (Standard gradient descent algorithm) ; can be used by methods 1, 2, 3, 5, 6

Parameter	Description	Default value
lr	learning rate	0.01

trainlm (Levenberg-Marquardt algorithm) ; can be used by methods 1, 2, 5, 6

Parameter	Description	Default value
mu	factor that sets the balance between the Gauss-Newton method and standard gradient descent.	0.0010 (initially)
mu_dec	decrease factor for mu	0.1
mu_inc	increase factor for mu	10
mu_max	maximum value of mu	10^{10}

Table 5.4: Training parameters for traingd and trainlm

Note that other Matlab training algorithms (e.g. *traingda*, *traingdx* etc.) are not shown because these are similar to *traingd*. Information on what algorithms can be used by which neural network types can be found in the documentation of each Matlab toolbox.

For the ASP experiments, the LM algorithm was used in most cases, with default parameter values. This choice was motivated by the speed of the LM algorithm: In all Matlab code implementations normal gradient descent is very slow for recurrent networks and for a large number of training epochs. LM has far less floating point operations (see [Peelen, 1999] for a comparison) and it uses very little epochs, and is therefore fast. More information about the Levenberg-Marquardt algorithm is given in section 4.3.

If the ModNet or NNet toolbox would have been written in C code the standard gradient descent algorithm could have been used. The FIR toolbox (written in C) shows that standard gradient descent in C code is much faster than any Matlab algorithm.

Training set

The training set used consists of 15 phonemes (5 of each class). The phonemes are put in a semi-random order that is fixed (not changed during the training process, in fact the order is the same for all ASP experiments with all network types).

For static neural networks and the k-NN classifier the order of the phonemes in the training set has no influence at all as each frame is dealt with independently. For dynamic neural networks the order highly influences the training process because past frames have influence on the classification.

The training set holds the input patterns and the targets. The n=1…N input patterns $\mathbf{p}(n)$ each hold 12 mel-cepstral coefficients $p_i(n)$, i=1…12. The N target patterns $\mathbf{t}(n)$ each hold 3 target values $t_i(n)$, i=1…3, one for each output of the neural network. Each target value corresponds to the class that the input pattern belongs to (1, 2, or 3 respectively). The target $t_i(n)$ is 1 if the pattern $\mathbf{p}(n)$ is of class i and zero otherwise:

class	target
1	**t**(n) = [1 0 0]
2	**t**(n) = [0 1 0]
3	**t**(n) = [0 0 1]

Note that each pattern (of 12 coefficients) is calculated from one speech frame. With the fixed phoneme duration of t_{PH} = 0.3(s) or N_S=4800 samples, the number of frames obtained with the Voicebox software is:

$$N_F = fix\left(\frac{N_S - N + I}{I}\right) = 36 \tag{5.12}$$

with the frame size N = 256, frame interval I = N/2 = 128 and the *fix* operation (Matlab) rounding down towards the nearest integer value. This is the maximum integer number of frames that can be extracted out of the sequence. The last { N_S-(N_F+1)*I } = 64 samples at the end of the phoneme sample sequence are not used because another N-sample frame can not be completely filled.

The training set has 15*N_F = 540 patterns in total. Each phoneme of 4800 samples is reduced to N_F*12 = 432 mel-cepstral coefficients.

5.5.4 Using a validation set (4)

The concept of the *validation set* is explained here in a qualitative way. A good mathematical treatment of this technique was not found, but should exist in literature because the technique is widely used. (see [Bishop, 1995] for the similar *cross-validation* technique.)

During training the performance of the neural network on an independent validation set is measured over time. The validation set consists of input data and targets like the training set, but it is independent, which means the data used in this set is different from the training set and this data is not used in the training algorithm.

The error made by the neural network on the training set obviously decreases during training because the algorithm will try to minimize this error. It is very likely the error made on the validation set also decreases because the 'general properties' of the training set are learned first. The general properties of the training data that are learned are also beneficial to the performance on the validation set.

At a certain point, the error on the validation set may start increasing. This happens when the neural network learns not only the general properties of the training data but also learns features that are highly specific only to the training set (i.e. the overtraining effect has started). So at the minimum of the validation error, training should be stopped because the expected error on future data sets is minimized at this point. The performance on the validation set effectively is an estimate of the generalization performance of the neural network. This quantity is what one would like to have minimized.

The validation stop was used in the ASP experiments. It was the most important stopping criterium, because the performance goal was set zero (effectively disabling this criterium) and the number of epochs was chosen very high (which ensures training is not stopped too early by the epochs stopping criterium).

But the training does not stop right away when an increase in the validation set error is observed: A parameter called *max_fail* states how many epochs the training algorithm should continue, despite of an increasing validation set error. During this period it can happen the validation set error starts decreasing again and training can proceed. However, when no decrease is observed, the training process is stopped after *max_fail* epochs. The training procedure finally returns the neural network weights, that led to a minimum value of the validation set error.

A small validation set was used of 3 phonemes (one of each class) in the class order 1, 2, 3. The *max_fail* parameter was set to 25 in most cases.

5.5.5 Repeated training (5)

Convergence of a neural network training algorithm (based on gradient descent) to a global minimum is never guaranteed. Instead a sub-optimal local minimum can be the result of a training procedure.

For this reason the training of a network is repeated some times in most experiments. This

- increases the possibility of finding at least one solution either at the global minimum or at a good-enough local minimum.
- gives some insight into the success rate of the algorithm: how often does the training procedure succeed or fail.

For each experiment a different random initialization of the weights (see step 2) is used. A typical repeat value used in the experiments is 10. See table 3.3 for the repeat values.

5.5.6 Evaluation of performance using a test set (6)

Finally the performance of a trained neural network should be evaluated. The best way to do this is with an independent test set. The validation set cannot be used for evaluation, because the neural network may have overfitted the validation set. This can occur because the validation stop procedure selects for the neural network that minimizes the error on the validation set.

The independent test set used in the ASP experiments has 6 phonemes of each class (18 in total). It contains 18 * N_F = 648 patterns $\mathbf{p}(n)$ (see equation 3.3). No postprocessing will be done on the outputs.

The simplest way to describe the performance (in a single number) on the test set is either the classification frame success rate or the classification frame error rate. The success rate will be used in this report. The same method was used in [ten Hove, 1996] and [Janssen, 1998] for evaluating a speech recognition system.

The frame success rate is calculated by evaluating the classification decision for each frame:

$$perf = \frac{N_{CORRECT}}{N_{FRAMES}} \cdot 100\,\% = \left(1 - \frac{N_{ERROR}}{N_{FRAMES}}\right) \cdot 100\,\% \tag{5.13}$$

The number of errors N_{ERRORS} is the number of frames for which the maximum i^{th} neural network output at frame n, $y_i(n)$, is not equal to the maximum target value $d_j(n)$, that is:

$$N_{ERROR} = \sum_{n=1}^{N_{FRAMES}} \left(1 - \delta^{Kro}\{\arg\max_i y_i(n), \arg\max_j d_j(n)\}\right) \tag{5.14}$$

It should be kept in mind this is a crude method of comparing performance. The actual performance of the neural network when placed into a full speech recognition system can be different, because in a full system a large amount of postprocessing is performed that can 'undo' some types of errors, for example.

Also the precise moment of phoneme transitions has influence on the performance value, when using the above method. But for real speech the precise moment in time that one phoneme stops and the following starts is often not well-defined.

It can be concluded that for a full speech recognition system evaluation a different evaluation method should be used. The Insertion/Deletion/Substitution error method in [Robinson, 1994] is the standard. However, the simple performance measure is of good use in the ASP experiments to get some insight into the performance of the 'bare' neural networks without pre-processing.

5.6 Results: Artificial phoneme recognition experiments

The results of the ASP experiments are presented in this section. In subsection 3.3.6 one result for a state-space network (a single structure) will be examined in detail. In subsections 1 to 3.4.2 the results for the 7 classification methods will be summarized. The results for all methods are summarized and compared in subsection 3.4.1. A list of methods, structures and parameters was already given in the previous section.

5.6.1 Results for a single state-space neural network structure

In this subsection the results of an initial ASP experiment with a state-space neural network will be discussed. These first results will be shown in this subsection in more detail than the rest of the experiments.

The method 2 structure is trained and has parameters N_1=2, N_2=5 and N_S=3. Both modules contain 2 layer static networks. Both modules have logistic sigmoid units in the hidden layer. The output neurons y_i also have logistic sigmoid units which keep the output between zero and one. The state neurons (the outputs of module 1) are linear neurons. The structure is visualized in figure 2.12.

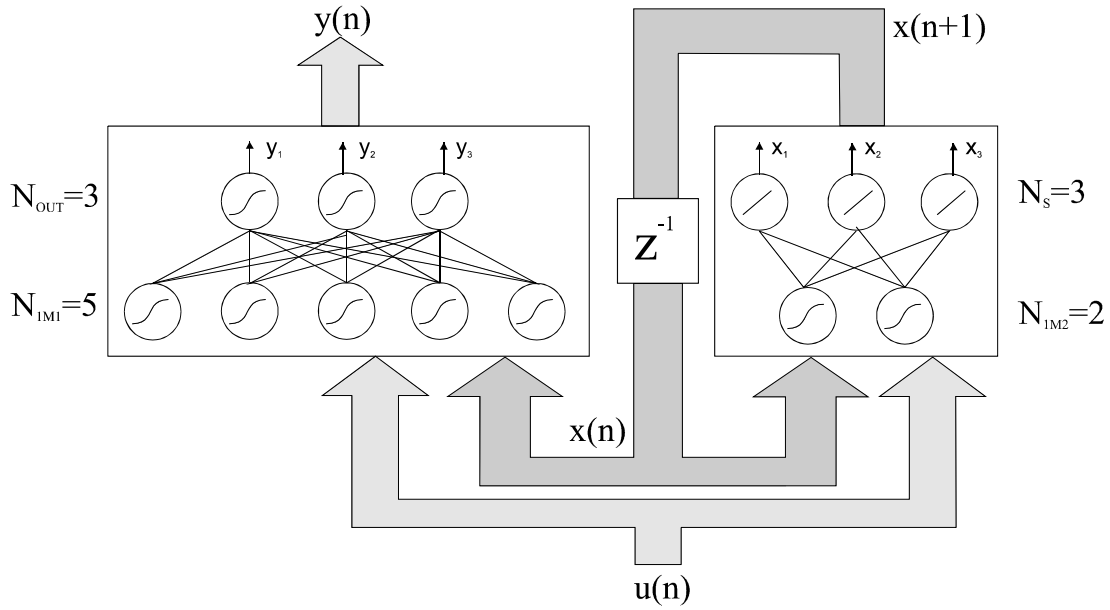


Figure 5.6: The 2-layer state-space neural network structure

The structure was trained with the training set described in the previous subsection, using the Backpropagation Through Time algorithm with Levenberg-Marquardt algorithm (BPTT-LM) of the ModNet toolbox. The training record which shows the performance of the network as a function of the number of epochs trained, is shown in figure 2.11.

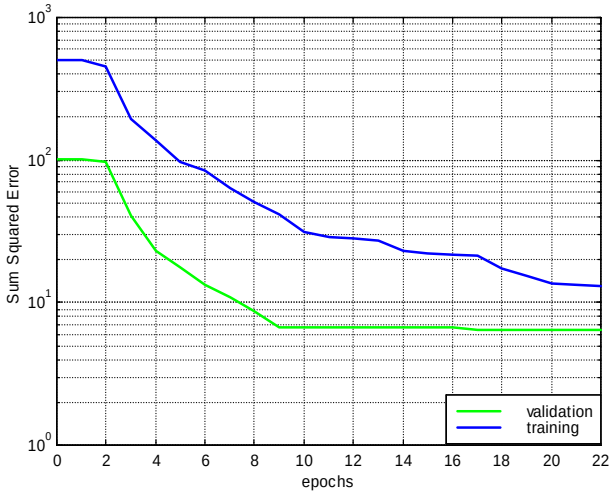


Figure 5.7: Training record (Sum Squared Error vs. epochs)

Both the performance on the training set (dark curve) and the validation set (light curve) are shown. For both the errors decreases comparably in the first 9 epochs of training. From epoch 9

on the error on the validation set decreases only minimally while the error on the training set continues to drop. The reason the training eventually stops after epoch 22 is stopping criterium number 5 ('maximum mu reached').

The validation error is smaller than the training error for all epochs because the validation set is five times smaller than the training set. This means the SSE, that depends on the number of examples and example lengths, is also smaller (also five times smaller for epochs 0 to 9).

The trained network is then simulated with the independent test set data. The output of all three outputs are plotted versus time *n* in figure 3.4.

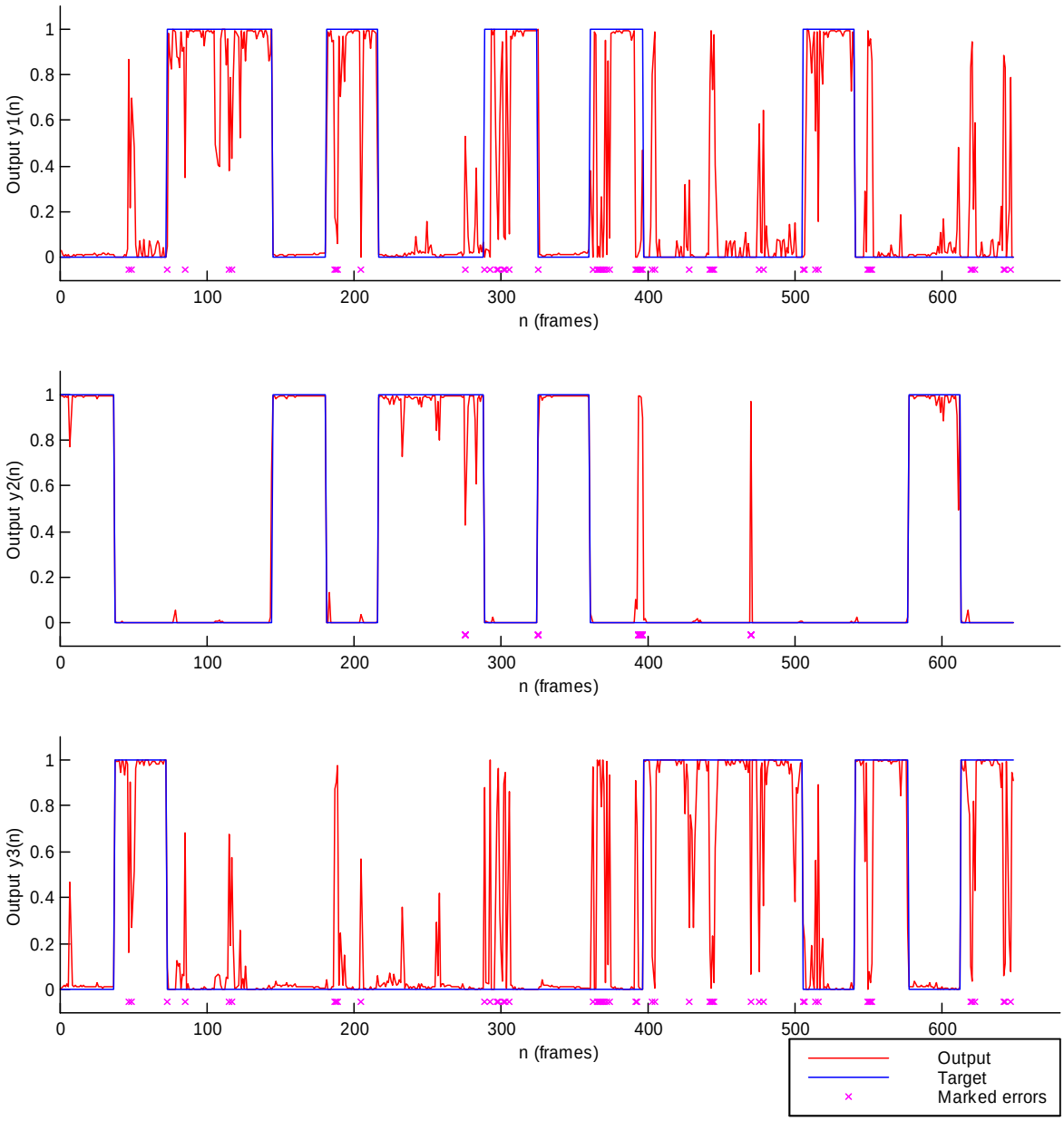


Figure 5.8: The three neural network outputs (and targets) for the 18 phoneme, 648-pattern test set

The dark curve (the 'square wave') is the one/zero target $t_i(n)$. The grey lines are the outputs of the neural network at each time *n*. The points in time where the network generates a wrong classification are marked with an 'x' placed near the x-axis. Outputs and targets are discrete points but they are connected for visibility.

The performance on the test set (rounded to 3 significant digits) was 91.0 % (590 frames classified correctly, 58 frames wrong). It is clear from the figure that some postprocessing would be useful for this particular network, to smooth the output signals and remove some of the 'spikes' and thereby reduce the error rate.

In the following figure (3.2), the classification decision is plotted for each frame *n* so the effect of the spikes on the decision can be observed.

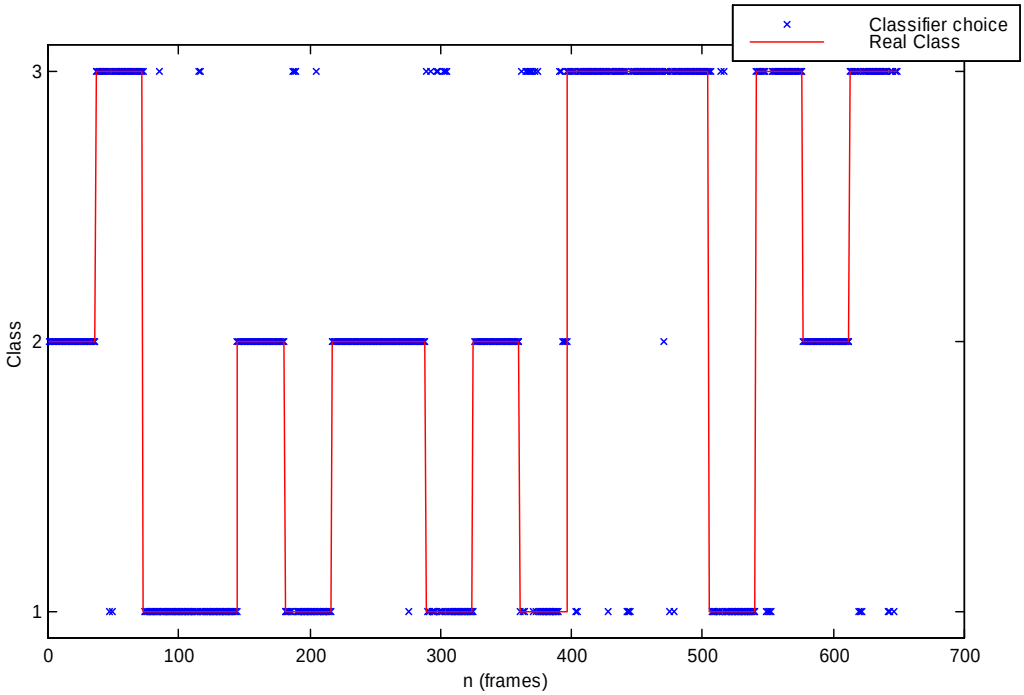


Figure 5.9: Visualization of the classification decision made by the network for each frame

The real class is plotted as a continuous line for clarity. The classification decision for each frame is marked with an 'x'. All phonemes in the test set are detected, but occasionally the classification oscillates between class 1 and 3.

5.6.2 Method 1: Static two-layer neural network (MLP)

Because only one structural parameter has to be chosen, the results for method 1 (the two-layer MLP) can be straightforwardly plotted. Figure 2.13 shows the performance (equation 3.1) of 10 times 25 trained neural networks on the test set for the 25 different values of the parameter N_1.

The success rate varies widely. The most successful networks are grouped in a 'band' around the 90% value. There are relatively few networks in the 70-85% region. A number of networks clearly perform sub-optimal because they lie far below the upper 'band'.

The reason of the stop (see subsection 3.1.1 for stopping criteria) of the training procedure was not recorded during this experiment because the NNet toolbox does not provide for this, so it was not investigated if one of the stopping criteria perhaps gave rise to these sub-optimal networks.

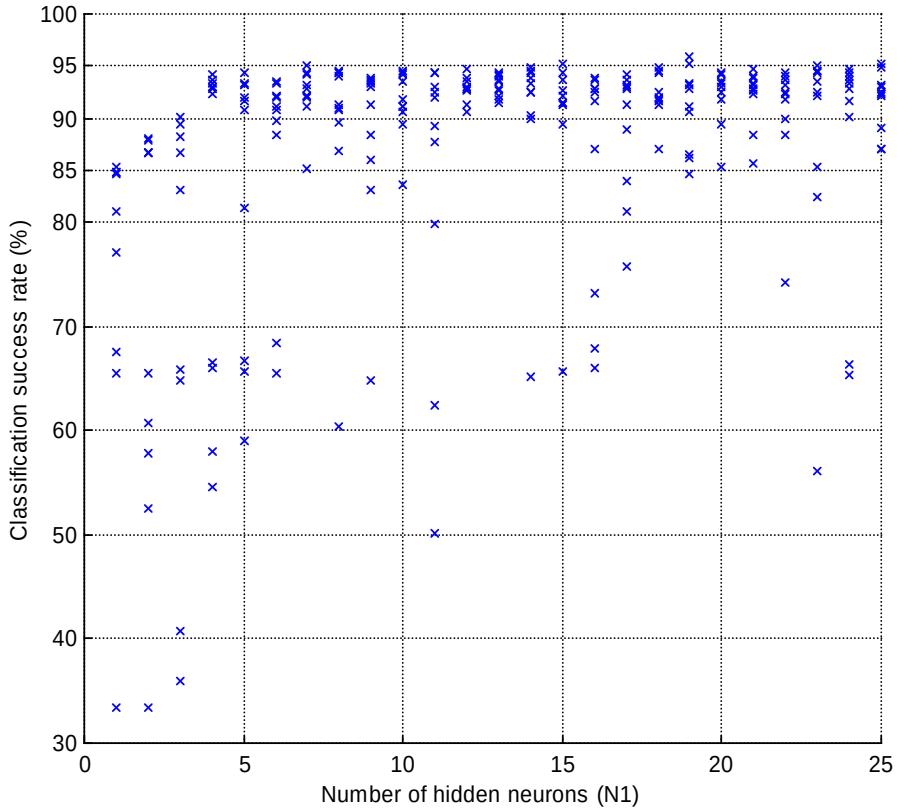


Figure 5.10: Method 1; classification performance of 250 networks (on the test set) as a function of varying number of hidden neurons N_1

The influence of the number of hidden neurons N_1 is visible. For the values 1 to 4 the performance of the group of best performing networks increases. For N_1=5 or higher, the top performances fluctuate, but stay approximately the same and there is not much influence of N_1 on performance. The number of 'suboptimal' networks decreases a bit with higher N_1.

The performance of the best neural network on the test set was 95.8% for N_1=19.

Distribution of performance values

An interesting observation is that the performance values of the networks that score around the 93% value (the successful band) seem normally distributed. This was visually checked using the Matlab *normplot* and *boxplot* functions. This indicated that the performance values in the 91-96% band are indeed almost normally distributed (μ = 93.2% ; σ = 1).

5.6.3 Method 2: the state-space neural network

The results for the state-space neural network are visualized in figure 2.4 the same way as for the previous method. This time the number of the structure used is plotted on the horizontal axis. Each unique combination of the three parameters corresponds to a different structure number. Directly plotting parameters versus performance is impossible because it would require a 4-dimensional plot.

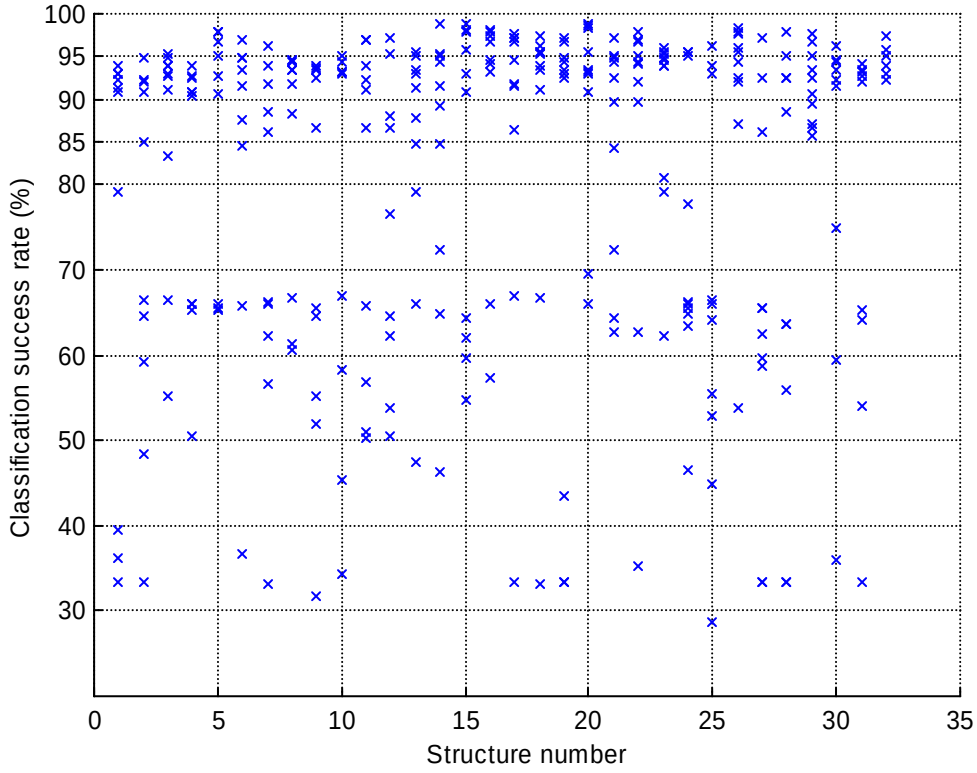


Figure 5.11: Method 2; classification performance of 316 networks (on the test set) for varying network structure

The parameters that describe each structure are listed in table 5.5 together with the structure number.

Structure number

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
N_1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
N_2	3	3	3	3	4	4	4	4	5	5	5	5	6	6	6	6
N_S	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4

Structure number

	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
N_1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3
N_2	7	7	7	7	8	8	8	8	3	3	3	3	4	4	4	4
N_S	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4

Table 5.5: Structural parameters and corresponding structure number

It can be seen in the figure that the structure does have some influence on the performance measured. The small structures 1 to 4 do not perform as well as most of the larger structures 5 and higher.

5.6.4 Method 3: 2-layer FIR network

The results for the FIR neural networks are depicted in figure 3.8. There are almost no 'failed' training runs, which can be attributed to the use of standard backpropagation instead of the LM

algorithm (which often stops on stopping criterium 5). Note the y-axis scale of the figure is different than for the previous methods.

There is a relation between the structure used and the performance. The smaller networks (structures 1-25) perform worse than the subsequent larger structures.

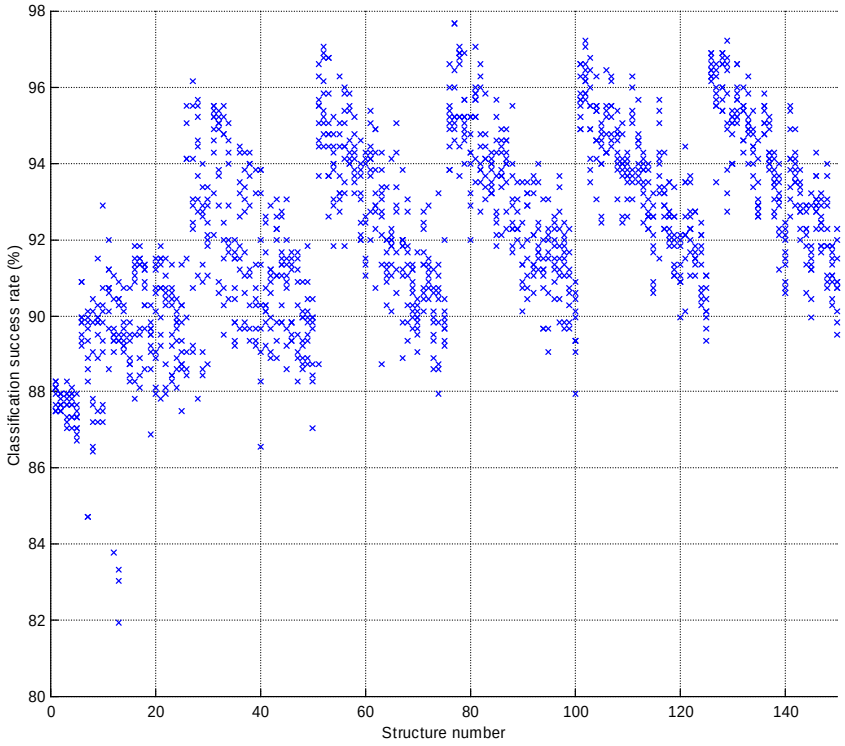


Figure 5.12: Method 3; classification performance of 150 networks (on the test set)

The parameters for each of the 150 structures are not listed in this report. To see which parameters lead to the best performance, the values of the three parameters N_1, T_1, T_2 are plotted as a function of the performance value in figure 3.9. This rectangle plot is a semi-3D plot. Bigger rectangles correspond to more neural networks that fall into the category plotted on the axes.

The y-values of the data points in the plot were obtained by counting the occurrence of parameter values of all networks, whose performance falls into the corresponding 'bin' on the x-axis. The x-axis bins were chosen at a distance of 0.5 of each other (so there are four bins/rectangles for every two percent performance increase).

Such a figure shows general trends (if any) about the relationship between performance and parameter values. From the upward trend in figure a) it can be concluded the most successful networks have a larger number of neurons N_1 (about 5-7). Figures b) and c) show a downward trend so good performance is obtained with small delay registers T_1 (about 2) and T_2 (about 2 or 3). Note that only the combination of large N_1 and small T_1, T_2 gives the best performance.

An explanation for this could not be thought of because one would say higher parameters T_i allows the network to use data of more time steps which aids the classification of the transitional phoneme signals.

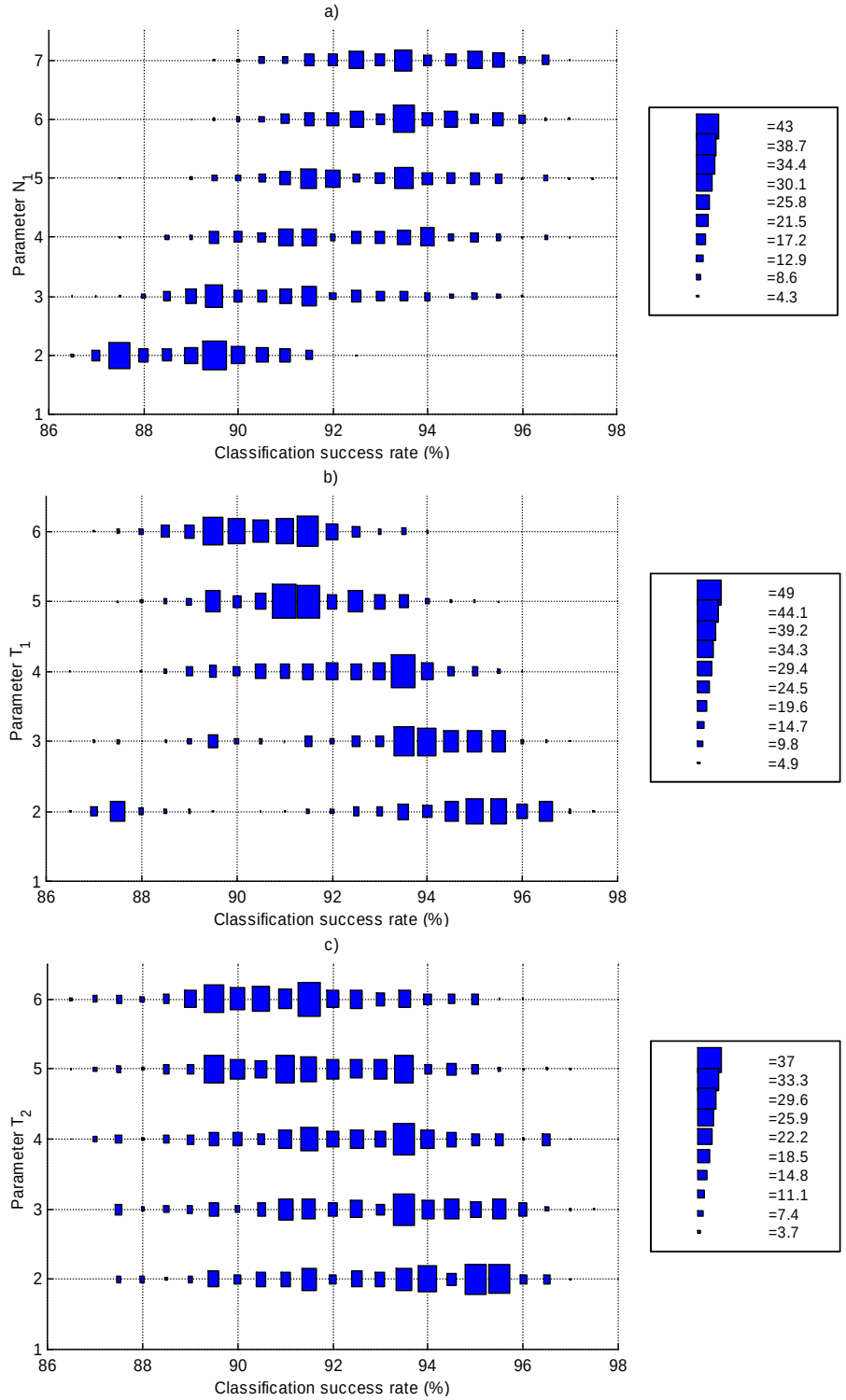


Figure 5.13: Relation between performance and parameter values N_1, T_1, T_2 (figures a,b,c respectively)

5.6.5 Method 4: the Nearest-Neighbor classifier

The performance on the test set for the various values of N used is plotted in figure 2.14.

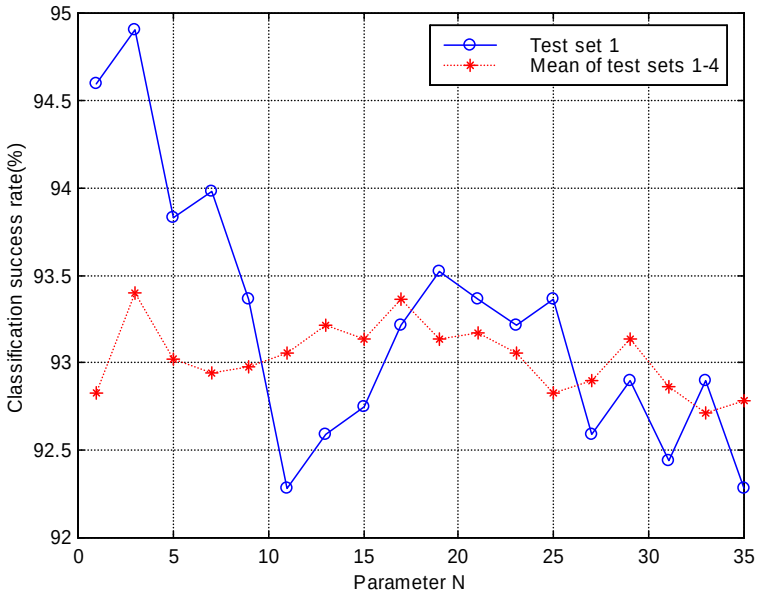


Figure 5.14: Method 4 ; classification performance on the test set with varying parameter N

The circles mark actual performance values for values of N used. These points are connected by a line for clarity.

With the k-NN method, the parameter is often selected by trial-and-error [van der Heijden, 1995]. In this case k = 3 performs best with 94.9% correct on the test set. For other test sets however, another value of k gave the best performing classifier. To obtain the best value for k, a number of test sets could be tried and results averaged. This averaging was done over four test sets (including the original one). The result is given in the figure (grey dotted line). Again the best choice was k=3 but the mean values do not differ very much, so the choice of k is not critical.

5.6.6 Method 5: FRNN

The results of method 5 (the FRNN) are presented in figure 2.16. For N=5 the best result is obtained (96%), but the variation in the results of different networks (with the same structure) is high.

5.6.7 Method 6: three-layer MLP

For the 3-layer MLP no clear link between parameters and performance was found, except that N_1 should be ≥4 for best performance. The top ten performing networks had both N_1 and N_2 between 4 and 10. The results are not shown here graphically because results were similar to the 2-layer MLP.

The best performing network had $N_1 = 10$ and $N_2 = 9$ and scored 96.1%.

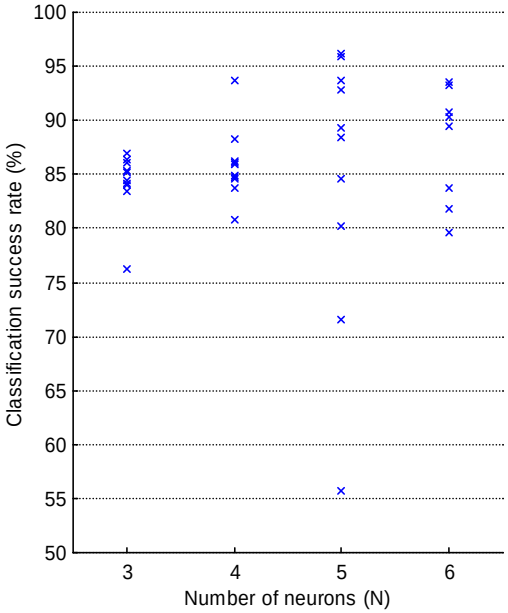


Figure 5.15: Method 5; classification performance on the test set with varying number of neurons N

5.6.8 Method 7: State-space network trained with Elman algorithm

The structure of the state-space network from method 2 is used. This structure can be easily implemented in the Matlab Neural Network Toolbox and trained with the supplied Elman training algorithm (see subsection 3.7.1 about this algorithm). The algorithm can be seen as a simplified version of the RTRL algorithm.

It would be interesting to know how this algorithm performs on the classification task. Therefore the best state-space neural network structure (see subsection 3.3.1) was taken and trained 10 times, this time with the Elman algorithm. The best performing network scored 98.1% on the test set.

Three more structures were tried but not more, because the implementation of the Matlab Elman algorithm for recurrent and time-delay networks is very slow.

5.6.9 Comparing methods

In this subsection the methods are compared. The results presented here will be used in the final conclusion on the ASP experiment in section 3.1.3.

For each method, the best network is taken for further evaluation. The performance figures on the test set of these best performing networks, which were already listed in the preceding subsections, are summarized in table 5.6 (column d).

a	b	c	d	e	f
Method	**experiment ID**	**Network type**	**Performance on test set of best network (%)**	**Structure of best network**	**Number of weights in network**
1	1	2-layer MLP	95.8	$N_1 = 19$	307
2	6	state-space (BPTT-LM)	**98.8**	$N_{HM1}=2$; $N_{HM2}=6$; $N_S=2$	147
3	9	FIR	97.7	$N_1=5$; $T_1=2$; $T_2=3$	173
4	4	k-NN	94.9	k=3	-
5	2	FRNN	96.1	N=5	90
6	8	3-layer MLP	96.1	$N_1=10$; $N_2=9$	259
7	11	state-space (Elman)	98.1	$N_{HM1}=2$; $N_{HM2}=6$; $N_S=2$	147

Table 5.6: Best performing networks, performance and structure

The best network structure is also given for reference (column e). The total number of weights in the network (i.e. the number of free parameters of the system) is listed in column f.

But different test sets lead to a different performance number. Therefore, the methods were tested using more test sets. Because new data for the test sets can be created easily, a large number of 100 additional test sets was used. The results are given in table 5.7. The mean performance over all 100 test sets is listed in column c. The performance values of table 5.6 (column d) are listed again in column b for comparison.

a	b	c	d	e
Method	**perf. of table 5.6 (%)**	**Mean perf μ over 100 test sets (%)**	**Std. deviation σ of perf (%)**	**95 % confidence interval on the mean μ (%)**
1	95.8	93.7	1.3	93.4 - 93.9
2	**98.8**	**97.4**	**1.1**	97.2 - 97.7
3	97.7	94.4	1.4	94.1 - 94.7
4	94.9	92.1	1.2	91.8 - 92.3
5	96.1	94.6	1.9	94.3 - 95.0
6	96.1	93.8	1.3	93.6 - 94.1
7	98.1	96.0	1.6	95.7 - 96.4

Table 5.7: Performance figures of methods on test set and 100 new test sets, respectively

The large number of test sets allows the use of statistics to get some more information about the performance. Using the Matlab function *normfit* the standard deviation σ was estimated (see column d) and the 95% confidence interval for the mean μ was calculated (see column e). A normal distribution was assumed. The normality of the distribution of performance figures was visually checked using the Matlab *normplot* and *boxplot* functions.

These values are visualized in figure 2.6, where the circles represent the mean performance (column c) and the whiskers represent the 95% confidence interval of the mean (column e).

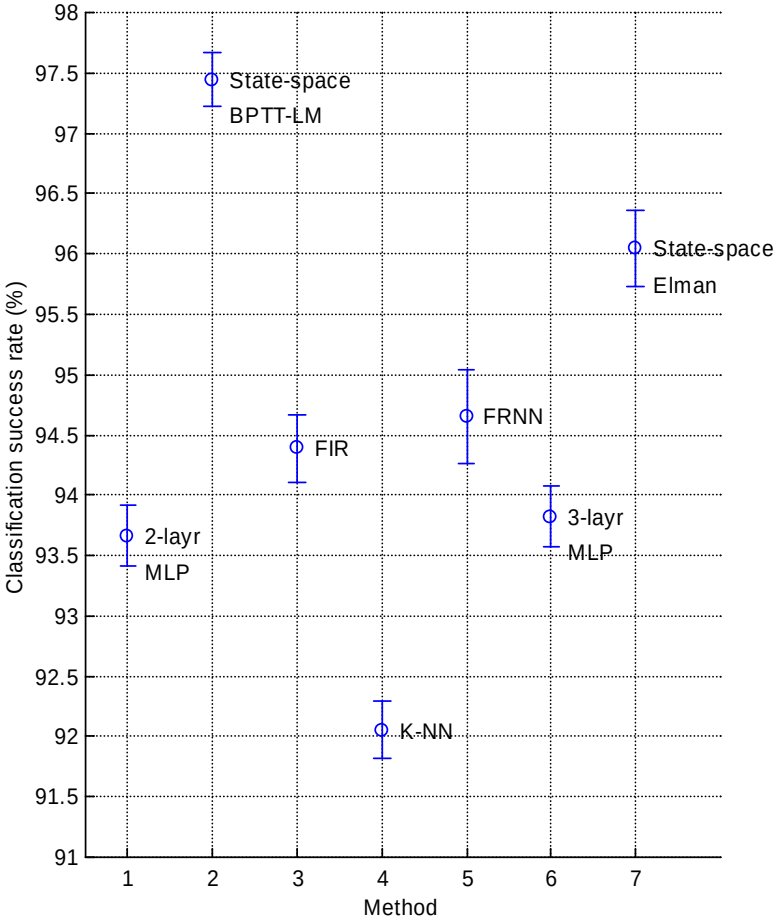


Figure 5.16: Mean performance and 95% confidence interval of the mean for the 7 methods

The relation between the classification success rate and the (mean) number of weights in the neural networks with a certain success rate is visualized in figure 2.5. Each mean value calculated is marked with a circle or cross. The marks are connected for visibility. The five lines are plotted in three separate figures for visibility. Each mean value is calculated by histogramming, i.e. averaging the number of weights over all neural network structures that fall into the 0.5%-wide bin on the x-axis. The x-value of each data point is the location of the center of one bin.

The figure makes the 'performance divided by the number-of-weights' ratio visible. (This will be called the *p/w-ratio*). A higher value means that less weights are needed in a neural network to obtain equal performance, so the network has learned a more compact/efficient representation of the relation between input data and targets. It can be concluded from this figure for the best-performing networks (>90%), that the general order from best to worst p/w-ratio is:

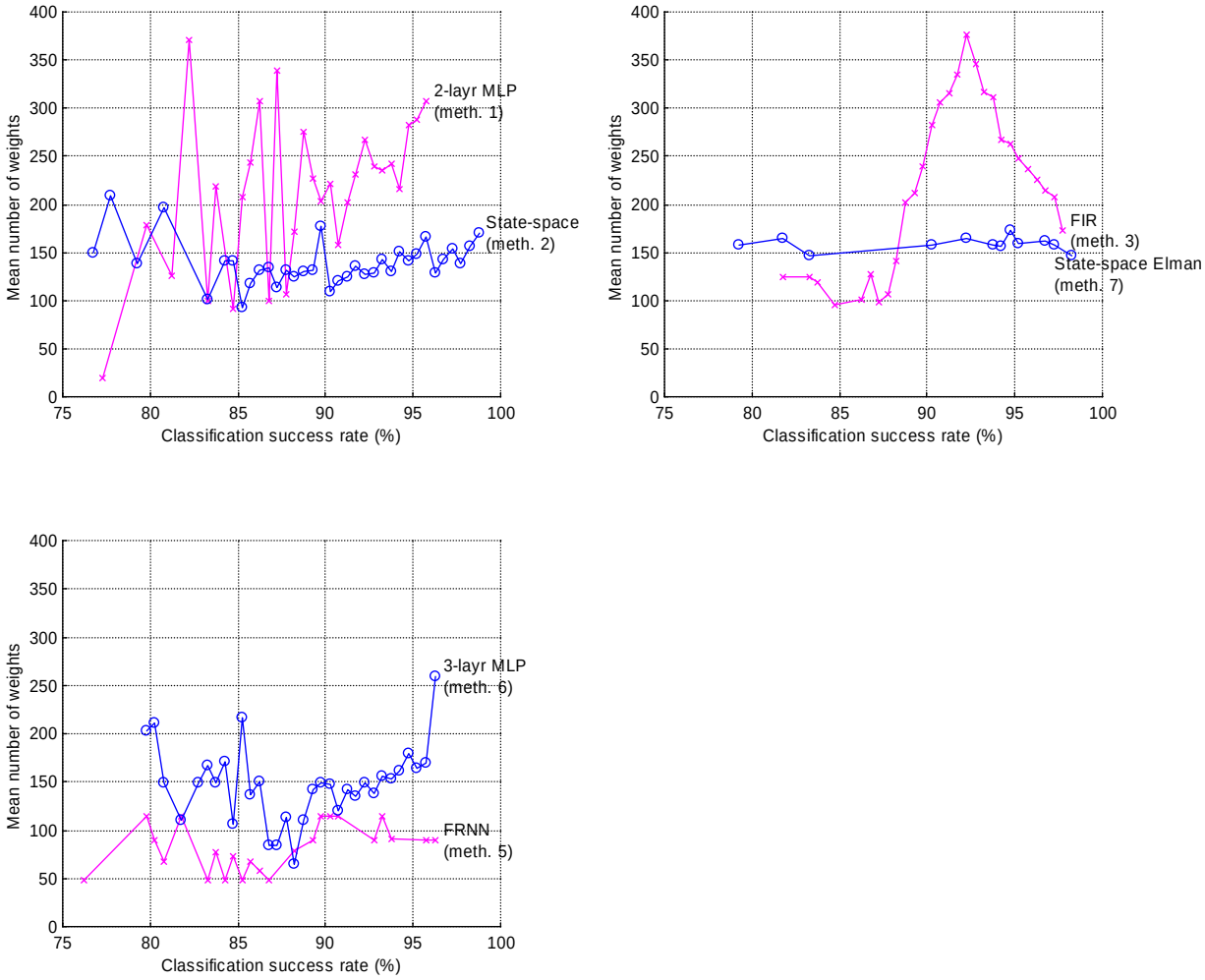


Figure 5.17: Relation between performance rate and mean number of weights for five different neural network methods

1. FRNN
2. state-space network (BPTT-LM)
3. state-space network (Elman)
4. 3-layer MLP
5. FIR network
6. 2-layer MLP

Note that for performances >95%, positions 4 and 5 are swapped.

5.7 Results: Preliminary speech classification experiment

The classification experiments were repeated with real speech signals instead of artificial ones. The same setup of a three-class classification system is used. The three classes are the phonemes /ay/ , /ey/ and /iy/ which are also the phonemes that were the models for the artificial phonemes.

The phoneme sounds were obtained from the Timit database. They were selected randomly from a series of ten different sentences spoken by five different male speakers from the dialect

region 1 (see Appendix D for information on Timit). Figure 2.15 shows the spectrograms of example phonemes, one of each class, taken from the Timit database.

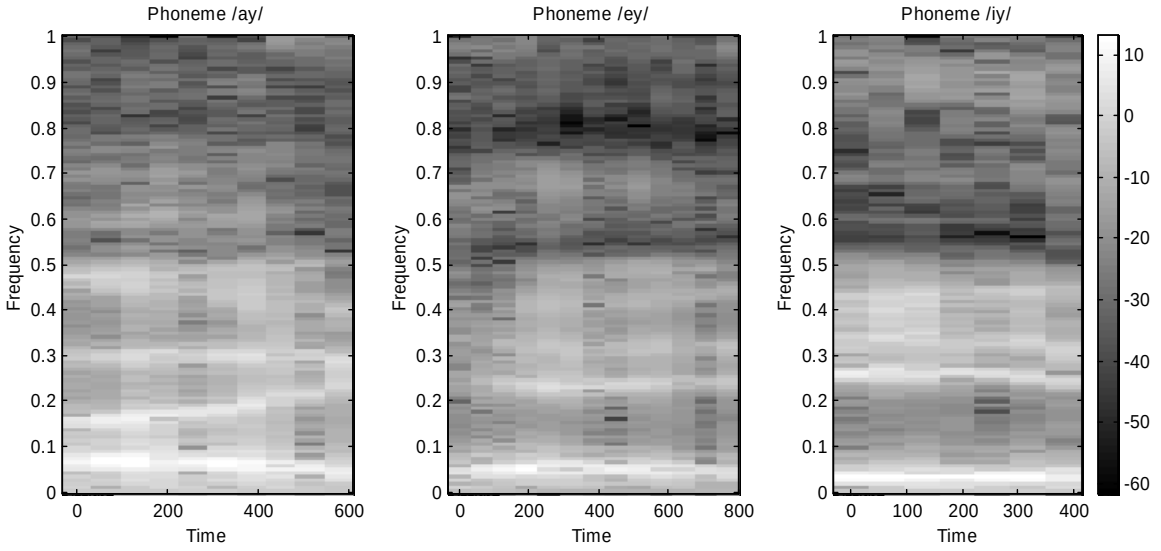


Figure 5.18: Spectrograms of three example real speech phonemes (one of each class, time in units [2 samples] , frequency in units [* F_s/2 (Hz)] Higher values represent more signal content at the specified frequency.)*

The contents of the training-, validation and test sets is shown in Table 5.8. For each class both the number of phonemes is listed and the resulting number of mel-cepstrum data frames. The numbers of phonemes were chosen such, that the number of frames of each class would be approximately the same.

	class 1 (/ay/)	class 2 (/ey/)	class 3 (/iy/)
number of phonemes in training set	8	13	14
number of frames in training set	120	136	114
number of phonemes in validation set	2	4	7
number of frames in validation set	45	48	44
number of phonemes in test set	6	8	12
number of frames in test set	109	89	99

Table 5.8: Phonemes used in the training-, validation- and test sets.

After training the networks the test set is used to assess the performance. The results are shown graphically in figure 2.17.

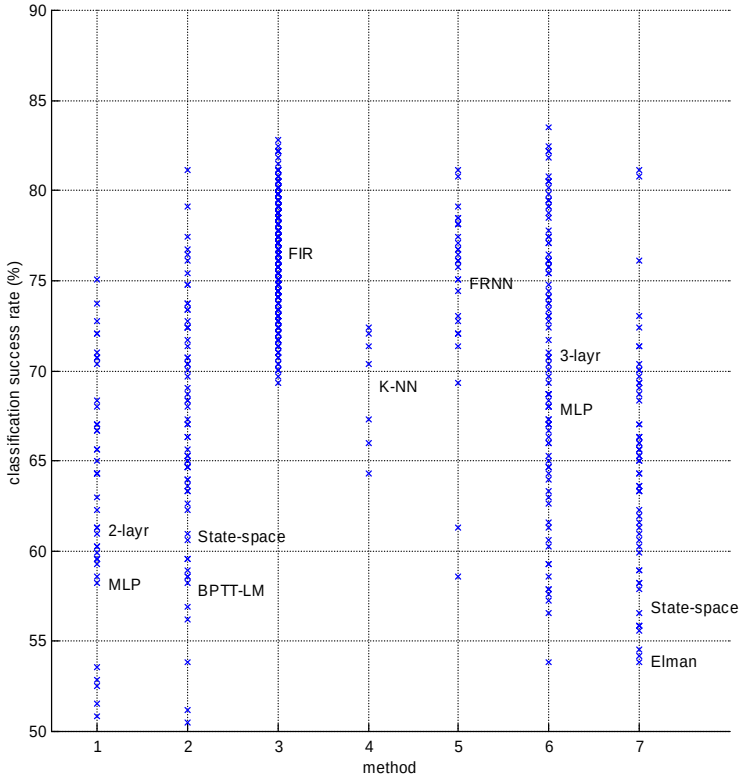


Figure 5.19: Classification success rates for each method.

The 7 different methods used are set out horizontally. The classification success rates are plotted on the vertical axis. The crosses correspond to individual neural network performance results: each network is the result of a separate training run.

For this task the 3 layer MLP and the FIR network are performing best. The spread in the results is high: this is partly caused by the many different structures that were used. These results were not extensively studied so no evaluation of the different structures is done.

The performance of the best network of each class is listed in Table 5.9.

Method	Best performance on the test set (%)
1	75.0
2	81.1
3	82.8
4	72.4
5	81.1
6	**83.5**
7	81.1

Table 5.9: Performance values of the best neural network for each method

It can be concluded that the expected advantage of recurrent networks for dynamic classification does not clearly show in the results. To isolate a possible problem in recurrent network training more experiments should be done in a systematical way (this experiment is only preliminary).

Using delta features in phoneme recognition

As was explained in subsection 2.2.3 the delta or delta-delta features are often added to the mel-cepstrum feature set to obtain better recognition of transitional phonemes. A new mel-cepstrum

feature set with 12 delta coefficients added was used in a small scale experiment with only a few neural network structures. The following best networks were obtained:

Method	Best classification performance (%)	Best network structure
two-layer MLP	85.2	$N_1 = 4$
state-space network BPTT-LM	79.1	k = 9
k-NN classifier	85.9	$N_1 = 2$; $N_2 = 5$; $N_S = 2$

Table 5.10: Classification experiment results using mel-cepstrum and delta features

It can be seen that the best performance can be increased for each method by adding delta features. The delta-delta features were not tried.

5.8 Conclusions

Best network

The results in table 5.7 and figure 2.6 clearly show the state-space architecture performs best. The confidence interval calculations can be used to state it is 95% certain, that the state-space network trained with BPTT-LM performs at least 2.2% better that the number two network structure, the FRNN.

The FIR network performance is close to the FRNN. The 2 and 3-layer static networks perform worse than the FIR network with 95% certainty. The k-NN has worst performance.

Variation in performance

The state-space neural network also scores best in the estimated standard deviation σ of performance values over 100 test sets (see table 5.7) because the value is smallest. This means that the state-space network looks like the most reliable method because the variation of the performance over different sets of data is smallest.

Performance versus number of weights

The comparison of the p/w-ratio in subsection 3.4.1 confirms the common assumption found in recurrent neural network literature [Bengio, 1996] [Robinson, 1994], that recurrent networks can perform equally well as static neural networks or time-delay networks (like the FIR network), while less weights are needed. Recurrent networks thus offer a more compact representation of a relation between input data and target than other network types.

Given the above results of the ASP experiments, recurrent neural networks seem a promising approach to phoneme recognition. However, the possibility of using delta and delta-delta mel-cepstrum features was not investigated. These features can be used to provide some dynamic processing capability to static neural networks.

Preliminary experiments on phoneme recognition

On phoneme recognition the FIR neural network and the 3-layer static MLP performed best. The state-space networks used did not perform very well compared to the other types.

Using delta mel-cepstrum features clearly increased performance: actually a static network trained with delta-features performs better than any recurrent network trained without delta features.

The available time did not permit an investigation of all possible causes of the mediocre performance of the recurrent networks at this task. Instead, only a few possible causes of problems for recurrent networks are given here:

- the one/zero targets are not the best choice, other target types could be used
- the size of the training set can be made larger. (The set used is actually quite small. In a full speech recognition systems it is not uncommon that a large part of the speech database is used in training.) Recurrent networks may need more training data than static networks because interdependencies between input data frames have to be learned.
- other network structures could give better results. The complexity of this real speech task is higher than for the ASP experiments so this suggests network structures should be made bigger.
- the performance may depend on training parameters. Not many different combinations of training parameters were tried so there may be still room for improvement in making this choice. Specifically, the parameter *max_fail* used in the validation stop criterion has a large influence on when training is stopped.
- the training data is currently presented to the network in one big sequence because this is required for the LM algorithm. However, partitioning training data into multiple epochs is the usual way of training. Then, the sequences can be presented to the network in a random order. (To try this, the standard gradient descent training method could be used.)

CHAPTER 6 CONCLUSIONS AND RECOMMENDATIONS

In this chapter the final conclusions of this report will first be presented. These conclusions are a summary of the individual conclusions that were given at the end of previous chapters.
The second part of this chapter gives recommendations for continuation of research in dynamic neural networks and applications to speech recognition.

Conclusions

The aims of this assignment were listed in section 1.3. In the following conclusions references will be made to these (numbered) aims to show what work was done on each one of the specific aims.

Investigation of recurrent neural network structures and training algorithms (aims 1, 3)

Recurrent Neural network architectures

Several types of recurrent neural networks have been listed in this report. Because some architectures offered a more general description than others, a hierarchical 'classification' of these networks could be made (see figure 2.1).

The most general description of architectures is offered by the general modular network framework. This framework was not directly implemented into software for two reasons: first to avoid a complex (and error-prone) program and secondly to keep acceptable speed of the neural network training and simulation in the Matlab environment.

The state-space network was selected as second-best because it holds some other often-used architectures as special cases. It was found these special cases were used in literature for phoneme recognition tasks. It was shown (chapter 4) that the FRNN, a special case of the state-space network, can not simulate all state-space systems.

Neural network methods for speech recognition (aim 3)

During the survey of neural network architectures special attention was paid to the application of speech recognition. References to existing methods were given.

There exist more neural network architectures that are not included in this report. Specific examples of structures that were not investigated, are:

- Adaptive neural network structures (see 'training algorithms' below)
- Higher-order neural networks (which use multiplicative combinations of inputs as additional inputs).
- The relation between Recurrent Neural Networks and Hidden Markov Models (HMM). Recently the hybrid ANN/HMM structure [Bengio, 1995] was introduced, a structure that combines an Artificial Neural Network (ANN) with a HMM.

Implementation of the state-space neural network

The state-space network was implemented in a Matlab toolbox (ModNet) together with training algorithms (see below). The options to *fix* or *prune* (see subsection 2.2.3) arbitrary connections is provided. This allows sub-types of the state-space network to be trained and simulated. These include the FRNN, the PRN, the multilayer RNN, the state-space SRN and the multilayer FRNN.

Training Algorithms

Training algorithms for recurrent neural networks have been investigated. The Backpropagation Through Time (BPTT) and Real-time Recurrent Learning (RTRL) learning algorithms have been the main focus.

Both learning algorithms were given for FRNN, state-space networks and the general case of the modular network framework. This implies they can be used with all neural network structures in this report which are specific cases of the modular network framework.

As the best algorithm for a phoneme recognition task BPTT was chosen, because:

- training for a speech recognition system is off-line (not real-time) so that epochwise training can be used. Epochwise BPTT has some advantages over epochwise RTRL, which will be listed below.
- Epochwise BPTT performs an exact minimization of the error measure over an example sequence (while RTRL performs an approximate minimization)
- the computational requirements for epochwise BPTT are less than for RTRL in a typical phoneme classification task
- The epochwise BPTT algorithm can be combined with the second-order Levenberg-Marquardt (LM) training algorithm. LM can speed up certain learning tasks dramatically.

The following training approaches were not investigated:

- Algorithms for automatic structure adaptation and automatic modularization.
- Other neural network training paradigms such as information-theoretic principles (MMI, MDL), *Support Vector Machines* and *Bayesian Inference* (listed in subsection 3.1.2).

Implementation of the BPTT and RTRL algorithms for state-space networks
Both the BPTT and RTRL algorithms, and a combined BPTT / Levenberg-Marquardt algorithm (for speeding up training), were implemented in a Matlab toolbox (ModNet). The modular setup of the state-space network structure and algorithms allows for an extension of the software to other modular networks.

Initial experiments on linear and non-linear system identification tasks were performed to test the algorithms. The Matlab code runs at an acceptable speed but is still much slower than a implementation in C would be.

Classification capabilities of recurrent neural networks (aim 2)
Classification capabilities of recurrent networks were investigated. It can be concluded that there exists no thorough mathematical treatment in literature of sequence classification (it was not found, anyway) and how recurrent networks can perform such classification. Some ideas about this were given in section 4.2.

Dynamic neural networks (both time-delay networks and recurrent networks) have an advantage over static neural networks because of their ability to process sequences of data instead of stand-alone data patterns only.

An advantage of recurrent networks over time-delay networks is their ability to process (and therefore classify) sequences of variable length.

Classification experiments using recurrent neural networks (aims 2, 3 and 4)
The state-space neural network was evaluated in an experiment (the ASP experiment, see chapter 5) against other neural network approaches and one non-neural approach. For this classification task artificial ‘speech’ signals were used.

Feature extraction (aim 3)
To obtain a feature extraction front-end for the phoneme recognition system, existing speech recognition methods were first studied. The standard mel-cepstrum method was chosen and described in this report.

Results
From the ASP experiments it can be concluded the state-space network outperforms the other approaches. Recurrent networks performed best, followed by time-delay networks, followed by static neural networks. So state-space networks are promising for use in phoneme recognition.

A comparison of the number of weights versus performance in the neural networks used, confirms the general assumption (in literature) that recurrent networks offer a more compact representation of an input-output data relation than non-recurrent network types.

Using recurrent neural networks for phoneme recognition (aim 4)
The preliminary experiment on phoneme recognition did not show a performance advantage for the recurrent neural networks. The experiments with delta features suggest that the conventional static neural network can be upgraded with dynamic classification capabilities. More experiments will be needed to find out which method eventually performs best in a real speech recognition system.

To aid future research into phoneme recognition, a Matlab toolbox (Timit_Tools) was created that enables the use of Timit speech data in Matlab.

Recommendations

Recommendations for future research on neural networks

Neural network structure selection
The more general a neural network architectural description is, the more structural choices are left to the user of the network. It should be investigated what guidelines should be followed to select a structure. Besides 'manual' selection, an algorithmic (automated) structure selection/adaptation procedure is also a possible option.

Neural network analysis
When a fully trained neural network is used to perform classification of data, the network 'interior' can be analyzed. For example, it may be possible to find out how each neuron is contributing to the classification. Another example is to analyze the meaning of the state in a state-space network. In other words: investigate *how* a neural network performs its task and not only how well.

Targets for neural networks trained for a classification task
In this report the most simple option was used for the training targets for a classification task: one/zero targets (see subsection 4.3.3 for possible target choices). The most interesting option for classification seems partial or weighted supervision, combined with double threshold targets. Within this project no experiments could be done yet to test such targets so it would be interesting to evaluate these on performance. Note that a different way of performance measurement (given below) is really needed when using partial supervision targets.

Recommendations for future research on neural network based speech recognition

Evaluating recurrent neural networks for speech recognition
To measure how recurrent networks perform in a real speech recognition task, the best thing to do would be to use these networks in an existing (possibly neural network-based) speech recognition system. The performance could then be compared to that of the original system, using non-recurrent networks.

Performance measurement in phoneme recognition experiments
Performance on phoneme recognition was measured using the rather simple frame error rate (defined in subsection 5.5.6). It has some disadvantages. Future experiments should use the standard phoneme INS/DEL/SUB (Insertion, Deletion, Substitution) errors system [Robinson, 1994] to be able to compare results with literature. The easiest way to implement this is to use an existing speech recognition system as suggested above.

Feature extraction
Of course a new speech recognition system can be created instead of using an existing one. In that case the feature extraction methods should be carefully chosen. Different phoneme types may need a very different treatment.

Phoneme recognition
In this report the assumption is made that phoneme recognition is used for any large vocabulary speaker independent speech recognition system. Phonemes are the 'basic units' of speech. In more recent years another 'basic unit' of speech has been investigated for use in speech recognition: the bi-phone. Bi-phones model all possible transitions from one phoneme to the next. Undoubtedly, other basic units have been used (e.g. tri-phones) and research still continues on the problem what features are best and on how humans recognize speech.

It is recommended that recent literature is investigated, to find out what basic unit is most appropriate for speech recognition.

Minor recommendations

Levenberg-Marquardt training algorithm for classification
It should be investigated whether the Levenberg-Marquardt algorithm is really appropriate for a classification task. In the experiments the LM algorithm did often stop early during training yielding a suboptimal weight vector, whereas the standard gradient descent algorithm did not make many suboptimal stops.

Gradient calculation routines can be tested with the finite differences method
Although the gradient calculation routines in the ModNet toolbox were tested by comparing them with Matlab NNet calculations (see Appendix E), the easiest method is calculating *finite differences* (i.e. making a small change in each weight, one at a time, and measuring the corresponding change in the error measure). It is strongly recommended that if new training algorithms are to be implemented, they are tested on correctness using this method. It can always be used, no matter how complex the network/algorithm.

Psycho-acoustic processing
As was stated in subsection 5.3.3, psycho-acoustic processing can be quickly tested in any speech recognition system. The performance increase may be worth the try.

REFERENCES

Baldi, P., (**1995**), *Gradient Descent Learning Algorithms: A Unified Perspective*, Book chapter in [Chauvin e.a., 1995]

Baltersee, J., Chambers, J.A., (**1998**), Nonlinear Adaptive Prediction of Speech with a Pipelined Recurrent Neural Network, *IEEE Transactions on Signal Processing*, Vol. 46 No. 8, pp. 2207-2216

Bengio, Y., Frasconi, P., Gori, M., Soda, G., (**1993**), Recurrent Neural Networks for Adaptive Temporal Processing, *Proc. of the 6th Italian workshop on Parallel Architectures and Neural Networks WIRN93*, pp. 85-117

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

Bengio, Y., (**1996**), *Neural Networks for Speech and Sequence Recognition*, International Thomson Computer Press, London, UK

Chauvin, Y., Rumelhart, D.E., (**1995**), *Backpropagation: Theory, Architectures and Applications*, Lawrence Erlbaum Associates Publishers, New Jersey, USA

Chen, R., Jamieson, L., (**1996**), *Experiments on the Implementation of Recurrent Neural Networks for Speech Phone Recognition*, Conference record: Asilomar Conference on Signals, Systems and Computers 1996, IEEE, New York, USA

Fritsch, J., Finke, M., (**1998**), *ACID/HNN: Clustering Hierarchies of Neural Networks for Context-Dependent Connectionist Acoustic Modeling*, Proceedings of ICASSP 1998, Vol. 1., Paper No. 1968

Frasconi, P., Gori, M., Maggini, M., Soda, G., (**1995**), Unified integration of explicit rules and learning by example in recurrent networks, *IEEE Transactions on Knowledge and Data Engineering*, pp. 340-346

Giles, C.L, Miller, C.B., (**1994**), *The effect of higher order in recurrent neural networks: experiments*, Book chapter in [Mammone, 1994]

Haykin, S., (**1998**), *Neural Networks: A comprehensive foundation*, second edition, Prentice Hall, New Jersey, USA

van der Heijden, F., (**1995**), *Image Based Measurement Systems*, Wiley & Sons, Chichester, UK

Hertz, J., Krogh, A., Palmer, R.G., (**1991**), *Introduction to theory of neural computation*, Addison-Wesley Publishing Company, CA, USA

Hove, W. ten, (**1996**), *Speech Recognition, Feature Extraction and the Artificial Neural Network of Kohonen*, , Report BSC 060N96, Signals & Systems group, Department of Electrical Engineering, University of Twente, Enschede, The Netherlands

Janssen, M.J., (**1998**), *Phoneme recognition with Neural Networks using transitional information*, Report S&S 040N98, Signals & Systems group, Department of Electrical Engineering, University of Twente, Enschede, The Netherlands

Kasper, K., Reininger, H., Wolf, D., Wust, H., (**1994**), *Fully Recurrent Neural Networks for Phoneme Based Speech Recognition*, Proceedings of EUSIPCO 1994, pp. 1705-1708

Koizumi, T., Mori, M., Taniguchi, S., Maruya, M., (**1996**), *Recurrent Neural Networks for Phoneme Recognition*, Proceedings of the International Conference on Spoken Language Processing (ICSLP) 1996, Vol. 1 (ThP1P1)

Kwakernaak, H., Sivan, R., (**1991**), *Modern Signals and Systems*, Prentice-Hall, New Jersey, USA

Lee, C.H., Soong, F.K., Paliwal, K.K., (**1996**), *Automatic Speech and Speaker recognition: Advanced Topics*, Kluwer Academic, USA

Lewis, F.L., Jagannathan, S., Yesildirek, A., (**1999**), *Neural Network Control of Robot Manipulators and Nonlinear Systems*, Taylor & Francis, London, UK

Lin, T., Horne, B.G., Giles, C.L., (**1998**), How embedded memory in recurrent neural network architectures helps learning long-term temporal dependencies, *Neural Networks* , Vol. 11, pp. 861-868

Lokerse, S.H., (**1995**), *Phoneme recognition with the Kohonen network*, Master's thesis Report no. BSC 064N95, Signals & Systems group, Department of Electrical Engineering, University of Twente, Enschede, The Netherlands

Mammone, R.J. , (editor), (**1994**), *Artificial Neural Networks for Speech and Vision*, Chapman & Hall, London, UK

McDonnel, J.R., (**1994**), Evolving Recurrent Perceptrons for Time-Series modeling, *Neural Networks* , Vol. 5 No. 1

Mozer, M.C., (**1995**), *A Focused Backpropagation Algorithm for Temporal Pattern Recognition*, Book chapter in [Chauvin e.a., 1995]

Nakagawa, S., Shikano, K., Tohkura, Y., (**1995**), *Speech, Hearing and Neural Network Models*, Ohmsha ltd., Tokyo, Japan

Patterson, D.W., (**1996**), *Artificial Neural Networks*, Prentice Hall, London, UK

Peelen, B.F., (**1999**), *Neural Dynamic Signature Verification system based on Stroke Velocity Profiles*, Master's thesis, Report no. S&S-NT 99N018, Signals & Systems group, Department of Electrical Engineering, University of Twente, Enschede, The Netherlands

Petridis, V., Kehagias, A., (**1996**), Modular Neural Networks for MAP Classification of Time Series and the Partition Algorithm, *Neural Networks*, Vol. 7 No. 1, pp. 73-86

Plataniotis, K.N, Androutsos, D., Venetsanopoulos, A.N., (**1997**), *A New Time Series Classification Approach*, Proceedings of ICASSP 1997, pp. 3345-3347

Rabiner, L., Juang, B.H., (**1993**), *Fundamentals of Speech Recognition*, Prentice Hall, New Jersey, US

Rissanen, J., (**1996**), *Information Theory and Neural Nets*, Book chapter in [Smolensky e.a., 1996]

Robinson, A.J., Fallside, F., (**1991**), A Recurrent Error Propagation Network Speech recognition system, *Computer Speech and Language*, Vol. 5 No. 3

Robinson, A.J., (**1994**), An Application of Recurrent Nets to Phone Probability Estimation, *Neural Networks*, Vol. 5 No. 2, pp. 298-305

Rumelhart, D.E, Durbin, R., Golden, R., Chauvin, Y., (**1995**), *Backpropagation: the basic theory*, Book chapter in [Chauvin e.a., 1995]

Santini, S., Del Bimbo, A., (**1995a**), Recurrent Neural Networks can be trained to be a Maximum A posteriori Probability Classifiers, *Neural Networks* , Vol. 8 No. 1, pp. 25-29

Santini, S., del Bimbo, A., Jain, R., (**1995b**), Block-Structured Recurrent Neural Networks, *Neural Networks*, Vol. 8 No. 1, pp. 135-147

Santini, S., del Bimbo, A., (**1995c**), Properties of Feedback Neural Networks, *Neural Networks*, Vol. 8 No. 4, pp. 579-596

Smolensky, P., Mozer, M.C., Rumelhart, D.E., (**1996**), *Mathematical Perspectives on Neural Networks*, Lawrence Erlbaum Associates, Inc., New Jersey, USA

Veelenturf, L.P.J, (**1995**), *Analysis and Applications of Artificial Neural Networks*, Prentice Hall International (UK) Ltd., UK

Weigend, A.S., (**1996**), *Time Series Analysis and Prediction*, Book chapter in [Smolensky e.a., 1996]

Wentink, M., (**1996**), *Identification of Dynamic Systems using Neural Networks*, Report BSC 067N96, Signals & Systems group, Department of Electrical Engineering, University of Twente, Enschede, The Netherlands

Williams, R.J., Zipser, D., (**1989**), Experimental Analysis of the Real-time Recurrent Learning Algorithm, *Connection Science*, Vol. 1, No. 1

Williams, R.J, Zipser, D., (**1995**), *Gradient-Based Learning Algorithms for Recurrent Networks and their computational complexity*, Book chapter in [Chauvin e.a., 1995]

Zamarreno, Vega, (**1998**), State space neural network. Properties and application, *Neural Networks*, Vol. 11, No. 6, pp. 1099-1112

Zwart, H., Nijmeijer, H., (**1996**), *Systeem- en besturingstheorie,* (Lecture notes), Department of Applied Mathematics, University of Twente, The Netherlands

Software

Software references are marked [<name>, <year>^{S}**]**

Matlab version 5.2, (**1997** ^{S}**)**

using Matlab Neural Network toolbox (NNet) version 3.0. Matlab name: 'nnet'.
Matlab Control System toolbox version 4.1: Matlab name: 'control'.
Matlab Statistics toolbox version 2.1.1. Matlab name: 'stats'.

Brookes, M., (**1998** ^{S}), VoiceBox toolbox for Matlab version 1.0, 1998. Matlab name: 'voicebox'. Website: http://www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html

Dijk, E.O., (**1999** ^{S}), ModNet modular state-space neural network toolbox version 1.0, 1999. Matlab name: 'modnet'.

Dijk, E.O., (**1999** ^{S}), TIMIT tools toolbox version 1.0, 1999. Matlab name: 'Timit_Tools'.

Janssen, M.J., (**1998** ^{S}), FIR neural network Matlab toolbox version **2.0**, 1998. Matlab name: 'fir'.

(The toolboxes except the NNet toolbox can be found in the user directory in users\dijk\matlab.)

APPENDIX A - ARCHITECTURES

A.1 SRN written as state-space networks

Here it is shown for two SRN structures that they can be written as state-space networks.

Definitions
The following vector functions are defined, that each describe the computation of the layer output $\mathbf{y}_{\cdot}(n)$ of the neural network given the input vector $\mathbf{x}_{\cdot\cdot}(n)$ to the layer at time n:

- input layer function $\mathbf{y}_I(n) = \mathbf{F}_I(\mathbf{u}(n))$
- hidden layer function $\mathbf{y}_H(n) = \mathbf{F}_H(\mathbf{x}_H(n))$
- output layer function $\mathbf{y}_O(n) = \mathbf{F}_O(\mathbf{x}_O(n))$
- context layer function $\mathbf{y}_C(n) = \mathbf{F}_C(\mathbf{x}_C(n))$

The state of all SRN is the input to the context layer $\mathbf{x}_C(n)$. α is the parameter vector of the feedback weights α_i.

SRN example a)
Given the definitions the computation by SRN example a) (figure 2.10a) can be written as:

$$\mathbf{y}_O(n) = \mathbf{F}_O\{\mathbf{F}_H[\mathbf{F}_C(\mathbf{y}_H(n-1)), \mathbf{F}_I(\mathbf{u}(n))]\} \tag{A.1}$$

This equation can be separated into a state-space process equation $\mathbf{F}_S(.)$ and an output equation $\mathbf{G}_S(.)$:

$$\begin{aligned} \mathbf{x}_C(n+1) &= \mathbf{F}_H[\mathbf{F}_C(\mathbf{x}_C(n)), \mathbf{F}_I(\mathbf{u}(n))] &&= \mathbf{F}_S(\mathbf{x}_C(n), \mathbf{u}(n)) \\ \mathbf{y}_O(n) &= \mathbf{F}_O\{\mathbf{F}_H[\mathbf{F}_C(\mathbf{x}_C(n)), \mathbf{F}_I(\mathbf{u}(n))]\} &&= \mathbf{G}_S(\mathbf{x}_C(n), \mathbf{u}(n)) \end{aligned} \tag{A.2}$$

SRN example b)

$$\mathbf{y}_O(n) = \mathbf{F}_O\{\mathbf{F}_H[\mathbf{F}_C(\mathbf{y}_H(n-1), \mathbf{y}_O(n-1)), \mathbf{F}_I(\mathbf{u}(n))]\} \tag{A.3}$$

This equation can be separated into a state-space process equation $\mathbf{F}_S(.)$ and an output equation $\mathbf{G}_S(.)$:

$$\begin{aligned} \mathbf{x}_C(n+1) &= \begin{bmatrix} \boldsymbol{\alpha} \cdot \mathbf{F}_H[\mathbf{F}_C(\mathbf{x}_C(n), \mathbf{F}_I(\mathbf{u}(n))] \\ \mathbf{F}_O\{\mathbf{F}_H[\mathbf{F}_C(\mathbf{x}_C(n), \mathbf{F}_I(\mathbf{u}(n))]\} \end{bmatrix} &&= \mathbf{F}_S(\mathbf{x}_C(n), \mathbf{u}(n)) \\ \mathbf{y}_O(n) &= \mathbf{F}_O\{\mathbf{F}_H[\mathbf{F}_C(\mathbf{x}_C(n), \mathbf{F}_I(\mathbf{u}(n))]\} &&= \mathbf{G}_S(\mathbf{x}_C(n), \mathbf{u}(n)) \end{aligned} \tag{A.4}$$

The two above state-space descriptions show that there is no clear separation between two networks (one that computes the process equation and one that computes the output equation), because several layers (like $\mathbf{F}_H$, $\mathbf{F}_C$ and $\mathbf{F}_I$) appear both in the process equation and in the output equation.

A.2 Two-layer RMLP cannot be described by the BFN framework

The two-layer RMLP structure (subsection 2.4.1) will be used as an example of a recurrent network that can not be described by the BFN framework (subsection 2.4.2).

The RMLP
The two-layer RMLP operates with the following equations (one for each layer):

$$\begin{aligned} \mathbf{y}_1(n) &= \mathbf{F}(\mathbf{A}_1\mathbf{y}_1(n-1) + \mathbf{B}_1\mathbf{u}_1(n)) \\ \mathbf{y}_2(n) &= \mathbf{F}(\mathbf{A}_2\mathbf{y}_2(n-1) + \mathbf{B}_2\mathbf{u}_2(n)) \end{aligned} \tag{A.5}$$

The connections of layers and external inputs/outputs impose the restrictions:

$$\begin{aligned}\mathbf{u}_1(n) &= \mathbf{u}_{\text{ext}}(n)\\ \mathbf{u}_2(n) &= \mathbf{y}_1(n)\\ \mathbf{y}_{\text{ext}}(n) &= \mathbf{y}_2(n)\end{aligned} \tag{A.6}$$

Now we have:

$$\begin{aligned}\mathbf{y}_1(n) &= \mathbf{F}(\mathbf{A}_1\mathbf{y}_1(n-1) + \mathbf{B}_1\mathbf{u}_{\text{ext}}(n))\\ \mathbf{y}_{\text{ext}}(n) &= \mathbf{F}(\mathbf{A}_2\mathbf{y}_{\text{ext}}(n-1) + \mathbf{B}_2\mathbf{y}_1(n))\end{aligned} \tag{A.7}$$

The output can also be written as a function of input and network state only:

$$\mathbf{y}_{\text{ext}}(n) = \mathbf{F}(\mathbf{A}_2\mathbf{y}_{\text{ext}}(n-1) + \mathbf{B}_2\{\mathbf{F}(\mathbf{A}_1\mathbf{y}_1(n-1) + \mathbf{B}_1\mathbf{u}_{\text{ext}}(n))\}) \tag{A.8}$$

BFN recurrent network

To model the recurrent connections of the RMLP, the BFN 'feedback' block is used. One feedback block realizes the equation:

$$\mathbf{y}(n) = \mathbf{N1}(\mathbf{F}(\mathbf{A}\cdot\mathbf{y}(n-1) + \mathbf{B}\cdot\mathbf{u}(n))) \tag{A.9}$$

where **N1**(.) the function realized by the embedded block N1.

Both recurrent connections of the RMLP must be modeled so two feedback blocks are needed. The first will be called N1 and the second N2. Block N2 is embedded into N1. Block N2 must also have an embedded block, but as no more recurrent connections are needed a 'dummy' block $N_3(\mathbf{u_3}(n)) = \mathbf{u_3}(n)$ is taken. The intermediate sums $s_i(n)$ of each block *i* are defined as the result of the computation **F**.

The three blocks compute the functions:

$$\begin{aligned}\mathbf{y}_1(n) &= \mathbf{N}_2\{\mathbf{s}_1(n)\} = \mathbf{N}_2\{\mathbf{F}(\mathbf{A}_1\mathbf{y}_1(n-1) + \mathbf{B}_1\mathbf{u}_1(n))\}\\ \mathbf{y}_2(n) &= \mathbf{N}_3\{\mathbf{s}_2(n)\} = \mathbf{N}_3\{\mathbf{F}(\mathbf{A}_2\mathbf{y}_2(n-1) + \mathbf{B}_2\mathbf{u}_2(n))\}\\ \mathbf{y}_3(n) &= \mathbf{s}_3(n) = \mathbf{u}_3(n)\end{aligned} \tag{A.10}$$

The connections of layers and external inputs/outputs impose these restrictions:

$$\begin{aligned}\mathbf{u}_1(n) &= \mathbf{u}_{\text{ext}}(n)\\ \mathbf{u}_2(n) &= \mathbf{s}_1(n)\\ \mathbf{u}_3(n) &= \mathbf{s}_2(n) = \mathbf{y}_2(n)\\ \mathbf{y}_{\text{ext}}(n) &= \mathbf{y}_1(n)\\ \mathbf{y}_1(n) &= \mathbf{y}_2(n)\end{aligned} \tag{A.11}$$

Now we have:

$$\begin{aligned}\mathbf{s}_1(n) &= \mathbf{F}(\mathbf{A}_1\mathbf{y}_{\text{ext}}(n-1) + \mathbf{B}_1\mathbf{u}_{\text{ext}}(n))\\ \mathbf{y}_{\text{ext}}(n) &= \mathbf{F}(\mathbf{A}_2\mathbf{y}_{\text{ext}}(n-1) + \mathbf{B}_2\mathbf{s}_1(n))\end{aligned} \tag{A.12}$$

The output can also be written as a function of input and network state only:

$$\mathbf{y}_{\text{ext}}(n) = \mathbf{F}(\mathbf{A}_2\mathbf{y}_{\text{ext}}(n-1) + \mathbf{B}_2\{\mathbf{F}(\mathbf{A}_1\underline{\underline{\mathbf{y}_{\text{ext}}}}(n-1) + \mathbf{B}_1\mathbf{u}_{\text{ext}}(n))\}) \tag{A.13}$$

The BFN recurrent network realizes a different function than the RMLP (equation 2.19). The difference is underlined in the above BFN equation. The above equation actually has only one feedback loop that goes to the two blocks N1 and N2.

APPENDIX B - LEARNING ALGORITHMS

B.1 Derivation of the BPTT algorithm for FRNN by unfolding the network in time

The derivation is based upon the unfolded network N_R^* as defined in equations 3.9 in section 3.4. The gradient was decomposed in equation 3.11, repeated here for convenience:

$$\frac{\partial E(n_0,n)}{\partial w_{ij}} = \sum_{m=n_0}^{n} \frac{\partial E(n_0,n)}{\partial w_{ij}(m)} \frac{\partial w_{ij}(m)}{\partial w_{ij}} = \sum_{m=n_0}^{n} \frac{\partial E(n_0,n)}{\partial w_{ij}(m)} \tag{B.1}$$

Now the expression for the partial derivatives $\partial E(n_0,n)/\partial w_{ij}(m)$ will be written down. Just like for any feedforward network, we can write for network N_R^*:

$$\frac{\partial E(n_0,n)}{\partial w_{ij}(m)} = \frac{\partial E(n_0,n)}{\partial y_i(m)} \frac{\partial y_i(m)}{\partial s_i(m)} \frac{\partial s_i(m)}{\partial w_{ij}(m)} \tag{B.2}$$

For easier notation, the following definitions and relations are introduced first:

$$\begin{aligned} \varepsilon_i(m) &= -\frac{\partial E(n_0,n)}{\partial y_i(m)} \\ \delta_i(m) &= -\frac{\partial E(n_0,n)}{\partial s_i(m)} \\ \frac{\partial y_i(m)}{\partial s_i(m)} &= f'(s_i(m)) \\ \delta_i(m) &= \varepsilon_i(m) \cdot f'(s_i(m)) \end{aligned} \tag{B.3a,b,c,d}$$

Now equation 3.7 can be written (using equations 3.9 and $\delta_i(.)$) as:

$$\frac{\partial E(n_0,n)}{\partial w_{ij}(m)} = -\delta_i(m) \frac{\partial s_i(m)}{\partial w_{ij}(m)} = -\delta_i(m) z_j(m) \tag{B.4}$$

Because the weights $w_{ij}(m)$ not only influence the output $y_i(m)$ but also future values of $\mathbf{y}(.)$ and thereby the cost function $E(n_0,n)$, the expression for $\varepsilon_i(m)$ is split up into two parts. The first part represents the explicit influence of the current outputs $y_i(m)$ on the cost function. The second part accounts for the implicit influence of $y_i(m)$ on the cost function through future values of $y_i(.)$ (which will depend on current output values $y_i(m)$ because of the recurrent connections).

To be able to split the single partial derivative $\varepsilon_i(m)$ into two parts, a bit of new notation must be introduced. Let $y_i^*(m)$ denote a new variable with values $y_i^*(m) = y_i(m)$ for all $m=n_0 \ldots n$ and $i=1 \ldots N$. Then the following expression denotes the explicit influence of the outputs $y_i(m)$ on the cost function:

$$\varepsilon_{i,Explicit}(m) = -\left.\frac{\partial E(n_0,n)}{\partial y_i(m)}\right|_{Explicit} = -\frac{\partial E(n_0,n)}{\partial y_i^*(m)} \frac{\partial y_i^*(m)}{\partial y_i(m)} \tag{B.5}$$

The implicit influence is 'lost' because the new variable $y_i^*(.)$ is not part of the network dynamics (equations 3.9). Next, the expression for the implicit influence of the outputs on the cost function is needed. Any such influence is exercised by means of the 'next' neuron outputs $y_i(m+1)$ so the expression can be decomposed into a partial derivative of $E(n_0,n)$ with respect to the elements of the output vector $\mathbf{y}(m+1)$:

$$\begin{aligned}\varepsilon_{i,Implicit}(m) &= -\left.\frac{\partial E(n_0,n)}{\partial y_i(m)}\right|_{Implicit} = -\sum_{l=1}^{N}\frac{\partial E(n_0,n)}{\partial y_l(m+1)}\frac{\partial y_l(m+1)}{\partial y_i(m)} = \\ &= -\sum_{l=1}^{N}\frac{\partial E(n_0,n)}{\partial y_l(m+1)}\frac{\partial y_l(m+1)}{\partial s_l(m+1)}\frac{\partial s_l(m+1)}{\partial y_i(m)}\end{aligned} \tag{B.6}$$

The expression for $\varepsilon_i(m)$ now becomes the sum of both the explicit and implicit terms. In the following equation the sum is calculated and all partial derivatives are substituted with the appropriate definitions from equations 3.25. The partial derivative $\partial s_l(m+1)/\partial y_i(m)$ is the weight $w_{li}(m+1)$ so we can write:

$$\begin{aligned}\varepsilon_i(m) &= \varepsilon_{i,Explicit}(m) + \varepsilon_{i,Implicit}(m) = \\ &= -\frac{\partial E(n_0,n)}{\partial y_i^*(m)}\cdot 1 - \sum_{l=1}^{N}(-\varepsilon_l(m+1))\cdot f'(s_l(m+1))\cdot w_{li}(m+1) = \\ &= e_i(m) + \sum_{l=1}^{N} w_{li}(m+1)\delta_l(m+1)\end{aligned} \tag{B.7}$$

Because outputs at times $m > n$ do not influence the cost function $E(n_0,n)$, it follows that the partial derivative expression $\varepsilon_i(m)$ must be zero (by definition) for $m=n+1$. So for $m=n$ equation 3.32 becomes $\varepsilon_i(m)=e_i(m)$. The weight copies $w_{ij}(m)$ will also be replaced by the original weights w_{ij} from recurrent network N_R. The result is the recursive relation:

$$\begin{aligned}\varepsilon_i(m) &= \begin{cases} e_i(m) & m = n \\ e_i(m) + \sum_{l=1}^{N} w_{li}\delta_l(m+1) & m < n \end{cases} \\ \delta_i(m) &= \varepsilon_i(m)\cdot f'(s_i(m))\end{aligned} \tag{B.8}$$

The weight adaptation at time n can now be written, using equations 2.28, 3.26 as:

$$\Delta w_{ij} = -\eta\frac{\partial E(n_0,n)}{\partial w_{ij}} = \eta\sum_{m=n_0}^{n}\delta_i(m)z_j(m) \tag{B.9}$$

The values of $\delta_i(.)$ required for the adaptation can be recursively computed using equation 3.29a,b in a so-called single *backward pass*. The following *forward pass* is the use of equation 3.30 to calculate the weight updates.

To obtain $z_j(n_0)$ initial conditions of the delay elements are set zero, $y_j(n_0-1) = 0$.

B.2 Derivation of the RTRL algorithm for FRNN using the ordered derivative

The RTRL algorithm can be derived using the ordered derivative. This method was also used in [Bengio, 1996], but the following derivation is a bit more clear and shows exactly when chain rules are applied.

The weight update can be calculated using the gradient (equal to equation 3.33):

$$\Delta w_{kl}(n) = -\eta\frac{\partial^+ E(n)}{\partial w_{kl}} \tag{B.10}$$

The ordered derivative is used, because all indirect influence of w_{kl} on the error measure has to be taken into account. In the following derivation, the second chain rule for ordered derivatives will be frequently used. Because the parameters w_{kl} of neuron k can only affect the error

measure indirectly through the outputs $y_i(n)$ of all other neurons (i=1..N), the second chain rule can be applied as follows:

$$\begin{aligned}\frac{\partial^+ E(n)}{\partial w_{kl}} &= \frac{\partial E(n)}{\partial w_{kl}} + \sum_{i=1}^{N} \frac{\partial E(n)}{\partial y_i(n)} \frac{\partial^+ y_i(n)}{\partial w_{kl}} = 0 + \sum_{i=1}^{N} \frac{\partial E(n)}{\partial y_i(n)} \frac{\partial^+ y_i(n)}{\partial w_{kl}} = \\ &= \sum_{i=1}^{N} -e_i(n) \frac{\partial^+ y_i(n)}{\partial w_{kl}}\end{aligned} \tag{B.11}$$

Applying the second chain rule again for the last term in the above equation

$$\begin{aligned}\frac{\partial^+ y_i(n)}{\partial w_{kl}} &= \frac{\partial y_i(n)}{\partial w_{kl}} + \frac{\partial y_i(n)}{\partial s_i(n)} \frac{\partial^+ s_i(n)}{d w_{kl}} = 0 + \frac{\partial y_i(n)}{\partial s_i(n)} \frac{\partial^+ s_i(n)}{\partial w_{kl}} \\ &= f'(s_i(n)) \frac{\partial^+ s_i(n)}{\partial w_{kl}}\end{aligned} \tag{B.12}$$

Again the second chain rule can be applied for the last term in the above equation. This time we use the fact that $s_i(n)$ only depends on the weight w_{kl} directly and on the extended input vector $\mathbf{z}(n)$ indirectly.

$$\begin{aligned}\frac{\partial^+ s_i(n)}{\partial w_{kl}} &= \frac{\partial s_i(n)}{\partial w_{kl}} + \sum_{j=1}^{N+M} \frac{\partial s_i(n)}{\partial z_j(n)} \frac{\partial^+ z_j(n)}{\partial w_{kl}} = \\ &= \delta_{ik} \cdot z_l(n) + \sum_{j=1}^{N} w_{ij} \frac{\partial^+ y_j(n-1)}{\partial w_{kl}} + \sum_{j=N+1}^{N+M} w_{ij} \cdot \frac{\partial^+ u_{j-N}(n-1)}{\partial w_{kl}} = \\ &= \delta_{ik} \cdot z_l(n) + \sum_{j=1}^{N} w_{ij} \frac{\partial^+ y_j(n-1)}{\partial w_{kl}} + 0\end{aligned} \tag{B.13}$$

where δ_{ik} denotes the Kronecker delta function.

The difference between the standard approach and the ordered derivative approach can be seen clearly in the last equation. The first term in the above equation, which is:

$$\frac{\partial s_i(n)}{\partial w_{kl}} = \delta_{ik} \cdot z_l(n) \tag{B.14}$$

has its own notation. Opposed to this, the same term in the (standard approach) equation 3.36 can not be written using a derivative notation but results directly from the application of the chain rule for partial derivatives. This equation is repeated here for convenience:

$$\begin{aligned}\frac{\partial s_i(n)}{\partial w_{kl}} &= \sum_{j=1}^{N+M} \frac{\partial (w_{ij} z_j(n))}{\partial w_{kl}} = \sum_{j=1}^{N+M} \left[w_{ij} \frac{\partial z_j(n)}{\partial w_{kl}} + z_j(n) \frac{\partial w_{ij}}{\partial w_{kl}} \right] = \\ &= \sum_{j=1}^{N+M} \left[w_{ij} \frac{\partial z_j(n)}{\partial w_{kl}} \right] + \underline{\underline{\delta_{ik} z_l(n)}}\end{aligned} \tag{B.15}$$

The term corresponding to 2.11 is underlined. The fact that using ordered derivatives allows for an easier notation of the algorithm, is the reason it is used for deriving RTRL for modular networks in subsection 3.5.3. It allows to notate the term 2.11 without having to fill in all the details of this expression right away, i.e. it can be left unspecified until the moment that choosing a specific neural network allows further development of the term.

Substituting the result 4.4 in equation 4.3, a recursive equation is obtained for calculating the ordered derivatives $\partial^+ y_i(m) / \partial w_{kl}$:

$$\frac{\partial^+ y_i(n)}{\partial w_{kl}} = f'(s_i(n))\left[\delta_{ik} \cdot z_l(n) + \sum_{j=1}^{N} w_{ij} \frac{\partial^+ y_j(n-1)}{\partial w_{kl}}\right] \tag{B.16}$$

Using these results in equation 4.2 provides the ordered derivative needed in 3.46 for calculating the weight update at time *n*. The above equation is identical to equation 3.38 that was derived using the standard partial derivative.

B.3 The Matlab/Elman learning algorithm for RNN

The Matlab Neural Network Toolbox (NNet) version 3.0 can model and train several neural network structures. One class of network structures is based on static feedforward networks, with delayed connections added to provide for dynamic networks (both time-delay networks and recurrent networks). These network architectures may have arbitrary connections between layers with or without a delay (of an arbitrary number of time steps). Recurrent connections from a layer to the layer itself, or from a layer to a previous layer, must have a minimum delay of one.

For each training function (for example standard gradient descent with backpropagation) a single algorithm is used in the toolbox to train all possible network architectures including standard feedforward networks, time delay networks and recurrent networks.

The question arises what learning algorithm is used to train the large variety of possible networks. The algorithm used in the NNet toolbox is called 'Elman backpropagation'. The documentation did not provide more information so the Matlab source code for the standard gradient descent/backpropagation learning algorithm (*traingd*) was examined.

Here the Elman algorithm will be examined further for the special case of a FRNN.

Relation with the RTRL algorithm

In the Elman algorithm at every time step the instantaneous error measure E(*n*) is used, not the total error over the training sequence $E_{TOTAL}(n_0,n)$. The computation of gradients is done in a single forward pass over all time steps. For every time step, only information of the current and previous time step are used to compute the gradients. The computed gradients are used to update the weights at the end of an epoch. In an epoch, all training data is presented to the network for all time steps.

These properties make the algorithm an *epochwise real-time* learning algorithm like epochwise RTRL (see section 3.2). So the Matlab/Elman algorithm can be conveniently compared with the epochwise RTRL algorithm.

Comparison with the RTRL algorithm

The Matlab algorithm can be developed in the same way as the RTRL algorithm, using the instantaneous error measure E(*n*). So both derivations are exactly the same up to a certain point, so this first part of the derivation is not repeated here. See section 3.4 for the derivation of the RTRL algorithm for FRNN. Recall the weight update can be calculated as (equation 3.43)

$$\Delta w_{kl}(n) = \eta \sum_{i=1}^{L} e_i(n) \pi_{kl}^i(n) \tag{B.17}$$

where the variable π obeys the recursive relation (equation 3.42):

$$\pi_{kl}^i(n) = f'(s_i(n)) \cdot \left[\sum_{j=1}^{N} w_{ij} \pi_{kl}^j(n-1) + \delta_{ik} z_l(n)\right] \tag{B.18}$$

The Matlab algorithm calculates both the above equations for every *n*, but the following approximation is made for the *inner* term π in the above equation:

$$\pi_{kl}^i(n-1) = \frac{\partial y_i(n-1)}{\partial w_{kl}} \approx f'(s_i(n-1)) \cdot \delta_{ik} z_l(n-1) \tag{B.19}$$

In this approximation all terms $\pi_{kl}^{i}(n\text{-}2)$ have been set to zero. By removing the Kronecker delta function δ_{ik} the result is:

$$\pi_{kl}^{k}(n-1)=\begin{cases}\dfrac{\partial y_k(n-1)}{\partial w_{kl}} \approx f'(s_k(n-1))\cdot z_l(n-1) & \text{for i = k}\\ 0 & \text{for i} \neq \text{k}\end{cases} \tag{B.20}$$

This approximation simplifies the weight update equation 2.5:

$$\begin{aligned}\pi_{kl}^{i}(n) &= f'(s_i(n))\cdot\left[w_{ik}\pi_{kl}^{k}(n-1)+\delta_{ik}z_l(n)\right]\\ &= f'(s_i(n))\cdot\left[w_{ik}f'(s_k(n-1))\cdot z_l(n-1)+\delta_{ik}z_l(n)\right]\end{aligned} \tag{B.21}$$

Assuming independent weights over time

Note that performing the approximation is equivalent to 'suddenly assuming' that all weights at all times are independent variables $w_{il}(n)$. This assumption would lead to the following expression for the partial derivative:

$$\begin{aligned}&\frac{\partial y_i(n)}{\partial w_{il}(n)} = f'(s_i(n))\cdot z_l(n) \Leftrightarrow\\ &\frac{\partial y_i(n-1)}{\partial w_{il}(n-1)} = f'(s_i(n-1))\cdot z_l(n-1)\end{aligned} \tag{B.22}$$

This expression is equal to the approximation in equation 2.6, so the approximation corresponds to the independent weights assumption.

The Matlab Elman algorithm

1. set the initial epoch to 1 ; set the initial weights
2. initialize the network (default values of zero but they can be chosen by the user)
3. operate the network for the epoch (times n=n_0…n_1).

calculate $\pi_{kl}^{i}(n)$ for all *i,k,l* and *n* using the following equation

$$\pi_{kl}^{i}(n) = f'(s_i(n))\cdot\left[w_{ik}f'(s_k(n-1))\cdot z_l(n-1)+\delta_{ik}z_l(n)\right] \tag{B.23}$$

4. use the values $\pi_{kl}^{i}(n)$ and the error $e_i(n)$ to calculate the weight update for all *n*:

$$\Delta w_{kl}(n)=\eta\sum_{i=1}^{L}e_i(n)\pi_{kl}^{i}(n) \tag{B.24}$$

5. Update all weights w_{kl}:

$$w_{kl}^{new}=w_{kl}^{old}+\sum_{n=n_0}^{n_1}\Delta w_{kl}(n) \tag{B.25}$$

6. Test if a stopping criterium has been reached.
7. Increase epoch and go back to step 2

MATLAB routines used in network training

The m-files Matlab uses to train a network (standard gradient descent) are:

- `traingd.m` ; contains the toplevel routine to train the network
- `calcgx.m` ; is called to calculate the gradients. This script only calls calcgrad.m.
- `calcgrad.m` ; performs the gradient calculations.

These files were examined to obtain information about the Matlab learning algorithm for recurrent neural networks.

APPENDIX C - CLASSIFICATION WITH RECURRENT NEURAL NETWORKS

C.1 Proof of MAP classification capability

Here it will be shown that state-space neural networks are capable of MAP classification of a data sequence. Comments on this proof will be given at the end.

Definitions
In the following discussion, $\mathbf{u}(n)$ is defined as the neural network input vector of size M, and $\mathbf{y}(n)$ is the output vector of size L. More definitions on MAP classification were given in section 4.2.

MAP Classification by static neural networks
Summarizing the MAP classification capabilities for static networks, a static network N_0 can perform the MAP classification function of a single input vector

$$P(\omega_i|\mathbf{u}(n)) \tag{C.1}$$

Using k previous inputs, grouped in an extended input vector $\mathbf{U_k}(n)$ of size k*M, as the input of a bigger static neural network the same argument applies and MAP classification of a sequence of length k is possible by a static network N_k:

$$p(\omega_i|\mathbf{U}_k(n)) = p(\omega_i|\mathbf{u}(n),...,\mathbf{u}(n-k)) \tag{C.2}$$

MAP classification of sequences by recurrent state-space neural networks
Now the classification possibilities of recurrent state-space neural networks will be looked at. Define the recurrent network R_1 with two modules, F and G.

When the size of the state vector is chosen equal to that of the input vector and the following trivial function for network F is used:

$$\mathbf{x}(n+1) = \mathbf{F}(\mathbf{x}(n),\mathbf{u}(n)) = \mathbf{u}(n) \tag{C.3}$$

the RNN module G effectively has both $\mathbf{u}(n)$ and $\mathbf{u}(n\text{-}1)$ as input at each time n :

$$\mathbf{y}(n) = \mathbf{G}(\mathbf{x}(n),\mathbf{u}(n)) = \mathbf{G}(\mathbf{x}(n-1),\mathbf{u}(n)) \tag{C.4}$$

so it can realize the MAP classification

$$P(\omega_i|\mathbf{u}(n),\mathbf{u}(n-1)) \tag{C.5}$$

of a sequence of length 2. The network R_1 can therefore classify equivalently to static network N_1, which can classify a longer sequence than static network N_0.

This argument can be extended: now define a recurrent network R_k. In general, choosing a state vector of size k*M, the state vector can hold at each time n:

$$\mathbf{x}(n) = \begin{bmatrix} \mathbf{u}(n-1) \\ \mathbf{u}(n-2) \\ \vdots \\ \mathbf{u}(n-k) \end{bmatrix} \tag{C.6}$$

If the function F is now chosen to perform the following function that merely 'shifts' the state vector and adds a new input $\mathbf{u}(n)$, such that a memory of k past inputs can be kept:

$$\mathbf{x}(n+1) = \mathbf{F}(\mathbf{x}(n), \mathbf{u}(n)) = \mathbf{F}(\begin{bmatrix} \mathbf{u}(n-1) \\ \mathbf{u}(n-2) \\ \vdots \\ \mathbf{u}(n-k) \end{bmatrix}, \mathbf{u}(n)) = \begin{bmatrix} \mathbf{u}(n) \\ \mathbf{u}(n-1) \\ \vdots \\ \mathbf{u}(n-k+2) \\ \mathbf{u}(n-k+1) \end{bmatrix} \tag{C.7}$$

then the static module G of network R_k can perform MAP classification of a sequence of k+1 input vectors:

$$y(n) = G(\mathbf{x}(n), \mathbf{u}(n)) = P(\omega_i | \mathbf{x}(n), \mathbf{u}(n)) = P(\omega_i | \mathbf{u}(n), ..., \mathbf{u}(n-k)) \tag{C.8}$$

It is now proven that a recurrent neural network R_k constructed this way is capable of MAP classification of a sequence:

$$P(\omega_i | \mathbf{u}(n), ..., \mathbf{u}(n-k)) \tag{C.9}$$

Comments on the proof

Recurrent neural networks are capable of MAP classification. The above argumentation is used by [Santini e.a., 1995a] to prove that a RNN can be trained to be a MAP classifier of sequences. Summarizing: a large state vector is constructed such that it holds all relevant past input samples needed for classification and then a static neural network G module can use this information to perform a MAP sequence classification.

The module G of network R_k receives exactly the same input as the static network N_k defined earlier, and has to compute exactly the same function as network N_k. So in this case, the recurrent network is just a redundant and cumbersome implementation of the static network N_k.

Therefore in practice the use of such a proof is very limited, because the proof also points out that an equivalent static neural network N_k is a much simpler structure that can perform the same classification. As a rationale for using recurrent networks for sequence classification this proof is not useful.

Also, the value of k is fixed by choosing the recurrent network structure. This contradicts with the supposed advantage of recurrent networks to classify variable length sequences.

APPENDIX D - TIMIT SPEECH DATABASE

D.1 Phoneme definitions

The following information is used from [Janssen, 1998]. All sentences in the TIMIT-database are labeled with a set of 62 possible phonetic labels, according to the ASCII CMU/ARPAbet representation of the standard International Phonetic Association (IPA) symbol.

Phone	Example	Phone	Example	Phone	Example
/iy/	b**ea**t	/er/	b**ir**d	/z/	**z**oo
/ih/	b**i**t	/axr/	din**er**	/zh/	mea**s**ure
/eh/	b**e**t	/el/	bott**le**	/v/	**v**ery
/ae/	b**a**t	/em/	yes'**em**	/f/	**f**ief
/ux/	b**eau**ty	/en/	butt**on**	/th/	**th**ief
/ix/	ros**e**s	/eng/	Wash**ing**ton	/s/	**s**is
/ax/	th**e**	/m/	**m**om	/sh/	**sh**oe
/ah/	b**u**tt	/n/	**n**on	/hh/	**h**ay
/uw/	b**oo**t	/ng/	si**ng**	/hv/	Le**h**eigh
/uh/	b**oo**k	/ch/	**ch**urch	/pcl/	(p closure)
/ao/	b**ough**t	/jh/	**j**udge	/tcl/	(t closure)
/aa/	c**o**t	/dh/	**th**ey	/kcl/	(k closure)
/ey/	b**ai**t	/b/	**b**ob	/qcl/	(q closure)
/ay/	b**i**te	/d/	**d**ad	/bcl/	(b closure)
/oy/	b**oy**	/dx/	bu**tt**er	/dcl/	(d closure)
/aw/	ab**ou**t	/nx/	(flapped n)	/gcl/	(g closure)
/ow/	b**oa**t	/g/	**g**ag	/epi/	(epin. clos.)
/l/	**l**ed	/p/	**p**op	/h#/	(begin sil)
/r/	**r**ed	/t/	**t**ot	/#h/	(end sil)
/y/	**y**et	/k/	**k**ick	/pau/	(betw. sil)
/w/	**w**et	/q/	(glot. stop)		

Table D.1: TIMIT phonemes

Corrections
During work with the Timit database it was found that one more phoneme label existed that was not listed. This is the /ax-h/ or /axh/ phoneme. So there are 63 phoneme labels.

Phoneme reduction schemes
Some of the phonemes in this table can not be distinguished without knowing the neighboring phonemes. For example all the phonemes /pcl/, /tcl/, /kcl/ and /qcl/ are in fact silence but are labeled this way because the phones /p/, /t/, /k/ or /q/ are preceded by those phones. To allow single-phoneme recognition, the following simplifications are made:

Old	New	Old	New	Old	New
/dx/	/d/	/eng/	/ng/	/bcl/	/vcl/
/ux/	/uw/	/hv/	/hh/	/dcl/	/vcl/
/el/	/l/	/pcl/	/cl/	/gcl/	/vcl/
/axr/	/er/	/tcl/	/cl/	/h#/	/sil/
/em/	/m/	/kcl/	/cl/	/#h/	/sil/
/en/	/n/	/qcl/	/cl/	/pau/	/sil/
/nx/	/n/	/q/	/cl/		

Table D.2: Substitutions

Corrections
According to the information used, this reduces the number of phonemes from 62 to 42. But one new label /vcl/ is added, so this calculation is not entirely correct. The reduced number of phonemes is 63 - 20 + 1 = 44.

Second reduction scheme
There is a second reduction scheme that reduces the number of phonemes to 36 (this number will probably not be correct, like the preceding phoneme counts). This scheme is not shown here. It may be found in other reports.

D.2 The Timit tools toolbox

The Matlab functions that were available (from [Janssen, 1998] and others) to handle the Timit database data, were collected in the Timit_Tools toolbox [Dijk, 1999[S]]. The following functions are provided:

`readph`	Read phonemes from a Timit *.phn file
`reads`	Reads a Timit sampled sound wave *.wav file (Timit uses a custom NIST Sphere-headered wav format)
`readsents`	Read one or more sentence(s) together with phonemes from one or more speaker(s) in one go.
`timit_test1`	Demo that uses above functions.
`timit_test2`	Demo of using *readsents*.

Warning: preliminary version
There were some errors found in the *readph* function in the phoneme reduction schemes and they were inconsistent with the tables given in section 4.1.7. A careful look at these reduction schemes is needed to verify that they are correct. The phonemes used in this report (non-stationary vowels only) are not influenced by the reduction schemes and by the possible errors therein. So the Timit_Tools toolbox described here should be considered a preliminary release.

D.3 Data used for the phoneme recognition experiment

The speakers/sentences used for the phoneme recognition experiment (see section 5.7) are shown here. The following Matlab code is used to load the sentences (five male speakers, ten sentences per speaker):

```
timpath='m:\timit\timit\train' ;
%select dialect region
speakerpath='dr1';
%sample info
fs=16000; bits=16;
fullpath = [timpath '\' speakerpath] ;
% make an array of all 10 sentences per speaker.
speakers={'mdac0','mcpm0','mdpk0','medr0','mgrl0'};
sent{1}={'sa1','sa2','si1261','si1837','si631','sx181','sx271','sx361','sx451','sx91'};
sent{2}={'sa1','sa2','si1194','si1824','si564','sx114','sx204','sx24','sx294','sx384'};
sent{3}={'sa1','sa2','si1053','si1683','si552','sx153','sx243','sx333','sx423','sx63'};
sent{4}={'sa1','sa2','si1374','si2004','si744','sx114','sx204','sx24','sx294','sx384'};
sent{5}={'sa1','sa2','si1497','si2127','si867','sx147','sx237','sx327','sx417','sx57'};
% load files
[mstart_end,mph,first,last,y]=readsents(fullpath,speakers,sent,0,2,1);
```

APPENDIX E - THE MODNET TOOLBOX

The Matlab toolbox that was developed to train and simulate (modular) state-space neural networks is described in this appendix. The toolbox is called the *ModNet* toolbox [Dijk, 1999[S]]. The state-space neural network model was introduced in subsection 2.2.1 and discussed in a modular neural network context in subsection 3.5.1. The modular description was used to develop training algorithms and as a guideline to develop the Matlab code. The motivation for this approach was given in subsection 3.8.2.

This appendix will give an overview of the ModNet toolbox (in 2.2.3) and will show the implementation of the algorithms in Matlab pseudo-code (in 3.6.1). Pseudo-code means that only the most important parts of the basic algorithm are shown so that the code can be easily understood. The code shown therefore does not really work in Matlab. For the full code, see the ModNet toolbox m-files. Some verification experiments that were done verify the correctness of some aspects of the training algorithms, are described in section 2.2.4.

E.1 ModNet toolbox overview

This section aims to provide an overview of the ModNet toolbox. It will give:

- an overview of supported neural network structures (3.6.2) ;
- a list of toolbox functions (1) ;
- a description of the initialization functions (5.3.5) ;
- a description of the training functions (2.4.1) ;
- a description of the interfacing functions (2.4.2) ;
- a description of other functions (2.3).

E.1.1 Overview of supported neural network structures

The toolbox currently supports the state-space neural network (and its subtypes), with any one-layer or two-layer perceptron inside each module. (This is easily extendable to N-layer networks.) By pruning and/or fixing connections, the state-space Simple Recurrent Network (SRN) and the type 2 FRNN (and its subtypes) can be trained and simulated.

By choosing one-layer networks for both modules the partially recurrent network (PRN) is obtained.

E.1.2 Functions list

The toolbox provides the following functions.

Network Initialization functions

initmodnet	- initialize a state-space neural network
initsrn	- initialize a state-space Simple Recurrent Network (SRN)
initfrnn	- initialize a type 2 Fully Recurrent Neural Network (FRNN)

Network training functions

trainmodnet	- train a modular neural network (several algorithms)

Interfacing functions to other Neural Network toolboxes

modnet2nnt	- convert ModNet training set to NNT 3.0 format/FIR neural network toolbox format
nnt2modnet	- convert NNT 3.0/FIR neural network toolbox training set to ModNet format

Documentation functions

modnet_help	- explains data structures used
modnet_rev	- shows revision history

Demo functions

modnet_demo1	- trains several networks on a linear system identification task.
modnet_demo2	- use a state-space network for identification of the nonlinear Hénon system.

For all functions more details about their use can be found in Matlab (by using *help* <function>). The toolbox uses specific functions and data structures from the Matlab NNet toolbox version 3.0 [Matlab, 1997[S]]. Other versions of the NNet toolbox may not work.

A global overview of the capabilities of all the functions is given in the remainder of this section.

E.1.3 Initialization functions

initmodnet
Initializes a state-space neural network. The two static modules should be constructed by the user, using the standard NNet toolbox commands. The weights of both modules are initialized in this function, using the NNet toolbox *init* command.

initsrn
Initializes a state-space SRN. The only difference from a true state-space neural network is that the feedback weight matrix (from module 1 outputs to module 1 inputs) is forced to be diagonal. (In Matlab terms: the feedback weight matrix is the left part of size N_S-by-N_S of the input weights matrix of module 1 `net1.IW{1}`, with N_S the state vector size.) The off-diagonal weights that were first set zero are *kept zero* by using the local learning rate feature of the ModNet toolbox.

initfrnn
Initializes a type 2 FRNN. The FRNN structure can be obtained out of a state-space network by removing module 2 and using some or all of the module 1 outputs directly as external outputs. The function initfrnn does in fact not remove the module 2, but makes it a linear 'dummy' module with an output vector equal to the input, $\mathbf{y}(n)=\mathbf{u}(n)$. Using the local learning rate feature all weights of the linear module 2 are fixed. Effectively, module 2 is now disabled. Module 1 now acts alone as a FRNN.

E.1.4 Network training

Modular neural networks are trained by the *trainmodnet* function.

Training algorithms
The following algorithms are implemented within *trainmodnet*.

Matlab algorithm identifier	**Algorithm**
`rtrl`	Real-time Recurrent Learning (RTRL)
`rtrl_epoch`	Epochwise RTRL
`bptt_epoch`	Epochwise BPTT
`bptt_vt`	Epochwise BPTT using the Virtual Targets (VT) approach (see subsection 3.6.1)

The Matlab implementation of these algorithms is shown in more detail in section 3.6.1.

Training functions
Every algorithm can make use of different training functions (not to be confused with the term training function as in 'the trainmodnet training function'). These functions modify the basic training algorithm. The following training functions are provided:

Matlab training function identifier	Meaning
`traingd`	Standard gradient descent
`traingda`	traingd with adaptive learning rate
`traingdm`	traingd with momentum learning
`traingdx`	traingd with both adaptive learning rate and momentum learning
`trainlm`	the Levenberg-Marquardt (LM) algorithm. This function actually modifies the algorithm from *bptt_epoch* to the separately implemented *bptt_lm*. Note that RTRL can not be used with LM, as noted in subsection 3.6.1.

These training functions have the same name as their NNet toolbox equivalents. One can issue 'help <function>' in Matlab to get an idea of the function.

Validation set
Besides the training set a validation set can be supplied to the algorithm. Training will stop whenever the error on the validation set has reached a minimum. See the Matlab NNet toolbox or subsection 5.5.4 for more information on validation sets.

Local learning rate feature
If the local learning rate feature is used, each weights can have its own unique learning rate multiplier. During training, the update for each weight is first multiplied by this local learning rate value before it is applied. This opens up the possibility of fixing weights (see subsection 2.2.3) by setting certain local learning rates to zero (then, the weights can not change value anymore so they can not be trained). This weight fixing is used in the ModNet toolbox for obtaining subsets of the state-space network, like the FRNN and the state-space SRN, out of the full state-space network. This feature could also be used to obtain other custom neural network structures (for an example see again subsection 2.2.3).

Weighted supervision feature
The toolbox can use the weighted supervision technique which was briefly explained in subsection 4.3.3. To use weighted supervision, for every target matrix in the training set an equally-sized supervision matrix must be supplied that assigns a weighting constant to each corresponding scalar target value.

E.1.5 Interfacing functions

Two interfacing functions are provided to exchange training sets between the ModNet toolbox and the Matlab NNet toolbox. Both the NNet matrix form of training sets and the cell-array form are supported. See the NNet toolbox for more information about the matrix form (for static networks only) and the cell-array form (for dynamic networks).

E.1.6 Other functions

Other functions provided are demos, documentation functions and utility functions. See the ModNet documentation for more information.

E.2 Implementation of the algorithms

The algorithms are computed by the routines *calc_bptt* and *calc_rtrl*. These are shown in subsections 1.1 and E.2.2 respectively.
The gradients for each static module are calculated in separate m-files, which are shown in subsection 3.1.1.

E.2.1 BPTT

```
% calc_bptt
%
% ! NOTE !
% THIS IS THE MATLAB 'PSEUDO-CODE   LISTING. IT WILL BEST EXPLAIN THE
% WORKING AND IMPLEMENTATION OF THE RTRL ALGORITHM. FOR THE FULL
% SOURCE, SEE ModNet\calc_bptt.m !
%
% Inputs: P – input patterns in TS-by-Ninp matrix
%         T – targets in TS-by-Noutp matrix
%         weights – neural network weights
%         modnet – modular network (excluding newest weights)
%         Ai – initial state vector for network
%         epoch – current epoch nr.
%         <other_parameters> - see calc_bptt.m
%             this includes numLayers, numInputs, numOutputs, numNeurons,
%              numNeuronInputs, numWeights
% where
%         Ninp  - number of external inputs to network
%         Noutp – number of external outputs to network
%
%
function [modnet,perf,normGradient,weights] = calc_bptt(modnet,P,T,S,Ai,weights,tp,tc,
epoch)

f                     = modnet.f ;
%---Constants
module1 = 1;
module2 = 2;
% this defines the ext. inputs, which have the state vector as inputs
stateRange = [1 numOutputs(module1)] ;
lastLayer2 = numLayers(2);
lastLayer1 = numLayers(1);

%---get nr of time steps TS of example sequence
 [TS numExtInputs] = size(P);

%----Initialize the deltaw_* etc. to zeros
E1 = zeros( TS+1, numOutputs(1));
E2 = zeros( TS+1, numOutputs(2));
deltaw1 = zeros( 1, numWeights(1));
deltaw2 = zeros( 1, numWeights(2));
if (epoch > 1)
 deltaw_prev1 = modnet.system.deltaw1; % get previous deltaw out of modnet (if epoch>1)
 deltaw_prev2 = modnet.system.deltaw2;
else
   deltaw_prev1 = deltaw1;                  % epoch=1, so fill the deltaw_prev with
initial zeros
   deltaw_prev2 = deltaw2;
end
deltaw_total1= deltaw1;
deltaw_total2= deltaw2;

%---get weight vectors
w1  = weights{1};
w2  = weights{2};

%--------------SIMULATE NETWORK FOR ENTIRE SEQUENCE-------------
state = Ai;
for ts=1:TS
   P_now = P(ts,:)';
   P_ext(ts,:) = [ state ; P_now ;1]';       % create an extended input vector
   [Y2{ts},state,Y1{ts},S1{ts},S2{ts},dYdS1{ts},dYdS2{ts}] = simmodnet_single_c(P_now,
state,w1,w2,f,<modnet_info>);
```

```
    % collect the last layer output (external output)
   Yout2(ts,:) = Y2{ts}(lastLayer2,1:numOutputs(2));
   Yout1(ts,:) = Y1{ts}(lastLayer1,1:numOutputs(1));
end
%--Error calculation: calculate error vector (from module 2 output and target)
E_ts = (T - Yout2) ;
%--------------END: SIMULATE NETWORK FOR ENTIRE SEQUENCE-------------

%-----------LOOP BACKWARDS OVER TIME-----------------
%--First for timestep TS, the 'trivial' case
E2(TS,:) = - E_ts(TS,:) ;
%E1 is already zero for t=TS

%---Then loop back for other times < TS
for ts=[ (TS-1):-1: 1 ]

   %----Calc E2 for module 2
   E2(ts,:) =  -E_ts(ts,:) ;

   % Calculate dYdY11 and dYdY21
   dYdY11 = calc_dYdX(module1, w1, dYdS1{ts}, stateRange,<modnet_info>);
    dYdY21 = calc_dYdX(module2, w2, dYdS2{ts}, stateRange,<modnet_info>);

   %----Calc E1 for module 1
    E1(ts,:) = E1(ts+1,1:numOutputs(module1)) * dYdY11 + ...
               E2(ts+1,1:numOutputs(module2)) * dYdY21 ;
end %-----------END LOOP OVER TIME-----------------

% ------------------------------------------------
% ---- BPTT Algorithm
%
%--------------LOOP FORWARDS OVER TIME-------------
% to calc the weight adjustments deltaw using dYdW gradients
for ts=1:TS
    %----Calc dYdW and adaptations deltaw using E1/E2, of module 2..
    dYdW2 = calc_dYdW(module2, w2, dYdS2{ts}, Y2{ts}, P_ext(ts,:)', <modnet_info>);
    deltaw2 = deltaw2 - E2(ts,:) * dYdW2 ;
    % ..for module 1
    dYdW1 = calc_dYdW(module1, w1, dYdS1{ts}, Y1{ts}, P_ext(ts,:)', <modnet_info>);
    deltaw1 = deltaw1 - E1(ts,:) * dYdW1 ;
end
%-----------END LOOP OVER TIME-----------------

%----Return performance and norm of gradient
deltaw_total1 = deltaw1;
deltaw_total2 = deltaw2;
normGradient=sqrt(sum(sum(deltaw_total1.^2)) + sum(sum(deltaw_total2.^2)) );
normGradient1=sqrt(sum(sum(deltaw_total1.^2)) );
normGradient2=sqrt(sum(sum(deltaw_total2.^2)) );

%----Adapt weights module 1&2
w1 = w1 + lr * deltaw1 ;
w2 = w2 + lr * deltaw2 ;

%----Convert vector back to weight cell array, to return
weights{1} = w1;
weights{2} = w2;

% return calculated performance
perf=calc_perf3(modnet.performFcn, E_ts, Yout2, T );

% put delta into modnet
modnet.system.deltaw1 = deltaw1;
modnet.system.deltaw2 = deltaw2;

%end
```

E.2.2 RTRL

```
% Calc_Rtrl – calculates RTRL algorithm for one example sequence
%
% ! NOTE !
% THIS IS THE MATLAB 'PSEUDO-CODE   LISTING. IT WILL BEST EXPLAIN THE
% WORKING AND IMPLEMENTATION OF THE RTRL ALGORITHM. FOR THE FULL
% SOURCE, SEE ModNet\calc_rtrl.m !
%
% Inputs: P – input patterns in TS-by-Ninp matrix
%         T – targets in TS-by-Noutp matrix
%         weights – neural network weights
%         modnet – modular network (excluding newest weights)
%         Ai – initial state vector for network
%         epoch – current epoch nr.
%         <other_parameters> - see calc_rtrl.m
%             this includes numLayers, numInputs, numOutputs, numNeurons,
%              numNeuronInputs, numWeights
% where
%         Ninp  - number of external inputs to network
%         Noutp – number of external outputs to network
%

function [perf,normGradient,weights] = calc_rtrl(modnet,P,T,Ai,weights,epoch,
<other_parameters>)

%----Constants
module1 = 1;
module2 = 2;

%----Init
f   = modnet.f;
w1  = weights{1};
w2  = weights{2};
lastLayer2 = numLayers(2);
% this defines which module inputs have the state vector as inputs
stateRange = [1 numOutputs(module1)] ;

%----get nr of time steps TS of example sequence
 [TS numExtInputs] = size(P);

%----Initialize the deltaw_* etc. to zeroes
PI_prev11   = zeros( numOutputs(1), numWeights(1) ) ;
PI_prev21   = zeros( numOutputs(2), numWeights(1) ) ;
deltaw1 = zeros( 1, numWeights(1));
deltaw2 = zeros( 1, numWeights(2));
deltaw_prev1 = deltaw1;
deltaw_prev2 = deltaw2;
deltaw_total1= deltaw_prev1;
deltaw_total2= deltaw_prev2;

%---set init. state
state = Ai;

%--Begin algorithm:
%-----------LOOP OVER TIME-----------------
for ts=[1:TS]
   T_now  = T(ts,:);
   P_now  = P(ts,:)';            % this is the current input
   P_ext = [ state ; P_now ;1];   % create the extended input vector

   % ----Simulate network for 1 timestep with pattern P_now
   [Y2,state,Y1,S1,S2,dYdS1,dYdS2] = simmodnet_single_c(P_now,state,w1,w2,
<modnet_structure> );

   % ----calculate error vector (from module 2 output and target)
   E_now  = T_now - Y2(lastLayer2,1:numOutputs(2));
   E_ts(ts,:) = E_now ;   % store errors over time in E_ts

    %----Calculate dYdY11 and dYdW1 and PI11 for module 1
   dYdY11 = calc_dYdX(module1, w1, dYdS1, stateRange, <modnet_structure>);
   dYdW1 = calc_dYdW(module1, w1, dYdS1, Y1, P_ext, <modnet_structure>);
    %---equation 3.78
    PI11 = dYdW1 + dYdY11 * PI_prev11 ;
```

```
    %----Calc dYdY21, dYdW2, and PI21 for module 2
    dYdY21 = calc_dYdX(module2, w2, dYdS2, stateRange, <modnet_structure>);
    [dYdW2] = calc_dYdW(module2, w2, dYdS2, Y2, P_ext,<modnet_structure>);
    %---equation 3.77
   PI21 = dYdY21 * PI_prev11 ;
   PI22 = dYdW2;                  % equation 4.7
   %---Calc adaptations module 1&2 , equation 3.80
    deltaw1 = E_now * PI21  ;
    deltaw2 = E_now * PI22 ;

    %----Adapt weights module 1/2
    w1 = w1 + lr * deltaw1 ;
    w2 = w2 + lr * deltaw2 ;

   %----Save some previous values of variables
   deltaw_prev1    = deltaw1;                    % prev. weights of module 1
   deltaw_prev2    = deltaw2;
   PI_prev11       = PI11;                       % all previous PI values

end %-----------LOOP OVER TIME-----------------

%----Return performance and norm of gradient
perf           =feval(modnet.performFcn, E_ts);
normGradient   =sqrt(sum(sum(deltaw_total1.^2)) + sum(sum(deltaw_total2.^2)) );
normGradient1  =sqrt(sum(sum(deltaw_total1.^2)) );
normGradient2  =sqrt(sum(sum(deltaw_total2.^2)) );

%--return last state
Af=state;

%--return weights
weights = {w1 w2};
```

E.2.3 Gradient calculations routines (calc_dYdW, calc_dYdX)

calc_dYdW

The function calc_dYdW calculates the partial derivatives of the network outputs Y with respect to all weights W.

```
function [dYdW] = calc_dYdW(module, w, dYdS, Y, P_ext, numNeurons, numNeuronInputs,
numOutputs, numLayerWeights, numLayers, numWeights)
%CALC_dYdW
%
weights1 = get_layerWeights( w, 1, numLayerWeights(module,:), numNeurons(module,:),
numNeuronInputs(module,:) );
if ( numLayers(module) >= 2 )
   weights2 = get_layerWeights( w, 2, numLayerWeights(module,:), numNeurons(module,:),
numNeuronInputs(module,:) );
end

dYdW = zeros( numOutputs(module) , numWeights(module) );

%---1 Layer network
if ( numLayers(module) == 1)
   lay=1; pos=1; Psize = numNeuronInputs(module,lay);
   for q = 1:numOutputs(module)
      dYdW(q, pos:pos+Psize-1 ) = dYdS(lay,q) .* P_ext(1:Psize)' ;
      pos = pos + Psize ;
   end %q
end   %if numLayers

%---2 Layer network
if (numLayers(module) ==2)
   k=1;
   %----------------LOOP over all parameters of module -----------------------
   for lay = 1:numLayers(module)
      %----Layer 1 of 2
      if (lay==1)
         for neur = 1:numNeurons(module,lay)
            for neur_input = 1:numNeuronInputs(module,lay)
               for l=1:numOutputs(module)
                  dYdW(l,k) = dYdS(lay+1,l) .* weights2(l,neur) ...
```

```
                    .* dYdS( lay, neur ) .* P_ext(neur_input );
               end %l
               k=k+1;
            end %neur_input
         end %neur
      %----Layer 2 of 2
      elseif (lay==2)
          for neur = 1:numNeurons(module,lay)
            for neur_input = 1:numNeuronInputs(module,lay)
               dYdW(neur,k) = dYdS(lay,neur) * Y(1,neur_input) ;
               k=k+1;
            end %neur_input
         end %neur
      end %if lay==2
   end %lay
   %----------------END LOOP over all parameters of module -----------------------
end % if numLayers
```

calc_dYdX

The function calc_dYdX calculates the partial derivatives of the network outputs Y with respect to all network inputs X.

```
function [dYdX] = calc_dYdX(module, w, dYdS, WhichInputs, numNeurons, numNeuronInputs,
numOutputs, numLayerWeights, numLayers, isBias)
%CALC_dYdX
%
lay1=1;
lay2=2;

weights1 = get_layerWeights( w, 1, numLayerWeights(module,:), numNeurons(module,:),
numNeuronInputs(module,:) );
if ( numLayers(module) >= 2 )
   weights2 = get_layerWeights( w, 2, numLayerWeights(module,:), numNeurons(module,:),
numNeuronInputs(module,:) );
end

%----Get from the WhichInputs argument: the start and end values of the range of module
%   external inputs X for which dYdX has to be calculated
numStartInput  = WhichInputs(1);
numEndInput    = WhichInputs(2);
numInputs_X    = numEndInput - numStartInput + 1;
numNeuronInputs_noBias = numNeuronInputs - isBias ;

% 1-LAYER NETWORK CASE
if numLayers(module) == 1
   for l=1:numOutputs(module)
      dYdX(l,1:numInputs_X) = dYdS(lay1,l) .* weights1(l,numStartInput:numEndInput) ;
   end

% 2-LAYER CASE
elseif numLayers(module) == 2
   for l=1:numOutputs(module)
    for q=1:numInputs_X
     dYdX(l,q) = sum ( weights2(l,1:numNeuronInputs_noBias(module,lay2)) ...
     .* dYdS(lay1,1:numNeurons(module,lay1) ) .* weights1(:, (q-1+numStartInput) )' ) ...
     .* dYdS(lay2,l);
    end %q
   end % l
end %if numLayers..
```

E.3 Verification experiments

The verification experiments described here are done to show correctness (by verification of numerical results) of certain aspects of the ModNet algorithms. Some verification of the algorithms is needed, because an error in the implementation of an algorithm can not always be spotted on first sight.

Later when the verifications listed here were completed, it was found that the *finite differences* method (see recommendations in chapter 6) is actually an easier way to test gradient calculation routines.

Verification of the RTRL algorithm

In this experiment the intention was to verify the RTRL algorithm, for a simple linear network (1-layer modules), against the RTRL algorithm for Partially Recurrent Networks (see subsection 2.2.4) contained in the REC toolbox [Janssen, 1998] for FRNN. Both network structures were equal and the same initial weights were used. Equal results would show the new RTRL algorithm is probably correct.

The numerical results however differed so RTRL calculations had to be done by hand for the first three time steps. These matched with the RTRL calculations of the ModNet toolbox but not with the calculations for the partially recurrent network in the REC toolbox.

The conclusion is the REC toolbox apparently contains an error. The ModNet RTRL algorithm equals the hand-made calculations.

`File: Exp_Validation\test_valid1`

Verification of the gradient calculation routines against Matlab routines

The epochwise BPTT algorithm (bptt_epoch) was verified in this experiment against the 'virtual target' BPTT algorithm (bptt_vt). The gradient calculations in the virtual target BPTT algorithm are done by standard Matlab NNet routines, so this comparison can validate the gradient calculations performed in the ModNet normal BPTT algorithm. Both algorithms were run with identical initial weights and the numerical results of calculations were identical.

The identical numerical results imply that the gradient calculations are done equally to the MATLAB toolbox, so this actually verifies the correct working of the gradient calculation routine *calc_dYdW* which calculates all gradients $\partial y_{ij}/\partial w_{kl}$. The verification was done for several networks, having both 1-layer and 2-layer static networks and linear, tansig and logsig neuron transfer functions.

`File: Exp_Validation\test_valid2`

Verification of the partial supervision option in BPTT/RTRL algorithms

In this experiment the partial supervision option (also called 'weighted targets' option) was tested. See subsection 4.3.3. All targets were weighted equally with a factor 1 so in this case the weighted targets option should produce identical results compared to the original algorithms, which it did. Using the partial supervision with mixed 0/1 target weightings was also used in an experiment.

`File: Exp_Validation\test_valid3`