Optimization of Recurrent Neural Networks for Time Series Modeling

Ph.D. Thesis

Morten With Pedersen

LYNGBY 1997

IMM-PHD-1997-37

IMM

IMM
DEPARTMENT OF MATHEMATICAL MODELLING

Technical University of Denmark
DK-2800 Lyngby – Denmark

1997-08-25
mwp

Optimization of Recurrent Neural Networks for Time Series Modeling

Ph.D. Thesis

Morten With Pedersen

LYNGBY 1997

IMM-PHD-1997-37

IMM

ISSN 0909–3192

Printed by IMM, Technical University of Denmark

Abstract

The present thesis is about optimization of recurrent neural networks applied to time series modeling. In particular is considered fully recurrent networks working from only a single external input, one layer of nonlinear hidden units and a linear output unit applied to prediction of discrete time series. The overall objectives are to improve training by application of second-order methods and to improve generalization ability by architecture optimization accomplished by pruning. The major topics covered in the thesis are:

- The problem of training recurrent networks is analyzed from a numerical point of view. Especially it is analyzed how numerical ill-conditioning of the Hessian matrix might arise.
- Training is significantly improved by application of the damped Gauss-Newton method, involving the *full* Hessian. This method is found to outperform gradient descent in terms of both quality of solution obtained as well as computation time required.
- A theoretical definition of the generalization error for recurrent networks is provided. This definition justifies a commonly adopted approach for estimating generalization ability.
- The viability of pruning recurrent networks by the Optimal Brain Damage (OBD) and Optimal Brain Surgeon (OBS) pruning schemes is investigated. OBD is found to be very effective whereas OBS is severely influenced by numerical problems which leads to pruning of important weights.
- A novel operational tool for examination of the internal memory of recurrent networks is proposed. The tool allows for assessment of the length of the effective memory of previous inputs built up in the recurrent network during application.

Time series modeling is also treated from a more general point of view, namely modeling of the joint probability distribution function of the observed series. Two recurrent models rooted in statistical physics are considered in this respect, namely the "Boltzmann chain" and the "Boltzmann zipper" and a comprehensive tutorial on these models is provided. Boltzmann chains and zippers are found to benefit as well from second-order training and architecture optimization by pruning which is illustrated on artificial problems and a small speech recognition problem.

Resumé

Nærværende afhandling omhandler optimering af rekursive neurale netværk anvendt til tidsseriemodellering. Specielt betragtes fuldt rekursive netværk virkende fra blot et enkelt eksternt input, et lag af ikke-lineære skjulte enheder samt en lineær outputenhed, der anvendes til prædiktion af diskrete tidsserier. De overordnede mål er at forbedre træningen ved anvendelse af anden-ordens metoder samt at forbedre generaliseringsevnen ved arkitekturoptimering udført ved beskæring. De væsentligste emner, som er dækket i afhandlingen, er:

- Problemet med at optræne rekursive netværk er analyseret ud fra et numerisk synspunkt. Specielt analyseres, hvorledes numerisk dårlig konditionering af Hessian matricen kan opstå.

- Optræningen forbedres betydeligt ved anvendelse af den dæmpede Gauss-Newton metode, der involverer *hele* Hessianen. Denne metode findes helt at udkonkurrere gradientnedstigning udtrykt i både kvalitet af opnået løsning samt krævet beregningstid.

- Der gives en teoretisk definition af generaliseringsfejlen for rekursive netværk. Denne definition retfærdiggør en almindeligt benyttet måde at estimere generaliseringsevne på.

- Anvendeligheden af beskæring af rekursive netværk med Optimal Brain Damage (OBD) og Optimal Brain Surgeon (OBS) beskæringsmetoderne undersøges. OBD findes at være meget effektiv, hvorimod OBS er slemt påvirket af numeriske problemer, hvilket fører til beskæring af vigtige vægte.

- Et nyt operationelt værktøj til undersøgelse af den interne hukommelse i rekursive netværk foreslås. Værktøjet tillader vurdering af længden af den effektive hukommelse omkring tidligere inputs, der er bygget op i det rekursive netværk gennem anvendelsen.

Tidsseriemodellering behandles også ud fra en mere generel anskuelse, nemlig modellering af den simultane tæthedsfunktion for de observerede serier. To rekursive modeller med rødder i den statistiske fysik betragtes i denne henseende, nemlig “Boltzmann chain” og “Boltzmann zipper”, og en omfattende beskrivelse gives af disse modeller. Boltzmann chain og zipper findes ligeledes at blive gavnet af andenordens optræning og arkitekturtilpasning ved beskæring, hvilket illustreres på kunstige problemer samt et mindre talegenkendelsesproblem.

Preface

The present thesis has been submitted in partial fulfilment of the requirements for the Ph.D. degree in electrical engineering. The work documented in this thesis was carried out at the Technical University of Denmark at Department of Mathematical Modelling, Section for Digital Signal Processing (the former Electronics Institute) and was supervised by assoc. prof. Lars Kai Hansen and assis. prof. Jan Larsen. The work was initiated in September 1994 and completed in August 1997.

During the course of writing, this thesis has turned out to become pagewise significantly more extensive than anticipated, even though much of the originally intended material has been left out "on the fly" due to time and space considerations. Even so the pile of paper grew quickly as the remaining material was put together. At a late stage of writing it was considered to expose the thesis to a novel textual pruning procedure tentatively denoted as Optimal Thesis Damage (OTD) but unfortunately this exciting new tool for text size optimization remained at a conceptual level of implementation.

At this point I would like to thank my advisors Lars Kai Hansen and Jan Larsen for always having their door open when I had a question (which was rather often towards the end of this work) and for creating an enthusiastic and pleasant atmosphere at the Section for Digital Signal Processing. Jan Larsen is furthermore acknowledged for pleasant company during our trips to NIPS.

Søren Riis is thanked for his valuable comments and suggestions to improvement of this manuscript. Unfortunately I was not able to comply with all of them as the OTD pruning scheme was never fully implemented. Søren is also thanked for his pleasant company at the office during the years.

I also thank the rest of the staff and Ph.D. students at the Section for Digital Signal Processing for many joyful moments. I am sure that the Tour de Ph.D. '95 will enter the history books at some point in time.

I sincerely wish to express my gratitude to Dr. David G. Stork for inviting me to work at Ricoh California Research Center from February to August 1996. It was a very rewarding stay and turned out to be six truly unforgettable months. The rest of CRC are thanked as well for their great hospitality.

This work was granted by the Danish Natural Science and Technical Research Councils through the Computational Neural Network Center, CONNECT. The Otto Mønsted Foundation is acknowledged for financial support to travel activities.

Technical University of Denmark, August 1997

Morten With Pedersen

Contents

Chapter 1

Introduction

This introductory chapter contains a presentation of the background and motivation for the work documented in this thesis. The presentation in section 1.1 is brief in nature as the details of the background material are placed in subsequent chapters. Section 1.2 outlines the objectives of this work and section 1.3 provides an overview of the thesis.

1.1 Background

Modeling of discrete time series is an important task within many fields of research, including medicine, economics, communication, meteorology, speech processing, mechanical engineering, control, fluid dynamics and biology to name just a few. The problem of time series modeling is often cast in terms of *prediction*, i.e., to provide an estimate of future values of the series based on previously observed values. Such an estimate may be obtained from a mathematical model of the mechanism which generates the time series. If there are known underlying deterministic equations they can be solved to predict future observations based on knowledge of the initial conditions. An example is equations describing the planetary orbits which e.g., allowed for the successful landing of the Mars Pathfinder on the surface of Mars after a journey lasting for seven months.

It is, however, very often the case that little or no physical insight about the time series generating mechanism is available, rendering the deductive approach to modeling impossible. In this case the "rules" that govern the generating mechanism must be inferred from regularities in past observations. These rules may be specified as an assumed functional relationship between previous and future values of the time series. The functional relationship can be implemented by use of *adaptive models* which provide a mapping of previous observations onto estimates of future values. Characteristic for adaptive models is their flexible parametrization which allows the mapping implemented by the model to be adjusted to the problem at hand.

A nonlinear adaptive model type which has gained significant popularity during the last decade is the *feed-forward artificial neural network*, or simply feed-forward network, which has proved itself to be a highly flexible tool for time series modeling in many different applications. A substantial theoretical as well as practically oriented framework for e.g., training, architecture optimization and model verification has been built up around this model type. The framework includes methods for training which are more efficient than the traditional gradient descent (e.g., second-order methods [Møl93, Nør96]) architecture optimization by pruning (e.g., OBD [CDS90] and OBS [HS93]) as well as an extensive statistical theory for assessment of model quality (e.g., estimation of generalization ability [Whi89, GBD92, Moo91]); see e.g., the textbooks [HKP91, Hay94, Bis95, Rip96] for a detailed general overview of the framework for feed-forward networks.

The overall aim of the continuing research in adaptive models like neural networks may be seen as devising ever more powerful and flexible models. Further, to develop automatic procedures that make these models as simple as possible to use and this way reduce or, ultimately, completely automate the choices and decisions that are otherwise needed to be made by the user in connection with application. In this respect, a decisive choice which needs to be made prior to the application of feed-forward networks to time series modeling is the so-called *lag space*, i.e., determination of the previous values of the time series on which to base prediction. Once the lag space has been chosen it remains fixed which limits the flexibility of the feed-forward network.

A more general model type is obtained if the connectivity of the feed-forward network is extended to include *feedback* connections from the outputs of the units back to their inputs; in this case the resulting model is denoted a *recurrent* network. The most general recurrent network is obtained if the output from *every* unit in the network is fed back to the inputs of *all* units; these networks are denoted *fully* recurrent networks. The advantage of recurrent networks compared to feed-forward networks is due to an internal *memory* of past inputs, introduced by the feedback connections. This internal memory of recurrent networks is *adaptive*, i.e., during training it may be adapted to encompass those previous inputs which are relevant to the problem at hand. Such adaptive memory may in fact completely relieve the user from specifying a lag space as a fully recurrent network is able to work entirely from its own internal memory, created from only a *single* external input.

Despite their advantage of being able to completely eliminate the often difficult procedure of choosing a proper lag space, recurrent networks have not gained popularity similar to that of feed-forward networks. An explanation for this might be that it is generally agreed upon in the literature that it is more difficult to handle recurrent networks in practice than it is feed-forward networks. In particular it has been found [Moz93, PF94, TB94] that training using the widely preferred gradient descent method is not sufficiently "powerful" to train recurrent networks. Slow convergence is frequently encountered and often it might completely fail to provide an acceptable solution to the problem at hand. The training problem seems to be especially pronounced for fully recurrent networks. This was emphasized by prof. Ah Chung Tsoi during a visit in April 1994 at the former Electronics Institute at the Technical University of Denmark. His general advice regarding the application of fully recurrent networks was "*Don't touch them!*". Despite the poor results with gradient descent training there are very few examples in the literature of attempts to employ training methods which are significantly different from gradient descent.

A seeming result of the limited popularity of recurrent networks is that much of the theoretical framework for e.g., model validation built up around feed-forward models has not been extended to feedback models. It is for instance not clear how to define the generalization error for *dynamic systems* like recurrent networks. Furthermore, the theory behind analytical generalization error estimates like e.g., the Final Prediction Error estimate [Aka69] has not been justified for recurrent networks. In addition, the more practically oriented framework, e.g., for improving generalization ability by pruning has only been attempted to a very limited extent for recurrent networks [GO94, CFP94]. The OBD and OBS pruning schemes which have been found to be very successful for feed-forward networks have apparently not been introduced in the context of recurrent networks.

Besides the difficulties with training it is furthermore problematic to analyze recurrent networks once they are trained. Feed-forward networks may be examined in terms of e.g., a sensitivity analysis of the elements in the lag space in order to assess the importance of individual inputs; this way, insight may be obtained regarding the functionality of the network. However, no such tools exist for recurrent networks and analysis of their functionality is belived to be difficult. This was emphasized at the NIPS*96 workshop on dynamical recurrent networks by Lee Feldkamp from the Ford Motor Company Laboratories who jokingly stated that "*Life is too short to understanding dynamical systems.*"

1.2 Objectives

The overall objectives of this work are to improve training of recurrent neural network structures applied to time series modeling and to improve generalization ability by pruning. The work is primarily focused on fully recurrent networks working from only a single input, one layer of nonlinear hidden units and a linear output unit. This choice of network is made despite the claim that "*...the growing consensus seems to be that the architecture is inadequate for difficult temporal-processing and prediction tasks.*" [Moz93] It is common to attempt to overcome the seeming inadequacy of this basic recurrent network architecture by increasing the number of hidden layers, increasing the complexity of the feedback connections by employing multiple time delays etc. However, the approach here is to improve performance by adopting a more efficient training algorithm than gradient descent.

Training of recurrent networks is performed *offline* and is viewed as a nonlinear unconstrained function optimization problem. The type of training methods considered are second-order methods which have been found to be successful for feed-forward networks. The specific method considered here is the damped Gauss-Newton method as it was found in [Ped94] to completely outperform both gradient descent as well as Newton's method and the *pseudo* Gauss-Newton method when applied to multi-input recurrent networks. In order to make the second-order method perform optimally, the work includes an analysis of the training problem for recurrent networks. The analysis is carried out from a numerical point of view and involves investigation of factors which might lead to an ill-conditioned Hessian matrix. Further, the possibilities of handling the problems of ill-conditioning by use of regularization are investigated.

In order to determine the quality of a trained recurrent network it is necessary to determine its generalization ability. An attempt is therefore made to provide a theoretical definition of the generalization error of a dynamic system like a recurrent network. In order to improve generalization the applicability of the Optimal Brain Damage and Optimal Brain Surgeon pruning schemes will be investigated in the context of recurrent networks. These pruning schemes are chosen as they are among the most successful for feed-forward networks.

In order to allow for an interpretation of recurrent networks as well as an analysis of their functionality it is attempted to *measure* their memory. A definition of the *effective* memory of recurrent networks is formulated as this model type in principle has an infinite memory. The definition of effective memory leads to the proposal of an operational approach by which to determine the memory of a recurrent network.

Examples of applications of recurrent networks are provided for the Santa Fe laser series and the Mackey-Glass series. In particular is generated learning curves for the two series on which all further experiments with pruning and memory measurement are based. In addition it will be examined whether the networks from the learning curves are capable of simulating the dynamics underlying the series just as a comparison will be made to learning curves generated by use of comparable feed-forward networks.

A more general approach towards time series modeling than prediction is to model the joint probability distribution function of the observed series. This approach is generally adopted within e.g., speech recognition and biological sequence modeling. During a stay at Ricoh California Research Center the opportunity arose to work with two recently proposed recurrent network models for this task; the models, rooted in statistical physics, are denoted Boltzmann chains and Boltzmann zippers and are intended for modeling of discrete valued time series. These recurrent network models are treated in this thesis as well and it is attempted to train these models by second-order methods and to perform architecture optimization by pruning.

1.3 Thesis overview

This thesis is organized into 13 chapters including the present and 14 appendices. Chapters 2–11 constitute the main part of the thesis and is about fully recurrent neural networks applied to time series prediction. Chapter 12 is an independent, self-contained chapter which describes the work on Boltzmann chains and zippers for joint p.d.f. modeling of discrete valued time series. The contents of the individual chapters are as follows:

Chapter 2 reviews the concept of time series modeling for prediction by assuming a mathematical model of the generating system. Relevant feed-forward and feedback models incorporating noise are described and the statistical framework for making predictions is reviewed.

Chapter 3 describes various models which can be used as a guess of the structure of the system *teacher function*. The chapter contains an overview of the various ways in which feedback may be employed in recurrent networks. Finally, the fully recurrent network model considered in this work is described in detail.

Chapter 4 defines the training problem and reviews the *offline* training methods which are considered in the present work. It is furthermore described how to compute the gradient and the Hessian for recurrent networks.

Chapter 5 contains an attempt to provide a theoretical definition of the generalization error for dynamic system models like recurrent networks and it is discussed how to optimally estimate the generalization error. The chapter is concluded by a brief description of Akaike's Final Prediction Error Estimate.

Chapter 6 is about model complexity optimization in terms of both regularization and pruning. Expressions are provided for OBD and OBS in the case of a regularized cost function. Attention is directed to the problem of nuisance parameters and it is suggested to define saliencies in terms of the change in *generalization* error rather than training error.

Chapter 7 contains an analysis of the problem of training recurrent networks, considered from a *numerical* point of view in terms of ill-conditioning of the Hessian matrix. The potential causes of ill-conditioning are examined and quadratic weight decay is praised for its merits in handling ill-conditioning.

Chapter 8 illustrates experimentally on a small example that the rather speculative causes of ill-conditioning presented in chapter 7 actually occur in practice. The chapter also contains a *quantitative* comparison between the numerical problems in feed-forward and recurrent networks, respectively, as well as an experimental justification of the Gauss-Newton approximation to the Hessian.

Chapter 9 quantitatively compares training by damped Gauss-Newton and gradient descent. *Learning curves* are generated for recurrent networks trained on the Santa Fe laser series and the Mackey-Glass series and similar learning curves are generated for comparable *feed-forward* networks. Finally it is investigated whether the recurrent networks are capable of simulating the dynamics of the chaotic time series.

Chapter 10 illustrates pruning by OBD of recurrent networks chosen from the learning curves. The quality of the saliency estimates are investigated for both OBS and OBD and the accuracy of the second-order approximation is illustrated for the two methods.

Chapter 11 introduces a novel tool for measuring the *memory* of a recurrent network. The suggested memory measure is defined in terms of both an average memory and a time-local memory. These quantities are then illustrated for various networks from the learning curves.

Chapter 12 contains a comprehensive tutorial on Boltzmann chains and Boltzmann zippers. This work has introduced both second-order methods for training and architecture optimization by pruning to these recent models. Experiments are described for both artificial problems as well as for the construction of a small *speechreading* system.

Chapter 13 wraps up the overall conclusions of the thesis and contains suggestions to future work.

Appendix A contains a description of the datasets which are used for prediction, namely the Santa Fe laser series and the Mackey-Glass series. Furthermore the correlation dimension of the attractor is computed for each of these chaotic time series.

Appendix B compares two different updating schemes for the units in a recurrent network, namely a *layered* or asynchronous update in which the units are organized into sequential layers and a synchronous update where all units (including output units) are located in the same layer.

Appendix C analyzes the computational complexity of using the matrix inversion lemma for *offline* iterative computation of the inverse Hessian matrix and compares to the complexity when using a standard matrix inversion.

Appendix D contains a derivation of the mathematical prerequisites in order to examine the eigenvalues of (the Gauss-Newton approximation to) the Hessian matrix in terms of the columns of the Jacobian matrix.

Appendix E reviews the perturbation theory which indicates the influence of the condition number on the solution to a system of linear equations in the likely case of e.g., small rounding errors.

Appendix F reviews Mackay's "recipe" for conversion of a Boltzmann chain into a corresponding HMM.

Appendices G–N contain the papers which have been authored and co-authored during this work. Each paper is preceeded by a small summary of the material which is presented.

Chapter 2

Systems modeling

This chapter reviews the concept of time series modeling for prediction by inferring a model of the generating system from observations of the series itself. Section 2.1 describes the idea of assuming a mathematical formulation of the underlying system. Modeling of such a system is traditionally accomplished in terms of a functional relationship between previous and future values of the observed time series. This approach is formally justified by Takens' theorem described in section 2.2. Whereas Takens' theorem assumes a noise free situation, real-world systems are subject to noise as described in section 2.3 and noise therefore has to be taken into account when formulating the model of a system. This is done in section 2.4 which describes two general model structures relevant to this work and reviews the statistical framework for making predictions.

2.1 A general system model

A discrete time series may be seen as discrete instances of a signal. In order to introduce the concept of time series modeling it is thus appropriate to abstractly define the notion of the signal from which the series is obtained. A signal can be characterized as an observed, measurable quantity of interest obtained from the output of an entity denoted a *system*. It is difficult to provide an *exact* formal definition of a system that will apply universally but we may abstractly describe a system as comprising the physical mechanisms which directly (and possibly indirectly) influence the generation of the quantity which is observed as a signal.

A system may generate its output exclusively from mechanisms, or *dynamics*, internal to the system in which case we denote it an output system. If the output is furthermore influenced by externally applied *input* signals then the system is denoted an input/output system. Figure 2.1 provides a conceptual model of an input/output system, where the input to the system is denoted $u(t)$ and the output from the system is denoted $x(t)$.

When attempting to model a signal we are in fact trying to "imitate" the mode of operation of the mechanisms comprising the system which generated the signal. The purpose of such modeling may be predictive, i.e., with the aim of predicting future outputs of the system. The purpose may also be explanatory, i.e., a means by which to obtain knowledge of the system by investigation of the model. In order to provide a mathematical model of a system we must assume that the internal mechanisms of the system may be described mathematically by either a deterministic functional expression or by a stochastic process. For now it will be assumed that the system under consideration is deterministic

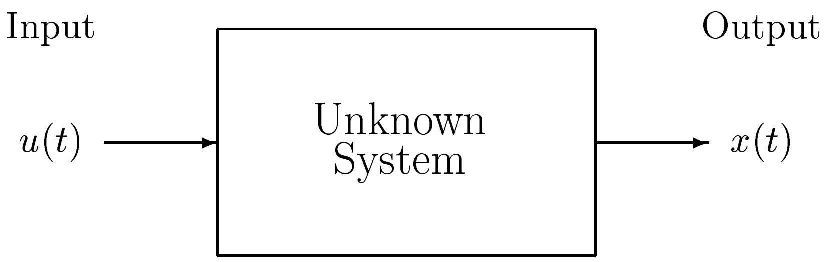


Figure 2.1: A conceptual model of an input/output system.

in nature but we will return to stochastic modeling in section 2.4 and in chapter 12.

The present work is focused around modeling of output systems generating a single output signal. A very general assumption regarding the mathematical structure of the internal dynamics of such a system is that it may be described by a *dynamic system* [PC89, CEFG91],

$$\mathbf{s}(t) = \mathbf{f}^t(\mathbf{s}(0)) \tag{2.1}$$

where $\mathbf{s}(t)$ is the internal *state vector* of finite dimension d at time t and $\mathbf{f}^t : \Re^d \to \Re^d$ is a mapping function assumed to be a diffeomorphism[1] and denoted the time-t map that takes an initial state $\mathbf{s}(0)$ to a state $\mathbf{s}(t)$. The time variable t can be either continuous or discrete. Assuming that the system is noise free the observed output is related to the internal dynamics of the system by

$$x(t) = h(\mathbf{s}(t)) \tag{2.2}$$

where the function $h : \Re^d \to \Re$ is called the *measurement function* [CEFG91] which is generally assumed to be differentiable. Figure 2.2 illustrates how an output system may be described by a dynamical system in the absence of noise.

In many applications the system output $x(t)$ will be an analog, continuous signal. However, in order to process the signal using digital computers we need to *sample* the output signal, leading to the observed time series. In the following it will be assumed that sampling has been performed using a *properly* chosen sampling period τ; how to properly choose the sampling period τ will not be treated here. Sampling of linear systems is treated in detail in e.g., [OS89]; proper sampling of nonlinear systems is, however, more involved and has to the knowledge of the author not been treated from a general point of view. Generally, if τ is chosen too small the resulting discrete time series will be subject to redundancy; in this case each new sample will provide little additional information about the system dynamics making modeling from the time series $\{x\}$ difficult. On the other hand, if τ is chosen too large the dynamics at one time become effectively causally disconnected from the dynamics at a later time; essential information regarding the system dynamics is lost, making accurate modeling impossible.

Assuming τ has been properly chosen we observe the sequence of system outputs

$$\ldots\ ,\ x(t-2\tau)\ ,\ x(t-\tau)\ ,\ x(t)\ ,\ x(t+\tau)\ ,\ x(t+2\tau)\ ,\ \ldots$$

[1] A one-to-one differentiable function with a one-to-one differentiable inverse.

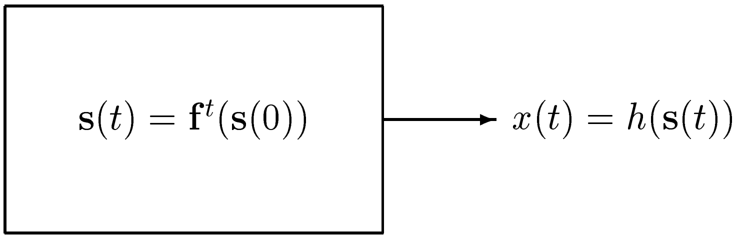


Figure 2.2: Output system described by a (noise free) dynamic system.

Usually the sampling period τ is not explicitly stated and we will write the observed sequence as

$$\ldots\ ,\ x(t-2)\ ,\ x(t-1)\ ,\ x(t)\ ,\ x(t+1)\ ,\ x(t+2)\ ,\ \ldots$$

thus implying the sampling period τ.

When constructing a model of a system we should naturally utilize any prior knowledge such as insight into the physical mechanisms of the system, somehow incorporating this knowledge into the model. However, often we do *not* have any prior knowledge about the mechanisms of the system, the only information available to us is the time series $\{x\}$ of measured system outputs. From the data alone we must infer a model which is able to account for the system dynamics; this is also referred to as *black-box* modeling [Lju87]. An approach that apparently dates back to [Yul27] is to model an output system like the one illustrated in Figure 2.2 from a *lag vector* $\mathbf{x}(t)$ defined as

$$\mathbf{x}(t) = [\ x(t)\ \ x(t-1)\ \ \ldots\ \ x(t-L+1)\ ]^{\mathrm{T}} \tag{2.3}$$

where L denotes the dimensionality; $\mathbf{x}(t)$ is often also referred to as the *delay vector* or the *delay line*. If the purpose of modeling is to predict the system output one time step into the future, the prediction $\widehat{x}(t+1)$ of the system output at time $t+1$ is obtained as

$$\widehat{x}(t+1) = F[\mathbf{x}(t)] = F[\ x(t)\ ,\ x(t-1)\ ,\ \ldots\ ,\ x(t-L+1)\ ] \tag{2.4}$$

where F is a function chosen to represent the system dynamics. This approach has been empirically proven to be very effective; see e.g., [WG93] and references herein. The approach was mathematically justified by Takens [Tak81] who demonstrated that assuming the system is noise free and measurements of the system output are performed with infinite precision, it is possible to exactly recreate the internal dynamics of the system illustrated in Figure 2.2 from a delay vector $\mathbf{x}(t)$, provided its dimensionality $L \geq 2d+1$. Here, d is the dimensionality of the state vector[2] in Eq. (2.1). This finding is denoted *Takens' theorem.*

2.2 Takens' theorem

This section contains a heuristic derivation of Takens' theorem adopted from [CEFG91]. Takens studied the *delay reconstruction map* $\mathbf{\Phi} : \Re^d \to \Re^L$ which maps the d-dimensional state space of a dynamic system defined by Eqs. (2.1–2.2) into an L-dimensional (lag) space,

[2]It was later proved in [SYC91] that d may be replaced by the *box counting* dimension of the attractor (refer to appendix A) on which $\mathbf{s}$ moves.

$$\begin{aligned}\mathbf{\Phi}(\mathbf{s}(t)) &= \Big(h(\mathbf{s}(t)) \, , \, h(\mathbf{f}^{-\tau}(\mathbf{s}(t))) \, , \, \ldots \, , \, h(\mathbf{f}^{-(L-1)\tau}(\mathbf{s}(t))) \Big) \\ &= (\, x(t) \, , \, x(t-1) \, , \, \ldots \, , \, x(t-L+1) \,) \\ &= \mathbf{x}(t)\end{aligned} \tag{2.5}$$

where $f^{-\tau}$ denotes the dynamics of Eq. (2.1) "reversed in time". $\mathbf{\Phi}$ may be regarded as a set of L simultaneous nonlinear equations in d variables, mapping the internal state of the system onto a delay vector. If the surface of $\mathbf{x} = \mathbf{\Phi}(\mathbf{s})$ is smooth and contains no self-intersections, i.e., a bijective one-to-one mapping, then there will be a unique solution of $\mathbf{s}$ in terms of $\mathbf{x}$. If this solution depends smoothly on $\mathbf{x}$ then the map $\mathbf{\Phi}$ is denoted an *embedding*.

It can be shown [CEFG91] that when $L = d+1$ then the set of self-intersections of the surface defined by $\mathbf{\Phi}$ will be of dimension *at most* $d-1$. E.g., if $\mathbf{s}$ is a $d = 2$ dimensional state vector (i.e., orbitting in a plane) which is smoothly mapped into $L = d+1 = 3$ dimensions then the set of self-intersections of the map will be of dimension at most $L = d-1 = 1$ (e.g., a line). If L is *increased* by one, the dimension of the set of self-intersections will *decrease* by one, and by induction we find that the map $\mathbf{\Phi}$ is guaranteed to be an embedding provided that

$$L \geq 2d+1 \tag{2.6}$$

which constitutes Takens' theorem. Note that the map $\mathbf{\Phi}$ *may* be an embedding with L as small as $L = d$, e.g., in the case of a linear system.

When $\mathbf{\Phi}$ is an embedding it will have a unique inverse $\mathbf{\Phi}^{-1}$. In this case it is possible to define a smooth dynamics $\mathbf{F}$ on the space of delay vectors $\mathbf{x} \in \Re^L$ which evolves the delay vector a period of time τ into the future. Setting τ equal to unity, the dynamics $\mathbf{F} : \Re^L \to \Re^L$ is *defined* as

$$\begin{aligned}\mathbf{F}(\,\mathbf{x}(t)\,) &= \mathbf{\Phi} \circ \mathbf{f}^{\tau} \circ \mathbf{\Phi}^{-1}(\,\mathbf{x}(t)\,) \\ &= \mathbf{\Phi} \circ \mathbf{f}^{\tau}(\,\mathbf{s}(t)\,) \\ &= \mathbf{\Phi}(\,\mathbf{s}(t+1)\,) \\ &= \mathbf{x}(t+1)\end{aligned} \tag{2.7}$$

In this expression $\mathbf{\Phi}^{-1}(\,\mathbf{x}(t)\,)$ transforms the delay vector $\mathbf{x}(t)$ into the corresponding internal state vector $\mathbf{s}(t)$. The internal state vector $\mathbf{s}(t)$ is advanced by $\mathbf{f}^{\tau}$ to $\mathbf{s}(t+\tau)$, resulting in $\mathbf{s}(t+1)$ for $\tau = 1$. Finally, the internal state vector $\mathbf{s}(t+1)$ is transformed by the embedding $\mathbf{\Phi}$ into the corresponding delay vector at time $t+1$. From Eq. (2.7) we thus see that provided $\mathbf{\Phi}$ is an embedding it is possible to generate dynamics which are equivalent to the original system Eqs. (2.1–2.2) from the system output $x(t)$ alone.

In Eq. (2.7) we can now isolate the component of the vector function $\mathbf{F}$ which corresponds to prediction,

$$x(t+1) = F(\,\mathbf{x}(t)\,) \tag{2.8}$$

which demonstrates that it is indeed theoretically possible to predict future outputs of a system from measurements of previous outputs provided that sufficiently many previous values are used. The function $F : \Re^L \to \Re$ can be regarded as a *teacher function* which generates the true prediction $x(t+1)$ corresponding to a particular lag vector $\mathbf{x}(t)$. Modeling of a system for prediction may now be cast in terms of identifying the teacher function F.

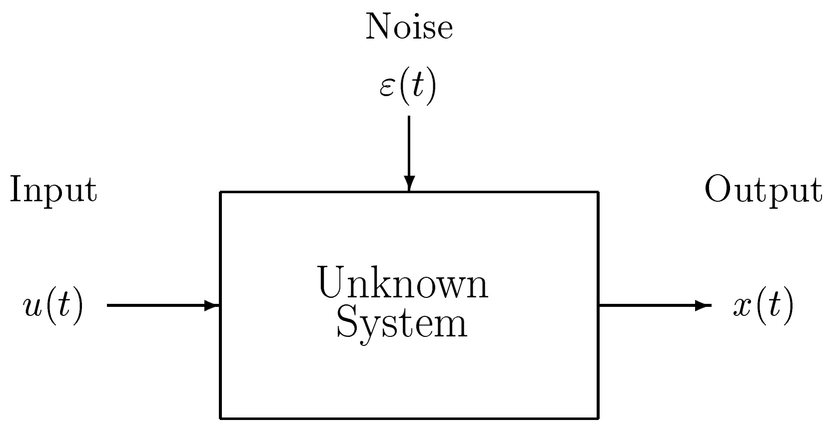


Figure 2.3: A conceptual model of an input/output system influenced by noise.

From the above heuristic derivation of Takens' theorem we learn that unobservable dynamics internal to an output system may be captured and modeled from measurements of the system output only. In [Cas92] it was further argued that Takens' theorem can be generalized to incorporate input/output systems as well, demonstrating that the internal dynamics of such systems may be modeled from delay vectors containing delayed values of both the externally provided input as well as the measured output. This way, Takens' theorem provides a mathematical justification of the approach towards modeling that has been traditionally adopted in various fields.

2.3 Noise

Takens' theorem relies on assumptions about a noise free system for which the outputs are measured with infinite precision. In this case it is theoretically possible to predict the system output for an arbitrary period of time into the future as seen from the dynamics in Eq. (2.7), where τ may be extended to an arbitrary time step. In practice however, these assumptions are generally unrealistic. Measurements of the system output are performed using limited precision just as the system may be subject to disturbances of various kinds, e.g., in the form of inaccessible and possibly random external influences. These disturbances will affect the internal dynamics of the system in unpredictable ways, thus limiting the accuracy by which predictions can be made. The noise effects of the disturbances are furthermore likely to accumulate over time, thus limiting the size of the time step by which it is possible to make useful predictions of future outputs. Figure 2.3 illustrates a conceptual model of an input/output system under the influence of noise.

2.4 Models of systems for prediction

The literature is rich on mathematical models of system structures in the presence of noise; see e.g., [Lju87, Sør94, Nør96] for an overview. In the following will be described two system structures which are relevant for the non-linear time series prediction problems and the model architectures considered in this work. The first system structure assumes that the system under consideration generates the time series $\{x\}$ according to a Non-linear

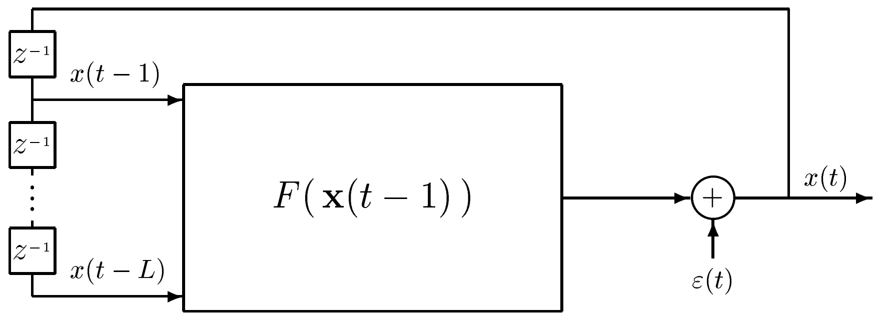


Figure 2.4: *True* system running in "closed-loop", generating data according to a NAR process.

Auto-Regressive (NAR) process, given by

$$x(t) = F(\mathbf{x}(t-1)) + \varepsilon(t) \quad , \tag{2.9}$$

where t is the time index, $x(t)$ is the scalar system output signal, $F(\cdot)$ constitutes a mapping of arbitrary complexity of the L-dimensional input vector $\mathbf{x}(t-1) = [x(t-1), \ldots, x(t-L)]$ containing previous system output values and $\varepsilon(t)$ is a disturbance term representing the system noise. The inherent system noise $\varepsilon(t)$ is assumed to be a stationary white noise sequence with zero mean and a finite variance. The generating process is illustrated in Figure 2.4.

Introduction of the noise term leads to a statistical framework of the system. The system output $x(t)$ may now be interpreted as a random variable, conditioned on the lag vector $\mathbf{x}(t-1)$ representing the input to the system. Assume for now that we have knowledge of the *teacher function* $F(\cdot)$ of the "true" system model Eq. (2.9). Assume further that we have observed previous system output values corresponding to the lag vector $\mathbf{x}(t-1) = [x(t-1), \ldots, x(t-L)]$ and wish to *predict* the next system output. The *optimal* prediction $\widehat{x}(t)$ of $x(t)$ is generally defined as the conditional expectation of $x(t)$ given $\mathbf{x}(t-1)$ [Lju87], i.e., as the value of $x(t)$ that will be realized "on average", given the particular realization $\mathbf{x}(t-1)$ of the input vector; this definition of the prediction is optimal in the least squares sense [Lju87]. From Eq. (2.9) we see that the optimal prediction of $x(t)$ given $\mathbf{x}(t-1)$ is obtained as

$$\widehat{x}(t) = E[x(t)|\mathbf{x}(t-1)] = E[F(\mathbf{x}(t-1)) + \varepsilon(t)|\mathbf{x}(t-1)] = F(\mathbf{x}(t-1)) \quad . \tag{2.10}$$

The prediction defined by Eq. (2.10) in terms of the *true* system is thus the *best* prediction of the next system output it is possible to make, given observations of $\mathbf{x}(t-1)$.

Within the statistical framework adopted, the most general and complete description of the generating system defined by Eq. (2.9) is in terms of the joint probability density function $P(x(t), \mathbf{x}(t-1))$ relating the model input and output. This entity is however not available as it requires complete knowledge of the system as well as the characteristics of the disturbances. When modeling a time series we therefore adopt a model which has a structure similar to our beliefs regarding the generating system. A general NAR model corresponding to Eq. (2.9) is

$$x(t) = g(\mathbf{x}(t-1); \mathbf{w}) + e(t) \quad , \tag{2.11}$$

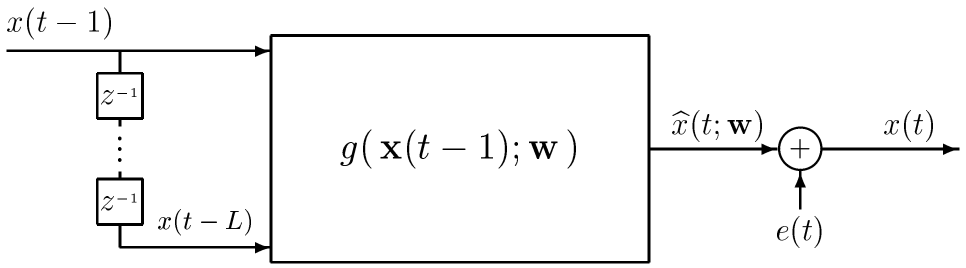


Figure 2.5: *Predictive* NAR model running in "open-loop", generating predictions $\widehat{x}(t)$ from *known* observations $x(t-1)$ from the true system.

where $g(\cdot)$ is a function which represents our "guess" at the true system teacher function $F(\cdot)$ parametrized by $\mathbf{w}$ and $e(t)$ denotes the model error. A model of this type is also denoted a *feed-forward* model as will be described in section 3.

When using the model (2.11) for prediction, the prediction of $x(t)$ given $\mathbf{x}(t-1)$ is obtained as

$$\widehat{x}(t;\mathbf{w}) = g(\mathbf{x}(t-1);\mathbf{w}) \approx E[x(t)|\mathbf{x}(t-1)] \tag{2.12}$$

The error term $e(t)$ now denotes the *prediction error*, obtained as

$$e(t) = x(t) - \widehat{x}(t;\mathbf{w}) \quad . \tag{2.13}$$

One-step ahead predictions of the system output are obtained by conditioning on the inputs to the model, in this case previous outputs of the true system. At the time we wish to make a prediction of the next output $x(t)$ we thus have complete knowledge of the previous values $\mathbf{x}(t-1)$ which are observed from the true system; these true system outputs may be viewed as particular, known realizations from a stochastic process defined by (2.9). When employing our model Eq. (2.11) for one-step ahead predictions we therefore operate it in "open-loop", as illustrated in Figure 2.5. The inputs to the model are generated "externally" by the true system, not by the model. Thus, the model defined by Eq. (2.11) on predictive form is purely *feed-forward* in nature.

Another possible description of a system is in terms of a so-called state space model. The state space model may also be seen as a dynamic system with a noise term added, and the models considered here are of the form

$$\begin{aligned} \mathbf{s}(t) &= \mathbf{f}(\,\mathbf{s}(t-1), x(t-1)\,) \\ x(t) &= h(\,\mathbf{s}(t)\,) + \varepsilon(t) \end{aligned} \tag{2.14}$$

where $\mathbf{s}(t)$ denotes an unmeasurable d-dimensional state vector internal to the system, $x(t-1)$ is the system output at the previous time step, $\mathbf{f}(\cdot)$ is a vector function of arbitrary complexity generating the internal dynamics of the system and $h(\cdot)$ is the measurement function generating the output of the system; $\varepsilon(t)$ is once more a disturbance term representing the system noise. As previous the noise $\varepsilon(t)$ is assumed to be a stationary white noise sequence with zero mean and a finite variance. In Figure 2.6 is illustrated a system which generates its output $x(t)$ according to the state space desription of Eq. (2.14).

It is simple to demonstrate that the dynamic system description in Eq. (2.14) contains *all* possible NAR descriptions defined by Eq. (2.9) as a special case. Assume

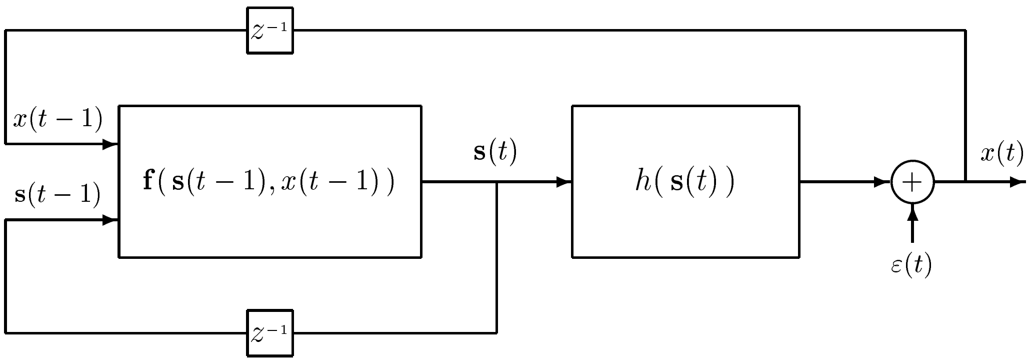


Figure 2.6: *True* system running in "closed-loop", generating data according to a state space model.

that the teacher function $F(\cdot)$ works from an L-dimensional input vector $\mathbf{x}(t-1) = [x(t-1), \ldots, x(t-L)]$. One of the many ways in which the state space model of Eq. (2.14) may implement such a NAR-model is by choosing the state vector $\mathbf{s}$ to be of dimension $d = L$ and specifying the update of each element of the state as

$$\begin{aligned} s_1(t) &= x(t-1) \\ s_2(t) &= s_1(t-1) \\ &\vdots \\ s_L(t) &= s_{L-1}(t-1) \ . \end{aligned} \tag{2.15}$$

In this implementation the dynamics of the internal state merely implements a shift register, or a delay line. If the function $h(\cdot)$ of Eq. (2.14) is chosen as $h = F$ then the two system descriptions are completely identical. Consequently, the dynamic system description is of a more *general* type than the NAR description. In fact, the NARMAX model (Non-linear AutoRegressive Moving Average model with eXogenous inputs) may be obtained from an extension of the state space model considered here [LB85].

In line with the approach adopted in [Lju87] we may generally formulate the dynamic system description in Eq. (2.14) equivalently as

$$x(t) = F_t(\mathbf{X}^{t-1}) + \varepsilon(t) \tag{2.16}$$

Here, $\mathbf{X}^t$ is a compact notation for a vector containing *all* observations generated by the true system from the starting time $t = 1$ up to time t, defined as

$$\mathbf{X}^t = (\, x(t), x(t-1), \ldots, x(1)\,) \tag{2.17}$$

The formulation in Eq. (2.16) explicitly states – in terms of the function $F_t(\cdot)$ – the dependency of the current output of the dynamic system upon *all* previously generated outputs. The subscript t denotes the dependence on time in terms of the increasing number of observations on which the function is based.

In order to formulate a statistical framework similar to the description for the feed-forward-type NAR process above in order to make predictions we will assume [Lju87] a probability density function P_t for the observed sequence $\mathbf{X}^t$ generated by the true system,

$$P_t\left(\mathbf{X}^t\right) = P_t\left(x(t), \mathbf{X}^{t-1}\right) \tag{2.18}$$

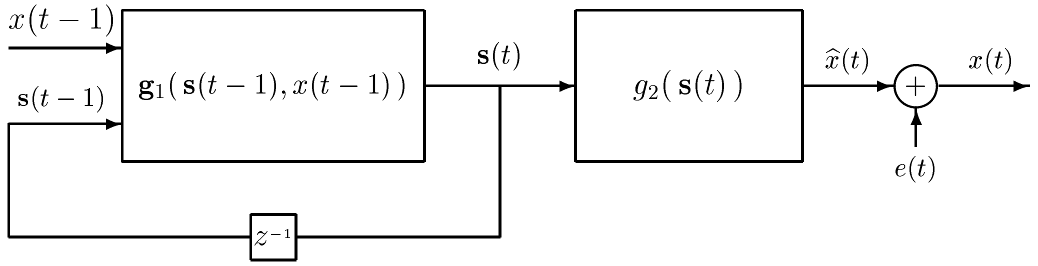


Figure 2.7: *Predictive* state space model running in "open-loop", generating predictions $\widehat{x}(t)$ from *known* observations $x(t-1)$ from the true system.

which is the most general and complete way of describing the system defined by Eq. (2.14) and Eq. (2.16). This joint probability density function of measurements up to time t may also be formulated as

$$P_t\left(\mathbf{X}^t\right) = P_t\left(x(t) \,|\, \mathbf{X}^{t-1}\right) \cdot P_{t-1}\left(\mathbf{X}^{t-1}\right) \tag{2.19}$$

Given this probability density function it is, at least *conceptually* [Lju87], possible to define the optimal predictor as the conditional mean of $x(t)$ given $\mathbf{X}^{t-1}$, that is,

$$\widehat{x}(t) = E[x(t) \,|\, \mathbf{X}^{t-1}] = F_t(\mathbf{X}^{t-1}) \ . \tag{2.20}$$

When modeling the dynamic system in order to make predictions, the mapping $F_t(\cdot)$ corresponding to the *noise free* dynamic system Eq. (2.14) is thus the teacher function to be identified.

When modeling a dynamical system believed to be of the form Eq. (2.14) we may employ a parametrized model with a similar structure,

$$\begin{aligned} \mathbf{s}(t) &= \mathbf{g}_1(\,\mathbf{s}(t-1), x(t-1); \mathbf{w}_1\,) \\ x(t) &= g_2(\,\mathbf{s}(t); \mathbf{w}_2\,) + e(t) \end{aligned} \tag{2.21}$$

where $\mathbf{g}_1(\cdot)$ and $g_2(\cdot)$ parametrized by $\mathbf{w}_1$ and $\mathbf{w}_2$, respectively, represent our "guesses" of the corresponding functions in the true model Eq. (2.14) and $e(t)$ once more denotes the model error.

Just as it was done for the true system in Eq. (2.16) we may formulate the model defined by Eq. (2.21) on a completely equivalent form as

$$x(t) = g_t(\mathbf{X}^{t-1}; \mathbf{w}) + e(t) \tag{2.22}$$

where $\mathbf{w} = (\mathbf{w}_1, \mathbf{w}_2)$. When using this model for prediction, the prediction of $x(t)$ given $\mathbf{X}^{t-1}$ is obtained in terms of the model as

$$\widehat{x}(t; \mathbf{w}) = g_t(\mathbf{X}^{t-1}; \mathbf{w}) \approx E[x(t) \,|\, \mathbf{X}^{t-1}] \tag{2.23}$$

in line with Eq. (2.20), and the function $g_t(\ \cdot\ ; \mathbf{w})$ is thus an approximation to the true system teacher function $F_t(\cdot)$ just as for the NAR-model type. As for the NAR model described above, predictions are thus obtained by operating the state space model in "open-loop". This is illustrated in Figure 2.7. The input $x(t-1)$ to the model on predictive form is an observation from the *true* data generating system Eq. (2.14) and is thus a known quantity; $e(t)$ represents the prediction error $e(t) = x(t) - \widehat{x}(t; \mathbf{w})$. Even though the model is operated in open-loop it still maintains a feedback of previous values of the internal state $\mathbf{s}$ as seen in Figure 2.7; this makes the predictive model a *recurrent* model as will be described in section 3.3.

Note that the function $g_t(\cdot)$ is parametrized by the *same* parameters $\mathbf{w} = (\mathbf{w}_1, \mathbf{w}_2)$ as in the state space formulation of Eq. (2.21) and may be generated by "unfolding" the predictor model in time; refer to Figure 4.1 on page 41 for an illustration of a time-unfolded model structure.

At this point a few comments about modeling dynamic systems will be made. A practical problem when modeling dynamic systems is that generally one cannot expect the *model* Eq. (2.21) to start iterations at the same time $t = 1$ as the *true system* Eq. (2.14) began iterations as e.g., observations $\{x\}$ are only available from a later time $t_0 > 1$. Consequently, the model output will not be based on the same (possibly infinite) observation sequence $\mathbf{X}^t$ as the true system but rather on a limited one $\mathbf{X}_{t_0}^t = [x(t), x(t-1), \ldots x(t_0)]$ which will lead to a discrepancy between model and true system output even in the case of perfect identification of the teacher function. Further, even if *all* observations $\mathbf{X}^t$ of the true system from the starting time $t = 1$ onwards were available we would still be ignorant regarding the unmeasurable internal initial state of the true system, also leading to a discrepancy between model and true system output.

In order for modeling to be possible at all it is therefore necessary to assume that the *influence* of previous observations $x(t - n')$, $\forall n' > n > 0$ on the true system output $x(t)$ will decay to zero at an appropriate rate for increasing n. This assumption will eventually allow the model output to converge towards the true system output after a *transient* period. Also, in order to model a dynamic system it must be assumed that the teacher function is *time invariant*, i.e., invariant under a shift of absolute time in the sense that the dynamics of the true system are not influenced by the iteration starting time. This requirement is equivalent to the functional mappings $\mathbf{f}(\cdot)$ and $h(\cdot)$ in Eq. (2.14) being fixed.

In practice, in order to conceptually handle the problems caused by lack of the complete observation sequence it is customary to assume [Lju87] that the available observed sequence *is* in fact complete and furthermore that the initial state of the true system is identical to e.g., zero. Thus, modeling becomes *conditioned* on these assumptions.

Chapter 3

Model architectures

When modeling a system from a time series it is necessary to provide a guess of the structure of the "teacher function" as was learned from the discussion in the previous chapter. This chapter provides a description of various model structures including an overview of recurrent network structures. The chapter is concluded by a description of the specific recurrent network structure considered in this work.

3.1 The linear FIR filter

The simplest structure we might assume for the teacher function is a *linear* relationship between previous and future values observed from the system. Assuming an auto-regressive modeling problem, the output $y(t)$ at time t from a linear model corresponding to the prediction $\widehat{x}(t+1)$ of the next value in the time series is calculated as

$$\widehat{x}(t+1) = y(t) = \sum_{k=0}^{L-1} w_k \, x(t-k) + w_{bias} \tag{3.1}$$

where w_k denotes adaptive weights applied to the L previous values of the time series and w_{bias} denotes an arbitrary offset or *bias* of the series to be modeled. The structure of the model given by Eq. (3.1) is recognized as a linear FIR (Finite Impulse Response) filter well-known from e.g., the digital signal processing literature [WS85, OS89], and is illustrated in Figure 3.1. The number of inputs L on which the output y is based is denoted the *filter order*.

Linear models have been widely applied to systems modeling in various fields. Their popularity is due to a variety of factors. Among these are the ease with which linear models are trained and operated. Furthermore, it is possible to provide an interpretation of linear models as they can be fully characterized in terms of e.g., their transfer function and impulse response. Consequently, an extensive theory exists regarding the properties of linear models.

Even though good results have been obtained in numerous applications of linear models, their modeling capability is often not adequate for satisfactory approximation of the teacher function. Many problems are inherently nonlinear in nature, and in this case a linear model is likely to fail to capture essential functionality with poor performance as a result.

A possible enhancement of the modeling capabilities of the linear model which retains the advantages of linearity is to apply a static nonlinear transformation of the inputs

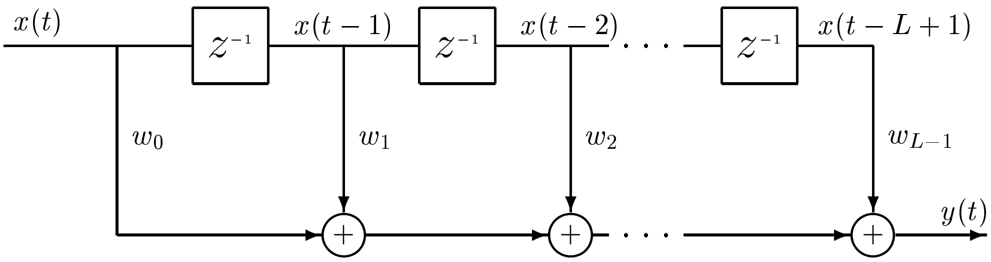


Figure 3.1: Linear FIR-filter of order L.

before they enter the linear model. An example is to approximate the teacher function by a polynomial expansion, in which case the output of the linear model reads

$$\begin{aligned} y(t) &= \sum_{k=0}^{L-1} w_k\, x(t-k) + \sum_{k=0}^{L-1}\sum_{l=0}^{L-1} w_{kl}\, x(t-k)x(t-l) + \ldots \\ &\quad + \sum_{k=0}^{L-1}\sum_{l=0}^{L-1}\sum_{m=0}^{L-1} w_{klm}\, x(t-k)x(t-l)x(t-m) + \ldots + w_{bias} \ . \end{aligned} \tag{3.2}$$

Such an expansion is theoretically capable of modeling an arbitrary nonlinear continuous function to arbitrary accuracy, provided the expansion is of a sufficiently high order. In practice however, a polynomial expansion is of limited use as the order of the expansion is often required to be very high in order to obtain a satisfactory approximation; this leads to an explosion in the number of parameters that needs to be adapted. Furthermore, the regressors resulting from the polynomial expansion tend to be highly correlated even for relatively low expansion orders which makes the adaption problem numerically ill-conditioned; refer to chapter 7 for an elaboration on ill-conditioning. The consequence is that even though a static nonlinearity increases the modeling capabilities of the linear model it is often in practice still not adequate in order to obtain a satisfactory approximation of the teacher function; a general, inherently nonlinear model structure is called for.

3.2 Feed-forward networks

A nonlinear model type that has gained significant popularity during the last decade is the so-called artificial neural network. The history of artificial neural networks goes back to [MP43] in which was presented a mathematical representation of an event in the nervous system in terms of a nonlinear processing element denoted an artificial neuron, which was combined into networks of units inspired by the connectivity of the brain. However, it was not until an efficient learning algorithm (the famous back-propagation) in [RM86] was distributed to a wide audience, as well as the general availability of sufficient computing facilities, that large-scale research into the possibilities of this model type was initiated. Since then the neural network model type has been established as an important modeling tool in numerous fields due to its high flexibility and modeling capabilities.

The most commonly adopted neural network structure is the feed-forward network. For a thorough introduction to feed-forward networks the reader is referred to one of the excellent text books, e.g., [HKP91, Hay94, Bis95]; here will only be provided a short summary of that model type. The term "network" refers to a collection of connected processing units, the term "feed-forward" refers to the connectivity between the units which only allows information to "flow" in one direction.

The processing units of a feed-forward network are organized into one or several so-called hidden layers as well as an output layer; the term "hidden" refers to the outputs from these units not being externally observable, i.e., they are internal to the network. Between layers, the units are connected by adjustable parameters denoted the weights, and the externally provided inputs to the network are connected by weights to the first hidden layer of units. In Figure 3.2 is illustrated a feed-forward network having a single hidden layer of units and a single unit in the output layer; the processing units are denoted by open circles and the adjustable weights are denoted by arrows, indicating the direction of information flow.

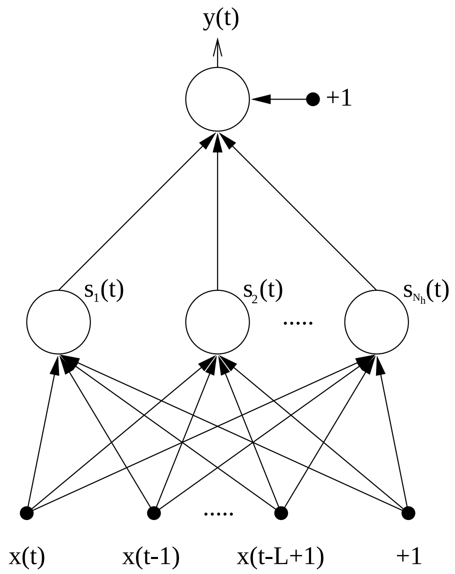


Figure 3.2: Feed-forward neural network.

When using feed-forward networks for auto-regressive time series modeling the inputs to the network at time t represent a lag space vector $\mathbf{x}(t) = [x(t), \ldots, x(t-L+1)]$ of previously observed values. Via the weighted connections these values enter the first hidden layer of units whose outputs $s_j(t)$ are calculated as

$$s_j(t) = f\left(\sum_{k=0}^{L-1} w_{jk}\, x(t-k) + w_{jb}\right) \tag{3.3}$$

where w_{jb} represents a bias weight and $f(\cdot)$ is a nonlinear sigmoid-shape *activation* function. In the experiments in this work involving feed-forward networks the function $f(x) = \tanh(x)$ has been used. In the case of multiple layers of hidden units, the layers are updated consecutively by expressions similar to Eq. (3.3), however using the outputs of the hidden units in the immediately preceeding hidden layer as inputs. Finally, the outputs from the last hidden layer are fed into the output layer units which are updated as

$$y(t) = \sum_{j=1}^{N_h} w_{oj}\, s_j(t) + w_{ob} \tag{3.4}$$

For auto-regression problems only a single output unit is used. This output unit has a linear activation function in order to allow for arbitrary dynamic range of the output. Feed-forward networks may also be applied to classification problems as described in e.g., [HMPHL96] included in appendix J; in this case the feed-forward network has several output units, one for each of the possible classes.

As seen from Eqs. (3.3–3.4) feed-forward networks provide a nonlinear mapping of an input onto one or several outputs. A feed-forward network may therefore be interpreted as a regular functional expression, the complexity of which depends on the number of processing units and layers.

A justification of the application of feed-forward networks to general nonlinear modeling problems has been given in terms of the *universal approximation theorem* which has been proved several times in the literature, e.g., in [Cyb89, Fun89, HSW89]. The essence of this theorem is that a function with a structure similar to that of a feed-forward network having only one layer of hidden units with nonconstant, bounded and monotone-increasing continuous activation functions and a linear output is capable of approximating any continuous function on a compact set of points to any desired degree of accuray, provided that *sufficiently* many hidden units are available. The universal approximation theorem is an *existence theorem* in the sense that it states the existence of an arbitrary good approximation; it does however not specify how *many* units is necessary in order to obtain a given accuracy. From the functional expression of a feed-forward network alone it is difficult to visualize how superpositions of the sigmoids may lead to the approximation of any function. An excellent illustration of how combinations of sigmoid outputs in a three-layer network lead to universality may be found on page 129 in [Bis95].

When using feed-forward networks for e.g., time series prediction, the modeler is not only faced with the problem of determining how many hidden units are appropriate, it is also necessary to choose the lag space vector used as input. It is necessary to determine how many previous values of the time series on which to base the prediction of the next value as well as the *delay time* between each of these values. If using too few previous values then it will not be possible to capture the dynamics of the system that generated the data from which we are modeling, and prediction accuracy will suffer. If using too many previous values on the other hand, training the network model of the generating system may possibly suffer due to redundancy in the inputs as well as from the increased number of parameters to be determined from the data. The proper number of lags to use must therefore be determined somehow. The literature contains several suggestions on how to choose an appropriate lag space; these will however not be explained in detail here. For references, suffice it to mention that the methods include trial-and-error, linear correlation analysis (not appropriate for nonlinear problems), information theoretic methods [PS93] and generalization based methods [Gou97]; for further references see e.g., the introduction in [WG93].

Various enhancements of the basic feed-forward architecture has been suggested in the literature in order to make the network "memory" of previous inputs more flexible, thus relaxing the accuracy with which the dimension of the externally provided lag space vector must be determined. In the following section it will be described how this flexibility can be obtained by making the networks *recurrent*.

3.3 Recurrent neural networks

For feed-forward networks it is necessary to determine the nature of an "optimal" lag space beforehand as described above, i.e., the number of previous observations to use as well as the delay time between these observations. The procedure of choosing a proper lag space however often poses a significant inconvenience for the modeler and it would therefore be of advantage if the determination of an optimal lag space could somehow be incorporated into the network. What is called for is a flexible memory of previous inputs internal to the network which can be adapted to the problem at hand.

Such a memory can be obtained by making the networks *recurrent*, i.e., by somehow providing a feedback of data generated by the network back into the units of the network to be used in future iterations. Feedback of previously generated data indirectly leads to a memory of in principle all previous inputs presented to the network. The key idea of this approach is then to let the network develop an "effective" memory of previous inputs during *adaption*, i.e., an internal memory which utilizes the previous inputs which are relevant to the problem at hand. Such an internal memory may naturally extend beyond the externally provided lag space, thus relaxing the requirements for the external lag space. An adaptive memory internal to the network may in fact completely relieve the modeler from choosing a lag space; this will be the case if the network is able to work entirely from its own internal memory, created from one external input only. Creation of this type of internal memory is the ultimate purpose of applying feedback.

The literature is rich on various ways in which to introduce feedback into neural networks and numerous recurrent network architectures have been proposed. A detailed presentation of all these specific architectures is beyond the scope of this presentation. Rather, a brief overview of the principal feedback paths and the modeling capabilites of the resulting networks will be provided. For a more detailed description of specific recurrent network architectures the reader is referred to e.g., [TB94] and the numerous references therein. Attention is also directed towards the excellent overview given in [Moz93] of the various ways in which memory of previous inputs can be provided for neural networks applied to temporal sequence processing like e.g., time series prediction.

The *type* of feedback to be considered in the following will be from the output side of the units in the network back to the input side, always passing through nonlinear saturating activation functions. This way, the feedback networks considered here are inherently stable in the bounded input, bounded output sense.

One of the simplest ways of introducing feedback into neural networks is by the so-called *local* feedback, where *adaptive* feedback connections are provided only from each hidden unit back to itself, as illustrated in Figure 3.3. The resulting networks are called locally recurrent networks and were treated by e.g., Frasconi, Gori and Soda in [FGS92]. Introduction of the feedback changes the nature of the network model from a static input/output mapping to a *dynamic* system as described in section 2.4. The network now computes its output $y(t)$ not only from the externally provided inputs $\mathbf{x}(t)$ but also from the previous values of the hidden units, the hidden unit *state vector* $\mathbf{s}(t-1)$. Through the hidden unit state vector the network output will therefore indirectly be a function of all previously applied inputs.

The advantages of applying a local feedback only is mainly related to implemental issues. The information necessary for computing the hidden unit outputs is local to each unit in the sense that no information from other hidden units in the same layer is required

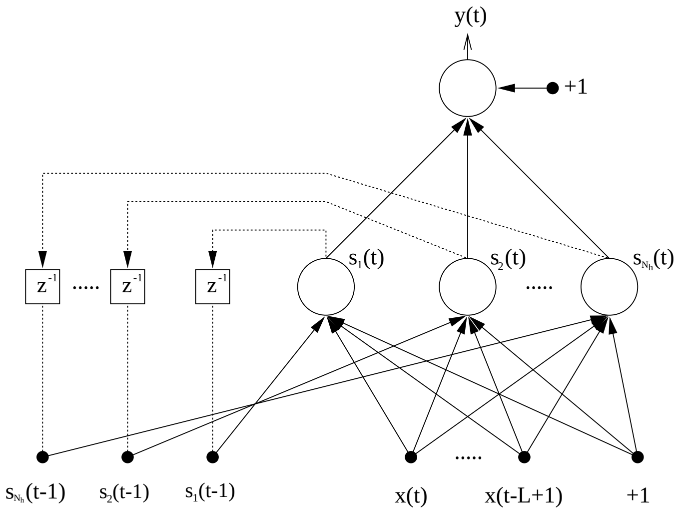


Figure 3.3: Architecture of a local feedback recurrent network.

in order to compute the output of the unit. This property makes the necessary extensions to algorithms for simulation and adaptation in order to handle local feedback very simple, leaving the computational complexity equivalent to that of feed-forward networks.

From a modeling point of view however, the advantages obtained from a local feedback are rather limited. As the hidden units are mutually disconnected they cannot *interact* and the dynamics which such a network is able to implement are rather limited in nature. E.g., locally recurrent networks are not capable of implementing an arbitrary NAR model by working from only a *single* external input as they cannot implement a "shift register" as described on page 13 due to their inability to exchange information between the hidden unit states. Thus, in order to model a time series generating system by use of a locally recurrent network it is still necessary to provide an external lag space input, of which the dimension L must be chosen appropriately. The use of this type of network rather than purely feedforward networks therefore seems questionable.

A more complex network architecture is obtained by use of *global* feedback, where adaptive feedback connections are provided between every pair of hidden units. Global feedback thus means that each hidden unit is fully connected to the hidden unit state vector $\mathbf{s}(t-1)$ containing the previous hidden unit outputs, as illustrated in Figure 3.4. The resulting networks are called fully (or globally) recurrent networks and are sometimes also denoted Elman networks [Elm90].

Due to the higher connectivity of globally recurrent networks their modeling capabilities are much richer than for the simple local feedback networks. In fact is has been proven that a globally recurrent network with a single layer of hidden units have the capacity to model an arbitrary, smooth nonlinear dynamic system, e.g., as illustrated in Figure 2.2 and described by Eq. (2.14). This result has been stated in several papers but was apparently proved for the first time in [SL91], where it was shown that a globally recurrent network model of a dynamic system can be made arbitrarily accurate over any fixed finite length of time. The proof is obtained as an extension of the results from e.g., [Cyb89, Fun89, HSW89] in which the universality of feed-forward networks was proved, as described above. In [SL91] it is shown that a globally recurrent network can perform a mapping of arbitrary accuracy corresponding to *both* the "true system" output as well as

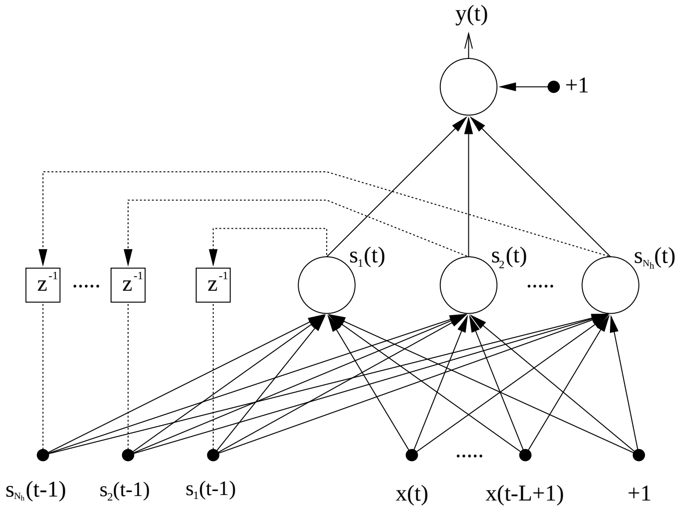


Figure 3.4: Architecture of a globally recurrent (Elman) network.

the true system hidden states. Referring to the dynamic system defined by Eq. (2.14), the principle underlying the proof is to model the true system hidden states $\mathbf{v}$ generated by the internal dynamics $\mathbf{F}$ by a *subset* of the recurrent network hidden units; refer to Figure 3.4. These hidden unit outputs are combined linearly by their corresponding feedback weights leading back to the hidden units whereby it is possible to form the true system hidden states $\mathbf{v}$ according to the universal approximation theorem for feed-forward networks. Additional hidden units *without* feedback weights are assigned to form a purely feed-forward mapping corresponding to the output $h(\mathbf{f}(\mathbf{s}, x))$ of the dynamic system, where x represents an arbitrary input vector. Finally it is shown how a maximum allowed error of ε after say, T time steps can be traced back to an initially allowed maximum error of δ on the hidden state $\mathbf{s}$, which concludes the proof.

The fact that fully recurrent networks can model arbitrary dynamic systems means that this model type is capable of modeling any time series, working from only a single external input. If delayed inputs are vital for the modeling they can be provided by a subset of the hidden units implementing a shift register as described on page 13. This way, fully recurrent networks is a model class for which the sometimes very tedious procedure of choosing a proper dimension of the lag space can be completely eliminated. It is of course still possible to provide an external lag space in order to "assist" the fully recurrent network with memory of previous inputs as indicated in Figure 3.4 but it is no longer a necessity. The improved modeling capabilities however comes at the price of non-local algorithms and a possibly significantly increased computational burden compared to feed-forward networks as will be described in chapter 4.

The fully recurrent network structure illustrated in Figure 3.4 can be augmented by a feedback path from the output back to the hidden units. This type of recurrent network is sometimes denoted Williams-Zipser networks [TB94, WZ89].

If feedback from the output to the hidden units is maintained while eliminating the feedback connections between the hidden units we arrive at a recurrent network structure as illustrated in Figure 3.5. This simplified network structure may generally be termed output feedback recurrent networks. In the literature this recurrent network structure is sometimes denoted as Jordan networks [Jor88].

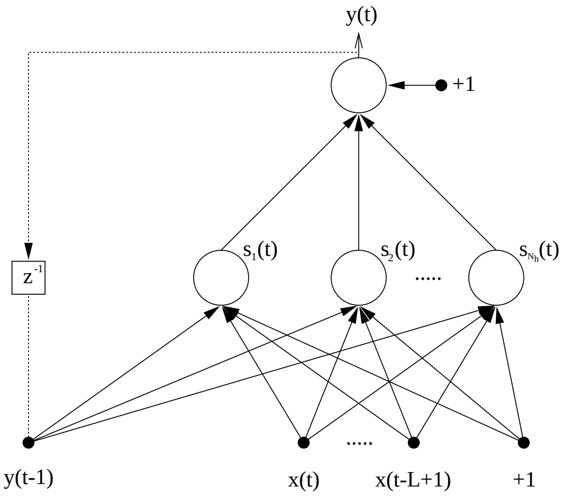


Figure 3.5: Architecture of an output feedback/output error recurrent network.

Referring to the system identification literature, a model which contains feedback of its own previous outputs only is denoted an *output error* model [Lju87, Sjö95, Nør96]. Apparently there seems to be a slight misunderstanding in the neural network literature regarding the model structure terminology, as output feedback recurrent networks are often erroneously denoted NARX networks (Nonlinear Auto Regressive with eXogenous inputs) [LHTG96, SHG97, GLH97, LHG96, LGHK97]. However, NARX models do *not* contain internal feedback, they are purely feedforward in nature as it is mentioned in e.g., [NRRU$^+$94, Sjö95]. The inputs to a NARX model is the exogenous inputs to the system to be modeled along with *observations* of the true system output; refer to Figure 2.1. On the other hand, an output error model as illustrated in Figure 3.5 works from the exogenous inputs along with *estimates* of the true system output, corresponding to the model output. The misunderstanding thus seems to have arised due to a confusion of *observed* system outputs with *estimated* system outputs.

Whereas local feedback networks are not capable of modeling an arbitrary dynamic system, it has been proved that output feedback networks indeed have the same modeling capabilities as fully recurrent networks. In [SHG97] it was shown how to construct an output feedback network having $N + 1$ hidden units and a feedback of the previous $2N$ network output values, capable of simulating a fully recurrent network having N hidden units; both network types are assumed to work from a single external input only. The output feedback model however suffers from what in [SHG97] is denoted linear slowdown, i.e., in order to simulate *one* time step of the fully recurrent network having N hidden units the output feedback network needs to be iterated $N + 1$ times with the external input kept constant. Nevertheless, as any fully recurrent network can be simulated by an output feedback network, output feedback networks are in principle capable of modeling arbitrary dynamic systems, as is the case for fully recurrent networks.

The basic feedback paths have been described above for networks having only a single layer of hidden units. It is of course possible to extend the network structures to multiple layers of hidden units as well. The literature contains numerous suggestions to "novel" recurrent network architectures which are obtained by combining multiple layers of hidden units, each layer possibly having a different feedback type (local, fully connected), as well as various forms of connectivity between layers. In addition, the simple delays and connections used in the illustrations above may be replaced by arbitrary *filters*. E.g., linear FIR filters were suggested in [Wan90, BT91, Wan93], linear IIR (Infinite Impulse Response) filters were suggested in [BT91] and a filter type denoted *gamma memory* was suggested in [dVP92]. In the case of replacing feedback connections with linear FIR filters of order D, D is sometimes referred to as the *memory order*. This way, the number of possible combinations of feedback types and connection types leading to different recurrent network architectures seems inexhaustible.

The motivation for increasing e.g., the number of hidden layers and the memory order is to somehow *assist* the basic recurrent network architectures outlined above in modeling the problem at hand, even though fully recurrent networks with simple delays and a single layer of hidden units are theoretically capable of modeling an arbitrary dynamic system. Increasing the memory order will lead to a direct access to information relating several steps back in time, thus relieving the network of implicitly creating this memory in the hidden units. Furthermore, increasing the number of hidden layers tends to increase the complexity of the mappings it is possible to implement, compared to a single layer network containing the same number of parameters. Even so, deviating from the basic recurrent network architectures leads to a variety of additional choices that need to be addressed by the modeler in terms of e.g., memory order, number of hidden layers, number of units in each layer etc. The Cartesian product of the spaces of all possible choices might easily lead to an explosion in the number of possible architectures to explore, making it difficult to determine an "optimal" architecture for the problem at hand. It therefore seems reasonable to initially explore the possibilities of recurrent network architectures utilized to their *full* and involving the *least* number of choices to be made, before moving on to consider more complicated network constructions.

3.3.1 RNN architectures considered in this work

The recurrent network architectures *initially* considered in this work are fully recurrent networks inspired from [WZ89] as illustrated in Figure 3.6. During this work this general architecture has however been slightly "trimmed" as will be described in more detail at the end of the section. The full architecture is described for completeness of presentation.

The motivation for choosing this particular recurrent network structure has been that it is theoretically capable of modeling an arbitrary dynamic system working from only a single externally provided input, thus making it possible to completely eliminate the need for an externally provided lag space as stated above. Consequently, for a fully recurrent network the only choice necessary to make by the modeler is the number of hidden units to use in the network. As mentioned above, output feedback networks are capable of modeling arbitrary dynamic systems as well, but for this network type it is still necessary to provide an estimate of the memory order to use for the output feedback, thus in effect converting the choice of a proper input dimension into the choice of a proper memory order. Therefore the single hidden layer fully recurrent network structure considered in this work seems to be the basic recurrent network architecture which provides maximum modeling ability while requiring the fewest possible number of choices to be made by the modeler.

The specific update formulas for the units of the recurrent networks considered are provided in the following; refer to Figure 3.6. Let $\mathbf{x}(t)$ denote a vector containing the L

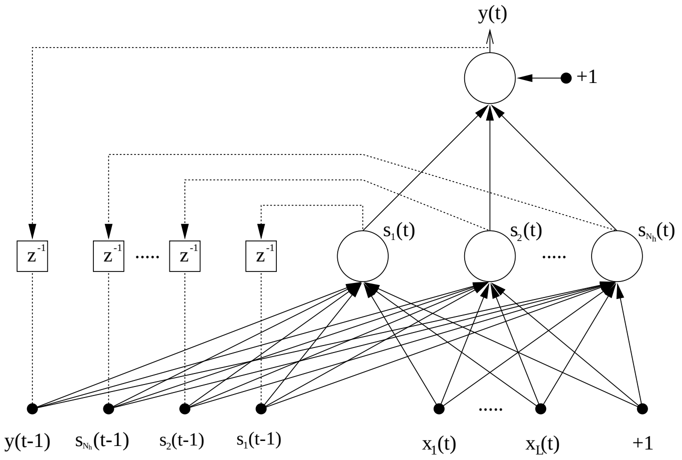


Figure 3.6: RNN architecture *initially* considered in this work.

external inputs at time t, let $\mathbf{s}(t)$ denote a vector containing the N_h hidden unit outputs at time t and let $y(t)$ denote the output unit output. For convenience in the weight labeling we collect the combined inputs to the hidden units at time t in a *state vector* $\mathbf{z}^h(t)$ whose kth element is defined as

$$z_k^h(t) = \begin{cases} x_k(t) & , \quad k \in I \\ s_k(t-1) & , \quad k \in H \\ y(t-1) & , \quad k = o \end{cases} \tag{3.5}$$

where we have arranged the indices on $\mathbf{x}$, $\mathbf{s}$ and y so that I denotes the set of L indices for which z_k^h is an input, H denotes the set of N_h indices for which z_k^h is the output of a hidden unit and o denotes the element corresponding to the output unit output. The activation of the kth hidden unit is now calculated according to

$$\begin{aligned} s_k(t) &= f\left[v_k(t)\right] \quad , \quad k \in H \qquad (3.6) \\ &= f\left[\sum_{j \in I \cup H \cup o} w_{kj} z_j^h(t) + w_{kb}\right] \\ &= f\left[\sum_{j \in I} w_{kj} x_j(t) + \sum_{j \in H} w_{kj} s_j(t-1) + w_{ko} y(t-1) + w_{kb}\right] \end{aligned}$$

where $v_k(t)$ denotes the weighted input to hidden unit k, $f(\cdot)$ is the nonlinear activation function set to $f(x) = \tanh(x)$ in this work, and w_{kb} is the bias weight. The hidden unit outputs are then forwarded to the output unit which sees the input vector $\mathbf{z}^o(t)$,

$$z_k^o(t) = s_k(t) \quad , \quad k \in H \tag{3.7}$$

The output $y(t)$ of the recurrent networks considered here is linear in order to allow for arbitrary dynamic range; this is furthermore required in order for the universal approximation theorem to apply. The network output is thus updated according to

$$\begin{aligned} y(t) &= \sum_{j \in H} w_{oj} z_j^o(t) + w_{ob} \qquad (3.8) \\ &= \sum_{j \in H} w_{oj} s_j(t) + w_{ob} \end{aligned}$$

where w_{ob} is the output unit bias. Note that the output unit does *not* have feedback of its own previous value as feedback in a linear unit is very likely to result in stability problems [Ped94].

When performing the first iteration at time, say, $t = 1$ it is customary [WZ89] to set the values of the hidden and output unit outputs in the previous time step to zero, i.e.,

$$s_k(0) = 0 \ , \quad k \in H \tag{3.9}$$
$$y(0) = 0 \ . \tag{3.10}$$

Setting the initial previous outputs to zero means that these values will not influence the network output in the first iteration, and the recurrent networks considered here will therefore perform a purely feed-forward mapping in the first iteration.

The feed-forward mapping in the first iteration is due to the layered update of the units in the recurrent network, i.e., the hidden unit outputs are updated prior to the update of the output unit, just as for feed-forward networks. This layered update should be opposed to the synchronous update described in e.g., [WZ89] where all units in the network are updated *simultaneously.* In [Ped94, PH95] it was shown that when using fully recurrent networks for time series prediction, layered update is preferable as a synchronous update of the units effectively results in a two-step ahead predictor. A demonstration of this effect is provided in appendix B.

In the beginning of this section it was stated that the network illustrated in Figure 3.6 was the *initially* considered architecture. During this work it has however been recognized that the feedback from the linear output unit is completely superfluous in a fully recurrent network for which reason it should never be included in the network architecture; this will be demonstrated in section 7.2.1. Furthermore, a lag space of external inputs was initially employed but later on discarded as it was recognized that this is indeed unnecessary for fully recurrent networks. The ultimate resulting recurrent network architecture with which the experiments in the following have been performed is illustrated in Figure 3.7.

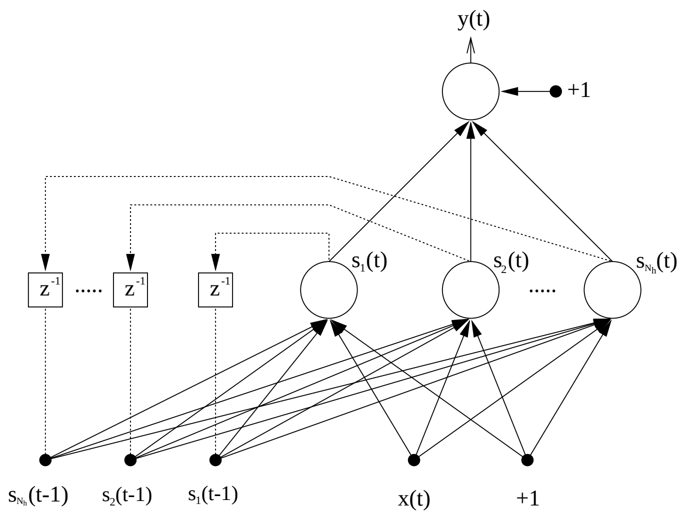


Figure 3.7: *Ultimate* RNN architecture considered in this work.

Chapter 4

Training adaptive models

Once the model structure has been selected the parameters must be adapted to the problem at hand; this problem is also called "learning" or "training" in the neural network literature. For recurrent networks the most commonly adopted approach towards training is by gradient descent-like procedures. This is so even though it is generally acknowledged that such methods are highly ineffective for recurrent network training [Moz93, PF94, TB94] due to long training times and lack of ability to find a good solution.

Many successful applications of gradient descent to *feed-forward* networks have been reported in the literature. However, significant effort has been directed towards improving the basic training method in terms of both speed and quality of obtained solutions for this model type. This has lead to the application of a multitude of different training methods for feed-forward networks including novel methods as well as methods wellknown from other contexts. The reader is referred to e.g., [Møl93] for an overview of training methods for feed-forward networks. Among the most successful [Møl93, Nør96] are the class of *second-order* methods, involving second-order information of the cost function.

Despite the good results with second-order methods for feed-forward networks very few attempts have been reported in the literature to apply training methods to recurrent networks which are *significantly* different from the basic gradient descent. Among the attempts is e.g., [PF94] where recurrent networks were trained using an Extended Kalman Filtering method with encouraging results. In [BSF94] a more traditional optimization method was used, namely a so-called pseudo-Newton method, involving a diagonal approximation to the Hessian matrix with second derivatives.

The type of training considered in this work is "offline" or "batch" training, i.e., update of the model parameters only after presentation of all examples to the model. The training problem has therefore been viewed as a nonlinear unconstrained function optimization problem. The approach in this work has been to apply *full* Newton type training methods, involving the full Hessian matrix. In particular, the *damped Gauss-Newton* method has been considered as it has turned out to be very efficient.

This chapter briefly reviews the optimization theory which is of relevance to the present work. For a more elaborate presentation and description of alternative optimization methods the reader is referred to e.g., the classic text books [GMW81, DS83]. In section 4.1 the training problem is defined in terms of minimizing a cost function and section 4.2 reviews the framework for optimization. Section 4.3 describes the traditionally used gradient descent and is followed by section 4.4 which describes the class of second-order methods considered in this work. Section 4.5 contains a discussion of various stopping criteria and section 4.6 reviews the alternative to offline methods, namely *online* training methods.

The chapter is concluded by sections 4.7 and 4.8 which describe the calculation of the gradient and Hessian matrix, respectively, for the recurrent networks considered in this work.

4.1 Definition of the training problem

In order to train the selected model structure we need to obtain a database $\mathcal{T}$ of examples consisting of inputs $\mathbf{x}(t)$ and corresponding desired outputs $d(t)$, $\mathcal{T} = \{\mathbf{x}(t), d(t)\}_{t=1}^{T}$ where T is the number of training examples. E.g., for autoregressive one step ahead prediction problems we have $\mathbf{x}(t) = [x(t), x(t-1), \ldots, x(t-L+1)]$ and $d(t) = x(t+1)$. We then need to train the model, i.e., adjust the parameters so as to obtain a "good" model. We thus need to somehow characterize a good model. Here, it seems reasonable to demand that a good model describes the observed data well, i.e., makes small errors $e(t)$,

$$e(t) = d(t) - y(t|\mathbf{w}, \mathbf{x}(t)) \tag{4.1}$$

where $d(t)$ is the desired output at time t and $y(t|\mathbf{w}, \mathbf{x}(t))$ is the model output given the external inputs $\mathbf{x}(t)$ and concatenated set of parameters $\mathbf{w}$; in the following the dependency upon the parameters will often be implicitly assumed and the model output at time t is simply denoted as $y(t)$. Note that whereas the order in which the examples are presented to the model is not important for feed-forward networks, the examples need to be presented to recurrent networks in proper time ordering in order for the output to make sense.

Based on Eq. (4.1) we can compute the errors for all examples in the database, or training set, $\mathcal{T}$. The parameters $\mathbf{w}$ of the model should be chosen so that the resulting errors $e(t), t = 1, \ldots, T$ become as small as possible. A commonly used criterion for measuring the size of the errors is by the *quadratic cost function*, defined as

$$E(\mathbf{w}) = \frac{1}{2} \sum_{t=1}^{T} [e(t)]^2 \tag{4.2}$$

where the factor $\frac{1}{2}$ is included for convenience in subsequent derivations and where the dependency on the training database $\mathcal{T}$ is implicitly assumed. A criterion like Eq. (4.2) which maps the model parameters $\mathbf{w}$ onto a scalar value is termed a *cost function*. We can now define the training problem as the problem of determining a set of parameters $\widehat{\mathbf{w}}$ which *estimates* the true minimizer $\mathbf{w}^*$ of the cost function. Minimization of Eq. (4.2) belongs to the class of *Prediction Error Methods* [Lju87] for which training is generally defined as a minimization of the sum of transformed prediction errors.

A reassuring property of the quadratic cost function $E(\mathbf{w})$ is that it results from several sound basic principles from which the learning problem may be interpreted. Among these is the simple geometrical interpretation where training is viewed as minimization of the Euclidean distance between two vectors containing the desired outputs and the model outputs, respectively. Another interpretation is the principle of Maximum Likelihood where the object is to maximize the probability of the observed data. Assuming that the prediction errors are Gaussian with known variance, the likelihood is maximized by minimization of $E(\mathbf{w})$. For further details and interpretations of the quadratic cost the reader is referred to the excellent review in [Lju87].

4.2 Optimization

Once the cost function has been determined, training becomes an optimization problem in which the object is to find a set of parameters $\mathbf{w}^*$ that minimizes the cost function,

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} E(\mathbf{w}) \tag{4.3}$$

For the quadratic cost function Eq. (4.2) the optimization problem is known as the least squares problem which is thoroughly treated in the literature; see e.g., [Bjö96, DS83].

In order to characterize a minimum $\mathbf{w}^*$ we employ the Taylor expansion to second order of $E(\mathbf{w})$ around $\mathbf{w}^*$,

$$E(\mathbf{w}) = E(\mathbf{w}^*) + (\mathbf{w} - \mathbf{w}^*)^{\mathrm{T}}\mathbf{g}(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^{\mathrm{T}}\mathbf{H}(\mathbf{w}^*)(\mathbf{w} - \mathbf{w}^*) \tag{4.4}$$

where we have defined the gradient vector as

$$\mathbf{g}(\mathbf{w}^*) = E'(\mathbf{w}^*) = \left.\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\right|_{\mathbf{w}=\mathbf{w}^*} \tag{4.5}$$

and the Hessian matrix with the second derivatives as

$$\mathbf{H}(\mathbf{w}^*) = E''(\mathbf{w}^*) = \left.\frac{\partial^2 E(\mathbf{w})}{\partial \mathbf{w}\partial \mathbf{w}^{\mathrm{T}}}\right|_{\mathbf{w}=\mathbf{w}^*} \tag{4.6}$$

In order for $\mathbf{w}^*$ to be a *local* minimum of $E(\mathbf{w})$ the necessary *and* sufficient conditions are [GMW81]

1. $\mathbf{w}^*$ is a stationary point, $\mathbf{g}(\mathbf{w}^*) = 0$
2. $\mathbf{H}(\mathbf{w}^*)$ is positive definite, $\mathbf{w}^{\mathrm{T}}\mathbf{H}\mathbf{w} > 0 \;,\; \forall \mathbf{w}$

In the general case it is not possible to determine a minimizer $\mathbf{w}^*$ of $E(\mathbf{w})$ analytically and we therefore have to resort to iterative methods. The principle of iterative methods is to start from an initial parameter estimate $\mathbf{w}_0$ possibly chosen at random and then proceed by generating a sequence $\{\mathbf{w}_k\}$ of parameter estimates that (hopefully) converges towards a minimizer $\mathbf{w}^*$. Convergence of $\{\mathbf{w}_k\}$ to $\mathbf{w}^*$ is characterized by

$$\lim_{k\to\infty} \|\mathbf{w}_k - \mathbf{w}^*\| = 0 \tag{4.7}$$

where $\|\cdot\|$ is a suitable vector norm. The general form of the parameter update is usually

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \eta\Delta\mathbf{w}_k \tag{4.8}$$

where $\mathbf{w}_k$ is the kth *iterate* of the parameters, $\Delta\mathbf{w}_k$ denotes a *search direction* and η specifies the *step size*, that is, the relative size of the step taken in the direction of search.

In the general case of a nonlinear model structure the quadratic cost function may have several points $\mathbf{w}^*$ satisfying the conditions for a local minimum as listed above. Points $\mathbf{w}^*$ satisfying

$$E(\mathbf{w}^*) \leq E(\mathbf{w}) \;\;,\;\; \forall \mathbf{w} \tag{4.9}$$

are termed *global* minima and are naturally the points of prime interest. Unfortunately there is no known general method for finding a global minimum of the cost function, and iterative methods as outlined in Eq. (4.8) can only be guaranteed to converge towards a local minimum. The only way in which to render a local minimum possible as a global minimizer is to repeat the iterative optimization method from different initial starting points $\mathbf{w}_0$ and compare the value of the cost for the resulting solutions. When this procedure is applied to model structures like neural networks one should be aware that all local minima $\mathbf{w}^*$ will be ambiguous. Due to symmetries in general neural network models it is possible to create permutations of the units in the networks that will leave the functional mapping implemented by the model unchanged. These permutations lead to a reorganization of the weights in the parameter vector $\mathbf{w}$ and thus to the ambiguity of the minima.

4.3 First-order methods

When choosing the search direction $\Delta\mathbf{w}_k$ it seems reasonable to choose a direction that leads "downhill" for the cost function $E(\mathbf{w})$, i.e., a direction from the current iterate $\mathbf{w}_k$ in which E decreases. This allows for determination of a next iterate $\mathbf{w}_{k+1}$ in such a way that $E(\mathbf{w}_{k+1}) < E(\mathbf{w}_k)$. This will be the case if the directional derivative of E at $\mathbf{w}_k$ in the direction $\Delta\mathbf{w}_k$ is negative, that is, if

$$\mathbf{g}(\mathbf{w}_k)^{\mathrm{T}}\Delta\mathbf{w}_k < 0 \tag{4.10}$$

A natural choice for $\Delta\mathbf{w}_k$ is the direction in which E descreases most rapidly from $\mathbf{w}_k$, namely the opposite of the gradient direction. This direction is referred to as the direction of steepest-descent. The parameter vector is then updated in each iteration as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta\mathbf{g}(\mathbf{w}_k) \tag{4.11}$$

which is known as the *method of steepest descent* or simply gradient descent. Methods of the form of Eq. (4.11) are generally termed first-order methods as they only involve first-order derivatives of the cost function. For the quadratic cost function used here the ith component of the gradient is calculated as

$$g_i(\mathbf{w}) = \frac{\partial E(\mathbf{w})}{\partial w_i} = -\sum_{t=1}^{T} e(t)\frac{\partial y(t)}{\partial w_i} \tag{4.12}$$

where $\partial y(t)/\partial w_i$ is the partial derivative of the model output wrt. parameter w_i.

Important to the convergence towards a local minimum is the choice of the step size η. If we choose η small enough we can always obtain a decrease of the cost function; however, a small η leads to slow convergence towards a local minimum. A larger value of η might lead to faster convergence but too large a value will lead to divergence and thus make the training algorithm unstable.

A commonly applied strategy in the neural network community is to keep η fixed throughout training [Hay94, HKP91]. Provided we are close enough to a local minimum $\mathbf{w}^*$ to allow for fairly accurate description of the cost function by a second-order Taylor expansion Eq. (4.4) it can be shown [WS85] that in this case η must be chosen as

$$0 < \eta < \frac{2}{\lambda_{max}} \tag{4.13}$$

where λ_{max} is the largest eigenvalue of the Hessian evaluated in the local minimum, $\mathbf{H}(\mathbf{w}^*)$. If employing a first-order method for training we usually do not have second-order information of the cost function available; furthermore, we do not know $\mathbf{w}^*$ beforehand. In practice, η is therefore often chosen "by hand" in a rather ad hoc fashion by trial-and-error.

An appealing alternative to keeping the step size fixed is to invoke a *line search* [DS83] in each iteration. A line search works by examining the cost function starting from the current iterate in the direction of search and choosing an η_k which leads to a reduction of the cost function. An *exact* line search seeks an η_k that solves

$$\min_{\eta_k > 0} \; E(\mathbf{w}_k + \eta_k \Delta\mathbf{w}_k) \tag{4.14}$$

If the η_k chosen only approximates the minimizing step size the line search algorithm is called an *inexact* line search. The literature is rich on various line search algorithms, see e.g., [DS83]. In this work a simple inexact line search adopted from [DS83] is used. The idea is to choose an initial value for η_k and then to keep halving it until a decrease in the cost is obtained. Here, the method is slightly refined in that it is examined whether further reduction of η_k will lead to a further decrease of the cost function.

From this procedure we note that employing a line search involves an increased computational burden compared to using a fixed step size. However, the tedious (and computationally demanding as well) work of choosing a proper step size "by hand" is eliminated. In [Ped94] it was empirically found that the halving line search suggested above in practice works just as well or better than an "optimally" chosen fixed step size.

Gradient descent methods are globally convergent optimization methods meaning that they will converge to a local minimum $\mathbf{w}^*$ or a *saddlepoint*[1] from almost every starting point $\mathbf{w}_0$ [DS83]. Usually, convergence to a saddlepoint is not a problem in practice. However, the convergence towards $\mathbf{w}^*$ is only linear. Linear convergence means that in each iteration the error is reduced as

$$\|\mathbf{w}_{k+1} - \mathbf{w}^*\| \le c\|\mathbf{w}_k - \mathbf{w}^*\| \;\;, \;\; c \in [0,1) \tag{4.15}$$

Often convergence is very slowly linear, i.e., the constant c is very close to unity. We shall return to the convergence properties of gradient descent in chapters 7 and 9.

Despite the slow convergence, the choice of gradient descent-like methods has been dominating training in the neural network community. The reasons for this are probably rooted in the ease of implementation as well as in the history of neural networks. Gradient descent was the method of choice in [RM86] which made the efficient computation of gradients for feed-forward networks known as *backpropagation* widely known, and revived the application of and research in neural networks. From an optimization point of view, however, the method was not an optimal choice.

4.4 Second-order methods

A class of optimization methods which has been found to be very efficient is the second-order methods. Whereas first-order methods like gradient descent involve only first derivatives of the cost function, second-order methods include second derivative information as well. The principle underlying second-order methods is to expand the cost function to second order around the current iterate $\mathbf{w}_k$ in each iteration,

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}_k) + \Delta\mathbf{w}^{\mathrm{T}}\,\mathbf{g}(\mathbf{w}_k) + \frac{1}{2}\Delta\mathbf{w}^{\mathrm{T}}\,\mathbf{H}(\mathbf{w}_k)\,\Delta\mathbf{w} \tag{4.16}$$

[1]Stationary point in which the Hessian is indefinite.

where $\mathbf{w} = \mathbf{w}_k + \Delta\mathbf{w}$ and where element ij in the Hessian for the quadratic cost function Eq. (4.2) is calculated as

$$H_{ij}(\mathbf{w}_k) = \left.\frac{\partial^2 E(\mathbf{w})}{\partial w_i \partial w_j}\right|_{\mathbf{w}=\mathbf{w}_k} = -\sum_{t=1}^{T} \left[e(t) \frac{\partial^2 y(t)}{\partial w_i \partial w_j} - \frac{\partial y(t)}{\partial w_i} \cdot \frac{\partial y(t)}{\partial w_j} \right] \tag{4.17}$$

A search direction is then determined based on the second-order expansion. In the following various second-order methods will be reviewed.

4.4.1 The Newton method

In the optimization algorithm known as Newton's method the next iterate $\mathbf{w}_{k+1}$ is chosen as the minimizer of the second-order expansion $\widetilde{E}(\mathbf{w})$. The step $\Delta\mathbf{w}_k$ leading to the minimum is determined by setting the derivative of Eq. (4.16) to zero,

$$\nabla \widetilde{E}(\mathbf{w}) = \mathbf{g}(\mathbf{w}_k) + \mathbf{H}(\mathbf{w}_k)\,\Delta\mathbf{w}_k = 0 \tag{4.18}$$

where the symmetry of the Hessian is used. From this expression we see that the $\Delta\mathbf{w}_k$ leading to the minimum can be obtained as the solution to a system of linear equations,

$$\Delta\mathbf{w}_k = -\left[\mathbf{H}(\mathbf{w}_k)\right]^{-1} \mathbf{g}(\mathbf{w}_k) \tag{4.19}$$

The resulting weight change is denoted the *Newton step* and the parameters are updated according to $\mathbf{w}_{k+1} = \mathbf{w}_k + \Delta\mathbf{w}_k$, i.e., with step size $\eta = 1$.

Newton's method is based on the assumption that the approximation $\widetilde{E}$ is fairly accurate so that minimizing Eq. (4.16) will lead to a decrease of the true cost function E. In practice, however, the expansion is only valid in a certain neighbourhood around the expansion point, and taking the full Newton step might actually lead to an *increase* of the cost function. Newton's method should therefore be combined with a line search using Eq. (4.19) as search direction, leading to the parameter update

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta\left[\mathbf{H}(\mathbf{w}_k)\right]^{-1} \mathbf{g}(\mathbf{w}_k) \tag{4.20}$$

This combination is called a *damped* Newton method [DS83].

A problem for Newton methods is that the search direction $\Delta\mathbf{w}_k$ obtained from Eq. (4.19) will only be a descent direction, i.e., satisfying Eq.(4.10), if the Hessian is positive definite. Far away from a local minimum the Hessian might be indefinite or even negative definite, resulting in a search direction towards a local maximum. For this reason the Newton method is only *locally* convergent meaning that it will only converge "close" to a local minimum. Newton's method should therefore be combined with a globally convergent method like gradient descent, switching whenever the Newton step is not a descent direction.

The advantage of the Newton method compared to gradient descent is that close to a minimum $\mathbf{w}^*$ convergence is quadratic [DS83],

$$\|\mathbf{w}_{k+1} - \mathbf{w}^*\| \leq c\|\mathbf{w}_k - \mathbf{w}^*\|^2 \ , \quad c \geq 0 \tag{4.21}$$

and therefore much faster than for gradient descent. Quadratic convergence can be conceived as the number of correct digits roughly doubling in each iteration close to $\mathbf{w}^*$.

Despite the fast local convergence of Newton's method it is rarely used for training adaptive models. This not only due to the lack of global convergence abilities but also due to the often vast computational burden associated with calculation of the second derivatives of the model output $y(t)$ entering Eq. (4.17).

4.4.2 The Gauss-Newton method

Instead of using the true Hessian Eq. (4.17) when solving least squares problems we can use an *approximation* to the second derivatives,

$$H_{ij}(\mathbf{w}_k) = \left.\frac{\partial^2 E(\mathbf{w})}{\partial w_i \partial w_j}\right|_{\mathbf{w}=\mathbf{w}_k} \approx \sum_{t=1}^{T} \frac{\partial y(t)}{\partial w_i} \cdot \frac{\partial y(t)}{\partial w_j} \tag{4.22}$$

where we simply neglect the term in Eq. (4.17) involving second derivatives of the model output. The approximation will be exact in the limit of an infinite number of examples T provided that the model is capable of implementing the teacher function (refer to chapter 2) for a set of parameters $\mathbf{w}^*$. In the limit the prediction errors $\{e(t)\}$ will have zero mean and be independent in which case the second-order term will be zero [Lju87]. Thus, for a "good" model trained on a large number of examples, the second-order term will be "small" close to $\mathbf{w}^*$ which justifies the approximation.

Using this approximation when determining the weight change in Eq. (4.19) leads to the so-called *Gauss-Newton method* [DS83, Bjö96], and the resulting step is denoted the Gauss-Newton step. We note that the approximation does not require additional information to that already available from the gradient calculation, leading to a fairly straightforward implementation.

The Hessian $\mathbf{H}$ for the quadratic cost function Eq. (4.17) may also be written as

$$\mathbf{H} = \mathbf{J}^{\mathrm{T}}\mathbf{J} + \mathbf{S} \tag{4.23}$$

where

$$J_{ti} = \frac{\partial y(t)}{\partial w_i} \quad , \quad S_{ij} = -\sum_{t=1}^{T} e(t) \frac{\partial^2 y(t)}{\partial w_i \partial w_j} \tag{4.24}$$

The matrix $\mathbf{J}$ is called the *Jacobian* matrix and $\mathbf{S}$ is the matrix with second-order terms omitted in the Gauss-Newton approximation, which reads

$$\mathbf{H} \approx \mathbf{J}^{\mathrm{T}}\mathbf{J} \tag{4.25}$$

Besides relieving from the computational burden of computing the second derivatives of the model output, it is clearly seen from Eq. (4.25) that the Gauss-Newton approximation to the Hessian is always positive (semi-) definite, ensuring that the Gauss-Newton step is a descent direction.

As we are still working from an approximation to the cost function, taking the full Gauss-Newton step might actually increase the cost function and may prevent the Gauss-Newton method from converging. The Gauss-Newton method should therefore be combined with a line search, which will make the method globally convergent [DS83]. Involving a line search leads to a parameter update equivalent to Eq. (4.20) and the resulting method is called the *damped* Gauss-Newton method [DS83].

In this work is used the step size halving line search described in section 4.3 in combination with the Gauss-Newton method. The initial value for the step size η is usually set to unity [DS83], $\eta = 1$, which corresponds to taking the full Gauss-Newton step. One might consider to set the initial step size to a value somewhat larger than one. This will in theory allow the damped Gauss-Newton method to escape from the neighbourhood of a local minimum at which the value of the cost function is relatively high, to a lower-lying region of the cost function surface. The benefit from this approach is however rarely observed in practice.

The *local* convergence properties of the Gauss-Newton method will clearly depend on whether the omitted second-order term $\mathbf{S}$ is significant at the local minimum $\mathbf{w}^*$, that is, whether $\mathbf{S}(\mathbf{w}^*)$ is "large" compared to $\mathbf{J}^T\mathbf{J}$ in Eq. (4.23). For models which are linear in the parameters the second-order term is always zero and the approximation thus exact; in this case the Gauss-Newton method finds the (global) minimum in a single iteration. For the so-called *zero residual* problems where all the errors $e(t)$ are zero in $\mathbf{w}^*$ the omitted second-order term $\mathbf{S}(\mathbf{w}^*)$ will be zero as well and the Gauss-Newton method will have quadratic convergence just as the Newton method.

If the second-order term is not negligible at the local minimum, convergence will generally only be linear [DS83]. Even though the theoretical local convergence properties for the Gauss-Newton method in this case are not better than for gradient descent, experience has shown that the damped Gauss-Newton method completely outperforms gradient descent. A drawback of Newton type methods seems to be the requirement of solving a system of linear equations in each iteration. Even so, the damped Gauss-Newton method usually converges orders of magnitude faster than gradient descent measured in actual computation time, as will be illustrated in chapter 9.

Looking at Eq. (4.19) one might get the impression that in order to obtain the search direction we actually invert the Hessian matrix and take its product with the gradient vector. Usually it is however more effective regarding both precision and computational speed to solve the system of linear equations resulting from Eq.(4.18). This is so even if using the *matrix inversion lemma* [Lju87] for iterative computation of the inverse Hessian, as suggested in e.g., [HS93]. In appendix C it is demonstrated that in most cases it is actually more computation intensive to iteratively compute the inverse Hessian and take the product with the gradient than it is to solve the system of linear equations.

Furthermore, solving the system might be more robust in case of a singular Hessian. If the Hessian is singular, the inverse $\mathbf{H}^{-1}$ cannot be computed. However, as long as the gradient $\mathbf{g}$ in Eq. (4.18) lies in the subspace spanned by $\mathbf{H}$, i.e., the *range* of $\mathbf{H}$, the system of linear equations will have a solution which can be obtained by the Singular Value Decomposition (SVD) method [PFTV92]. We are however still likely to run into problems if the Hessian is singular or very *ill-conditioned*. We shall return to the problems of singularity and ill-conditioning of the Hessian in more detail in chapter 7.

As a final note it is here suggested to always precede the damped Gauss-Newton method by a few iterations of gradient descent even though the damped Gauss-Newton is globally convergent. The reason for this is that initially the errors $e(t)$ are generally relatively large as we start from a randomly chosen set of parameters. The large errors mean that the second-order term $\mathbf{S}$ omitted in the Gauss-Newton approximation Eq. (4.25) is large, which will lead to a fairly poor search direction even though it is still a descent direction. Just a few iterations of gradient descent will reduce the initial errors significantly and lead to better search directions for the damped Gauss-Newton method. Even though this strategy is not strictly necessary it tends to speed up convergence and thus reduce overall computation time.

4.4.3 The pseudo Gauss-Newton method

Even though the Gauss-Newton approximation is much simpler to compute than the true Hessian we still need to solve a system of linear equations in each iteration in order to determine the search direction. In order to avoid solving the system of equations it has been suggested [HKP91] to introduce a diagonal approximation to the Gauss-Newton

Hessian, assuming that the diagonal elements are dominating. This leads to the *pseudo* Gauss-Newton method in which the search direction in iteration k for the ith parameter is simply computed as

$$\Delta w_i = -g_i(\mathbf{w}_k)/H_{ii}(\mathbf{w}_k) \tag{4.26}$$

The search direction resulting from the pseudo Gauss-Newton method is thus equivalent to using the Gauss-Newton method separately for each weight. As was the case for both Newton's method and the Gauss-Newton method, pseudo Gauss-Newton should be combined with a line search. In practice it has been found that convergence for the pseudo method when applied to the training of fully recurrent networks is better than for gradient descent [Ped94] but not nearly as fast as for the *full* Gauss-Newton method.

4.5 Stopping criteria

In order to determine when to stop the training we need to employ one or several stopping criteria. The stopping criteria should indicate whether the iterates have converged towards a satisfactory solution $\mathbf{w}^*$ *or* whether further iterations will be fruitless. In the following some common choices of stopping criteria will be discussed.

Maximum number of iterations. An obvious criterion which should always be used is to set an upper limit on the number of iterations performed. The maximum number of iterations allowed indicates our patience with the training algorithm and guarantees that it will terminate in finite time. Maximum number of iterations should however not be the only stopping criterion as it does not give any information regarding the quality of the solution obtained. Rather, it should be used to indicate that something went wrong as the training algorithm did not find a "satisfactory" solution within the expected time.

Size of the gradient. A necessary condition for a solution $\mathbf{w}^*$ to be a local minimum is that the gradient is zero, $\mathbf{g}(\mathbf{w}^*) = \mathbf{0}$, indicating a stationary point. Due to finite precision and the iterative nature of the training algorithms we cannot expect to find a solution that satisfies this condition exactly. In practice, an adequate test for whether the kth iterate satisfies this condition is

$$\|\mathbf{g}(\mathbf{w}_k)\|_2 \leq \epsilon_1 \tag{4.27}$$

where $\|\cdot\|_2$ denotes the Euclidean norm and ϵ_1 is a *sufficiently* small positive constant. This test for a stationary point is not only a necessary condition, but in fact also a *sufficient* condition for being close to a local minimum. This is so because convergence towards a saddlepoint or a local maximum is practically impossible if we make sure always to use a descent direction in combination with a line search [DS83].

Lower bound for parameter change. As a criterion for whether the training algorithm is progressing satisfactory or has ground to a halt, we can use a lower bound on the minimally accepted parameter change over a number Δ of iterations,

$$\|\mathbf{w}_{k+\Delta} - \mathbf{w}_k\|_\infty \leq \epsilon_2 \tag{4.28}$$

where $\|\cdot\|_\infty$ denotes the sup norm and ϵ_2 indicates our tolerance with progress as well as the number of significant digits we desire [DS83]. Whenever this criterion is satisfied it is an indication that the training algorithm has either converged or it has stalled due to e.g., numerical problems. In order to decide between the two possibilities we need to employ the gradient criterion.

Lower bound for change in cost function. In some presentations [Hay94] the iterations are considered to have converged when the absolute rate of change in the cost function is sufficiently small,

$$E(\mathbf{w}_{k+1})/E(\mathbf{w}_k) \leq \epsilon_3 \tag{4.29}$$

where ϵ_3 typically lies in the range 0.0001 – 0.01 [Hay94]. Such a stopping criterion however seems rather inappropriate when working with nonlinear models like neural networks. Experience has shown [HKP91] that the cost function surface often possesses very flat "plateaus" and "rain gutter"-like passages beyond which the error all of a sudden drops significantly. Passing through such regions may leave the cost practically unchanged while the weights undergo a dramatic change, and the size of the gradient will often indicate farness from a local minimum. As this stopping criterion furthermore does not provide any information regarding the quality of the obtained solution it seems rather superfluous.

From the above it seems clear that the gradient stopping criterion should always be employed as it indicates the closeness to a local minimum. A proper choice of threshold ϵ_1 will however be problem dependent [DS83] and the investigation of parameter changes should therefore ideally be employed as well in order to ensure that iterations are not terminated prematurely.

In the experiments reported in this work the stopping criteria used has been maximum number of iterations combined with the gradient stopping criterion. The bound on parameter change was omitted as experience showed that if ϵ_1 was chosen in the interval $[10^{-6}; 10^{-3}]$ no further significant weight changes took place within this range of a local minimum.

4.6 Online training methods

The methods for training reviewed above are known as "batch" or "offline" methods, as the training of the model is not integrated into the application. Rather, an example database $\mathcal{T}$ of corresponding input and output values from the system to be modeled is acquired beforehand. The model is then being fully trained before it is applied to the prediction task in question, after which no further training takes place. Typically when training offline, an iteration of the training algorithm corresponds to presenting all examples to the model before updating the parameter estimate.

An alternative to this approach is to adopt an "online" method where the parameters are updated immediately after the presentation of each example to the model. This opens up for the possibility of integrating training into the application, continuously improving the model. Such methods are also known as recursive methods [Lju87] or "pattern mode" methods [Hay94]. When updating the parameters immediately after the presentation of each example we can choose to base the parameter update on the most recent example only, i.e., on the *instantaneous* quadratic error measure,

$$J(t) = \frac{1}{2}[e(t)]^2 \; . \tag{4.30}$$

When training using this error measure it is common to use gradient methods [HKP91, Hay94] as the Hessian is extremely ill-conditioned (has rank one) when based on a single example only. A classical signal processing example of this approach is the Least Mean Square (LMS) algorithm [WS85], derived for linear models.

As an alternative to the instantaneous quadratic error measure we can choose to compute the parameter update from an *accumulated* quadratic error measure, which at time t is computed as

$$E(t;\mathbf{w}) = \frac{1}{2t}\sum_{t'=1}^{t}[e(t')]^2 \ . \tag{4.31}$$

Using an accumulated error measure opens up for the application of "online versions" of the Gauss-Newton method, which are known as Recursive Least Squares (RLS) methods [Lju87]. These methods are generally more effective than LMS-type methods, but as for "offline" optimization methods at the cost of increased computation in each iteration. In order to avoid a matrix inversion at each time step (or more appropriately solving a system of linear equations), RLS-type methods rely on the matrix inversion lemma treated in appendix C to compute the inverse of the Gauss-Newton approximation to the Hessian in an iterative manner.

When training a model using "batch" methods, we assume that the system to be modeled is stationary. By stationary we understand that the parameters defining the teacher function do not vary with time and that the additive noise contributions are generated by a stationary stochastic process; refer also to section 2.4.

If the assumption of a stationary environment cannot be justified we need to resort to online methods which allow the model to continuously adapt itself to the changing conditions of operation. If using an accumulated error measure we need to introduce a weighting scheme of the individual errors in Eq. (4.31) assigning less weight to older errors relating to conditions that are no longer representative for the underlying system. A commonly applied approach is to introduce a *forgetting factor* λ, modifying the accumulated error measure as [Lju87]

$$E(t;\mathbf{w}) = \frac{1}{2t}\sum_{t'=1}^{t}\lambda^{t-t'}[e(t')]^2 \quad , \quad 0 \le \lambda < 1 \ . \tag{4.32}$$

This weighting scheme leads to an exponential decay of the relative weighting of old errors, the rate of the decay being determined by the value of λ. In order to make a proper selection of λ we need to estimate the rate at which the underlying system is likely to change.

Online methods are treated in more detail in e.g., [Lju87, Par92]. They have not been considered in the present work, training of the models has been done offline on a fixed training set thus assuming a *stationary environment* [Hay94].

4.7 Computing the gradient for RNNs

In order to train adaptive models we need to compute the gradient of the quadratic cost function Eq. (4.2). From Eq. (4.12) we see that the gradient is composed of a sum of terms involving the prediction errors as well as the partial derivatives of the model output at each time step. If modeling using a feed-forward neural network the model outputs $y(t)$ are independent as they are based on the current inputs only; in this case the partial derivatives $\partial y(t)/\partial w_i$ will be independent as well and are easily obtained by the backpropagation method; see e.g., [HKP91] for an excellent description of this method. If modeling using recurrent networks, the individual model outputs are however *not* independent, as the output at time t generally depends on both the external input at time t as well as unit outputs entering the model output at the *previous* time step $t-1$; refer to section 3.3.1. The partial derivatives for an RNN model output at time t will therefore involve derivatives of the *previous* unit outputs as well, making computation of the quadratic cost function gradient somewhat more involved than for feed-forward networks.

This section describes the two basic approaches towards computing the gradient of the cost function for fully recurrent networks. The expressions for the derivations apply specifically to the RNNs considered in this work, described in section 3.3.1, but the principles are easily applied to other recurrent structures as well. The two methods to be described in the following are *Back-Propagation Through Time* and *Real-Time Recurrent Learning* which differ significantly in the requirements of storage and computation time.

4.7.1 Back-Propagation Through Time

The first algorithm that was applied to the calculation of the cost function gradient for RNNs was called Back-Propagation Through Time (BPTT) [HKP91, Hay94]. The method works by *unfolding* the recurrent network in time. Unfolding in time is a procedure which turns an arbitrary recurrent network into an equivalent, deeply layered feed-forward network employing a massive weight sharing. For a sequence of length T we duplicate all the units in the network for each time step $t = 1, \ldots, T$. The hidden units of time step t are connected to the units of timestep $t+1$ through the feedback weights of the RNN as illustrated in Figure 4.1. The external input $x(t)$ is connected to the hidden units of time t through the input weights of the RNN, and the hidden unit outputs $s_i(t)$ are connected to the output unit through the output weights for computation of the output $y(t)$ at time t. As the weights do not change with time, the weights are identical between layers. The example in Figure 4.1 is for a recurrent network of the type described in section 3.3.1 with one external input, two hidden units and no feedback from the output unit back to the hidden units for convenience of presentation.

The resulting structure is equivalent to a feed-forward network and is updated likewise, layer by layer. This corresponds to the iteration of the recurrent network time step by time step. The quadratic cost function Eq. (4.2) may for this structure be interpreted as an "instantaneous" error measure for a multilayer feed-forward network having T output units, one in each layer as illustrated on Figure 4.1. The principle of BPTT is to calculate the gradient for this "instantaneous" error measure using simple back-propagation, an ideal tool for computing derivatives for feed-forward structures [HKP91]. Having output units in each layer of the unfolded network poses a major difference from traditional feed-forward structures which typically only have output units in the final layer. It is of no problem to the application of back-propagation however, the error signals computed for these units are simply propagated backwards from the layer in which they originate along with the errors for the hidden units in the same layer.

The back-propagation algorithm works by computing an *error* δ for each unit in the network. The derivative of the squared error wrt. a given weight $w_{\text{to,from}}$ is then computed as $\delta_{\text{to}} \times z_{\text{from}}$, where z_{from} is the value (hidden unit output or external input) at the origin of the weight and δ_{to} is the error for the destination unit of the weight; see e.g., [HKP91] for further details. This way, the back-propagation algorithm calculates the derivative of the error measure wrt. every weight in the feed-forward network. As the weights in the

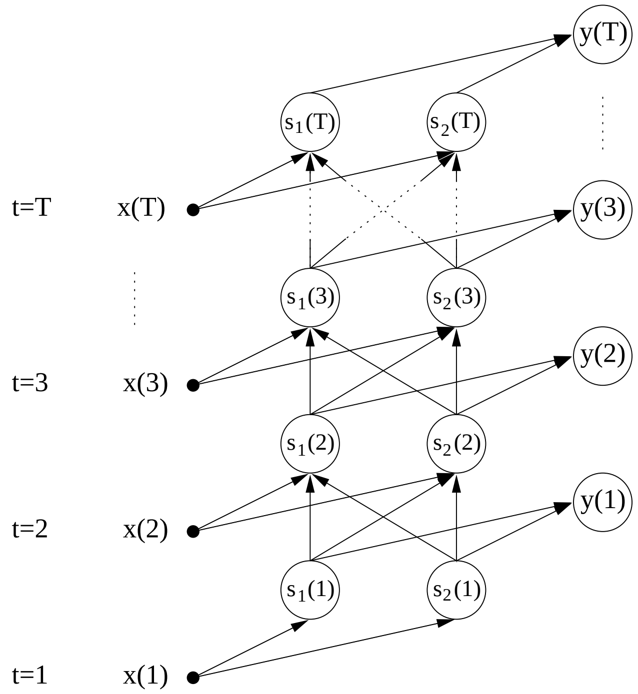


Figure 4.1: Unfolding in time of recurrent network with one external input and two hidden units. Note that in this example there is no feedback from the output unit back to the hidden units. The weights connecting the units are identical in each time step, i.e., between layers.

unfolded network are identical in each layer, the total derivative of the error measure wrt. a particular weight is obtained by summing the derivatives in all layers obtained for this particular weight. See e.g., [WP90] for a more detailed description of the BPTT algorithm.

A big advantage when computing the derivatives using the BPTT algorithm is that the number of operations required per time step scales as $\mathcal{O}(N_u^2)$ just as for ordinary back-propagation; N_u is here the number of units in the network. This is much faster than the alternative, Real-Time Recurrent Learning, which is described below.

A major criticism of the BPTT algorithm has been the requirements for storage [HKP91]. If the total number of units in the RNN is N_u and we are considering a sequence of training examples of length T, we need to store $N_u T$ activations. Thus, storage requirements scale with the length of the training sequence which might be impractical for long sequences. However, for most applications this is not a practical problem as computer memory is now available in large quantities at low cost.

Another and more severe critique of BPTT has been that in its standard formulation it is not practically possible to use it for online training of RNNs as both storage and computations required in each iteration will grow linearly with time. In order to overcome this problem, several modifications have been suggested; see e.g., [WP90]. The principle underlying the modified algorithms is to store the activations of only the h most recent time steps, thus all information older than h time steps is "forgotten" about. BPTT is then applied to these h most recent time steps only, leading to an *approximation* of the true gradient computed from the initial time step.

4.7.2 Real-Time Recurrent Learning

A method that overcomes the problems with *true* gradient following online training is the Real-Time Recurrent Learning algorithm (RTRL) [WZ89]. The key feature of this algorithm is an exact recursive update of the derivatives, eliminating the need to store all previous states of the units in the recurrent network. However, this advantage over BPTT comes at the price of a significantly increased computational burden as we shall see.

In the following the RTRL algorithm is derived for the type of recurrent networks used in this work. Whether using the instantaneous error measure $J(t)$ Eq. (4.30) or training using the accumulated error measures of Eq. (4.2) or Eq. (4.31) we need the partial derivatives of the instantaneous quadratic error measure at each time step t,

$$\frac{\partial J(t)}{\partial w_{ij}} = -e(t)\frac{\partial y(t)}{\partial w_{ij}} \tag{4.33}$$

Recall from section 3.3.1 that we combine the inputs to the hidden units and the output unit at time t in vectors $\mathbf{z}^h(t)$ and $\mathbf{z}^o(t)$, respectively, defined as

$$z_k^h(t) = \begin{cases} x_k(t) & , \quad k \in I \\ s_k(t-1) & , \quad k \in H \\ y(t-1) & , \quad k = o \end{cases} \quad , \quad z_k^o(t) = \{\; s_k(t) \quad , \quad k \in H \tag{4.34}$$

where I denotes the set of indices for the inputs, H denotes the set of indices for the hidden units and o denotes the output unit. For convenience of presentation in the following we will assume that the bias weights for the units in the network are provided indirectly by e.g., an additional hidden unit with an output value permanently set to +1 (and thus *not* receiving any inputs).

The derivatives of the output unit at time t Eq. (3.8) are then computed as

$$\frac{\partial y(t)}{\partial w_{ij}} = \delta_{oi} z_j^o(t) + \sum_{j' \in H} w_{oj'} \frac{\partial s_{j'}(t)}{\partial w_{ij}} \tag{4.35}$$

where δ_{oi} is the Kronecker delta. This expression contains the derivatives of the hidden unit outputs,

$$\frac{\partial s_k(t)}{\partial w_{ij}} = \frac{\partial f[v_k(t)]}{\partial v_k(t)} \cdot \frac{\partial v_k(t)}{\partial w_{ij}} \quad , \quad k \in H \tag{4.36}$$

where

$$\frac{\partial v_k(t)}{\partial w_{ij}} = \delta_{ki} z_j^h(t) + \sum_{j' \in H} w_{kj'} \frac{\partial s_{j'}(t-1)}{\partial w_{ij}} + w_{ko} \frac{\partial y(t-1)}{\partial w_{ij}} \tag{4.37}$$

Assuming that iterations start at time $t = 1$ we set the initial conditions to zero according to standard practice [WZ89]. I.e.,

$$\begin{aligned} s_k(0) &= 0 \;, \quad k \in H \\ \frac{\partial s_k(0)}{\partial w_{ij}} &= 0 \;, \quad k \in H \\ \frac{\partial y(0)}{\partial w_{ij}} &= 0 \end{aligned} \tag{4.38}$$

The recursive equations above form the essence of the RTRL algorithm. We note how the *exact* value of the partial derivatives of the network output at time t are computed in an iterative manner by accumulation of derivatives from the previous time step only. The partial derivatives obtained from the RTRL algorithm can be accumulated over a training sequence or used immediately, depending on the type of cost function and method used for training.

Note that in order compute the derivative of a unit output at time t wrt. a particular weight we need to compute and store derivatives of *all* units in the network wrt. this particular weight, which leads to a significantly increased computational burden compared to BPTT. Let the total number of units in the recurrent network be N_u and the number of inputs be N_I. The total number of weights in a fully connected network is then $N_u(N_u + N_I)$. From Eqs. (4.35–4.37) we see that we need to maintain the derivatives of every unit in the network wrt. each of these weights, $N_u^2(N_u + N_I)$ derivatives in total. Each of these derivatives are seen to involve around N_u operations leading to a total of $N_u^3(N_u + N_I)$ operations; an operation is here defined as a multiplication combined with an addition. The computational complexity of the RTRL algorithm in each iteration thus scales as $\mathcal{O}(N_u^4)$ whereas the storage requirements scale as $\mathcal{O}(N_u^3)$.

The iterative computation of exact derivatives and the significantly decreased demand for storage compared to BPTT is seen to come at the price of a significantly increased computational complexity. Even so, the RTRL approach towards computation of derivatives has been the method of choice in this work due to problems with computation of second derivatives using the BPTT approach, as will be explained below.

4.8 Computing the Hessian for RNNs

When training using second-order methods we need the Hessian matrix with second derivatives of the cost function. This section describes how to compute the Hessian for the quadratic cost function when modeling using recurrent networks. The description is divided into two parts, computation of the Gauss-Newton approximation and computation of the second-order part.

4.8.1 The Gauss-Newton approximation

When calculating the Gauss-Newton approximation to the Hessian, at each time step t we need to compute the outer product of a vector containing the partial derivatives of the network output at time t with itself, as can be seen from Eq. (4.22). In order to obtain these partial derivatives $\partial y(t)/\partial \mathbf{w}$ we need to employ the RTRL algorithm as it is unfortunately not possible to obtain these derivatives from the faster BPTT algorithm, as will be shown below.

When employing the BPTT algorithm we are in fact utilizing an alternative way of writing the partial derivatives for the quadratic cost function $E(\mathbf{w})$ given by Eq. (4.2). The derivatives in Eq. (4.12) can alternatively be written as [Hay94]

$$\frac{\partial E(\mathbf{w})}{\partial w_{ij}} = \sum_{t=1}^{T} \frac{\partial E(\mathbf{w})}{\partial s_i(t)} \frac{\partial s_i(t)}{\partial w_{ij}} \tag{4.39}$$

where w_{ij} is a weight leading to unit i in the network and where $s_i(t)$ is the output of unit i at time step t. t may also be conceived as indicating the layer number in the unfolded network; refer to Figure 4.1. As the weights are assumed to be constant when computing the cost function we will naturally obtain a contribution to the derivatives $\partial E(\mathbf{w})/\partial w_{ij}$ at each step t; this corresponds to the weights between each layer being identical when unfolding the recurrent network in time as described in section 4.7.1.

When employing the BPTT algorithm we are using the traditional back-propagation algorithm to obtain the elements of Eq. (4.39), starting from time/layer $t = T$ and working *backwards* to $t = 1$. Note however, that

$$\frac{\partial E(\mathbf{w})}{\partial s_i(t)} \frac{\partial s_i(t)}{\partial w_{ij}} \neq \frac{\partial J(t)}{\partial w_{ij}} \quad , \quad J(t) = \frac{1}{2}[e(t)]^2 \tag{4.40}$$

I.e., the element obtained at step t of the BPTT algorithm is not equal to the derivative of the squared error at time t. This can immediately be apprehended by noting that the lefthand side of Eq. (4.40) involves only terms computed from time T back to time t, while the righthand side involves only terms computed from time $t = 1$ up to time t, as seen from Eqs. (4.35–4.37). It is only when we take the sum over all t that we obtain equality between the gradients computed from Eq. (4.39) and Eq. (4.12) [Hay94].

Partial derivatives of a feed-forward network output are obtained by "back-propagating ones", i.e., by applying the back-propagation algorithm and back-propagate errors set equal to one, $e(t) = 1$. However, applying this strategy to the BPTT algorithm will *not* provide the terms $\partial y(t)/\partial \mathbf{w}$ needed for computation of the Gauss-Newton approximation to the Hessian, for the same reason as listed in Eq. 4.40. Only if we take the sum of $\partial y(t)/\partial \mathbf{w}$ over all t will equality be obtained to the sum of all the elements resulting from the RTRL algorithm. Therefore, BPTT cannot be applied to the computation of the Gauss-Newton approximation to the Hessian.

When computing the Gauss-Newton approximation to the Hessian we are thus forced to use the RTRL algorithm in order to obtain the partial derivatives $\partial y(t)/\partial \mathbf{w}$. Besides the $N_u^3(N_u + N_I)$ operations thus required in each iteration to compute the partial derivatives, it takes an additional $N_u(N_u + N_I)(N_u(N_u + N_I) + 1)/2$ operations to form the outer product if utilizing the symmetry of the Hessian.

This seemingly large cost in order to obtain the Hessian matrix combined with the need to solve a system of linear equations in each iteration has lead to an often stated claim that second-order methods are much more computationally demanding than first-order methods. Let us delve into this claim, investigating it in more detail for the RTRL algorithm.

For simplicity, we assume that the number of external inputs N_I is somewhat smaller than the number N_u of units in the network. Using this simplification there is approximately N_u^2 weights in the network, and the number of operations required at each time step for computation of the partial derivatives is thus N_u^4. Forming the outer product of the partial derivatives costs $N_u^2(N_u^2 + 1)/2$ operations, or around half the cost of the partial derivatives. For a training sequence with T examples, the total cost of computing the Gauss-Newton approximation to the Hessian is thus approximately $1.5N_u^4T$.

In order to obtain the search direction we need to solve a system of N_u^2 linear equations. The computational cost of this is approximately N_u^6 according to [PFTV92][2]. Usually we have much more examples available for training than there are weights in the network, as the training problem will otherwise be ill-posed due to a rank-deficient Hessian matrix. Let us however be pessimistic and say that the number of training examples T equals

[2]Experiments in MATLAB where the Gauss-Jordan elimination method is used indicated that the number of operations required for solving a system of N linear equations scales approximately as $0.7N^3$.

the number of weights N_u^2. The solution of the system of linear equations thus requires $N_u^6 \approx N_u^4 T$ operations.

From the above we see that the total cost of obtaining the search direction for the Gauss-Newton method is $2.5 N_u^4 T$. Comparing to the number of operations required for the computation of the gradient using the RTRL algorithm, $N_u^4 T$, we learn that the application of the Gauss-Newton method to RNNs is just a mere factor of 2.5 more computationally costly than gradient descent!

4.8.2 Computing the second-order term

As the Newton method described in section 4.4.1 is not used in practice due to lack of global convergence abilities as well as the vast computational burden generally associated with the computation of second derivatives of the model output, there is no real practical need for the second-order part of the Hessian Eq. (4.17). Second derivatives of the output for the recurrent networks considered in this work were however derived in [PH95] in order to investigate the "importance" of the second-order term.

The second derivatives are obtained by further differentiation of the expressions Eq. (4.35) and Eq. (4.37) entering the RTRL algorithm. The term in Eq. (4.17) denoting the second derivative of the RNN output is calculated as

$$\frac{\partial^2 y(t)}{\partial w_{ij} \partial w_{pq}} = \delta_{oi} \frac{\partial z_j^o(t)}{\partial w_{pq}} + \sum_{j' \in H} w_{oj'} \frac{\partial^2 s_{j'}(t)}{\partial w_{ij} \partial w_{pq}} + \delta_{op} \frac{\partial z_q^o(t)}{\partial w_{ij}} \tag{4.41}$$

which is obtained as the derivative of Eq. (4.35). This expression contains the second derivative of the hidden unit outputs which is calculated as

$$\frac{\partial^2 s_k(t)}{\partial w_{ij} \partial w_{pq}} = \frac{\partial^2 f[v_k(t)]}{\partial v_k(t)^2} \cdot \frac{\partial v_k(t)}{\partial w_{ij}} \cdot \frac{\partial v_k(t)}{\partial w_{pq}} + \frac{\partial f[v_k(t)]}{\partial v_k(t)} \cdot \frac{\partial^2 v_k(t)}{\partial w_{ij} \partial w_{pq}} \tag{4.42}$$

where the second derivative of the input to unit k is calculated as

$$\frac{\partial^2 v_k(t)}{\partial w_{ij} \partial w_{pq}} = \delta_{ki} \frac{\partial z_j^h(t)}{\partial w_{pq}} + \sum_{j' \in H} w_{kj'} \frac{\partial^2 s_{j'}(t-1)}{\partial w_{ij} \partial w_{pq}} + w_{ko} \frac{\partial^2 y(t-1)}{\partial w_{ij} \partial w_{pq}} + \delta_{kp} \frac{\partial z_q^h(t)}{\partial w_{ij}} \tag{4.43}$$

which is obtained by differentiation of Eq. (4.37). In line with the first derivatives, the second derivatives are set equal to zero at time $t = 0$.

The computational complexity of the calculation of the second derivatives is as follows. The number of weight combinations for the derivatives is equal to the number of weights squared, $N_u^2 (N_u + N_I)^2$. From Eqs. (4.41–4.43) we see that we need to maintain and store the second derivatives of *all* units in the network wrt. every weight combination, leading to a total of $N_u^3 (N_u + N_I)^2$ second derivatives. Each of these second derivatives is seen to involve around N_u operations leading to a total of $N_u^4 (N_u + N_I)^2$ operations per iteration. If utilizing the symmetry of the second derivatives, however, the storage requirements and number of operations is approximately halved. The computational complexity of each iteration when computing second derivatives from the method outlined above thus scales as $\mathcal{O}(N_u^6)$ whereas the storage requirements scale as $\mathcal{O}(N_u^5)$. Inclusion of the second order term in the Hessian therefore leads to a significantly increased computational burden compared to the Gauss-Newton approximation in its own.

Chapter 5

Generalization

In the previous chapter training of adaptive models was defined in terms of minimizing the prediction errors made on examples in the training set. Provided that the chosen model has been properly trained it therefore seems reasonable to believe that the errors made on the training examples are "small." From an application point of view the training error in itself is, however, a fairly uninteresting quantity for assessing the quality of a model; in problems where low error on the examples used for training is the primary object of modeling, one could simply use the training data to create a lookup table; this will naturally lead to zero error on the training examples.

A more relevant measure of quality for a trained model is its ability to *generalize*,[1] i.e., to provide good predictions on examples that were not present in the training set. A commonly adopted measure of the generalization ability of a model applied to time series prediction is the expected squared prediction error on a novel example.

Section 5.1 describes the theoretical definition of the generalization error for feed-forward models as well as an approximation computed on a test set. The theoretical definition of the generalization error for feed-forward networks is not directly applicable to dynamic system models like recurrent networks due to the dependency of the model output of all previous inputs. In section 5.2 an attempt is made to provide a theoretical definition of the generalization error for recurrent networks. Further, it is described how the generalization error has been estimated in this work. The chapter is concluded by a description of an *analytical* estimate of the generalization error, namely Akaike's Final Prediction Error estimate.

5.1 Generalization in feed-forward models

The generalization error of a model intended for prediction is for feed-forward models commonly defined as the expected squared prediction error on a novel example [HKP91, Hay94, Bis95], consisting of an input $\mathbf{x}(t)$ and a desired output $d(t)$; for auto-regressive modeling we might have $\mathbf{x}(t) = [x(t), x(t-1), \ldots, x(t-L+1)]$ and the desired output is the next value in the series, $d(t) = x(t+1)$. Assuming stationarity, the generalization error $G(\widehat{\mathbf{w}})$ of a model $g(\,\cdot\,;\widehat{\mathbf{w}})$ is obtained as

$$\begin{aligned} G(\widehat{\mathbf{w}}) &= E\{\,[\,d - g(\mathbf{x};\widehat{\mathbf{w}})\,]^2\,\} && (5.1)\\ &= \int [\,d - g(\mathbf{x};\widehat{\mathbf{w}})\,]^2 \cdot P(\mathbf{x}, d)\;\mathsf{d}\mathbf{x}\,\mathsf{d}d && (5.2) \end{aligned}$$

[1] A term apparently borrowed from psychology [Hay94].

where $\widehat{\mathbf{w}}$ denotes the parameter estimate resulting from training and $P(\mathbf{x}, d)$ denotes the joint probability density function for inputs $\mathbf{x}$ and corresponding outputs d as described in section 2.4.

Usually the joint probability density function is unknown just as only a finite amount of data is available. The generalization error is in practice therefore *estimated* on a separate set of examples denoted the test set. Assuming that test set comprises V examples and that the index of the first example is $T+1$ the generalization error is estimated as

$$\widehat{G}(\widehat{\mathbf{w}}) = \frac{1}{V} \sum_{t=T+1}^{T+V} [\, d(t) - g(\mathbf{x}(t); \widehat{\mathbf{w}})\,]^2 = \frac{1}{V} \sum_{t=T+1}^{T+V} e^2(t; \widehat{\mathbf{w}}) \tag{5.3}$$

If the generalization error is estimated on a single time series it must be assumed that the generating process is ergodic.

5.2 Generalization error for recurrent models

The theoretical definition of the generalization error for feed-forward networks is not directly applicable to dynamic system models like recurrent networks due to the dependency of the model output of all previous inputs. In the literature the generalization error of recurrent networks is traditionally simply estimated on a test in the same way as it is done for feed-forward networks as described above. However, as a recurrent network implements a function of in principle *all* previously applied inputs, it is not immediately obvious to what extent this approach may be theoretically justified. In this section an attempt is made to formulate the generalization error of recurrent models within a statistical framework in line with the approach which is traditionally adopted for feed-forward models.

5.2.1 Theoretical definition

As the output from a dynamic system model like a recurrent network is based on *all* previous outputs, time necessarily enters the definition of generalization error for this model type. In the following it will be assumed that the problem at hand is one step ahead time series prediction and the desired output is therefore set to $d(t) = x(t)$ for inputs $x(t-1), x(t-2), \ldots$ to the network; the expressions in the following are however easily modified to other applications.

We now formulate the expected squared prediction error which will be encountered at time step t as

$$G_t(\widehat{\mathbf{w}}) = E\left\{[\, x(t) - g_t(\mathbf{X}^{t-1}; \widehat{\mathbf{w}})\,]^2\right\} \tag{5.4}$$

$$= \int [\, x(t) - g_t(\mathbf{X}^{t-1}; \widehat{\mathbf{w}})\,]^2 \cdot P_t\left(\mathbf{X}^t\right) \, \mathsf{d}\mathbf{X}^t \tag{5.5}$$

where $x(t)$ denotes the true system output at time t and $g_t(\,\cdot\,; \widehat{\mathbf{w}})$ denotes the (time unfolded) recurrent network model output at time t; refer to the general model description given in section 2.4. This error may be interpreted as the average squared error resulting at time t if the *same* model $g_t(\,\cdot\,; \widehat{\mathbf{w}})$ was applied to infinitely many realizations of the observation sequence $\mathbf{X}^{t-1}$ from the true system. The error $G_t(\widehat{\mathbf{w}})$ defined by Eq. (5.5) may consequently also be conceived as the *ensemble* error at time t.

Due to lack of knowledge of the *complete* observation sequence $\mathbf{X}^t$ from the true system as well as the true system initial state the network model will initially enter a *transient* mode of operation as described in section 2.4, before asymptotically converging towards the approximated mode of operation of the true system; the asymptotic behaviour of the recurrent network model as $t \to \infty$ may also be referred to as the *steady-state* [PC89]. During the transient the generalization error $G_t(\widehat{\mathbf{w}})$ will typically not be representative for the generalization error obtained after entering the steady-state and thus the "normal" mode of operation; rather, the errors will on average tend to be too large. As the transient may last for an arbitrary period of time, the overall generalization error $G(\widehat{\mathbf{w}})$ of a dynamic system model is consequently *defined* as the average squared error to expect after entering the steady-state mode of operation. Ultimately, this will be the case after iteration of the model for an infinitely long period of time and the generalization error is therefore defined as

$$G(\widehat{\mathbf{w}}) \triangleq \lim_{t\to\infty} G_t(\widehat{\mathbf{w}}) = \lim_{t\to\infty} \int [\, x(t) - g_t(\mathbf{X}^{t-1}; \widehat{\mathbf{w}})\,]^2 \cdot P_t\left(\mathbf{X}^t\right) \, \mathsf{d}\mathbf{X}^t \qquad (5.6)$$

In order to define the generalization error this way it is necessary to assume that the limit exists, i.e., that the expected squared prediction error will not vary with time after the model has entered its steady-state mode of operation.

Consider a sequence of squared prediction errors $\{e(t; \widehat{\mathbf{w}})^2\}$ resulting from predictions of a *single*, infinitely long sequence of observations generated by the true system. Assuming this sequence of squared errors to be a *mean-ergodic* sequence [Pap84] the generalization error defined by Eq. (5.6) may equivalently be obtained as

$$G(\widehat{\mathbf{w}}) = \lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} [\, x(t) - g_t(\mathbf{X}^{t-1}; \widehat{\mathbf{w}})\,]^2 = \lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} e^2(t; \widehat{\mathbf{w}}) \ . \qquad (5.7)$$

When computing the generalization error this way the initial prediction errors might be subject to the effects of the transient, and therefore not representative for the errors in the steady-state mode of operation. The effects of the increased errors during the transient will, however, tend to zero when applying the limit.

5.2.2 Empirical estimate

In practice we only have a finite amount of observations from the true system available, usually extracted as part of a *single* realization of a true system observation sequence; i.e., only one series is available as it may not be possible to repeat the "experiment" which generated the obtained series. An *estimate* of the generalization error is therefore in practice obtained by holding back a finite length segment of the observed time series as an independent test set, and an estimate of the generalization error defined by Eq. (5.7) of the recurrent network is obtained as the average squared predition error on the observations in this test segment,

$$\widehat{G}(\widehat{\mathbf{w}}) = \frac{1}{V} \sum_{t=t_0+1}^{t_0+V} e^2(t; \widehat{\mathbf{w}}) \approx G(\widehat{\mathbf{w}}) \qquad (5.8)$$

where $t_0 + 1$ denotes the time index for the first "target value" in the test segment which comprises V elements[2].

[2] The generalization error estimates reported in this thesis are normalized by the empirical variance of the samples in the test set in order to comply with the Normalized Mean Squared Error defined in appendix A.

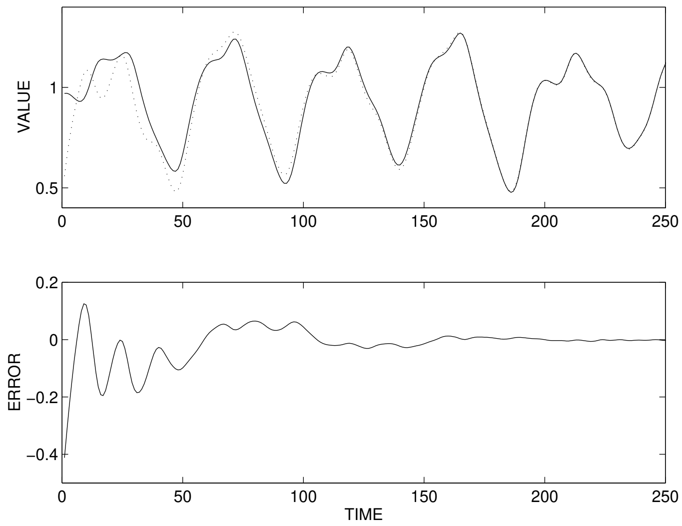


Figure 5.1: Illustration of transient when starting iterations on a test segment. Upper panel: Correct output (solid line) and network output (dotted line). Lower panel: Prediciton errors. Note how the prediction errors decay as the network approaches the steady-state mode of operation.

When estimating the generalization error on a finite length test segment the effect of the transient will generally *not* tend to zero and become negligible as is the case in the limit in Eq. (5.7). Rather, a bias is introduced into the estimate of the true generalization error. Naturally, it is of interest to reduce this bias as much as possible; this will be elaborated upon in the following.

The recurrent network models considered in this work are working solely from an *internal* memory of previous input values, built up in the hidden unit state vector during iterations of the network. When commencing iterations on e.g., a test segment the internal state of the network is usually set to zero as described in sections 2.4 and 3.3.1, and a memory of past observations *graduately* builds up during the first iterations until the internal memory contains a full representation of those previous values necessary for accurate predictions. The initial iterations of the recurrent network during which the internal memory is not fully developed thus mark a transient before convergence to the steady-state mode of operation as described in section 5.2.1.

During the transient the prediction errors will be larger than in the steady-state as illustrated in Figure 5.1. Here, a fully trained single-input recurrent network is applied to a test segment[3] by setting the internal hidden state to zero before starting the iterations. The recurrent network has a very low test error once the model has converged, but the initial errors are large and not representable for what should be expected once the network has converged to the steady-state. As iterations progress the prediction errors decay as the network approaches the steady-state mode of operation.

From Figure 5.1 it is seen how inclusion of the initial prediction errors when estimating the generalization error using Eq. (5.8) will introduce a bias, leading to an overly pessimistic estimate of the true generalization error. In order to avoid this bias the initial errors should be omitted from the estimate of the generalization error. Omission of these

[3]The time index of the first element in the test segment is here *arbitrarily* set to zero.

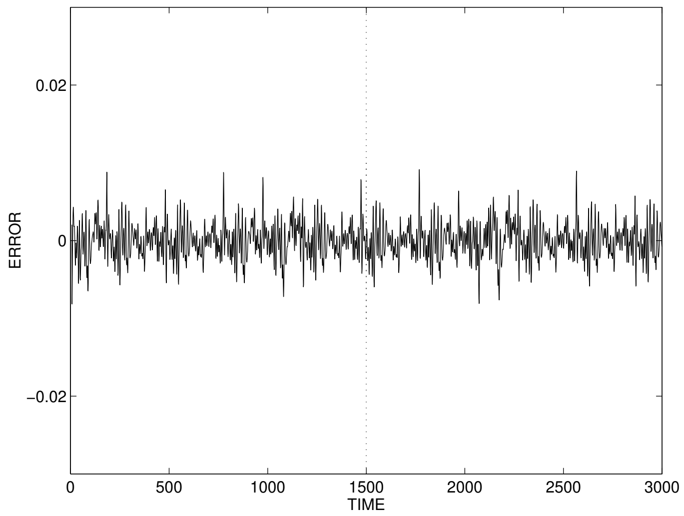


Figure 5.2: Illustration of the development of the prediction error when iterating from the training set into an immediately following test set (the same test set as used in Figure 5.1 but extended) for the *same* recurrent network as used in Figure 5.1. The vertical line denotes the train/test separation point. Note the small scale here on the ordinate axis compared to the lower panel of Figure 5.1.

non-representative errors poses at least two problems, however. First of all it is difficult to decide exactly how many of the initial prediction errors should be discarded, as the length of the transient will depend upon both the nature of the problem at hand as well as the particular network realization which resulted from training. Secondly, values of the possibly scarce test series are "wasted" during the transient as they are not contributing to the generalization error estimate, but rather used to initialize the internal memory of the network; the consequence is a reduction of the *effective* size of the test set leading to larger *variance* on the generalization error estimate.

In order to reduce the effect of transient bias on the generalization error estimate as much as possible while still utilizing all available data in the test series for the estimate, the approach adopted in the present work has been to always choose the test series to *immediately* follow the training series. When estimating the generalization error, iterations are initiated on the preceeding *training* series in order for the internal memory of the recurrent network to be close to the "normal mode of operation", i.e., the steady-state once iterations enter the test segment. This is illustrated in Figure 5.2 for the *same* network which was used in Figure 5.1 and using the *same* (extended) test set. Now, iterations are initiated on the training set at time $t = 1$ and continued into the test set which starts at time step 1500 as denoted by the vertical line. It is seen that the initial errors on the test set no longer suffer from transient effects; note the small scale on the ordinate axis compared to Figure 5.1.

This method will tend to introduce a bias into the generalization error estimate as well. The prediction errors obtained in the beginning of the test set might be overly optimistic compared to the steady-state mode of operation when $t \rightarrow \infty$ as the internal memory of the network is initially dominated by observations from the *training* set. The smaller initial prediction errors on the test set consequently introduce a *negative* bias into the generalization error estimate. The approach is, however, preferable considering

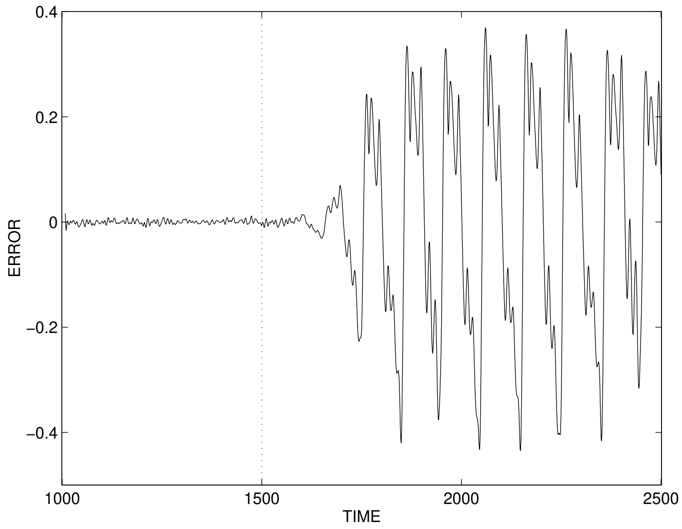


Figure 5.3: Illustration of the development of the prediction error when iterating from the training set into an immediately following test set for a recurrent network having a *very large* estimated generalization error. The vertical line denotes the train/test separation point. Note the small initial errors on the test set due to influence from the training set.

the smaller *magnitude* of the introduced bias (compare Figure 5.1 to Figure 5.2) and, considering that as the quality of training improves, the average test errors will approach the average training error (as it appears in Figure 5.2) and the relative effect of the bias will thus tend to zero.

The effect of smaller intial test errors due to influence from the training series is not easily seen from Figure 5.2 as the errors on the training and test set are practically identical. In order to more clearly demonstrate the effect, a similar illustration is provided in Figure 5.3. However, the recurrent network used exhibits severe *overtraining*, i.e., the average squared error of the fully trained network is much smaller on the training set than on the test as the network has adapted itself to features specific to the training set that are not present in the test set. From the figure it is seen that the initial prediction errors made on the test set are hardly distinguishable from the errors made on the training set, but as the influence from the points in the training set on the internal memory decays, the prediction errors increase dramatically in magnitude.

Estimation of the generalization error on a test series immediately following the training series is a commonly adopted procedure for feed-forward models as well, see e.g., [WHR90, WG93]. For these models the lag space extends back into the training series when predicting the first few values in the test series; this will introduce a negative bias into the estimate in the same way as was described for recurrent networks above. The ideal situation for both network types would be to have a very large test series far away in time from the training series, allowing for an unbiased generalization error estimate. However, practical limitations usually makes this ideal situation unattainable.

5.3 Analytical generalization error estimates

As an alternative to estimating the generalization error on a test set it has in the literature been suggested to derive *analytical* expressions for the generalization error by use of asymptotic theory. The concept underlying the analytic approach is to estimate the generalization error in terms of the training error and the model complexity. Analytical generalization error estimates include the AIC, BIC, FPE and NIC estimates. An often adopted estimate for neural networks is the Final Prediction Error (FPE) estimate [Aka69] which will be briefly outlined in the following.

5.3.1 The FPE-estimate

This section briefly outlines the derivation of the FPE-estimate. For a detailed derivation the reader is referred to e.g., [Lar93, Ras93].

The derivation of the FPE-estimate assumes that examples in terms of inputs $\mathbf{x}(k)$ and corresponding outputs $d(k)$ from the true system are generated by a teacher function, degraded by additive noise. It is assumed that the teacher function can be perfectly described by a network model $g(\,\cdot\,;\mathbf{w})$ for a set of *teacher* weights $\mathbf{w}^*$ and thus that examples are generated according to

$$d(k) = g\left(\mathbf{x}(k); \mathbf{w}^*\right) + \varepsilon(k) \tag{5.9}$$

where the noise samples $\varepsilon(k)$ are independent identically distributed stochastic variables with finite, but unknown variance σ^2. Further, it is assumed that the noise terms are independent of the corresponding inputs.

It is now assumed that an ensemble of *student* networks is available in which each network is provided with its own *individual* training set comprising T examples. Each network in the ensemble is trained, leading to an estimate $\widehat{\mathbf{w}}$ of the teacher weights $\mathbf{w}^*$ for each network. It is assumed that the parameter fluctuations $\delta\mathbf{w} = \widehat{\mathbf{w}} - \mathbf{w}^*$ around the teacher weights are *small*, which may be justified if the number of training examples T is *large*. This ensures the validity of an approximation of the *student* Mean Squared Error $\mathrm{MSE}(\widehat{\mathbf{w}})$[4] by a Taylor series approximation to second order in weigth fluctuations $\delta\mathbf{w}$ around $\mathrm{MSE}(\mathbf{w}^*)$ for the *teacher* function, calculated on the same training set as the student. Averaging over the entire ensemble leads to

$$\langle \mathrm{MSE}(\widehat{\mathbf{w}})\rangle_T = \left(1 - \frac{N}{T}\right) \cdot \sigma^2 + O\left((1/T)^2\right) \tag{5.10}$$

where $< \cdot >_T$ denotes the ensemble average over all possible training sets of size T, N denotes the number of weights in the networks and it is assumed that terms of Landau order $(1/T)^2$ can be neglected. Note that the ensemble average of the training errors is smaller than the noise level, corresponding to the students having learned some of the noise as well.

Similary, the generalization error $G(\widehat{\mathbf{w}})$ of each student network is approximated by an expansion of the *teacher* function generalization error $G(\mathbf{w}^*)$ to second order in parameter fluctuations $\delta\mathbf{w}$. The ensemble average is then obtained as

$$\langle G(\widehat{\mathbf{w}})\rangle_T = \left(1 + \frac{N}{T}\right) \cdot \sigma^2 + O\left((1/T)^2\right) \tag{5.11}$$

Elimination of the noise level σ^2 in Eq. (5.10) and Eq. (5.11) leads to a relation between the ensemble average training and generalization errors obtained as

$$\langle G(\widehat{\mathbf{w}})\rangle_T = \frac{T+N}{T-N} \cdot \langle \mathrm{MSE}(\widehat{\mathbf{w}})\rangle_T \tag{5.12}$$

[4]The Mean Squared Error relates to the error measure $E(\mathbf{w})$ defined by Eq. 4.2 as $\mathrm{MSE}(\mathbf{w}) = \frac{2}{T}E(\mathbf{w})$.

Usually, only *one* training set is available. The ensemble average of the training errors is then replaced by the single available observation, leading to the FPE-estimate of the generalization error for this particular model,

$$\widehat{G}(\mathbf{w}) = \frac{T+N}{T-N} \cdot \mathrm{MSE}(\widehat{\mathbf{w}}) \tag{5.13}$$

Often the student network is trained from an error function augmented by an additive *regularization* term as will be described in the following two chapters. The regularization term constrains the weights, thereby in effect limiting the degrees of freedom in the network model. The limitation of degrees of freedom may be seen as a reduction of the *actual* number of weights in the network to an *effective* number of weights N_{eff} [Moo92], $N_{\text{eff}} < N$. Let $\mathbf{A}$ denote the matrix with second derivatives of the quadratic error function, $\mathbf{D}$ denote the matrix containing the second derivatives of the additive regularization term and let $\mathbf{H} = \mathbf{A} + \mathbf{D}$ denote the second derivatives of the augmented cost function (refer also to page 59). The effective number of weights may then be estimated as [LH94]

$$N_{\text{eff}} = \mathrm{tr}(\mathbf{A}\mathbf{H}^{-1}\mathbf{A}\mathbf{H}^{-1}) \tag{5.14}$$

For a more rigorous definition of the effective number of parameters the reader is referred to [LH94]. When estimating the generalization error using the FPE-estimate for a network trained on a regularized error function the actual number of parameters N should be replaced by the effective number of parameters N_{eff},

$$\widehat{G}(\mathbf{w}) = \frac{T+N_{\text{eff}}}{T-N_{\text{eff}}} \cdot \mathrm{MSE}(\widehat{\mathbf{w}}) \tag{5.15}$$

From this expression it is seen that the estimated generalization error for a regularized network will be smaller than that of an unregularized network having the same training error. This is due to the effect that a (small) regularization term will reduce the ability of the networks in the ensemble to adapt themselves to the noise on the training set as seen from Eq. (5.10), i.e., to *overfit*, leading to an improved generalization ability as will be described in the following chapter.

The extent to which the derivations outlined above leading to the FPE-estimate applies to dynamic systems like e.g., recurrent models is not entirely clear. One complication of a formal derivation of the FPE-estimate for general dynamic systems will be the influence of transient effects when e.g., expanding the student network training error around the teacher network training error computed on the same finite training set. In this case the exact value of the teacher function training error will be dependent on the initial value of the unmeasurable internal state vector; this needs to be taken into account when calculating the ensemble average of the errors.

Despite the lack of formal justification it has been briefly attempted in this work to apply the FPE-estimate to recurrent networks in order to empirically assess the relevance of the estimator for this model type. In particular the FPE-estimate has been applied as a means for selection of the optimal model structure among the networks resulting from a pruning procedure as it was done for feed-forward networks in e.g., [SHL93, HMPHL96]. The results are reported in chapter 10.

Chapter 6

Model complexity optimization

In the previous chapter it was emphasized that the quality of a trained model should be measured in terms of the generalization ability rather than the error on the specific data set used for training the model, and several approaches towards estimating the expected generalization error were described. A measure of great importance to the generalization ability of a model is the *model complexity*, i.e., in the case of neural networks the number of hidden units and weights. If the chosen model is too simplistic it will not be flexible enough to emulate the dynamics of the system which produced the time series; this situation is often denoted as "underfitting" and will naturally lead to a large generalization error. On the other hand, if the chosen model is too complex the excess degrees of freedom will allow the model to fit not only the original signal but also the additional noise, due to the finite size training set. This situation is denoted as "overfitting" and will also lead to a large generalization error. The phenomena of underfitting and overfitting are illustrated in many text books [HKP91, Hay94, Bis95] in terms of a polynomial fit to noise contaminated data, using too low and too high order of the fitting polynomium, respectively. These illustrations emphasize the need to choose not only a proper functional family when modeling but also a proper model complexity.

The problems of underfitting and overfitting were illustrated mathematically in [GBD92] where it was shown that the generalization error splits into three components,

$$G(\widehat{\mathbf{w}}) = (\text{noise variance}) + (\text{model bias})^2 + (\text{model variance}) \ . \tag{6.1}$$

The first component is the variance of the additive noise which is independent of the chosen model and indicates an absolute lower bound on the obtainable generalization error. The second component is "model bias squared" which indicates an offset from the lowest possible generalization error due to a systematic difference on average from the teacher function. This term will dominate the generalization error if the chosen model is too simplistic and thus prone to underfitting as described above. The third term is model variance which indicates an offset from the lowest possible generalization error due to large variability in the models which result from training. A large model variance thus indicates that widely different models may result from training on different realizations of the fixed size training set. This will be the case for a model with excess degrees of freedom, hence prone to "overfitting."

In order to decrease the model bias and thereby the bias term of the generalization error we may increase the model complexity, i.e., the number of parameters. This will however increase the model variance and lead to an increasing variance contribution to the generalization error. Only in the case of an *un*biased[1] model and in the limit of an infinite training set will it be possible to eliminate both the bias term as well as the variance term in Eq. (6.1). In practice we are however training on a finite size training set and therefore need to choose a model complexity which makes a *tradeoff* between the bias term and the variance term of Eq. (6.1) in order to minimize the generalization error. This tradeoff is also denoted the "bias/variance dilemma" [GBD92]. The following contains a description of various methods for controlling the model complexity and thereby choosing an optimal model structure.

6.1 Regularization

A possible approach towards controlling the model complexity and thus seek for the optimal bias/variance tradeoff is by *regularizing* the cost function, i.e., to modify the original cost function in such a way that constraints are imposed on the parameters in order to reduce the degrees of freedom in the model. This approach is also called complexity regularization [Hay94]. Regularization of the cost function is usually accomplished by augmenting the original cost function $E(\mathbf{w})$ by an additive regularization term $R(\mathbf{w})$, resulting in the total cost function

$$C(\mathbf{w}) = E(\mathbf{w}) + R(\mathbf{w}) \tag{6.2}$$

A commonly used regularization term is of the form

$$C(\mathbf{w}) = E(\mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \ , \tag{6.3}$$

where α is a positive constant denoted the *weight decay*; the division by a factor of two is applied for convenience when computing derivatives. The effect of the weight decay is to bias the weights uniformly towards zero during training and thus in effect reduce degrees of freedom; the "persistence" of the bias is determined by the magnitude of the weight decay parameter α. The bias towards zero will reduce the magnitude of the weights and therefore bias the network towards a simple linear model as the sigmoid activation functions are approximately linear for small weights. For an illustrative elaboration on the simple weight decay the reader is referred to e.g., section 9.2 in [Bis95].

The simple weight decay is a special case of a more general regularizer of the form

$$C(\mathbf{w}) = E(\mathbf{w}) + \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{D}\mathbf{w} \tag{6.4}$$

where $\mathbf{D}$ is a positive definite matrix. It is common to keep the matrix $\mathbf{D}$ diagonal but an individual weight decay parameter is sometimes assigned with each weight, i.e., $D_{ii} = \alpha_i$ and $D_{ij} = 0$, $i \neq j$. Instead of assigning an individual weight decay parameter with each weight we can assign different weight decay parameters with certain *groups* of weights; e.g., in [SHL93] two different weight decay parameters were used for a feed-forward network, one for the weights on the input side of the layer of hidden units and one for the weights on the output side. For recurrent networks it is of course possible to do likewise. E.g., the weights of a recurrent network like the one illustrated in Figure 3.6 may be split into two groups, one group which contains the weights which would also be present in a feed-forward network, the other group containing the feedback weights. If the indices of the

[1]A model which is capable of implementing the teacher function. This is also denoted as a *complete* model [Lar93].

feed-forward weights are collected in the set F and the indices of the feedback weights are collected in the set B we may regularize the recurrent network as

$$C(\mathbf{w}) = E(\mathbf{w}) + \frac{1}{2}\alpha_{\mathrm{ff}} \sum_{i \in F} w_i^2 + \frac{1}{2}\alpha_{\mathrm{rec}} \sum_{i \in B} w_i^2 \tag{6.5}$$

where α_{ff} and α_{rec} are the weight decay parameters associated with the two groups of weights. The intriguing about this construction of the regularizer is that it allows for control of not only the overall model complexity but also the degree of feedback in the network; i.e., in the limit of $\alpha_{\mathrm{rec}} \to \infty$ the recurrent network will reduce to a feed-forward network.

The probably most commonly adopted method for determination of proper values for the weight decay parameters is by the trial-and-error method, i.e., by appropriate sampling of the weight decay parameter space, training a number of networks for each weight decay sample and then estimate the generalization ability of the resulting networks on a validation set set; this method has been adopted in the present work as well. However, the literature contains suggestions to algorithmic determination of the weight decays. In [HRSL94, HR94, HMPHL96] the weight decay parameters were determined by *alternating* minimization of the augmented cost function wrt. the weights and minimization of an analytical generalization error estimate (e.g., the FPE-estimate) wrt. the weight decay parameters, and encouraging results were obtained. A related approach was adopted in [LHSO96, LSAH97], where the weight decay parameters were determined by minimization of the estimated generalization error obtained on a *validation set*. These approaches towards algorithmic determination of the weight decay parameters has yet to be tested for recurrent networks.

It is finally noted that regularization is of importance not only from a model complexity point of view but during training also from a *numerical* point of view. This will be described in more detail in section 7.3.

6.2 Architecture optimization

As an alternative to the indirect control of the model complexity by regularization one may resort to a more direct manipulation of the network architecure in search of an optimal model for the problem (and amount of data) at hand, i.e., to directly manipulate the number of units in the network as well as their connectivity.

A naive approach towards exploring the space of possible network architectures is by exhaustive search of possible network architectures. A simple search method is to vary the number of hidden units N_h and determine the performance of the resulting models. This procedure is simple to apply but the approach may require significant computational effort as many different networks have to the trained. An additional drawback of this method is that it only searches a very restricted class of network models, as the connectivity of the networks is not being adjusted.

More selective procedures towards obtaining an optimal network architecture have been suggested in the literature. Two classes of such procedures are the *pruning algorithms* and the *growing algorithms*. Pruning alorithms work by training a network with excess degrees of freedom and gradually remove connections or complete units in order to obtain an optimal architecture. Growing algorithms work the other way around by starting with a very simple network architecture and gradually add complexity in terms of hidden units and connections. Actually these two approaches towards architecture optimization may be combined as it was done in e.g., [HP94]. Growing algorithms have not been considered in this work and the reader is referred to e.g., [HKP91, Bis95] for an overview of these methods.

The focus of this work has been on pruning algorithms in line with what seems to be the majority of the literature on architecture optimization. Part of the success of pruning algorithms over growing algorithms is owed to the renowed pruning algorithms Optimal Brain Damage (OBD) and Optimal Brain Surgeon (OBS) which will be described in detail in the following sections.

Both of these methods work from a network initially having excess degrees of freedom and trained to a local minimum of the cost function. The weights are then ranked according to relative importance, or *saliency*, and weights having low saliency are eliminated. The saliency of a weight is measured in terms of its effect on the training error, i.e., the increase in training error that will result if the weight is eliminated. The rationale behind this method is that if we eliminate the least salient weights according to training error we gracefully relieve the capability of the network to overfit the training data and thus improve generalization.

The optimal way in which to determine the saliency of a weight in terms of training error would be to eliminate each weight of the network in turn, each time retraining the reduced network to a new local minimim of the cost function and finally determining the saliency of the eliminated weight as the discrepancy between the training errors from before and after the weight was eliminated. The resulting "optimal" saliencies reflect the "true" consequences of eliminating a particular weight but this approach is computationally very costly to implement. Even so, this way of determining saliencies was applied in [Tho91] and has since been denoted as the "brute force" method. As an alternative to saliencies computed by brute force, OBD and OBS work from *approximations* to these optimal saliencies.

The weight elimination procedure is usually succeeded by retraining of the reduced network to a new minimum of the cost function after which the elimination procedure is repeated. This results in a nested family of network architectures among which we must choose the optimal. This choice is usually based on the generalization error of the resulting networks, estimated either on a validation set or by an analytical generalization error estimate like the FPE estimate. We now formulate a repice for the pruning procedure as follows:

1. Train the network to a minimum of the cost function.

2. Estimate the generalization error for the resulting network.

3. Determine whether any further weights should be eliminated. If this is not the case, go to item 6.

4. Determine the saliencies for all weights in the network.

5. Rank the weights according to saliency and eliminate the least salient weight(s). Go to item 1.

6. Stop the pruning procedure and choose the network with smallest estimated generalization error.

The following two sections explain how saliencies are estimated for the OBD and OBS pruning schemes and relevant expressions are derived. The derivations will be performed for a cost function augmented by an additive regularization term of the form of Eq. (6.4) as the cost function should generally be regularized in order to handle numerical problems, as discussed in chapter 7. The resulting expressions will therefore differ slightly from the original expressions in [CDS90, HS93]. In the following, $\mathbf{H}$ will denote the Hessian of the *augmented* cost function, obtained as

$$\mathbf{H} = \frac{\partial^2 C(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^{\mathrm{T}}} = \frac{\partial^2 E(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^{\mathrm{T}}} + \frac{\partial^2 R(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^{\mathrm{T}}} = \mathbf{A} + \mathbf{D} \tag{6.6}$$

Thus, $\mathbf{A}$ denotes the second derivatives of the error measure $E(\mathbf{w})$ and $\mathbf{D}$ is the second derivatives of the regularization term.

6.2.1 Optimal Brain Damage

This section describes the Optimal Brain Damage (OBD) pruning scheme which was introduced in [CDS90]. The overall pruning procedure is as outlined in the recipe above. In each "round" of weight eliminations the saliency of a weight is computed as the *estimated* increase in training error $E(\mathbf{w})$ if the weight is eliminated. Eliminating a weight is equivalent to setting it to zero as it then no longer contributes to the network output; this way, the saliency is obtained as the estimated change in training error if the weight is set to zero. Consequently, the saliency estimates of OBD do not include the effect of retraining the reduced network.

Let $\widehat{\mathbf{w}}$ denote the parameters resulting from training the network to a local minimum of the augmented cost function $C(\mathbf{w})$. As we are interested in changes in the error measure $E(\mathbf{w})$ we expand this to second order around the current parameter estimate,

$$E(\widehat{\mathbf{w}} + \delta\mathbf{w}) \approx E(\widehat{\mathbf{w}}) + \delta\mathbf{w}^T \nabla E(\widehat{\mathbf{w}}) + \frac{1}{2}\delta\mathbf{w}^T \mathbf{A} \delta\mathbf{w} \tag{6.7}$$

where $\delta\mathbf{w}$ denotes a perturbation of the parameter vector $\widehat{\mathbf{w}}$ and $\mathbf{A}$ denotes the Hessian matrix of the error measure E calculated at $\widehat{\mathbf{w}}$. Assuming the point $\widehat{\mathbf{w}}$ to be a local minimum of the total cost function C, we have

$$\begin{array}{l} \nabla C(\widehat{\mathbf{w}}) = \nabla E(\widehat{\mathbf{w}}) + \mathbf{D}\widehat{\mathbf{w}} = \mathbf{0} \\ \Updownarrow \\ \nabla E(\widehat{\mathbf{w}}) = -\mathbf{D}\widehat{\mathbf{w}} \end{array} \tag{6.8}$$

This expression is inserted into (6.7) which leads to

$$\begin{aligned} \delta E &= E(\widehat{\mathbf{w}} + \delta\mathbf{w}) - E(\widehat{\mathbf{w}}) \\ &= -\delta\mathbf{w}^T \mathbf{D}\widehat{\mathbf{w}} + \frac{1}{2}\delta\mathbf{w}^T \mathbf{A}\delta\mathbf{w} \end{aligned} \tag{6.9}$$

where δE denotes the change in the error measure E resulting from the weight perturbation $\delta\mathbf{w}$. Assuming that the perturbation of the parameter vector $\widehat{\mathbf{w}}$ corresponds to setting the jth parameter to zero,

$$\delta\mathbf{w} = -(\mathbf{e}_j^T \widehat{\mathbf{w}})\mathbf{e}_j \tag{6.10}$$

in which $\mathbf{e}_j$ denotes the jth unit vector, the resulting change in the error measure E is obtained as

$$\delta E_j = \widehat{w}_j \sum_i D_{ji}\, \widehat{w}_i + \frac{1}{2} A_{jj}\, \widehat{w}_j^2 \tag{6.11}$$

where $\widehat{w}_j$ denotes the jth element in $\widehat{\mathbf{w}}$. In the case of individual weight decay parameters α_j we thus estimate the saliency of the jth weight as

$$\delta E_j = \left(\alpha_j + \frac{1}{2} \frac{\partial^2 E(\mathbf{w})}{\partial w_j^2} \right) \widehat{w}_j^2 \quad . \tag{6.12}$$

The OBD pruning scheme allows for the elimination of more than one parameter at a time. In this case it is assumed that the total change δE in the error measure can be approximated by the sum of the saliencies resulting from setting the parameters to zero one at a time,

$$\delta E_P \approx \sum_{j \in P} \delta E_j = \sum_{j \in P} \left(\alpha_j + \frac{1}{2} \frac{\partial^2 E(\mathbf{w})}{\partial w_j^2} \right) \widehat{w}_j^2 \tag{6.13}$$

where P denotes the indices for the pruned weights. Approximating the total change in the error measure in this way is equivalent to the assumption that off-diagonal elements of the Hessian can be neglected,

$$\frac{\partial^2 E(\mathbf{w})}{\partial w_i \partial w_j} \approx 0 \quad , \quad i \neq j \tag{6.14}$$

The remaining diagonal terms of the Hessian are furthermore approximated as

$$\frac{\partial^2 E(\mathbf{w})}{\partial w_j^2} \approx \sum_{t=1}^{T} \frac{\partial y(t)}{\partial w_j} \cdot \frac{\partial y(t)}{\partial w_j} \tag{6.15}$$

which corresponds to the Gauss-Newton approximation Eq. (4.22) to the second derivatives as described in section 4.4.2.

As described above, the saliency estimates of the OBD pruning scheme do not include the effect of retraining the reduced network to a new local minimum of the cost function. As the training error $E(\mathbf{w})$ will *decrease*[2] during retraining the OBD saliency may be seen as an *upper* bound on the change in training error *after* retraining of the remaining parameters, as pointed out in [GHK+93]. Even so it is not clear from the OBD saliencies how retraining will affect the *ranking* of the weights.

6.2.2 Optimal Brain Surgeon

This section describes the Optimal Brain Surgeon (OBS) pruning scheme which was introduced in [HS93]. The basic idea of OBS is the same as for OBD, namely to estimate the change in training error if a weight is set to zero, working from a second-order expansion. As opposed to OBD however, OBS seeks to include in the saliency estimate the effect of retraining the reduced network and thereby provide a saliency estimate which is closer to the "optimal" saliency as described above. The retraining effect is incorporated into the saliency by reestimation of the remaining parameters in the network to a new minimum *within the quadratic approximation* of the training error $E(\mathbf{w})$, calculating the change in training error in this minimum. As for OBD, the expressions for OBS are here derived for a regularized cost function, the results of which were originally presented in [HP94].

[2]Note that it is theoretically possible for the error measure $E(\mathbf{w})$ to actually *increase* during retraining when training from a regularized cost function $C(\mathbf{w})$. However, if the weight decay parameters are *small* this will seldomly occur in practice.

Once more, let $\widehat{\mathbf{w}}$ denote the parameters resulting from training the network to a local minimum of the augmented cost function $C(\mathbf{w})$. The cost function is expanded to second order around $\widehat{\mathbf{w}}$,

$$C(\mathbf{w}) \approx C(\widehat{\mathbf{w}}) + \frac{1}{2}\delta\mathbf{w}^{\mathrm{T}}(\mathbf{A}+\mathbf{D})\delta\mathbf{w} \tag{6.16}$$

where $\delta\mathbf{w}$ denotes a perturbation of the parameter vector $\widehat{\mathbf{w}}$; the first order term vanishes as $\widehat{\mathbf{w}}$ is assumed to be a local minimum of C. Furthermore recall from Eq. (6.6) that the Hessian matrix $\mathbf{H}$ for the augmented cost function $C(\mathbf{w})$ is obtained as $\mathbf{H} = \mathbf{A}+\mathbf{D}$ where $\mathbf{A}$ is the matrix with second derivatives of the error measure $E(\mathbf{w})$ computed at $\widehat{\mathbf{w}}$.

We now set the jth weight to zero which is expressed as

$$\delta\mathbf{w}_j^{\mathrm{T}}\mathbf{e}_j + \widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{e}_j = 0 \;\Leftrightarrow\; \delta\mathbf{w}_j^{\mathrm{T}}\mathbf{e}_j = -\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{e}_j \tag{6.17}$$

where $\mathbf{e}_j$ denotes the jth unit vector. Reestimation of the remaining weights to a new minimum within the quadratic approximation is equivalent to determination of the minimum of Eq. (6.16) subject to an equality constraint given by Eq. (6.17). We thus introduce the Lagrange function

$$\tilde{C}_j(\mathbf{w}) = C(\mathbf{w}) + \lambda_j(\delta\mathbf{w}_j + \widehat{\mathbf{w}})^{\mathrm{T}}\mathbf{e}_j \tag{6.18}$$

in which $C(\mathbf{w})$ denotes the approximation from Eq. (6.16). The coefficient λ_j is the Lagrange multiplier and $\delta\mathbf{w}_j$ as well as λ_j are now to be determined in such a way that the expression (6.18) is minimized, at the same time satisfying Eq. (6.17). First, $\delta\mathbf{w}_j$ is determined so as to minimize Eq. (6.18). This is obtained by setting the derivatives of Eq. (6.18) equal to zero,

$$\begin{aligned} & \frac{\partial C(\mathbf{w})}{\partial(\delta\mathbf{w}_j)} + \lambda_j\mathbf{e}_j = \mathbf{0} \\ \Updownarrow \\ & \mathbf{H}\delta\mathbf{w}_j + \lambda_j\mathbf{e}_j = \mathbf{0} \\ \Updownarrow \\ & \delta\mathbf{w}_j = -\lambda_j\mathbf{H}^{-1}\mathbf{e}_j \end{aligned} \tag{6.19}$$

In order to determine λ_j we use Eq. (6.19) inserted into Eq. (6.17),

$$\begin{aligned} & \lambda_j\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{e}_j = \widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{e}_j \\ \Updownarrow \\ & \lambda_j = \frac{\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{e}_j}{\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{e}_j} \end{aligned} \tag{6.20}$$

The reestimation of the remaining parameters after setting the jth weight to zero thus leads to a change in the parameter vector which is determined by insertion of Eq. (6.20) into Eq. (6.19),

$$\delta\mathbf{w}_j = -\frac{\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{e}_j}{\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{e}_j}\mathbf{H}^{-1}\mathbf{e}_j \tag{6.21}$$

Using this expression it is now possible to estimate the change in the error function $E(\mathbf{w})$ resulting from the parameter change, i.e., the saliency. A Taylor expansion to second order around $\widehat{\mathbf{w}}$ leads to

$$E_j(\mathbf{w}) = E(\widehat{\mathbf{w}}) + \delta\mathbf{w}_j^T\nabla E(\widehat{\mathbf{w}}) + \frac{1}{2}\delta\mathbf{w}_j^{\mathrm{T}}\mathbf{A}\delta\mathbf{w}_j \tag{6.22}$$

The first order term in Eq. (6.22) may be determined from the first order derivatives of the regularized cost function $C(\mathbf{w})$ as it is assumed that $\widehat{\mathbf{w}}$ is a local minimum,

$$\nabla C(\widehat{\mathbf{w}}) = \nabla E(\widehat{\mathbf{w}}) + \mathbf{D}\widehat{\mathbf{w}} = \mathbf{0} \Leftrightarrow \nabla E(\widehat{\mathbf{w}}) = -\mathbf{D}\widehat{\mathbf{w}} \tag{6.23}$$

This expression is inserted into Eq. (6.22) which leads to

$$\delta E_j(\mathbf{w}) = -\delta\mathbf{w}_j^{\mathrm{T}}\mathbf{D}\widehat{\mathbf{w}} + \frac{1}{2}\delta\mathbf{w}_j^{\mathrm{T}}\mathbf{A}\delta\mathbf{w}_j \tag{6.24}$$

in which $\delta E_j(\mathbf{w}) = E_j(\mathbf{w}) - E(\widehat{\mathbf{w}})$. Finally the expression for the retraining Eq. (6.19) is inserted into Eq. (6.24) and the expression which estimates the saliency when setting the jth weight to zero is calculated as

$$\delta E_j(\mathbf{w}) = \lambda_j\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{D}\widehat{\mathbf{w}} + \frac{1}{2}\lambda_j^2\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{A}\mathbf{H}^{-1}\mathbf{e}_j \tag{6.25}$$

The expression (6.25) is computationally fairly complex but can be simplified somewhat if assuming that the regularization is a simple weight decay in which case $\mathbf{H} = \mathbf{A} + \mathbf{D} = \mathbf{A} + \alpha\mathbf{I}$; note that the weight decay parameters are assumed to be identical for *all* weights in the network. In this case the saliencies are "simplified" as

$$\begin{aligned}
\delta E_j(\mathbf{w}) &= \alpha\lambda_j\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\widehat{\mathbf{w}} + \frac{1}{2}\lambda_j^2\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}(\mathbf{H}-\mathbf{D})\mathbf{H}^{-1}\mathbf{e}_j \\
&= \alpha\frac{\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{e}_j(\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\widehat{\mathbf{w}})}{\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{e}_j} + \frac{1}{2}\frac{(\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{e}_j)^2}{(\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{e}_j)^2}\left(\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{e}_j - \alpha\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-2}\mathbf{e}_j\right) \\
&= \frac{1}{2}\frac{(\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{e}_j)^2}{\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{e}_j} + \alpha\left(\frac{\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{e}_j(\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\widehat{\mathbf{w}})}{\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{e}_j} - \frac{1}{2}\frac{(\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{e}_j)^2\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-2}\mathbf{e}_j}{(\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{e}_j)^2}\right) \\
&= \frac{1}{2}\frac{\widehat{w}_j^2}{\mathbf{H}_{jj}^{-1}} + \alpha\left(\frac{\widehat{w}_j(\mathbf{e}_j^{\mathrm{T}}\mathbf{H}^{-1}\widehat{\mathbf{w}})}{\mathbf{H}_{jj}^{-1}} - \frac{1}{2}\frac{\widehat{w}_j^2\mathbf{H}_{jj}^{-2}}{(\mathbf{H}_{jj}^{-1})^2}\right)
\end{aligned} \tag{6.26}$$

where $\widehat{w}_j$ denotes the jth component of the parameter vector at the local minimum $\widehat{\mathbf{w}}$ and $\mathbf{H}_{jj}^{-2}$ is obtained as the jth diagonal element of $(\mathbf{H}^{-1} \cdot \mathbf{H}^{-1})$. Careful accounting of the computational complexity reveals that Eq. (6.26) saves $N_w - 1$ vector dot products compared to Eq. (6.25), where N_w is the number of weights in the network.

From the reestimation vector Eq. (6.21) and the resulting saliency Eq. (6.26) we note that the OBS pruning scheme requires the inverse of the *full* Hessian whereas OBD requires the inverse of the diagonal elements only. The Hessian is taken as the Gauss-Newton approximation Eq. (4.22) as originally suggested in [HS93]; refer to section 4.8 for computation of the Hessian for recurrent networks. In [HS93] it was argued that one might save on computation by using an iterative scheme for calculation of the inverse Hessian, namely the matrix inversion lemma, rather than direct matrix inversion. However, a detailed count of operations reveals that it is only beneficial to use the iterative scheme in the atypical case $N_w > T$, i.e., when the number of weights in the network is larger than the number of training examples. Refer to appendix C for the detailed operations count.

Note from Eq. (6.26) that when working from a regularized cost function the saliency estimate may actually become *negative*; whereas a perturbation of $\widehat{\mathbf{w}}$ will always lead to an increase of the regularized cost $C(\mathbf{w})$ within the quadratic approximation it may be that a perturbation will actually decrease the training error $E(\mathbf{w})$ while increasing the regularization term, resulting in a negative saliency estimate. This has been experienced to occur in practice. In this case the weight selected for pruning is the one having the largest magnitude *negative* saliency.

If no regularization is used corresponding to $\mathbf{D} = \mathbf{0}$, then Eq. (6.26) reduces to the "original" saliency estimate presented in [HS93],

$$\begin{aligned} \delta E_{OBS,j}(\mathbf{w}) &= \frac{1}{2}\lambda_j^2 \mathbf{e}_j^{\mathrm{T}} \mathbf{A}^{-1} \mathbf{e}_j \\ &= \frac{1}{2} \frac{(\widehat{\mathbf{w}}^{\mathrm{T}} \mathbf{e}_j)^2}{\mathbf{e}_j^{\mathrm{T}} \mathbf{A}^{-1} \mathbf{e}_j} \\ &= \frac{1}{2} \frac{\widehat{w}_j^2}{\mathbf{A}_{jj}^{-1}} \end{aligned} \tag{6.27}$$

where $\mathbf{A}_{jj}^{-1}$ is the jth diagonal element of the inverse Hessian matrix $\mathbf{A}^{-1}$ for $E(\mathbf{w})$ computed in $\widehat{\mathbf{w}}$. Naturally, in this case the saliency estimates will always be positive.

As mentioned in the general description of pruning algorithms above, the weight elimination should always be succeeded by a retraining step which trains the reduced network to a new local minimum in order to ensure that the gradient is zero as assumed by both OBD and OBS. In [HS93] it was boldly claimed that no further retraining of the weights is necessary beyond the "built-in" reestimation of the remaining parameters. This claim relies heavily upon the accuracy of the quadratic approximation to the cost function. Whereas the OBS pruning scheme and thereby the built-in reestimation of remaining parameters is *exact* in the case of a linear model the second-order approximation may be very crude for nonlinear models like neural networks as it will be illustrated in chapter 10. In fact, experience from this work has shown that the OBS pruning scheme is practically useless if no further retraining by e.g., the Gauss-Newton method is applied.

6.2.3 Nuisance parameters

This section directs the attention towards a problem when estimating saliencies that seems not to be generally appreciated, namely the problem of *nuisance* parameters, e.g., [Lar93, Rip96]; nuisance parameters were also treated in [PHL96] included in appendix H. Nuisance parameters are weights which enter the neural network but have no influence on the output, regardless of their value.

As an example, consider the elimination of an output weight w_{oj} in a feed-forward network, described in section 3.2. In this case all the weights to the corresponding hidden unit are in effect also pruned away. Such a situation is well-known in the statistics literature on model selection where such "ghost" input weights are known as nuisance parameters. When using the OBS pruning scheme it is important to remove these parameters from the network function before estimating the saliency $\delta E_{\mathrm{OBS},oj}$, as they will otherwise give "spurious" contributions to both the saliency estimate as well as the corresponding reestimation vector. Applying OBS without taking this fact into consideration often results in sudden "jumps" in the level of the network error due to pruning of an important weight based on a corrupted saliency estimate.

This phenomenon is illustrated in Figure 6.1 for a feed-forward network trained on the Mackey-Glass series, described in appendix A.2, and pruned using the OBS pruning scheme. The figure illustrates the errors that resulted *after* further retraining of the

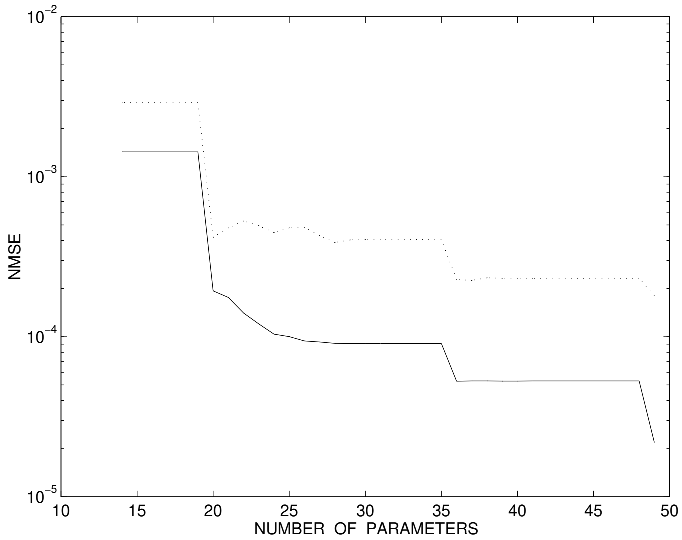


Figure 6.1: "Jumps" when pruning a feed-forward network using the OBS pruning scheme. The solid line denotes the training error, the dotted line denotes the error on a test set. The error levels resulted *after* further retraining by the damped Gauss-Newton method.

remaining weights by the damped Gauss-Newton method. The very first weight which is chosen for pruning is in fact an output weight, so that all the input weights to the corresponding hidden unit are in effect also pruned away. The poor choice of the weight to eliminate leads to a dramatic increase in both training and test error and the network cannot recover from the devastating effect even after further retraining of the weights; prior to the further retraining the increase in errors was several orders of magnitude larger than that in Figure 6.1. After elimination of the resulting nuisance parameters and further retraining there are 36 parameters left in the network; once more does OBS select an output weight for pruning, leading to yet another large increase in errors from which the network cannot recover.

In order to handle the problem of nuisance parameters when estimating the OBS saliency and corresponding reestimation vector for a particular weight it is required to remove the resulting nuisance parameters from the weight vector $\widehat{\mathbf{w}}$ and the corresponding rows and columns of the Hessian $\mathbf{H}$ before calculating these quantitites from Eq. (6.21) and Eq. (6.26). Removing rows and columns from the Hessian corresponding to the superfluous weights and forming a reduced (regularized) Hessian $\mathbf{H}_1$ is straightforward. However, inverting each of the different (sub-)matrices that will result when estimating saliencies for a given network architecture may be very computationally expensive. This cost can be considerably reduced by *rearranging* the rows and columns of $\mathbf{H}^{-1}$ as

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_1 & \mathbf{H}_2 \\ \mathbf{H}_3 & \mathbf{H}_4 \end{bmatrix} \quad \rightarrow \quad \mathbf{H}^{-1} = \begin{bmatrix} (\mathbf{H}^{-1})_1 & (\mathbf{H}^{-1})_2 \\ (\mathbf{H}^{-1})_3 & (\mathbf{H}^{-1})_4 \end{bmatrix} \tag{6.28}$$

where $\mathbf{H}_2$, $\mathbf{H}_3$ and $\mathbf{H}_4$ are the rows and columns corresponding to the nuisance parameters. Using a standard lemma[3] for partitioned matrices, we obtain

$$(\mathbf{H}_1)^{-1} = (\mathbf{H}^{-1})_1 - (\mathbf{H}^{-1})_2[(\mathbf{H}^{-1})_4]^{-1}(\mathbf{H}^{-1})_3 \tag{6.29}$$

[3]Schur's matrix inversion lemma.

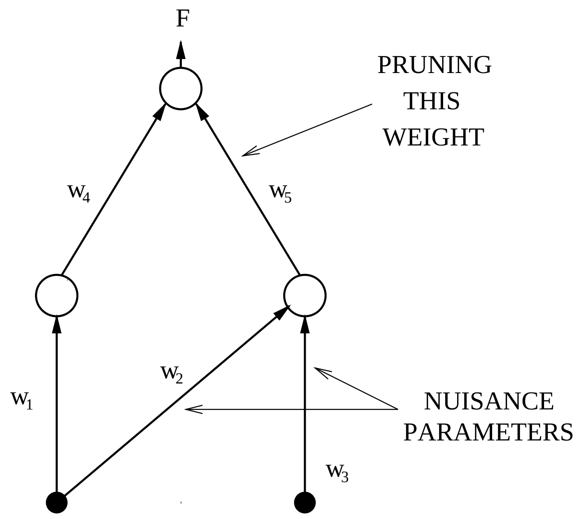


Figure 6.2: Example of nuisance parameters when pruning an output weight in a feed-forward network.

which only calls for inversion of the (small) submatrix $(\mathbf{H}^{-1})_4$ with a dimension equal to the number of nuisance parameters. Consequently, in each "round" of pruning it is only necessary to invert the full Hessian $\mathbf{H}$ once; in the case of nuisance parameters only smaller submatrices need to be inverted when computing the inverse of the reduced Hessian from Eq. (6.29), thus reducing the amount of computation required.

Figure 6.2 illustrates how the input weights to a hidden unit in a feed-forward network become nuisance parameters when estimating the saliency of the corresponding output weight. Before computing the saliency for the output weight the nuisance parameters need to be eliminated from the second-order expansion of the cost function in order not to give spurious contributions to the saliency estimate. This is obtained by partitioning the (inverse) Hessian as described above, leading to the submatrices

$$\mathbf{H} = \begin{bmatrix} H_{11} & H_{12} & H_{13} & H_{14} & H_{15} \\ H_{21} & H_{22} & H_{23} & H_{24} & H_{25} \\ H_{31} & H_{32} & H_{33} & H_{34} & H_{35} \\ H_{41} & H_{42} & H_{43} & H_{44} & H_{45} \\ H_{51} & H_{52} & H_{53} & H_{54} & H_{55} \end{bmatrix} \Rightarrow \begin{array}{lcl} \mathbf{H}_1 & = & \begin{bmatrix} H_{11} & H_{14} & H_{15} \\ H_{41} & H_{44} & H_{45} \\ H_{51} & H_{54} & H_{55} \end{bmatrix} \\ \\ \mathbf{H}_4 & = & \begin{bmatrix} H_{22} & H_{23} \\ H_{32} & H_{33} \end{bmatrix} \end{array}$$

where $\mathbf{H}_1$ corresponds to the reduced Hessian and $\mathbf{H}_4$ corresponds to the nuisance parameters.

Nuisance parameters will be a problem not only when estimating saliencies for an output weight in a feed-forward network but also if the weight in question is e.g., the sole input weight to a hidden unit; in this case the corresponding output weight will become a nuisance parameter. When estimating saliencies for the output weights in a *recurrent* network the input weights to the corresponding hidden unit will however not necessarily become nuisance parameters as the hidden unit may be connected to other hidden units in the network, thus still contributing to the network output; refer to Figure 3.6 on page 26 for an illustration of a recurrent network in order to visualize this. When estimating saliencies in general, nuisance parameters will result and must be taken care of either if the weight in question is the sole weight leading *to* a hidden unit or if the weight in question is the sole weight originating *from* a hidden unit.

The description of nuisance parameters in this section has been focused on the OBS pruning scheme. The reason for this is that nuisance parameters are not a problem when computing saliencies by the OBD pruning scheme. This is so as the saliency estimates of OBD involve only the diagonal element of the Hessian corresponding to the weight in question; hence, potential nuisance parameters will not influence the saliency estimate in any way.

For both OBD and OBS it might be that a weight is selected for pruning, leaving nuisance parameters in the reduced network, even if these nuisance parameters are accounted for when estimating the saliency. This will in fact *always* be the case when eliminating the final weights to a hidden unit, thus removing it entirely from the network; as only one weight is pruned at a time, removal of a hidden unit from the network will always leave at least one nuisance parameter in the network. Regardless of their origin, nuisance parameters should be identified and eliminated before commencing retraining of the reduced network as they may also be a nuisance to the training methods applied.

If it is decided to trust the quality of the built-in reestimation of the parameters in the reduced network supplied by OBS and thus neglect further retraining, possibly for a few weight elimination rounds only, computation may be reduced if applying the Schur matrix inversion lemma described above to compute the inverse of the Hessian for the reduced network. As only one weight is pruned at a time the inverse of the reduced network is especially simple to obtain [Lar93] as it merely involves inverting a scalar as seen from Eq. (6.29), unless of course nuisance parameters must be removed from the network as well.

6.2.4 Generalization based saliencies

As described above, the purpose of architecture optimization by pruning is to improve generalization ability of a network model by removal of the least important weights and in this way optimize the network architecture to the problem at hand. Architecture optimization may be viewed as a search for the optimal tradeoff between the model bias and the model variance contributions to the generalization error, due to a limited amount of training data. In order to determine the optimal network in the nested model family resulting from pruning, the generalization error of each model is estimated either on a validation set or by an analytical generalization error estimate. Even though model quality is assessed in terms of generalization error, the saliency measure of both OBD and OBS according to which weights are ranked is based solely on the change in *training* error which will result if a weight is eliminated. This approach is adopted in the hope that if the least salient weights according to training error are deleted we gracefully relieve the danger of overfitting, thus hopefully improving generalization ability. However, as the real object of pruning is to improve generalization, *why not let the saliency itself reflect the possible improvement in generalization error?* This approach was adopted in [PHL96], included in appendix H.

Defining saliency in terms of the change in estimated generalization error rather than training error leads to the concept of generalization based saliencies. The generalization error estimate from which the saliency is estimated may be an analytical expression like the FPE estimate; this is the approach adopted in [PHL96], leading to modified versions of the pruning schemes OBD and OBS, denoted as γOBD and γOBS. Generalization based saliencies may however also be calculated as the estimated change in error on a validation set. In this case OBD and OBS may be applied directly, however replacing the training error with the validation set error; this approach was adopted in [LHSO96].

Estimating saliencies as the expected change in the FPE estimate of the generalization error if a weight is pruned means estimating the change in

$$\widehat{E}_{\text{test}} = \frac{T + N_{\text{eff}}}{T - N_{\text{eff}}} E_{\text{train}} \approx \left(1 + \frac{2N_{\text{eff}}}{T}\right) E_{\text{train}} \quad , \tag{6.30}$$

where T is the number of training examples and N_{eff} is the *effective* number of parameters, estimated as $N_{\text{eff}} = \text{tr}(\mathbf{AH}^{-1}\mathbf{AH}^{-1})$ as described in section 5.3. While OBD and OBS are based on estimates of the change in E_{train} we see from Eq. (6.30) that in order to obtain saliencies that estimate the change in generalization error we must generally take the prefactor into account. We note that if the network is *not* regularized then $N_{\text{eff}} = \text{tr}(\mathbf{AH}^{-1}\mathbf{AH}^{-1}) = \text{tr}(\mathbf{I}) = N$, i.e., the total number of parameters left in the network after pruning, and the prefactor is only a function of the total number of weights. In this case ranking according to training error saliency is equivalent to ranking according to generalization error.

In the generic case of a regularized network we have $N_{\text{eff}} < N$, and we need to evaluate the change in the prefactor, i.e., in the effective number of parameters, associated with the pruning of every weight in the network. Denoting the generalization based saliency of weight $\widehat{w}_j$ as $\delta E_{\text{test},j}$ we find

$$\delta E_{\text{test},j} \approx \delta E_{\text{train},j} - \frac{2\,(N_{\text{eff}} - N_{\text{eff},j})}{T} E_{\text{train}} \tag{6.31}$$

where the number of parameters after pruning of weight j is $N_{\text{eff},j}$ and $\delta E_{\text{train},j}$ is the training error based saliency.

The reader is referred to appendix H for details on modifying the pruning schemes OBD and OBS to incorporate generalization based saliencies. In order to emphasize the use of generalization error for the ranking of weights in the modified methods the prefix γ is applied: γOBD and γOBS.

Both γOBD and γOBS were in [PHL96] applied to feed-forward networks trained on the Mackey-Glass series. However, generalization based saliencies have yet to be applied to recurrent networks. It has not been attempted during this work but remains an interesting task of the future.

Chapter 7

Ill-conditioning in recurrent networks

This chapter addresses the problem of training recurrent networks. Training recurrent networks is generally believed to be a difficult task [Moz93, PF94, TB94] and excessive training times and lack of convergence to an acceptable solution are frequently encountered problems. Many authors have made this observation but few have attempted to diagnose the reasons for these training problems. Among the most debated attempts is the work presented in [BSF94] where it was shown that a recurrent network cannot be trained to store, or *latch*, information for an *arbitrary* period of time using gradient based training algorithms, as the fraction of the gradient relating to information n time steps in the past approaches zero as n becomes large. Naturally, if the dependencies in the data span long time intervals, i.e., the desired output at time T depends on inputs presented at times $t \ll T$, this finding may explain difficulties in training a recurrent network to perform the proper mapping. This problem for recurrent networks of learning long-time dependencies seems to be of particular importance in the context of classification of sequences of arbitrary length, where e.g., the information relevant to a correct classification may be presented to the network already during the first few time steps. Sequence classification-like problems have been the dominating type of problems considered in the literature addressing the long-term dependency learning problem [BSF94, HS95, LHTG96, GLH97]; the problem has *not* been addressed in the context of time series prediction even though it has potential relevance to this application as well. In order to handle the problems of learning long-term dependencies it has been suggested to increase network complexity in various ways. E.g., in [LHTG96, GLH97] it was found that increasing the order of the internal memory facilitates learning of long-term dependencies as this corresponds to "jump-ahead" connections in the time-unfolded network. In [HS95] a rather exotic network construction involving special so-called "memory cells" controlled by high-order "gating units" was reported to be capable of completely eliminating the long-term dependency problem. The construction is capable of storing information for an arbitrary period of time without leading to a decaying gradient wrt. the stored information.

The problem of learning long-time dependencies is potentially relevant to time series prediction problems but it seems to be a manageable problem in this context. In order to be able to model the dynamic system underlying the time series at all it is necessary to assume that the influence of current observations on future observations will decay at an appropriate rate as discussed in section 2.4. The network model need therefore not be capable of storing information for an *arbitrary* period of time but rather for a limited period of time. Thus, the problems caused by insufficient "memory" of the network may be handled in the traditional way of increasing network memory, i.e., by adjusting the network complexity in terms of e.g., number of hidden units.

Another possible explanation for the problems of training recurrent networks was provided in [BGM94] where the problems of reaching satisfactory solutions were attributed to the many local minima of the cost function surface. In order to reach a local minimum of the cost function surface at all it is however a requirement of prime importance that the training algorithm applied is practically capable of reaching such a minimum. This will often not be the case as the performance of commonly used training algorithms rooted in optimization theory are severely hampered by *numerical* problems. Such problems are a seemingly neglected aspect of recurrent network training as it appears that so far no one have treated the recurrent network training problem from a general numerical point of view.

Slow convergence and thus long training times are also frequently encountered problems for feed-forward networks. Even so, it seems that also for this model type little research effort has been directed towards the question of understanding the numerical reasons for why various optimization methods perform poorly when applied to neural network training problems. Apparently, the only reference that has treated this problem in detail is [SBC93], in which the causes for numerical problems were analyzed for feed-forward networks applied to classification problems.

In this chapter the analysis of numerical problems is extended to recurrent networks applied to time series prediction problems but the analysis applies to any application of recurrent networks. In section 7.1 the numerical problems will be related to a large condition number of the Hessian matrix and it is described how a large condition number will affect the convergence of commonly adopted training algorithms. In section 7.2 it will be theoretically analyzed how ill-conditioning of the Hessian might arise; the analysis is performed in terms of the Jacobian matrix. In section 7.3 it is described how the problems of ill-conditioning may be handled by a simple quadratic weight decay term.

7.1 Ill-conditioning

When training adaptive models using traditional iterative optimization methods as described in chapter 4, a measure of great importance to the successful application of these methods is the *condition number* of the Hessian matrix of the cost function. The condition number of a matrix is defined in terms of the singular values obtained from a *Singular Value Decomposition* (SVD).

The SVD decomposition is based on a theorem from linear algebra which states that any matrix $\mathbf{A}$ of dimension $m \times n$ can be factorized into a product,

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}} \ , \tag{7.1}$$

where $\mathbf{U}$ of dimension $m \times n$ and $\mathbf{V}$ of dimension $n \times n$ are column-orthogonal matrices and $\mathbf{D}$ is an $n \times n$ diagonal matrix with elements $D_{ii} = \sigma_i \geq 0$ and $D_{ij} = 0 \ , \ i \neq j$. The elements σ_i are called the singular values of $\mathbf{A}$. For a matrix $\mathbf{A}$ where $m \geq n$ the singular values are obtained as the nonnegative square roots of the eigenvalues of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$, and the columns of $\mathbf{U}$ and $\mathbf{V}$ are the eigenvectors of $\mathbf{A}\mathbf{A}^{\mathrm{T}}$ and $\mathbf{A}^{\mathrm{T}}\mathbf{A}$, respectively. The number of non-zero singular values denotes the *rank* of $\mathbf{A}$, i.e., the number of linearly independent columns in $\mathbf{A}$; if $\mathbf{A}$ does not have full column rank it is said to be *rank-deficient.* See e.g., [PFTV92] for further details regarding the SVD decomposition.

The condition number of $\mathbf{A}$ is now defined as

$$\kappa(\mathbf{A}) = \frac{\sigma_{\max}}{\sigma_{\min}} , \tag{7.2}$$

where $\sigma_{\max}$ is the largest singular value and $\sigma_{\min}$ is the smallest singular value of $\mathbf{A}$. From this definition it is straightforward to see that if $\mathbf{A}$ is a symmetric, positive definite matrix the condition number is obtained as

$$\kappa(\mathbf{A}) = \frac{\lambda_{\max}}{\lambda_{\min}} , \tag{7.3}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the largest and smallest eigenvalues of $\mathbf{A}$, respectively.

If the condition number $\kappa(\mathbf{A})$ is infinite then the matrix $\mathbf{A}$ is singular and some of the columns in $\mathbf{A}$ are linearly dependent. If the condition number is finite but "large", this is an indication of some of the columns in $\mathbf{A}$ being "nearly" linearly dependent; in this case, $\mathbf{A}$ is said to be *ill-conditioned.* Except in special cases, it is in practice rare to find singular values *exactly* equal to zero in e.g., the Hessian matrix $\mathbf{H}$ during training. It is however not uncommon to encounter very small singular values, leading to a large condition number $\kappa(\mathbf{H})$, which may severely hamper the performance of the optimization algorithm employed.

In [GMW81, DS83] it is shown that within the region around a local minimum $\mathbf{w}^*$ where the cost function can be described by a second-order Taylor expansion, the sequence of parameter estimates $\{\mathbf{w}_k\}$ obtained using gradient descent will converge towards $\mathbf{w}^*$ as

$$\|\mathbf{w}_{k+1} - \mathbf{w}^*\| \approx c\,\|\mathbf{w}_k - \mathbf{w}^*\| \;\;,\;\; c = \frac{(\kappa - 1)}{(\kappa + 1)} \tag{7.4}$$

provided that an exact line search is used; κ is here the condition number of the Hessian evaluated in the local minimum, $\kappa(\mathbf{H}(\mathbf{w}^*))$. Thus, if the condition number of the Hessian is large, the constant c will be close to unity and gradient descent will converge very slowly.

If training using a Newton type optimization method we need to solve a linear system of equations Eq. (4.19) based on the Hessian matrix in each iteration in order to obtain the search direction. If the condition number of the Hessian becomes large, the solution might however become unreliable due to the influence of round-off errors when using finite-precision arithmetic. In e.g., [DS83] it is shown how the solution to a system of linear equations like Eq. (4.19) is likely to be entirely unreliable if the condition number $\kappa(\mathbf{H}) \geq \epsilon^{-1}$, where ϵ is the machine precision. According to [DS83] it is however "generally felt" that the solution *may* not be trustworthy already if $\kappa(\mathbf{H}) \geq \epsilon^{-\frac{1}{2}}$; this bound may thus be interpreted as a "rule of thumb", indicating when attention should be paid to the condition number. For the IEEE 64-bit floating point representation[1] the machine precision is $\epsilon = 2^{-52} \approx 2.22 \cdot 10^{-16}$ and the rule of thumb then reads $\kappa(\mathbf{H}) \geq 6.7 \cdot 10^{7}$. This might seem as a "large" number, but this order of magnitude is not uncommon in the framework of either feed-forward networks [SBC93] or recurrent networks, as we shall see in the next chapter. A very large condition number of the Hessian is an indication that the second-order approximation to the cost function is very "flat" in some directions, leading to very large components in these directions when solving for the search direction

[1] The IEEE 64-bit floating point representation is used for e.g., the datatype `double` in the C programming language on most platforms as well as in MATLAB.

vector. The large resulting Newton step, possibly combined with poor precision in the solution due to the large condition number, will lead to a very badly determined search direction and thus to slow convergence towards a local minimum. Thus, if the condition number is large, the otherwise highly effective Gauss-Newton method is likely to perform very poor. Appendix E describes the effect of the condition number when solving a system of linear equations.

As the condition number is very important for the convergence of the methods used for training, an analysis of possible causes for a singular or ill-conditioned Hessian matrix is appropriate. Such analysis was initiated in [CKS91] where it was theoretically indicated that if the training data to each individual input of a feed-forward network is scaled to zero mean, then an eigenvalue of the order of the number of inputs N_I can be suppressed. Further, it was shown how a similarly sized eigenvalue can be suppressed if a symmetric activation function like $\tanh(\cdot)$ is used. However, these simple countermeasures are not adequate for avoiding ill-conditioning in either feed-forward or recurrent networks as will become apparent.

In the following it will be analyzed what might lead to a singular or near-singular Hessian matrix for the quadratic cost function when applied to a recurrent network. The analysis will refer to the Gauss-Newton approximation to the Hessian which is of direct relevance to the Gauss-Newton training method. Furthermore, the Gauss-Newton approximation is the dominating part of the Hessian [SBC93], as will be indicated in section 8.3. In order to facilitate the analysis of causes for large condition numbers we write the Hessian as in Eq. (4.25),

$$\mathbf{H} \approx \mathbf{J}^{\mathrm{T}}\mathbf{J} \ \ , \ \ J_{ti} = \frac{\partial y(t)}{\partial w_i} \tag{7.5}$$

where $\mathbf{J}$ is the Jacobian matrix of dimension $T \times N_w$ whose columns are the partial derivatives of the network output at each time step in the training series; T is here the number of training examples and N_w is the number of weights in the network.

As the condition number of $\mathbf{H}$ is the square of $\kappa(\mathbf{J})$, $\kappa(\mathbf{H}) = \kappa^2(\mathbf{J})$, an analysis of $\mathbf{H}$ can be given in terms of the Jacobian $\mathbf{J}$. E.g., if some of the columns in $\mathbf{J}$ can be shown to be linearly dependent this means that the Jacobian is rank deficient which leads to a singular Hessian. On the other hand, if some of the columns in $\mathbf{J}$ are *nearly* linearly dependent small singular values of the Jacobian will result, leading to ill-conditioning of the Hessian $\mathbf{H}$.

7.2 Analysis of the Jacobian

The Jacobian matrix of a recurrent network will now be analyzed in order to determine in which situations a large condition number of the Jacobian, and thereby of the Hessian matrix, will result. The analysis is split into three parts. The first part treats exact collinearity between certain columns of the Jacobian, thus leading to an infinite condition number. The second part identifies situations that will lead to *nearly* collinear columns of the Jacobian, which is found to be equivalent to a decrease of the smallest eigenvalue of the Hessian, while the third part describes how large eigenvalues of the Hessian might arise.

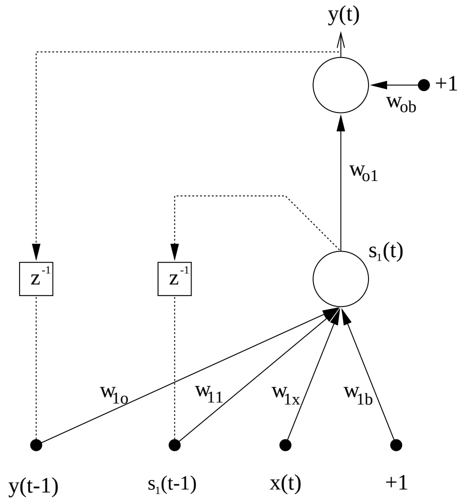


Figure 7.1: Simple recurrent network with redundant connection w_{10}.

7.2.1 Exact column collinearity

The Jacobian analysis is initiated by an identification of situations which will lead to exact linear dependencies between columns of the Jacobian. For the type of recurrent networks considered in this work, defined by Eq. (3.6) and Eq. (3.8), it turns out that there is a built-in rank deficiency in the Jacobian as it is easy to show that some of the columns in **J** will in theory *always* be linear combinations of each other, i.e., they are collinear. The cause of the collinearity is redundancy in the parametrization which will be illustrated by an example involving the small network shown in Figure 7.1. The results are not specific for this particular network but apply generally to networks having an arbitrary number of hidden units.

For simplicity, the network considered here involves only one external input and one hidden unit as seen from Figure 7.1, and the output is thus defined as

$$\begin{aligned} y(t) &= w_{o1}s_1(t) + w_{ob} && (7.6)\\ s_1(t) &= f\left(w_{11}s_1(t-1) + w_{1o}y(t-1) + w_{1x}x(t) + w_{1b}\right) && (7.7)\\ &= f\left((w_{11} + w_{1o}w_{o1})s_1(t-1) + w_{1x}x(t) + (w_{1b} + w_{1o}w_{ob})\right) && (7.8)\\ &= f\left(k_1 s_1(t-1) + w_{1x}x(t) + k_2\right) && (7.9) \end{aligned}$$

where Eq. (7.8) is obtained by insertion of Eq. (7.6) in Eq. (7.7). We see that the network output will remain unchanged as long as the total weighting k_1 of $s_1(t-1)$, $k_1 = w_{11} + w_{1o}w_{o1}$, and the total bias k_2 on the hidden unit, $k_2 = w_{1b} + w_{1o}w_{ob}$, remains constant. w_{o1} and w_{ob} cannot be changed without directly affecting the network output Eq. (7.6) and are therefore kept fixed which we denote by $*$. However, changes in w_{11}, w_{1o} and w_{1b} that satisfies *both* expressions

$$\begin{array}{ccccccc} w_{11} & + & w_{o1}^* \cdot w_{1o} & + & 0 & = & k_1 \\ 0 & + & w_{ob}^* \cdot w_{1o} & + & w_{1b} & = & k_2 \end{array} \qquad (7.10)$$

will leave the network output unchanged. The equations in (7.10) form hyperplanes in parameter space spanned by w_{11}, w_{1o} and w_{1b} and their line of intersection is computed

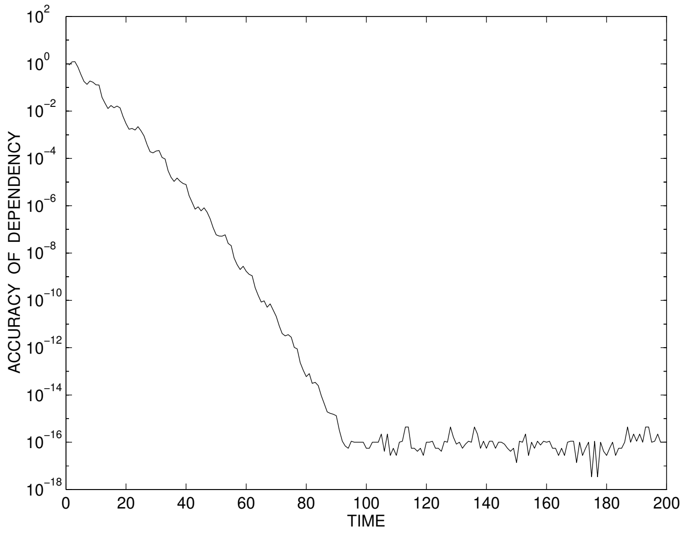


Figure 7.2: "Accuracy" of dependencies between columns in the Jacobian (see text).

as

$$(w_{11}, w_{1o}, w_{1b}) = (k_1, 0, k_2) + t(-w_{o1}^*, 1, -w_{ob}^*) \tag{7.11}$$

parametrized by t. The line defines a direction in parameter space in which the network output is constant. The constant network output means that derivatives in this direction are zero. Thus, columns in the Jacobian corresponding to (w_{11}, w_{1o}, w_{1b}) are linearly dependent.

When investigating Jacobians for the dependency problem outlined above it is however uncommon to encounter singular values *exactly* equal to zero; but according to the analysis above this clearly ought to be the case. The reason for this is the initialization of the units when starting up the network. If the recurrent network starts iterations at time $t = 1$ it is common practice [WZ89] to set the previous states of the hidden units as well as their derivatives to zero,

$$s_i(0) = 0 \quad , \quad \frac{\partial s_i(0)}{\partial \mathbf{w}} = 0 \; . \tag{7.12}$$

This startup procedure clearly marks an initial discontinuity in the recursive equations (7.6) and (7.7) governing the feedback network. Thus initially the partial derivatives wrt. the involved weights in the Jacobian will generally *not* be linearly dependent. But after iterating a few time steps indicating a transient, the dependency arises with increasing accuracy.

In order to illustrate this discontinuity an RNN with the same architecture as shown in Figure 7.1 was applied to the first 1000 points of the Santa Fe laser data series described in appendix A.1, and the Jacobian matrix was calculated. As described above, columns in this Jacobian corresponding to the weights (w_{11}, w_{1o}, w_{1b}) ought to be linearly dependent in the direction given by $(-w_{o1}^*, 1, -w_{ob}^*)$. This may also be expressed as

$$\frac{\partial y(t)}{\partial w_{10}} - w_{o1}^* \cdot \frac{\partial y(t)}{\partial w_{11}} - w_{ob}^* \cdot \frac{\partial y(t)}{\partial w_{1b}} = 0 \quad , \quad t = 1, \ldots, T \tag{7.13}$$

The lefthand side of Eq. (7.13) is illustrated in Figure 7.2 as time progresses from $t = 1$ onwards. Initially Eq. (7.13) does not hold due to the startup process described above, but as iterations progress the accuracy increases and after a "transient period" the expression is satisfied to machine precision, indicating linear dependency among the corresponding columns of the Jacobian.

The linear dependency described above is eliminated if we omit the feedback weights w_{io} , $i = 1, \ldots, N_h$, leading *from* the output unit back *to* the hidden units i, as the degeneracy can then no longer occur. This elimination has no influence on the modeling capabilities of the network since the remaining weights can be adjusted so that the network output remains unaffected. The weights w_{io} are thus completely redundant parameters and should be omitted.

7.2.2 Approximate column collinearity

Even though removal of the feedback weights w_{io} leading from the linear output unit back to the hidden units removes the problem of exact rank-deficiency of the Jacobian for recurrent networks, this does *not* eliminate ill-conditioning as experiments will show. The condition number of the Hessian very often grows large due to columns of the Jacobian being nearly collinear. The *degree* of collinearity between two vectors $\mathbf{x}$ and $\mathbf{y}$ may be expressed in terms of the angle θ between them, defined by

$$\cos\theta = \frac{\mathbf{x}^{\mathrm{T}}\mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} \approx 1 - \frac{\theta^2}{2} \tag{7.14}$$

where $\|\cdot\|$ is the Euclidean norm; this expression may also be interpreted as the *cross correlation coefficient* between $\mathbf{x}$ and $\mathbf{y}$. The approximation is obtained by a Taylor expansion of the cosine to second order. Naturally, if $\cos\theta = 1$ then the angle θ between the two vectors is zero and the vectors are exact multiples of each other. On the other hand, if the angle θ is "nearly" zero it is an indication of the vectors being nearly collinear; thus the smaller the θ, the closer the vectors $\mathbf{x}$ and $\mathbf{y}$ are to being collinear[2].

In [SBC93] it was shown that if the vectors $\mathbf{x}$ and $\mathbf{y}$ are taken as columns of the Jacobian $\mathbf{J}$, then the condition number of the Hessian $\mathbf{H} = \mathbf{J}^{\mathrm{T}}\mathbf{J}$ is bounded below by

$$\kappa(\mathbf{H}) \;>\; \frac{1}{\theta^2(4-\theta^2)} \cdot \left(\frac{\|\mathbf{x}\|^2}{\|\mathbf{y}\|^2} + \frac{\|\mathbf{y}\|^2}{\|\mathbf{x}\|^2} + 2 \right) \quad , \tag{7.15}$$

where $\|\cdot\|^2$ denotes the Euclidean norm squared. From this lower bound we learn that the condition number of the Hessian will grow at least inversely proportional to the square of the smallest angle θ obtained between columns of the Jacobian. The proof for this lower bound is provided in appendix D. There, it is furthermore demonstrated that a decrease of θ will lead to a decrease of only the *smallest* eigenvalue of the Hessian while the largest eigenvalue will remain unaffected.

In order to reveal what might cause near collinearity between columns of the Jacobian we need to analyze the expressions defining the partial derivatives of the model output, forming the columns of the Jacobian. Such analysis of the partial derivatives was performed for a feed-forward network with two hidden layers in [SBC93]. The analysis there lead to an enumeration of particular situations that will result in near collinearity between columns in the Jacobian and thus to ill-conditioning of the Hessian.

The focus of the present work is recurrent networks for which the expressions for the partial derivatives are somewhat involved as seen in section 4.7.2. In order to introduce

[2]This may naturally be equivalently expressed in terms of the cross correlation coefficient being close to unity.

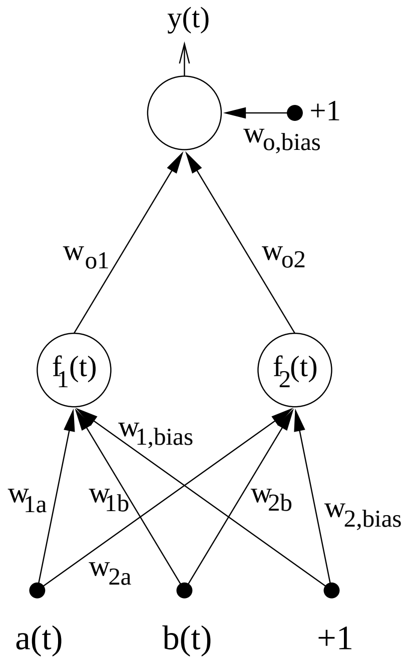


Figure 7.3: A simple feed-forward network having two external inputs and two hidden units.

the concept of partial derivative analysis we will therefore initially focus on feed-forward networks for which the partial derivative expressions are more easily comprehended, and then later relate the results to recurrent networks.

As in the previous section the issue will be addressed in terms of a small example. In particular we will consider a simple feed-forward architecture as illustrated in Figure 7.3, having two external inputs $\mathrm{a}(t)$ and $\mathrm{b}(t)$ and two hidden units. The output from this feed-forward network is calculated as

$$\begin{aligned} y(t) &= w_{o1} f_1[v_1(t)] + w_{o2} f_2[v_2(t)] + w_{o,bias} \\ &= w_{o1} f_1[w_{1a}\mathrm{a}(t) + w_{1b}\mathrm{b}(t) + w_{1,bias}] \\ &\quad + w_{o2} f_2[w_{2a}\mathrm{a}(t) + w_{2b}\mathrm{b}(t) + w_{2,bias}] + w_{o,bias} \ . \end{aligned} \tag{7.16}$$

In order to investigate the columns of the Jacobian for this network we caluclate the partial derivatives of the network output $y(t)$ as

$$\frac{\partial y(t)}{\partial w_{o1}} = f_1[v_1(t)] \tag{7.17}$$

$$\frac{\partial y(t)}{\partial w_{o2}} = f_2[v_2(t)] \tag{7.18}$$

$$\frac{\partial y(t)}{\partial w_{o,bias}} = 1 \tag{7.19}$$

$$\frac{\partial y(t)}{\partial w_{1a}} = w_{o1} f_1'[v_1(t)]\mathrm{a}(t) \tag{7.20}$$

$$\frac{\partial y(t)}{\partial w_{1b}} = w_{o1} f_1'[v_1(t)]\mathrm{b}(t) \tag{7.21}$$

$$\frac{\partial y(t)}{\partial w_{1,bias}} = w_{o1} f_1'[v_1(t)] \tag{7.22}$$

$$\frac{\partial y(t)}{\partial w_{2a}} = w_{o2} f_2'[v_2(t)]\mathrm{a}(t) \tag{7.23}$$

$$\frac{\partial y(t)}{\partial w_{2b}} = w_{o2} f_2'[v_2(t)]\mathrm{b}(t) \tag{7.24}$$

$$\frac{\partial y(t)}{\partial w_{2,bias}} = w_{o2} f_2'[v_2(t)] \tag{7.25}$$

We will now examine these derivatives in order to determine possible causes for (near) collinearity between the corresponding columns of the Jacobian. E.g., we find that if the external inputs $\mathrm{a}(t)$ and $\mathrm{b}(t)$ are linearly dependent for all t, then the columns of the Jacobian determined by Eq. (7.20) and Eq. (7.21) will be linearly dependent as well; this is also the case for the columns determined by Eq. (7.23) and Eq. (7.24). If two inputs are linearly dependent then one of them is clearly redundant and the problem can be solved by discarding one of the inputs[3].

Assuming that the external inputs are not linearly dependent there are however still situations where conditions *internal* to the network will lead to an almost rank-deficient and thus ill-conditioned Jacobian. In order to describe these situations we define a vector containing the outputs from the first hidden unit as

$$\mathbf{f}_1 = (f_1[v_1(1)], f_1[v_1(2)], \ldots, f_1[v_1(T)])^{\mathrm{T}} \tag{7.26}$$

where T is the number of training examples and define vectors $\mathbf{f}_2$, $\mathbf{f}_1'$ and $\mathbf{f}_2'$ likewise; here, prime denotes the derivative. Inspecting Eqs. (7.17–7.25) we may now identify the following situations which will lead to an almost rank-deficient Jacobian [SBC93]:

1. If the vectors $\mathbf{f}_1$ or $\mathbf{f}_2$ are nearly multiples of $(1, 1, \ldots, 1)^{\mathrm{T}}$ then columns in the Jacobian determined by Eq. (7.17) or Eq. (7.18) will be nearly multiples of Eq. (7.19). This situation arises e.g., if a hidden unit output is constantly saturated at the *same* value. This is likely to be the case if the magnitude of the weights in the network is large.

2. If the vectors $\mathbf{f}_1'$ and $\mathbf{f}_2'$ are nearly multiples of each other, then the pairs of columns in the Jacobian determined by Eq. (7.20) and Eq. (7.23), Eq. (7.21) and Eq. (7.24) as well as Eq. (7.22) and Eq. (7.25) will be nearly multiples of each other. This situation will arise if e.g., the hidden unit outputs are constantly saturated; they may however toggle between the two levels of saturation.

3. If the vectors $\mathbf{f}_1$ and $\mathbf{f}_2$ are nearly multiples of *each other*, then the columns in the Jacobian determined by Eq. (7.17) and Eq. (7.18) will be nearly multiples of each other. This situation will occur not only if the two hidden units both saturate, but also if they *specialize to the same features in the data.*

[3]Naturally, for a *recurrent* network working from only *one* external input this is not a relevant issue.

These situations which will lead to ill-conditioning are all related to intrinsic problems of the neural network architecture. All items in the list refer to situations which may be traced back to the modular structure of neural networks as they all describe situations where the outputs of different modules in terms of hidden units become highly correlated. Furthermore, all of the situations described may be caused by the saturating characteristic of the sigmoid functions usually applied as activation function for the hidden units.

We will now proceed to consider recurrent networks. For recurrent networks we may write the partial derivatives explicitly for a simple network as well, and examine them in the same way as was done for the feed-forward network above. Such analysis is however not as straightforward and intuitive to perform as in the feed-forward case, since most partial derivatives $\partial y(t)/\partial w_i$ involve the partial derivatives of *all* the N_h hidden units in the network wrt. ∂w_i, $\partial s_k(t)/\partial w_i$, $k = 1, \ldots, N_h$; refer to Eqs. (4.35–4.37). Furthermore, the partial derivatives depend on derivatives from previous time steps, complicating an analysis like for the feed-forward network above even further. It is however *not* necessary to perform a similar analysis as it is fairly straightforward to convince oneself that ill-conditioning of the Jacobian and thereby of the Hessian matrix for a recurrent network will result from the exact same situations as listed above for a feed-forward network. The reason for this is that all of the situations listed will lead to near redundancy in the parametrization of *any* network, in the same way as was found for the output unit feedback weights in section 7.2.1. E.g., if a hidden unit of a recurrent network is permanently saturated and constant ($\equiv \pm 1$) it will act as an extra bias input to the units with which it is connected, making the "original" bias weights redundant. This is so since addition of an arbitrary constant c to the weight leading from the saturated hidden unit to the other units in the network and addition of $-c$ to the "original" bias weights will leave the total bias to the hidden units unchanged. Likewise, if the outputs from two or more hidden units in a recurrent network are constantly proportional, the weights leading *from* these units *to* any unit (including the two proportional units them selves) could be combined into a *single* weight, thus also leading to redundancy in the parametrization. Hence, the situations causing ill-conditioning for a feed-forward network will cause ill-conditioning for a recurrent network as well as the situations are equivalent to redundant parametrization for both network architectures.

The effects of e.g., two proportional hidden unit outputs are generally much more severe for recurrent networks than for feed-forward networks due to the higher connectivity between the units of a recurrent network. If the outputs from two hidden units become proportional in a feed-forward network with one hidden layer, the only weights affected in terms of redundancy are the two which connect the hidden units with the output unit. In a fully connected recurrent network however, the two proportional hidden unit outputs are connected to *all* units in the network, and the weights affected are therefore every *pair* of weights leading *from* the two proportional units *to* every single one of the N_u units in the network. This way $2N_u$ weights will be affected, which should be contrasted to the mere two weights affected for the feed-forward network.

This finding is illustrated by a small example. Consider the feed-forward network displayed in Figure 7.4 having three hidden units. If the outputs from hidden units two and three become proportional then redundancy will result between the two weights w_{o2} and w_{o3} as indicated by dashed connections. Now consider the recurrent network displayed in Figure 7.5 which also has three hidden units thus $N_u = 4$ units in total. If the outputs from hidden units two and three become proportional then not only will the output weights

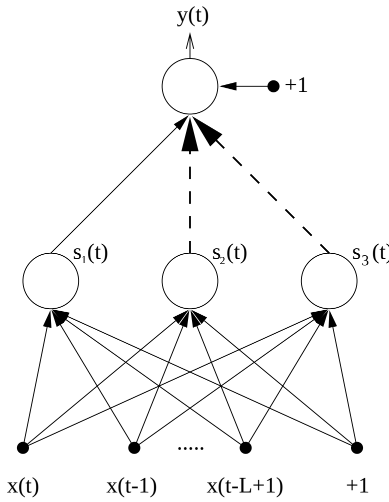


Figure 7.4: Feed-forward network having three hidden units. In case of the outputs from hidden units two and three becoming proportional the weights affected in terms of redundancy will be the ones indicated by dashed connections, two weights in total.

w_{o2} and w_{o3} be affected in terms of redundancy, the weight pairs (w_{12}, w_{13}), (w_{22}, w_{23}) and (w_{32}, w_{33}) will be affected as well, leading to eight weights affected in total.

The parameters between which redundancy occurs define directions in parameter space in which the cost function is approximately constant, as the parameters may be manipulated arbitrarily along these directions almost without affecting the network output, in line with the description of the redundant output feedback weights in section 7.2.1. The almost constant cost function causes negligible derivatives of any order in the directions involved, leading to an ill-conditioned Hessian.

The situations leading to ill-conditioning listed on page 77 and applying to both feed-forward as well as recurrent networks might seem rather speculative in nature as they

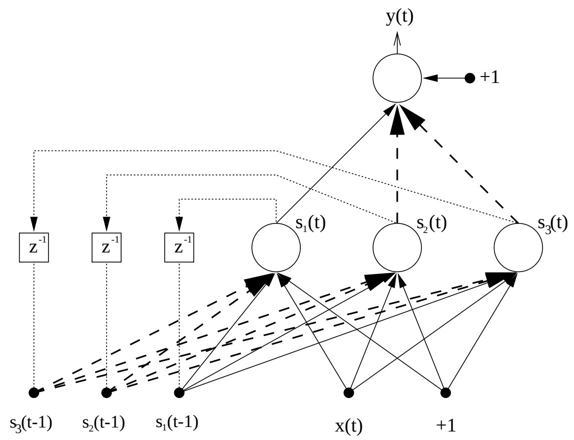


Figure 7.5: Recurrent network having three hidden units. In case of the outputs from hidden units two and three becoming proportional the weights affected in terms of redundancy will be the ones indicated by dashed connections, eight weights in total.

involve a lot of "if's". Even so, the situations have nevertheless been found to occur rather frequently in practice. Numerous examples of this were provided in [SBC93] for feed-forward networks applied to classification problems, and the same phenomena has been observed frequently during the present work for both feed-forward and recurrent networks applied to time series prediction problems. In particular it seems that item three in the list above, i.e., that two or more hidden units specialize to the same features in the training data making their outputs highly correlated, occurs very often during training. High correlation between the hidden unit outputs was also found in e.g., [WR91] where the covariance matrix of the hidden unit outputs from a feed-forward network applied to time series prediction on the sunspot problem was studied in terms of its eigenvalues. An experiment which illustrates the occurrence of high correlation between two hidden unit outputs and the problems that arise due to ill-conditioning will be described in section 8.1.

7.2.3 Column length disparity

So far the analysis of the Jacobian matrix has been focused around situations which will cause ill-conditioning of the Hessian due to increasing collinearity between columns of the Jacobian; this collinearity will lead to a decrease of the *smallest* eigenvalue as demonstrated in appendix D. However, from the lower bound on the condition number $\kappa(\mathbf{H})$ provided by Eq. (7.15) we learn that an increase of the condition number may result not only from a decreasing angle θ between columns of the Jacobian but also from an increasing disparity between the lengths of columns of the Jacobian. In appendix D it is shown that an increase of the largest column length will lead to an increase of the *largest* eigenvalue of the Hessian and that a decrease of the smallest column length similary will lead to a decrease of the smallest eigenvalue.

Based on these observations it therefore seems appropriate to identify situations which will lead to a large column length disparity in the Jacobian. In order to initiate such analysis we once more direct the attention towards the simple feed-forward network displayed in Figure 7.3 whose partial derivatives are given by Eqs. (7.17–7.25). By inspecting these partial derivatives we learn that the derivative wrt. the output bias weight $w_{o,bias}$ is constantly equal to one; the length of the corresponding column of the Jacobian will therefore be constant, indicating a bound to which other column lengths should be compared.

Inspecting the remaining derivatives in Eqs. (7.17–7.25) we note that if the output weights w_{o1} and w_{o2} grow large then the lengths of *some* columns of the Jacobian may grow large, namely the columns corresponding to the partial derivatives wrt. the weights leading *to* the hidden units Eqs. (7.20–7.25), as these derivatives scale with the output weights. Thus, an increasing magnitude of the output weights of a feed-forward network may generally lead to an increase in the magnitude of the partial derivatives wrt. weights leading *to* the hidden units and thus to an increase of the *largest* eigenvalue of the Hessian.

For recurrent networks, the partial derivatives of the model output $\partial y(t)/\partial w_{pq}$ are dominated by sums of weighted terms of the form

$$\frac{\partial s_k(t)}{\partial w_{pq}} \propto \sum_{j=1}^{N_u} w_{kj} \frac{\partial s_j(t-1)}{\partial w_{pq}} \tag{7.27}$$

as seen from Eqs. (4.35–4.37); N_u denotes the number of units in the network and w_{jk} denotes the weights connecting the units. From this expression we learn that if any of the weigths connecting units in a recurrent network grow large, elements of the Jacobian are likely to grow large as well.

By comparing the partial derivatives for a feedforward network Eqs. (7.17–7.25) to the partial derivatives for a recurrent network Eqs. (4.35–4.37) we learn that the problem of growing elements in the Jacobian due to growing weights is potentially much more severe for recurrent networks than for feed-forward networks. Not only are *more* columns of the Jacobian of a recurrent network likely to be affected by an increasing magnitude of the weights, the individual elements are also likely to grow much *larger* in magnitude than for a feed-forward network with weights of comparable magnitude. This is due to the structure of the partial derivatives which are calculated from *sums* of weighted hidden unit derivatives Eq. (4.35), each of which are themselves computed from *sums* of weighted hidden unit derivatives at the previous time step Eq. (4.37) and so on. Thus the recursive calculation of derivatives is likely to amplify the effects of large magnitude weights, compared to feed-forward networks. This will be demonstrated by a simple example in section 8.2.

The recursive structure of the RNN derivatives results from the high connectivity between the units in a recurrent network. From the analyses of Jacobian column collinearity and column length disparity above we learned that this network structure may lead to numerical problems which are potentially more severe than for a comparable feed-forward network. The numerical problems may result from both a decrease of the *smallest* eigenvalue as well as from an increase of the *largest* eigenvalue.

If the Hessian starts to become ill-conditioned due to *small* eigenvalues, the conditioning problem will soon become worse due to an increase in the values of the *largest* eigenvalues, especially if training using the Gauss-Newton method. The small eigenvalues indicate directions in parameter space in which the cost function is almost constant; calculating the Gauss-Newton search direction will lead to large components in these approximately constant directions. The domination of the search direction by these directions in parameter space will lead to an unrestrained growth in the magnitude of the affected weights. This growth will lead to an increased magnitude of the elements in some columns of the Jacobian and thereby to large eigenvalues of the Hessian, making the learning problem even more ill-conditioned. This effect will be illustrated experimentally in chapter 8.

7.3 Regularization

A traditional method for handling the problem of ill-conditioning is by *regularizing* the cost function [DS83, Hay94]. A simple yet highly effective regularization can be obtained by augmenting the cost function by a simple quadratic weight decay [HKP91, Hay94],

$$C(\mathbf{w}) = E(\mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \tag{7.28}$$

where α is a small positive constant. Simple weight decay is often primarily considered as a means for avoiding overfitting as it constrains the parameters and thus reduces the degrees of freedom as was described in chapter 6. Weight decay should however also (and for recurrent networks primarily) be considered for its numerical effects. The immediate effect of the weight decay is addition of α to the diagonal of the Hessian which puts a lower bound on the smallest eigenvalues when working from the positive semi-definite Gauss-Newton Hessian, since it is easy to show that

$$\lambda(\mathbf{H} + \alpha\mathbf{I}) = \lambda(\mathbf{H}) + \alpha \tag{7.29}$$

where $\lambda(\mathbf{H})$ is a vector with the eigenvalues of $\mathbf{H}$ and $\mathbf{I}$ is the identity matrix. The condition number of the regularized Hessian is now calculated as

$$\kappa(\mathbf{H} + \alpha\mathbf{I}) = \frac{\lambda_{\max} + \alpha}{\lambda_{\min} + \alpha} \approx \frac{\lambda_{\max}}{\alpha} \tag{7.30}$$

Another effect resulting from regularization of the cost function by a simple weight decay term is the limit imposed on the growth of the weights. Weight decay will bound the magnitude of the weights and thus indirectly bound the largest singular values of the Jacobian and thereby the largest eigenvalues of the Hessian. This is obtained by a reduction of the Jacobian column lengths according to the description in section 7.2.3. Consequently, a simple quadratic weight decay will bound the eigenvalues from both below and from above, leading to a more manageable condition number and thus highly improved training as will be illustrated in chapter 8.

An alternative way in which to handle problems caused by ill-conditioning during training by the Gauss-Newton method is by solving the system of linear equations Eq. (4.19) using the *pseudoinverse* [PFTV92] of the Hessian $\mathbf{H}$. The pseudoinverse of a matrix $\mathbf{A}$ is computed as the inverse of the righthand side of Eq. (7.1),

$$\mathbf{A}^{-1} = \mathbf{V}\mathbf{D}^{+}\mathbf{U}^{\mathrm{T}} \tag{7.31}$$

where $\mathbf{D}^{+}$ denotes the inverse of the diagonal matrix $\mathbf{D}$ in which some of the elements $1/\sigma_i$ are set to zero. If we want to eliminate all singular values/eigenvalues of the Hessian which are smaller than a threshold of e.g., 10^{-3}, then we set to zero all elements $1/\sigma_i$ in $\mathbf{D}^{+}$ for which $\sigma_i < 10^{-3}$ prior to the computation of the inverse from Eq. (7.31). Comparing to weight decay we note that only the directions of the smallest eigenvalues are affected by application of the pseudoinverse whereas weight decay affects all eigenvalues. Furthermore, the gradient is not modified by the pseudoinverse as is the case when using weight decay. Consequently, application of the pseudoinverse will not necessarily bound the magnitude of the weights during training. The weights may thus still grow large, leading to ill-conditioning due to *large* eigenvalues. In this situation the singular value/eigenvalue threshold must be increased in order to handle the resulting numerical problems. The pseudoinverse has only been briefly attempted during this work, verifying these observations.

The constraints put on the weights by the weight decay regularization has a smoothing effect on the cost function by making it more "regular", hence the name. This smoothing effect was clearly illustrated for a recurrent network in [PH95] included in appendix G. Here, a recurrent network was trained on the sunspot problem using a small weight decay parameter, and a slice through the cost function revealed a high complexity of the cost surface. The network was then retrained using a larger weight decay parameter, and the same slice through the retrained cost function revealed a much smoother cost surface.

Regularization of recurrent networks was also treated in [WM96b] presented at NIPS*95 and elaborated upon in [WM96a], where the use of a "smoothing" regularizer of a somewhat different form than in Eq. (7.28) was suggested. The motivation for the regularizer was however to improve the generalization ability, the numerical aspects of the regularizer were not considered. In both references it was stated that "*To our knowledge, recurrent learning with regularization has not been reported before.*" However, training recurrent networks using regularization *had* been reported before, as [PH95] was presented at NIPS*94 the previous year. The statement however supports the impression that the importance of regularization in order to prevent numerical problems, especially when training recurrent networks, is not generally recognized in the neural network community.

In addition to the quadratic weight decay, several other regularization terms have been suggested in the literature; see e.g., [HKP91, Hay94] for an introduction and references and e.g., [WM96a] for further references. These have however not been considered in this work due to a higher complexity which is seemingly not justified by an increased ability to handle the numerical problems described in this chapter, compared to the simple quadratic weight decay.

Chapter 8

Illustration of ill-conditioning

Whereas the previous chapter described hypothetically how ill-conditioning might arise in recurrent networks it is in this chapter demonstrated how ill-conditioning typically manifests itself in practice. Section 8.1 describes an example of recurrent network training in terms of which many of the aspects described in chapter 7 are extensively illustrated. Training is performed by the damped Gauss-Newton method as well as by gradient descent. Section 8.2 provides a quantitative comparison between the conditioning problems for comparable feed-forward and recurrent networks having *random* weights which illustrates the increased numerical problems for the recurrent structure. The chapter is concluded by a small example which serves to illustrate the significance of the often neglected second derivative term of the Hessian matrix.

8.1 Recurrent network training: An example

It will here be illustrated how ill-conditioning usually occurs in practice during training of recurrent networks. The illustration will be made in terms of a recurrent network applied to a time series prediction problem. The series to be modeled in this section is the laser data from the Santa Fe time series prediction competition described in appendix A.1. The first 1000 data points from the series were scaled to zero mean and unit variance and used for training, and the following 100 points scaled accordingly were used as a separate test set. The test set is however of minor importance here as the focus is on aspects of training.

The recurrent network used for the experiments was of the type described in section 3.3.1. For simplicity and ease of presentation a small network having only three hidden units was chosen; the small network does however not make the results less general as similar results may be obtained for more complex network structures. Furthermore the network received only one external input and *no* feed-back was present from the linear output unit back to the hidden units, as was found appropriate in section 7.2.1. The resulting network architecture thus involved 19 parameters in total and is illustrated in Figure 8.1.

The following describes how the recurrent network was trained by use of both the damped Gauss-Newton method as well as by gradient descent and illustrates numerous aspects of the network during training. Training was performed both without and with a weight decay term in order to illustrate the effects of applying this regularizer.

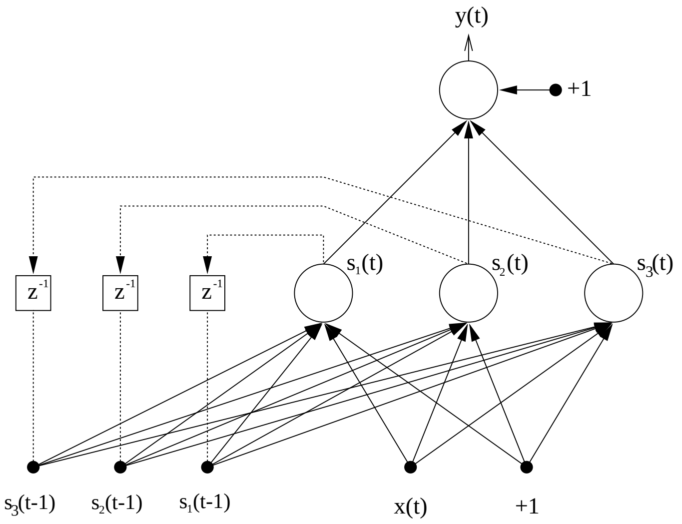


Figure 8.1: Architecture of the recurrent network used for experiments in this section.

8.1.1 Training by damped Gauss-Newton without weight decay

At first the network was trained by the damped Gauss-Newton method without weight decay. However, application of the second-order method was preceeded by five iterations of gradient descent in order to improve the accuracy of the Gauss-Newton approximation as suggested in section 4.4.2. In the left panel of Figure 8.2 is shown the *training curves*, i.e., the evolution of the Normalized Mean Squared Error (NMSE, refer to appendix A) for both the training and test set as iterations progress. Considering these errors alone it seems that training is converging towards a solution at a local minimum $\mathbf{w}^*$, as the errors are almost constant from iteration to iteration. This is however *not* the case as may be learned from the evolution of the weights shown in the right panel of Figure 8.2. Several of the weights seem to have started an unrestrained growth in magnitude. Furthermore, examining the evolution of the Euclidean norm of the gradient shown in the left panel of Figure 8.3 seems to confirm that the weight iterates are indeed not approaching a local

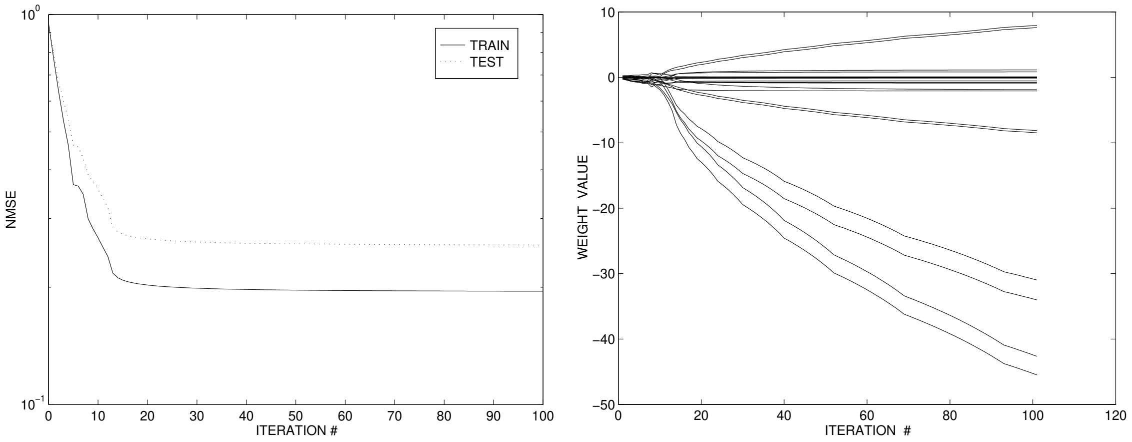


Figure 8.2: Training by damped Gauss-Newton, $\alpha = 0.0$. Left panel: Evolution of training and test errors. Right panel: Evolution of the weight values.

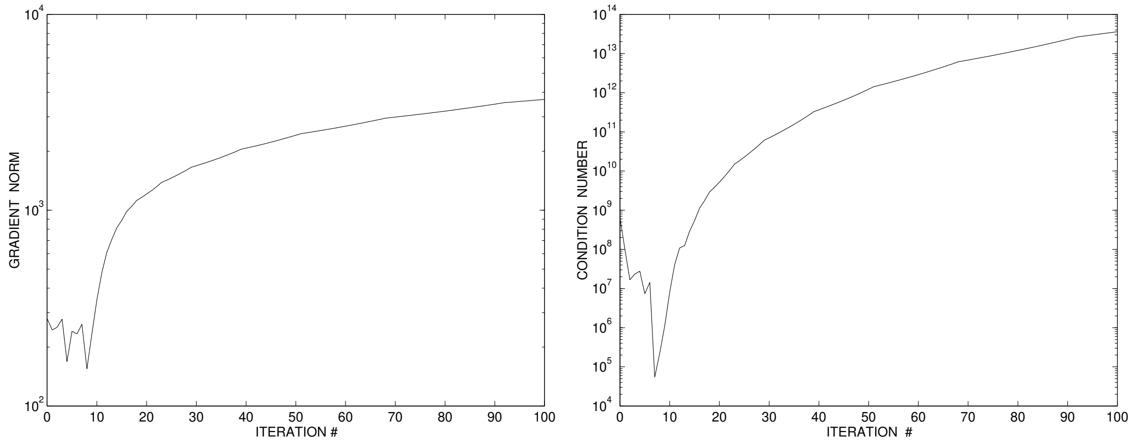


Figure 8.3: Training by damped Gauss-Newton, $\alpha = 0.0$. Left panel: Evolution of the gradient norm, $\|\mathbf{g}\|_2$. Right panel: Evolution of the condition number of the Hessian, $\kappa(\mathbf{H})$.

minimum, as the norm is growing very large. Thus, taking the weights and the gradient norm into consideration it seems like the training algorithm has failed.

The condition number of the Hessian during training is shown in the right panel of Figure 8.3, it is seen to grow enormously shortly after the damped Gauss-Newton method is applied. This reveals the reason for the failure of the training algorithm, as the Hessian is becoming extremely ill-conditioned. In order to investigate the possible cause for the ill-conditioning the outputs from the hidden units were examined. The left panel of Figure 8.4 shows the outputs from the three hidden units of the recurrent network after training iteration 100 as time progresses. It seems that the two low-amplitude outputs are highly anti-correlated. This observation is confirmed by the right panel of Figure 8.4 which illustrates the values of the cross correlation coefficients *defined* by Eq. (7.14) between the hidden unit outputs. The magnitude of the correlation coefficient between hidden

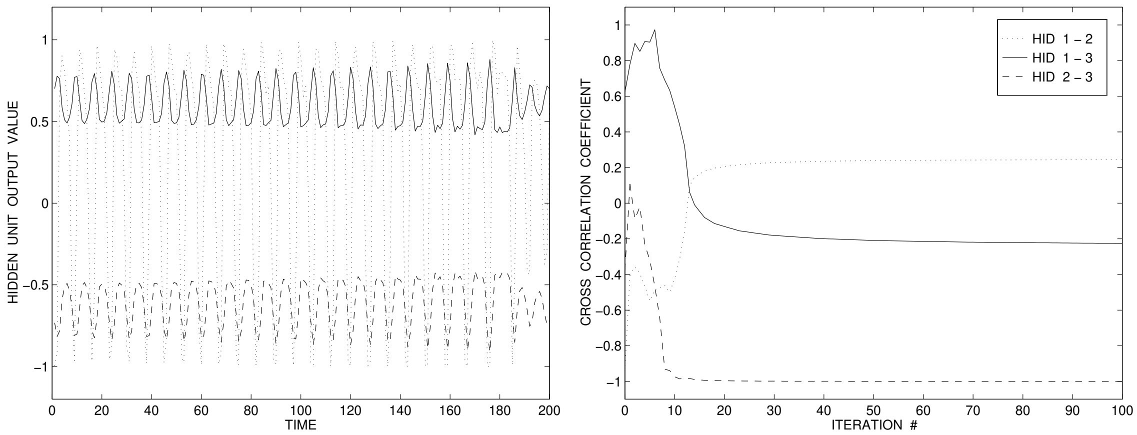


Figure 8.4: Training by damped Gauss-Newton, $\alpha = 0.0$. Left panel: Outputs from the hidden units after training iteration 100. Right panel: Evolution of the cross correlation coefficients between the hidden unit outputs (see text).

units two and three is close to unity (0.9998), indicating that the outputs from these two hidden units are proportional; this may also be interpreted as the vectors containing the hidden unit outputs at each time step as elements being collinear. The cause of the ill-conditioning thus seems to be that two of the hidden units have specialized on the same features in the data set, corresponding to situation 3 in the list on page 77 of possible sources of ill-conditioning.

The weights that grow in magnitude in the right panel of Figure 8.2 are the pairs of weights between which redundancy occurs, i.e., the weights leading *from* units two and three *to* every unit in the network, including the output unit. This is in line with the small example given in Figure 7.5 on page 79. Note that the error in the left panel of Figure 8.2 and thus the network output is unaffected by the growing weights since the effects of the weight changes cancel out due to the dependency between the hidden unit outputs. Note also that each pair of weights have values of the same sign; as the proportional outputs from the two hidden units are negatively correlated, the network output is left unchanged if the *discrepancies* between the weight values remain constant as is the case on the left panel of Figure 8.2.

The proportional hidden unit outputs lead to directions in parameter space in which the quadratic cost function is constant as was described on page 79. This is illustrated in Figure 8.5 which displays the surface of the cost function after iteration 10 when the

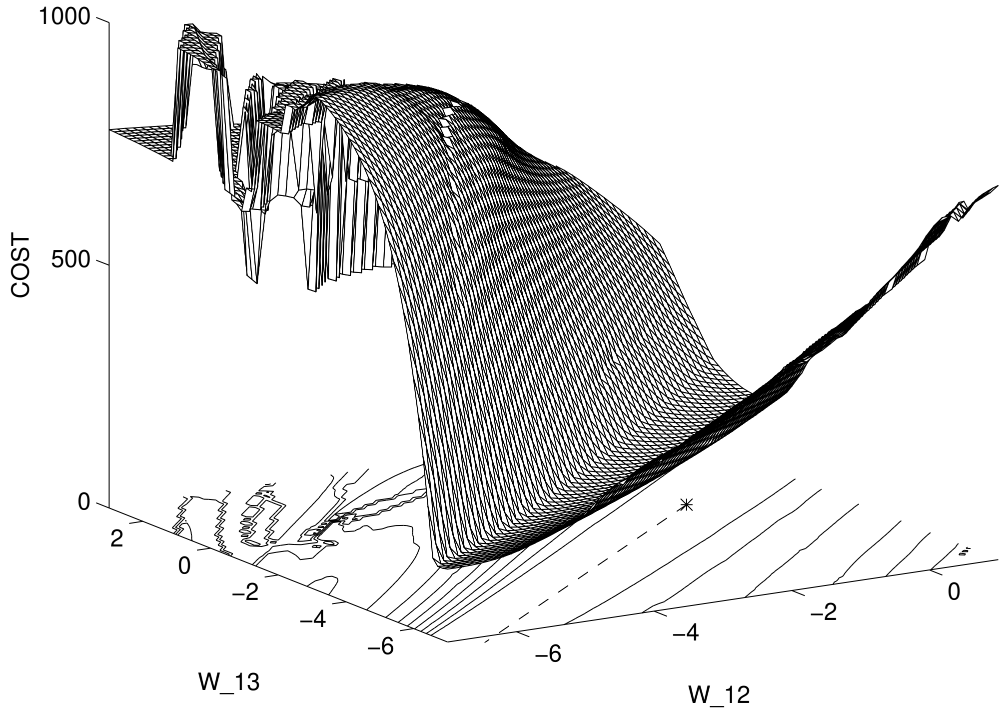


Figure 8.5: Training by damped Gauss-Newton, $\alpha = 0.0$. Cost function surface after 10 iterations illustrated as a function of the weights w_{12} and w_{13} leading *from* hidden units two and three *to* hidden unit one. The asterisk on the underlying contour plot denotes the current iteration point and the dashed line emanating from this point indicates the direction of the Gauss-Newton step. Note the "rain gutter"-like structure of the cost surface indicating a direction in which the cost is (practically) constant.

weights w_{12} and w_{13} leading *from* hidden units two and three *to* hidden unit one, a pair of weights affected by redundancy, are varied. The cost surface clearly reveals a "rain gutter"-like structure in the direction of constant cost. The parameter estimate after iteration 10 is indicated by an asterisk and is seen to be positioned approximately at the bottom of the "rain gutter." Forming the second-order expansion around this point leads to large components of the Gauss-Newton search direction in the constant direction as indicated by the dashed line.

The rapid increase in both weight magnitudes as well as the condition number displayed in the right panel of Figure 8.3 is seen to occur shortly after the initiation of the damped Gauss-Newton method. The second-order method quickly "discovers" the increasing dependency between the hidden unit outputs in terms of the directions of constant cost. The effects of the large components in the search direction are however handled by the line search which returns very small step sizes, as indicated by the smooth increase in the weight magnitudes. The explanation for the small step sizes is that the search direction does not lead *precisely* towards the bottom of the "rain gutter," partly due to imprecision caused by ill-conditioning when computing the Gauss-Newton step; refer to appendix E.

The drastic increase of the gradient norm displayed in the left panel of Figure 8.3 may also be described in terms of the "rain gutter"-like structure of the cost surface. It turns out that the parameter estimates are not located *exactly* at the bottom but rather a little bit "up the sides." As iterations progress the parameter estimate moves slightly down towards the bottom, however never quite reaching it. As the weight magnitudes increase it turns out that the sides of the "rain gutter" become steeper, leading to the increase of the gradient norm.

In Figure 8.6 is shown the eigenvalues of the Hessian after iterations 7, 20 and 100. At each of the iterations it is seen that the condition number results from both very small as well as very large eigenvalues and we note that as iterations progress the eigenvalues extend both upwards and downwards. This is consistent with the description given in sections 7.2.2 and 7.2.3 as well as the mathematical derivation given in appendix D.

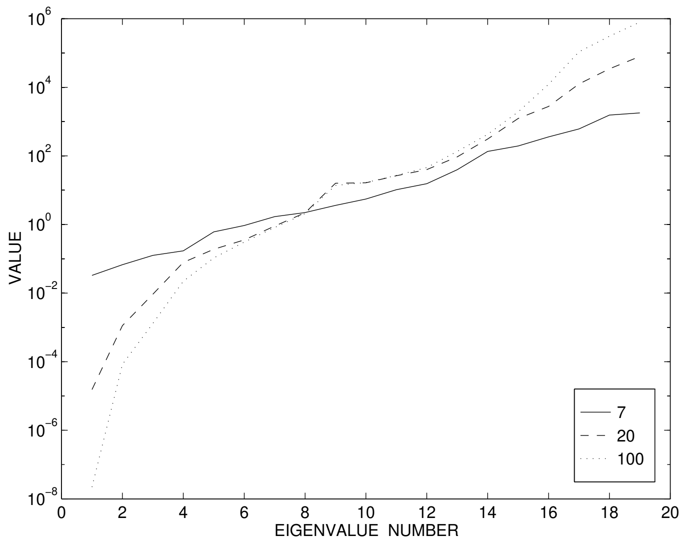


Figure 8.6: Training by damped Gauss-Newton, $\alpha = 0.0$. Eigenvalues of the Hessian at iterations 7, 20 and 100.

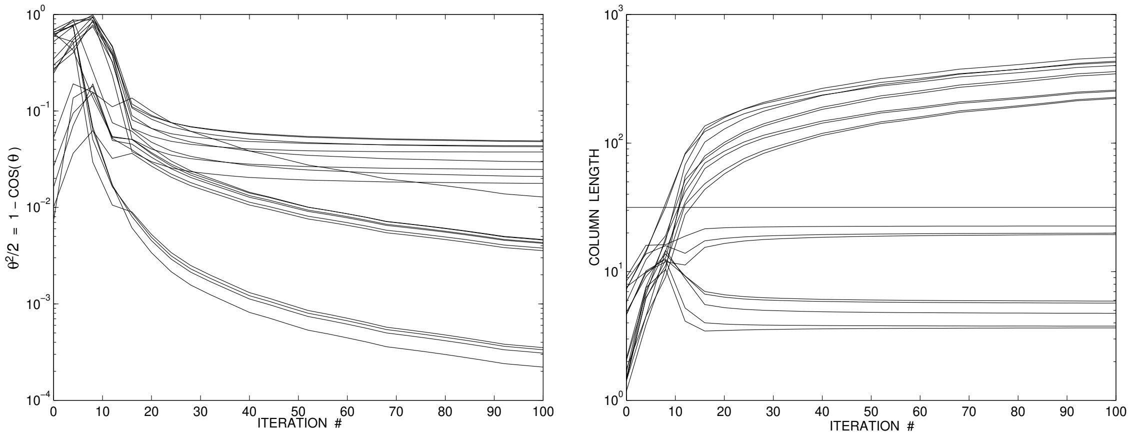


Figure 8.7: Training by damped Gauss-Newton, $\alpha = 0.0$. Left panel: Measuring the collinearity between the 20 most collinear column pairs of the Jacobian matrix. Right panel: Measuring the length of the columns of the Jacobian.

The decrease of the smallest eigenvalues is due to increasing collinearity between columns of the Jacobian, i.e., a decreasing angle between the corresponding column vectors. This is verified in the left panel of Figure 8.7 which illustrates half the square of the angle θ between the 20 *most* collinear pairs of columns of the Jacobian matrix as iterations progress. The values of $\theta^2/2$ were obtained as one minus the absolute value of the fraction given in Eq. (7.14). Several of the values are seen to decrease as iterations progress leading to a decrease of the smallest eigenvalues as described in appendix D.

The increase of the largest eigenvalues is due to an increasing disparity between the lengths of some columns of the Jacobian matrix as described in section 7.2.3. This is verified in the right panel of Figure 8.7 which illustrates the lengths of the columns of the Jacobian as iterations progress. Note the horizontal line which denotes the constant length of the derivatives wrt. the output unit bias. The lengths of several of the columns are seen to grow as iterations progress, leading to an increase of the largest eigenvalues; the increasing magnitude of some elements of the Jacobian is due to the increasing weight magnitudes as was also explained in section 7.2.3.

8.1.2 Training by damped Gauss-Newton using a small weight decay

The training of the recurrent network was then repeated using the exact same initial weights and the same training approach, but now with a simple weight decay regularization term added to the cost function as in Eq. (7.28), using a "small" weight decay $\alpha = 10^{-3}$. In the left panel of Figure 8.8 is shown the resulting evolution of the errors. The positive effect of the regularization is immediately evident, as the final errors are several orders of magnitude below the levels shown in Figure 8.2 obtained without regularization. In the right panel of Figure 8.8 we see that the regularization term limits the growth of the weights compared to Figure 8.2.

The limitation imposed on the weights by the weight decay leads to more well-defined local minima of the cost function, as the weights now cannot continue to grow indefinitely without significantly influencing the cost function at some point. The damped Gauss-Newton method is now able to locate a local minimum of the cost function as seen from the left panel of Figure 8.9. Here we see the norm of the gradient which suddenly de-

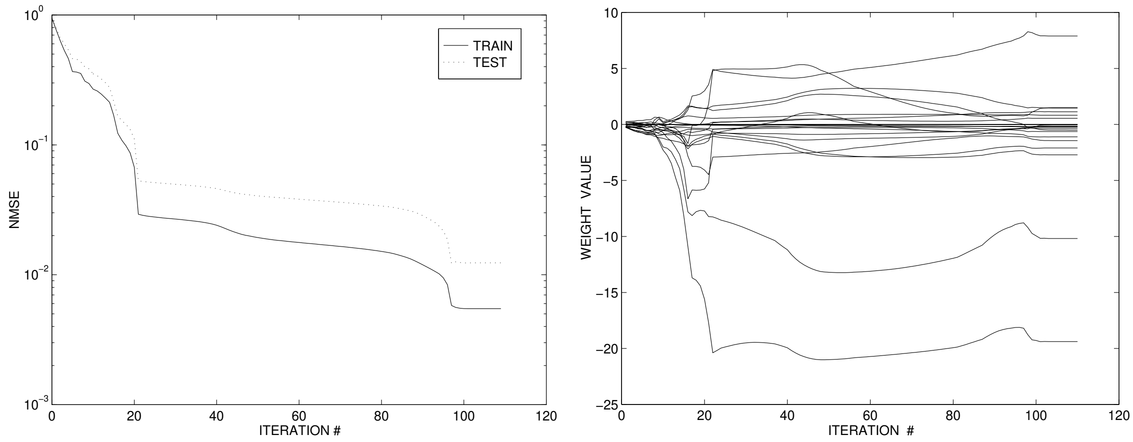


Figure 8.8: Training by damped Gauss-Newton, $\alpha = 10^{-3}$. Left panel: Evolution of training and test errors. Right panel: Evolution of the weight values.

creases dramatically as iterations enter the neighbourhood of a local minimum. Iterations terminate after satisfying the stopping criterion set to $\|\mathbf{g}\|_2 < 10^{-4}$. Note that the weights change very little during the last 8–9 iterations as the damped Gauss-Newton method performs the last "fine tuning" of the weights as it approaches the local minimum.

From the right panel of Figure 8.9, showing the values of the cross correlation coefficients between the hidden unit outputs, we see that hidden units two and three still specialize on the same features in the data set as their outputs are highly anti-correlated. At iteration 109 the magnitude of the correlation coefficient is 0.9972, approximately equal to the level reached without regularization; thus, the network is still prone to ill-conditioning. The condition number of the Hessian, shown in the left panel of Figure 8.10, grows to around 10^8 in magnitude where it stabilizes due to the effects of the weight decay. Despite the still large condition number, the damped Gauss-Newton manages to locate a local minimum of the cost function. Experience shows that when using the damped Gauss-

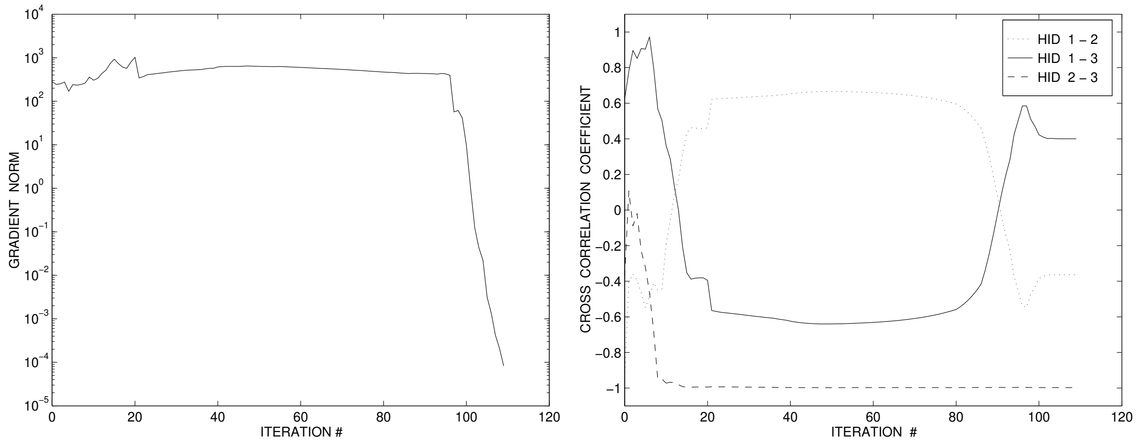


Figure 8.9: Training by damped Gauss-Newton, $\alpha = 10^{-3}$. Left panel: Evolution of the gradient norm, $\|\mathbf{g}\|_2$. Right panel: Evolution of the cross correlation coefficients between the hidden unit outputs.

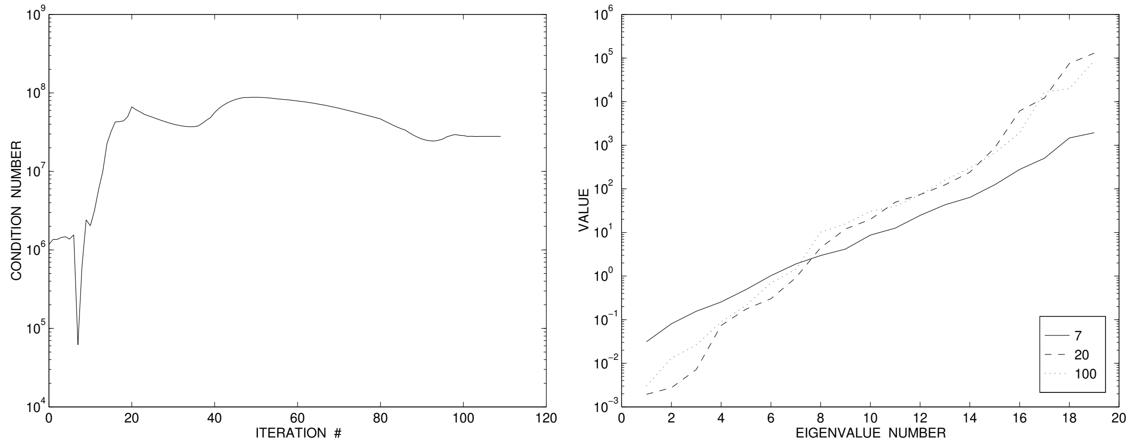


Figure 8.10: Training by damped Gauss-Newton, $\alpha = 10^{-3}$. Left panel: Evolution of the condition number of the Hessian, $\kappa(\mathbf{H})$. Right panel: Eigenvalues of the Hessian at iterations 7, 20 and 100.

Newton method, successful training to a local minimum is generally obtained for condition numbers up to about 10^8 in magnitude which is in line with the "rule of thumb" described in section 7.1. The exact bound is however problem dependent and may further depend on the decomposition algorithm used when solving for the search direction Eq. (4.19); in this work was used the fast and stable Cholesky factorization [DS83, PFTV92] adopted from [PFTV92].

From the right panel of Figure 8.10 it is seen that the reduction in the condition number is primarily obtained from the lower bound of 10^{-3} imposed on the smallest eigenvalues by the weight decay. Comparing to the eigenvalues in Figure 8.6, this is seen to have reduced the condition number by five orders of magnitude. The constraints put on the magnitude of the weights by the weight decay only lead to a comparably small decrease of the largest eigenvalues, about one order of magnitude. This is due to the still fairly large weight magnitudes as seen in Figure 8.8.

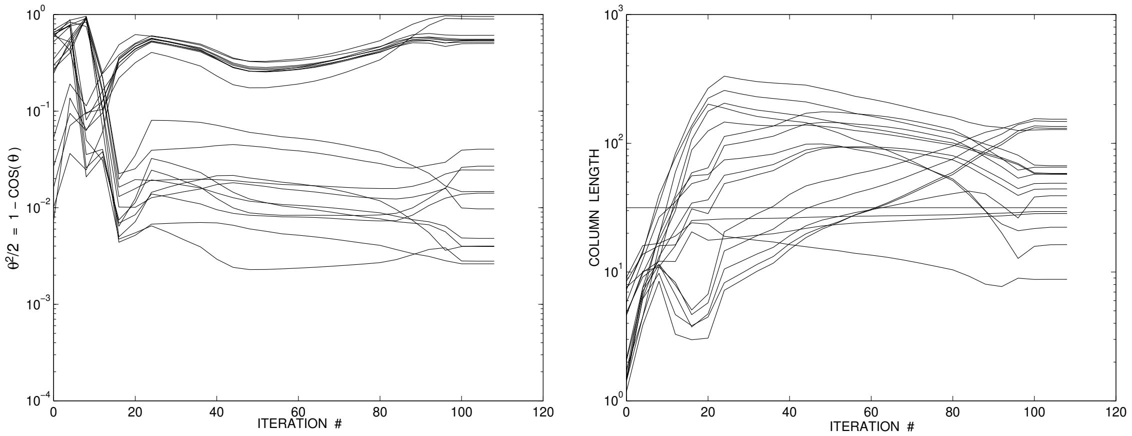


Figure 8.11: Training by damped Gauss-Newton, $\alpha = 10^{-3}$. Left panel: Measuring the collinearity between the 20 most collinear column pairs of the Jacobian matrix. Right panel: Measuring the length of the columns of the Jacobian.

The left panel of Figure 8.11 illustrates half the squared angle $\theta^2/2$ between the *same* columns of the Jacobian as was displayed in Figure 8.7. The reduced weight magnitudes due to the weight decay is seen to reduce the collinearity between the Jacobain columns as the smallest angles are somewhat larger than in Figure 8.7. However, the resulting increase of the smallest eigenvalues is probably dominated by the direct effect of the weight decay parameter being added to the diagonal of the Hessian.

In the right panel of Figure 8.11 is illustrated the evolution of the lengths of the columns of the Jacobian as iterations progress. The difference between the longest and the shortest columns is seen to be diminished compared to the equivalent illustration in Figure 8.7 when training without regularization. This diminution explains the reduction of the largest eigenvalues as seen in the right panel of Figure 8.10.

8.1.3 Training by damped Gauss-Newton using a larger weight decay

Naturally, the training problem will become more well-conditioned if the weight decay is increased even further. The weight decay was increased to $\alpha = 10^{-1}$, and training was started from the same initial weights as in the previous two examples. In the upper left panel of Figure 8.12 is shown the evolution of the errors. The level of the errors obtained

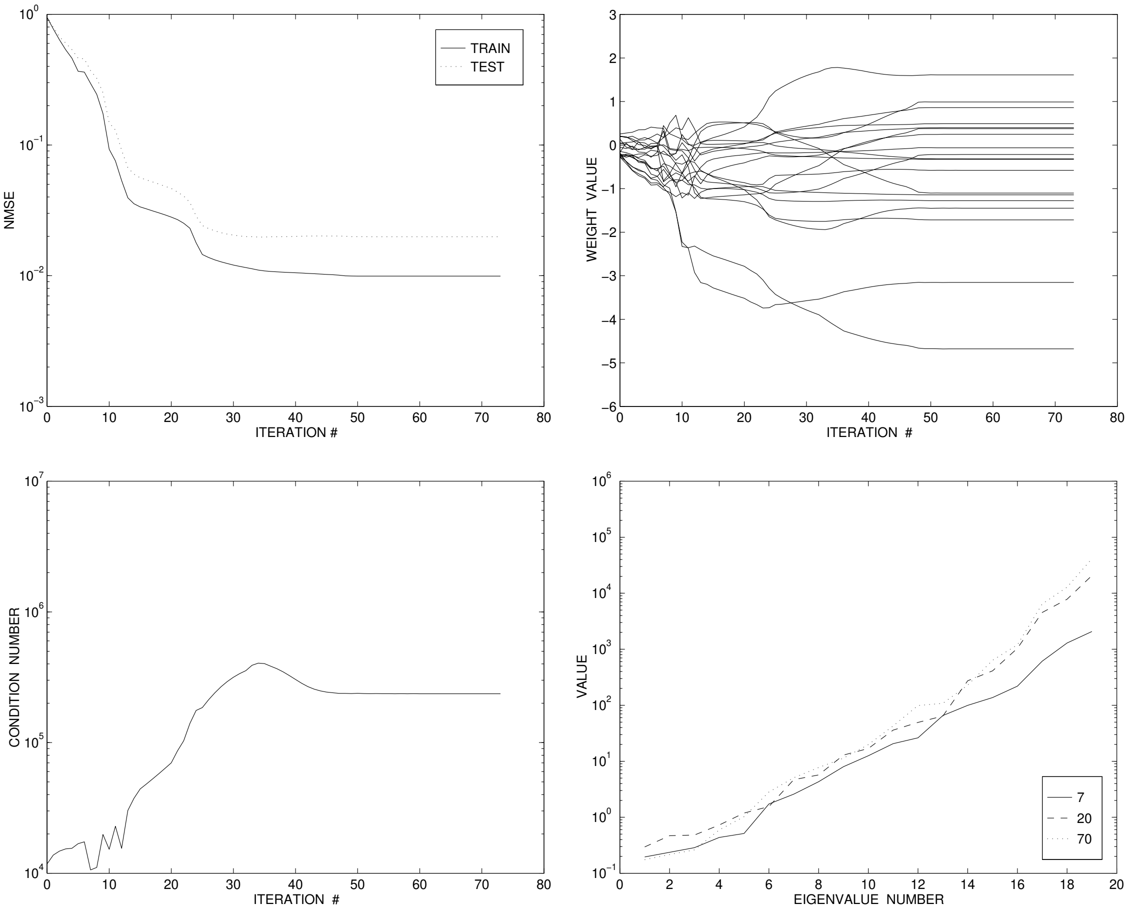


Figure 8.12: Training by damped Gauss-Newton, $\alpha = 10^{-1}$. Upper left panel: Evolution of training and test errors. Upper right panel: Evolution of the weight values. Lower left panel: Evolution of the condition number of the Hessian, $\kappa(\mathbf{H})$. Lower right panel: Eigenvalues of the Hessian at iterations 7, 20 and 70.

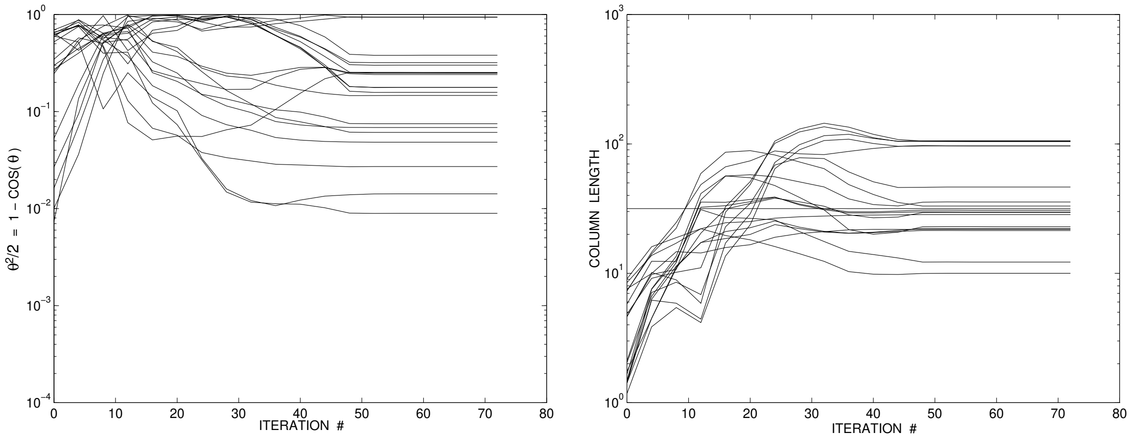


Figure 8.13: Training by damped Gauss-Newton, $\alpha = 10^{-1}$. Left panel: Measuring the collinearity between the 20 most collinear column pairs of the Jacobian matrix. Right panel: Measuring the length of the columns of the Jacobian.

are somewhat higher than in Figure 8.8 as the weights are now severely constrained by the increased weight decay as seen from the upper right panel of Figure 8.12; the magnitudes of the weights are significantly smaller than for the lower weight decays. As for the lower weight decay, iterations terminate due to satisfaction of the stopping criterion $\|\mathbf{g}\|_2 < 10^{-4}$ but now after only 73 iterations; the training problem has become more "quadratic", i.e., more well-behaved due to the larger weight decay.

As seen from the lower left panel of Figure 8.12 the condition number of the Hessian is reduced further compared to Figure 8.10, about three orders of magnitude. Comparing the lower right panel of Figure 8.12 to Figure 8.10 we learn that two orders of magnitude are due to the direct effect of the increased weight decay, increasing the smallest eigenvalues. The last order of magnitude is due to a reduction of the largest eigenvalue, resulting from the smaller weight magnitudes.

The improvement in the condition number is further reflected by Figure 8.13. In the left panel it is seen that the columns of the Jacobian are now less collinear than was the case in Figure 8.11 when using a lower weight decay. Also, the lengths of the columns of the Jacobian seen in the right panel of Figure 8.13 are further reduced due to the increased constraints on the weight magnitudes.

Augmenting the quadratic cost function by a weight decay term eliminates the directions of constant cost which were illustrated in Figure 8.5. When using a weight decay the total cost will no longer remain unchanged in these directions as is illustrated in Figure 8.14. Here, the total cost function surface is displayed for the same weights as in Figure 8.5, now after 21 iterations using $\alpha = 10^{-1}$. It is seen that the direction in which the cost was constant when training without a weight decay is now effectively "cealed off." This elimination of constant directions in parameter space also eliminates the corresponding large components which previously dominated the Gauss-Newton search directions, leading to improved training of the network.

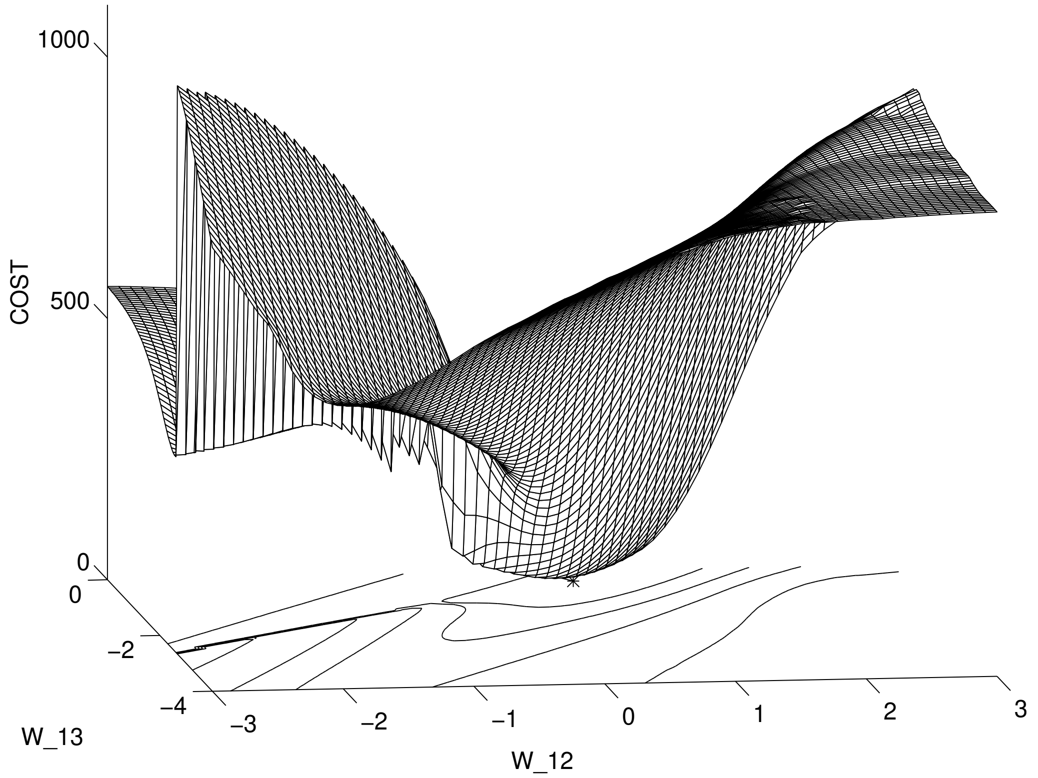


Figure 8.14: Training by damped Gauss-Newton, $\alpha = 10^{-1}$. Cost function surface after 21 iterations illustrated as a function of the weights w_{12} and w_{13} leading *from* hidden units two and three *to* hidden unit one. The asterisk on the underlying contour plot denotes the current iteration point. Note how the weight decay has eliminated the "rain gutter"-like structure of the cost surface.

8.1.4 Training by gradient descent

The examples above have given a practical illustration of how ill-conditioning might arise when training recurrent networks using the damped Gauss-Newton method, how ill-conditioning of the Hessian may actually increase when using this method and how weight decay helps the conditioning problems by bounding the eigenvalues from both below and above thus improving efficiency of the training.

We now turn to investigate the effects of ill-conditioning when training using gradient descent featuring a line search, as described in chapter 4. In order to allow for a direct comparison to the damped Gauss-Newton method, experiments were performed on the exact same problem as considered above. Training was started from the same initial weights as above, but instead of switching to the Gauss-Newton method after five iterations, training was continued using gradient descent.

In the left panel of Figure 8.15 is seen the evolution of the errors when training *without* weight decay for 1000 iterations. Comparing to the Gauss-Newton method without weight decay in Figure 8.2 it is seen that the levels of the errors reach a lower level before they flatten out due to apparent convergence. As was the case for the Gauss-Newton method, iterations are however *not* close to a local minimum. This is indicated by the small but significant weight changes displayed in the right panel of Figure 8.15 and in particular by

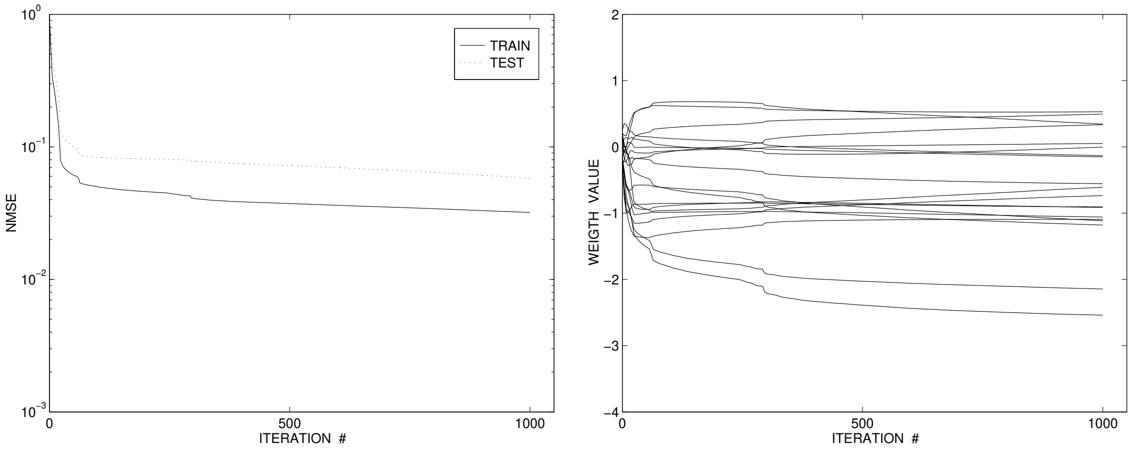


Figure 8.15: Training by gradient descent, $\alpha = 0.0$. Left panel: Evolution of training and test errors. Right panel: Evolution of the weight values.

the Euclidean norm of the gradient, shown in the left panel of Figure 8.16. The gradient norm is seen to be large compared to the smallest norm obtained in Figure 8.9 at which no further weight changes took place.

In the right panel of Figure 8.16 is shown the cross correlation coefficients between the hidden unit outputs. Comparing to Figures 8.4 and 8.9 for the damped Gauss-Newton method it is seen that the hidden unit outputs do not become nearly as correlated when training using gradient descent. This is reflected by the evolution of the condition number, shown in the left panel of Figure 8.17. It is seen to be small compared to the condition numbers which resulted from Gauss-Newton training even when using a relatively large weight decay. From the right panel of Figure 8.17 it is seen that the relatively small condition numbers result from both a relatively large magnitude of the smallest eigenvalue and from a relatively small magnitude of the largest eigenvalue.

From the left panel of Figure 8.18 it is seen that the columns of the Jacobian matrix

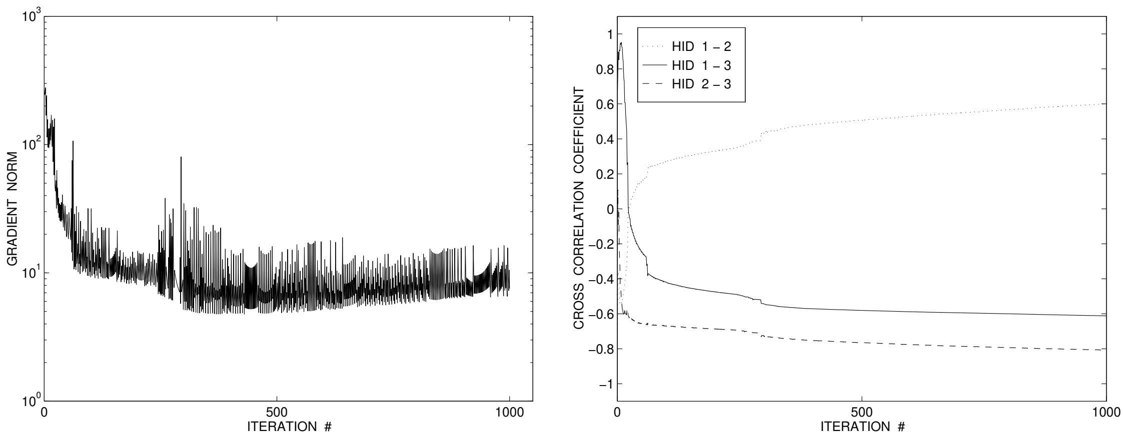


Figure 8.16: Training by gradient descent, $\alpha = 0.0$. Left panel: Evolution of the gradient norm, $\|\mathbf{g}\|_2$. Right panel: Evolution of the cross correlation coefficients between the hidden unit outputs.

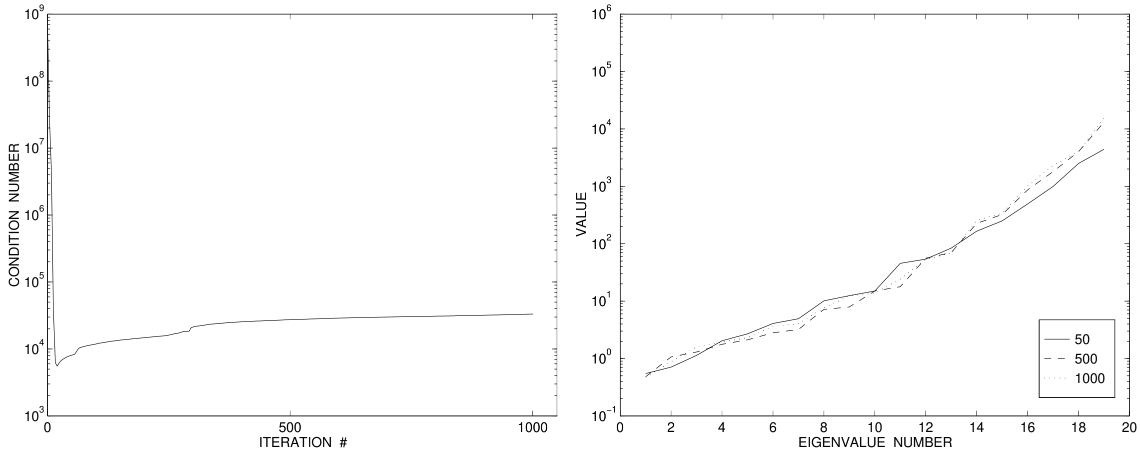


Figure 8.17: Training by gradient descent, $\alpha = 0.0$. Left panel: Evolution of the condition number of the Hessian, $\kappa(\mathbf{H})$. Right panel: Eigenvalues of the Hessian at iterations 50, 500 and 1000.

do not become nearly as collinear as was the case in Figure 8.7 when training by the damped Gauss-Newton method without weight decay. Rather, the magnitudes of the smallest $\theta^2/2$ are comparable to Figure 8.13 where a relatively large weight decay was used. The decreased collinearity is due to the decreased correlation between the hidden unit outputs leading to a relatively large magnitude of the smallest eigenvalue. The right panel of Figure 8.18 illustrates the length of the columns of the Jacobian matrix which are seen to be relatively small compared to the lengths in Figure 8.7. This is due to the small magnitude of the weights, leading to a relatively small value of the largest eigenvalue of the Hessian.

From the observations made above it is seen that *initally* the convergence of gradient descent is better than for the Gauss-Newton method without weight decay as it is not influenced in the same way by directions in which the cost function is almost constant. This leads to a lower level of error, before convergence slows down considerably. Whereas the

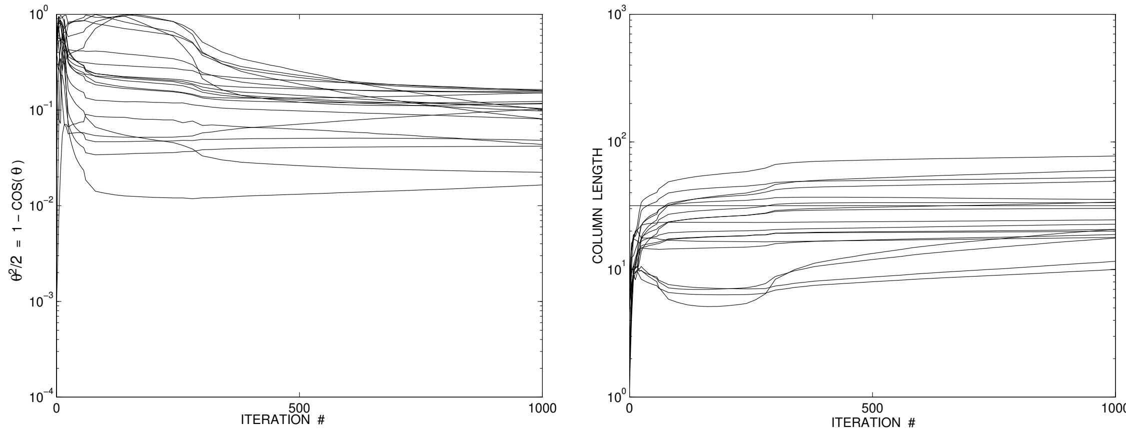


Figure 8.18: Training by gradient descent, $\alpha = 0.0$. Left panel: Measuring the collinearity between the 20 most collinear column pairs of the Jacobian matrix. Right panel: Measuring the length of the columns of the Jacobian.

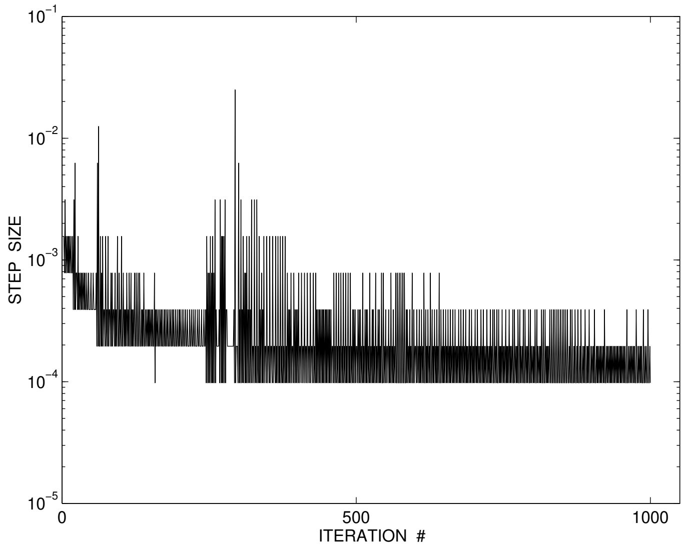


Figure 8.19: Training by gradient descent, $\alpha = 0.0$. Step size found by line search in each iteration.

damped Gauss-Newton method would take large steps in this situation, gradient descent can take steps no larger than $\eta = 2/\lambda_{\max}$ (Eq. 4.13) in order to ensure a decrease in the cost function. From the eigenvalues shown in the right panel of Figure 8.17 we learn that the maximum step size is $\eta \approx 2 \cdot 10^{-4}$. Figure 8.19 shows the step sizes actually chosen by the line search and we see that they are indeed matching the theoretical limit. Thus, the steps taken by gradient descent are very small, leading to slow convergence.

From the examples above using damped Gauss-Newton we learned that this method could easily handle condition numbers of a magnitude of 10^4 which is commonly encountered when training recurrent networks. Condition numbers of this *relatively* small order of magnitude will however lead to very slow convergence of gradient descent as illustrated theoretically by Eq. (7.4) and in practice by the example above.

In an attempt to improve convergence for gradient descent we may augment the quadratic cost function by a weight decay term in order to reduce the condition number as was successfully applied for the damped Gauss-Newton method. From the right panel of Figure 8.17 we learn that the magnitude of the weight decay should be $\alpha \approx 1$ in order to increase the smallest eigenvalues.

Training by gradient descent was repeated several times from the same initial weights as previously, using an increasing magnitude of weight decay. As expected, weight decays of a magnitude less than one had no significant influence as the weights resulting during training and thus the errors and condition number differed only in insignificant decimal places from the values which resulted from training without weight decay. At no point was obtained errors smaller than displayed in Figure 8.15. It turned out that in order to decrease the condition number by one order of magnitude, the weight decay should be $\alpha = 5$.

The evolution of the errors when training using a weight decay of $\alpha = 5.0$ is shown in the upper left panel of Figure 8.20. One of the effects of the large weight decay is seen to be increased errors compared to Figure 8.15 which is due to the significant constraints put onto the weights as seen from the upper right panel of Figure 8.20. Comparing the magnitude of the gradient norm shown in the lower left panel of Figure 8.20 with the levels

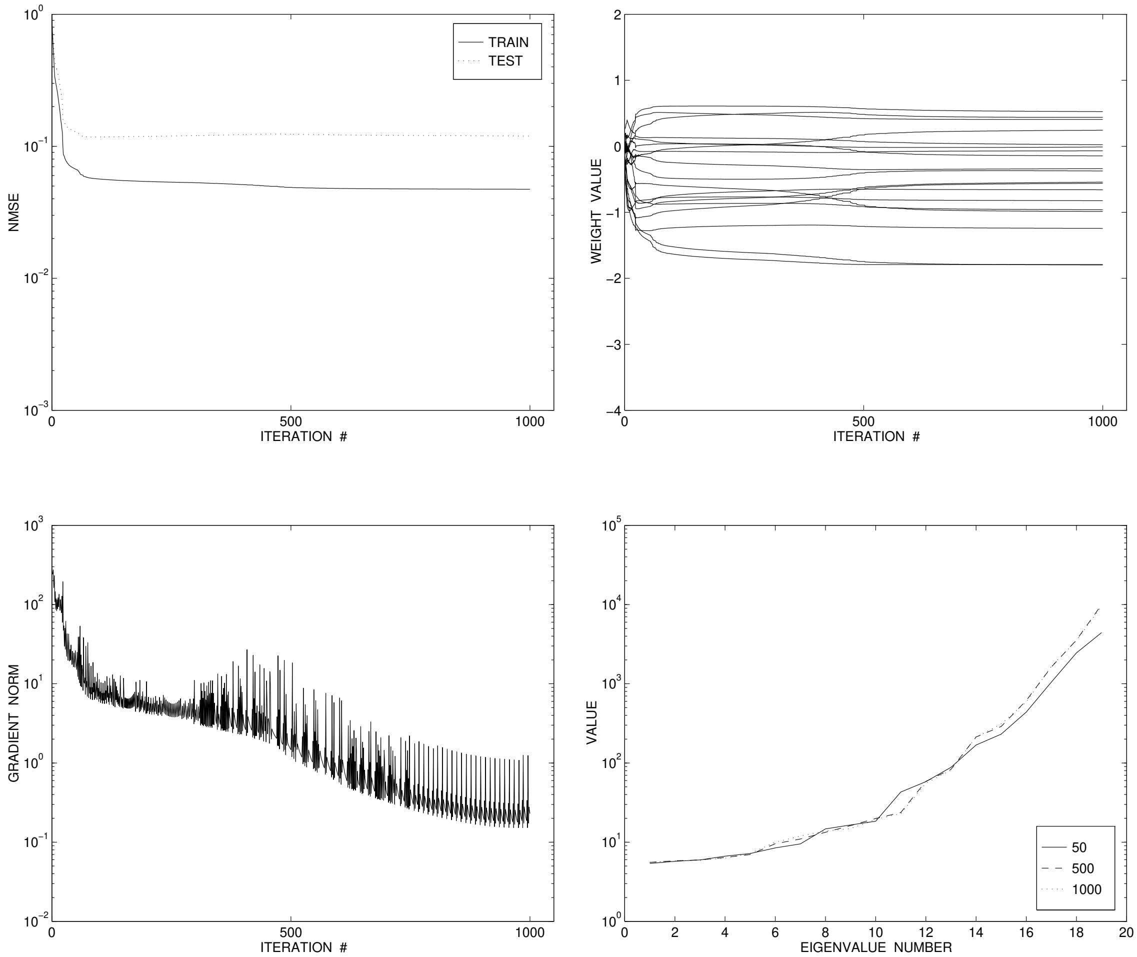


Figure 8.20: Training by gradient descent, $\alpha = 5.0$. Upper left panel: Evolution of training and test errors. Upper right panel: Evolution of the weight values. Lower left panel: Evolution of the gradient norm, $\|\mathbf{g}\|_2$. Lower right panel: Eigenvalues of the Hessian at iterations 50, 500 and 1000.

of the gradient norm obtained without weight decay shown in Figure 8.16, it seems that iterations have come nearer to a local minimum of the total cost within the 1000 iterations allowed as the gradient norms are about two orders of magnitude smaller. The improved convergence towards a local minimum of the augmented cost function however comes at the price of reduced performance on the *primary* cost, the quadratic cost function, as the assisting weight decay term has become too dominating.

From the lower right panel of Figure 8.20 showing the eigenvalues of the Hessian at various stages of training it is seen that the slightly improved convergence is obtained from the decrease of the condition number by one order of magnitude, compared to Figure 8.17. It is seen that the reduction is obtained mainly from an increase of the smallest eigenvalues, the largest eigenvalues are of about the same order of magnitude as without weight decay.

As a curiosity it is mentioned that in order to satisfy the stopping criterion $\|\mathbf{g}\|_2 < 10^{-4}$ within 1000 iterations when using gradient descent the weight decay had to be $\alpha = 150$, leading to a mere condition number ≈ 20. Naturally, a weight decay of this order of magnitude leads to completely degenerate networks. For comparison, the Gauss-Newton method needed a weight decay as low as $\alpha = 10^{-7}$ in order to spend *more* than 500 iterations in order to satisfy the same stopping criterion.

The proper choice of weight decay is dependent upon both the problem at hand as well as the specific network architecture used. Even so, the examples above have indicated that the damped Gauss-Newton method is capable of converging much faster to a local minimum than gradient descent, and this in the presence of condition numbers many orders of magnitude larger than can be handled by gradient descent. Consequently, the weight decay needed by the damped Gauss-Newton method in order to converge is many orders of magnitude smaller relative to the magnitude needed if training by gradient descent. Using a smaller weight decay means that the influence on the modeling capabilities of the network is less pronounced, leading to better adapted network models.

8.2 Comparison: feed-forward vs. recurrent networks

In section 7.2.2 it was described how the situations found leading numerical problems for neural network structures were potentially much more severe for recurrent networks than for feed-forward networks. The reason for this was found to be rooted in the higher connectivity of recurrent networks. This leads to many more weights affected if e.g., two hidden units specialize on the same features in the data during training and become proportional as was illustrated in the examples above. Furthermore, the recurrent structure of the partial derivatives, which are calculated partly as weighted sums of the hidden unit derivatives from the previous time step, will tend to increase the magnitude of the largest eigenvalues compared to feed-forward networks as was described in section 7.2.3.

In an attempt to illustrate the increased numerical problems one is facing when working with recurrent rather than feed-forward networks, a quantitative comparison between the two network structures was made. Networks with an increasing number of hidden units were initialized with random values, drawn from a uniform distribution centered around zero. The networks were then applied to the first 1000 points of the Santa Fe laser data and the Hessian matrix computed for the resulting unregularized cost function, and the smallest and the largest eigenvalue were determined. For each network structure this procedure was repeated 100 times for each of a number of increasing magnitudes of the bounds on the uniform distribution from which the weights were drawn.

In Figure 8.21 is shown the results for the recurrent network structures, each having a single external input and three, five and eight hidden units, respectively. The mean of the logarithm of the *largest* eigenvalues is indicated by an 'x' while the dotted vertical line indicates the standard deviation. The mean of the logarithm of the *smallest* eigenvalues is indicated by an 'o' with the solid vertical line indicating the standard deviation. By comparing the levels of the largest eigenvalues in the three panels it is clearly seen that the magnitude of the largest eigenvalues increase as the number of hidden units in the network increases. From a level of 10^6 for the network in the upper left panel of Figure 8.21 having three hidden units the magnitude increases to a level around 10^9 for the network in the lower panel having eight hidden units. This increase in magnitude can be explained by the increasing number of terms entering the calculation of the partial derivatives wrt. the hidden unit weights; refer to Eq. (4.37). The more terms entering the calculation, the higher the probability of large magnitude partial derivatives and thus the higher the probability of large magnitude eigenvalues. Within each plot in Figure 8.21 it *appears* that the magnitudes of the largest eigenvalues are not influenced by the increase in the weight

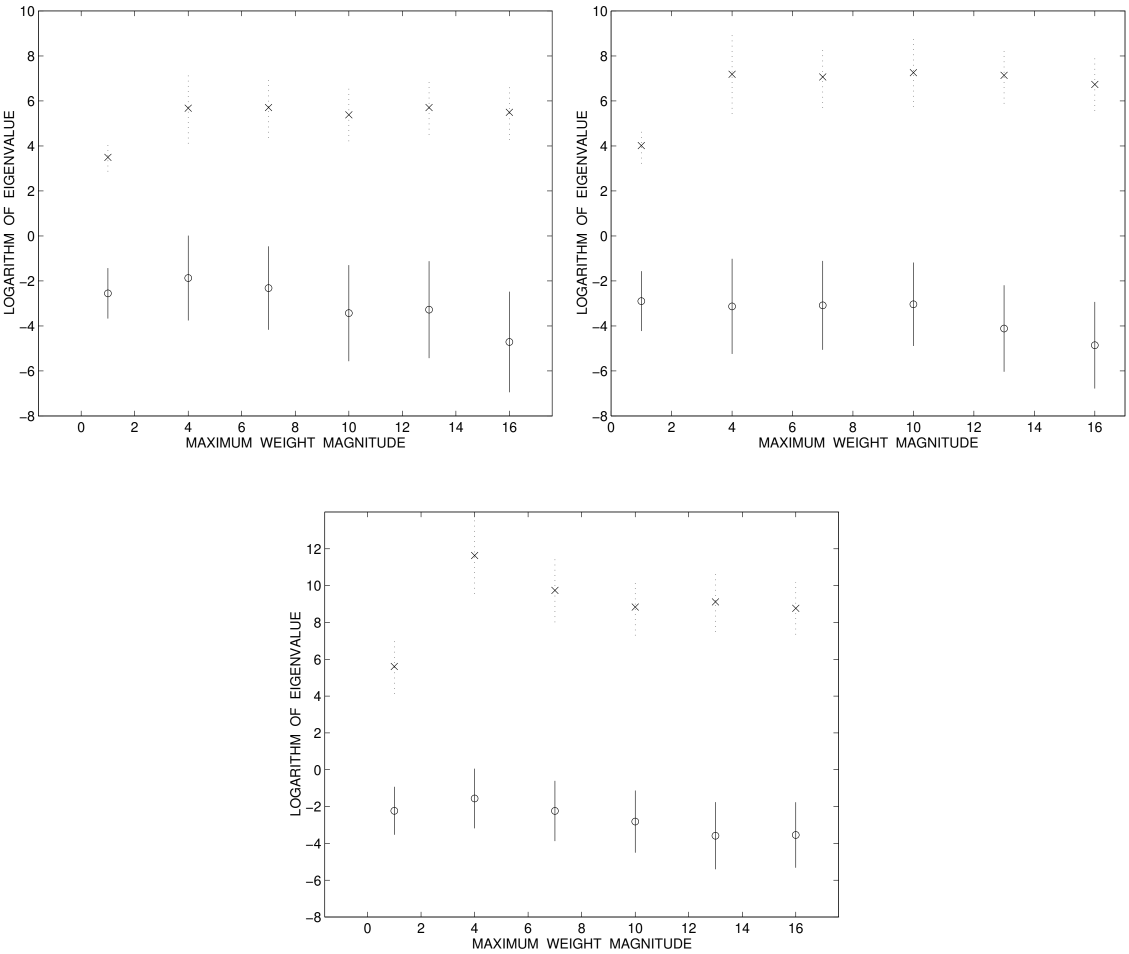


Figure 8.21: Mean values and standard deviations of the logarithm to the largest ('x') and smallest ('o') eigenvalues of the Hessian for *recurrent* networks with random weights of increasing magnitude. Upper left panel: One input, three hidden units (19 weights). Upper right panel: One input, five hidden units (41 weights). Lower panel: One input, eight hidden units (89 weights).

magnitudes. This is due to the large variability within each weight magnitude as well as the logarithmic scale used.

The levels of the smallest eigenvalues are seen to be unaffected by the increasing number of hidden units. The reason for this is the lack of specialization of the hidden units on the same features in the training set. As the networks are completely random the hidden units have of course not specialized to anything and the hidden unit outputs are therefore not likely to become proportional for this reason. However, as the magnitude of the random weights increases so does the probability of several hidden units becoming saturated and thus proportional, leading to small magnitude of the smallest eigenvalues. In fact, if the magnitude of the weights is extended beyond the largest magnitude used in Figure 8.21 this will lead to a significant drop in the levels of the smallest eigenvalues. This drop is indicated by the slightly decreasing levels of the smallest eigenvalues seen in Figure 8.21 as the weight magnitudes are increased.

In Figure 8.22 is shown the results for the feed-forward network structures, each hav-

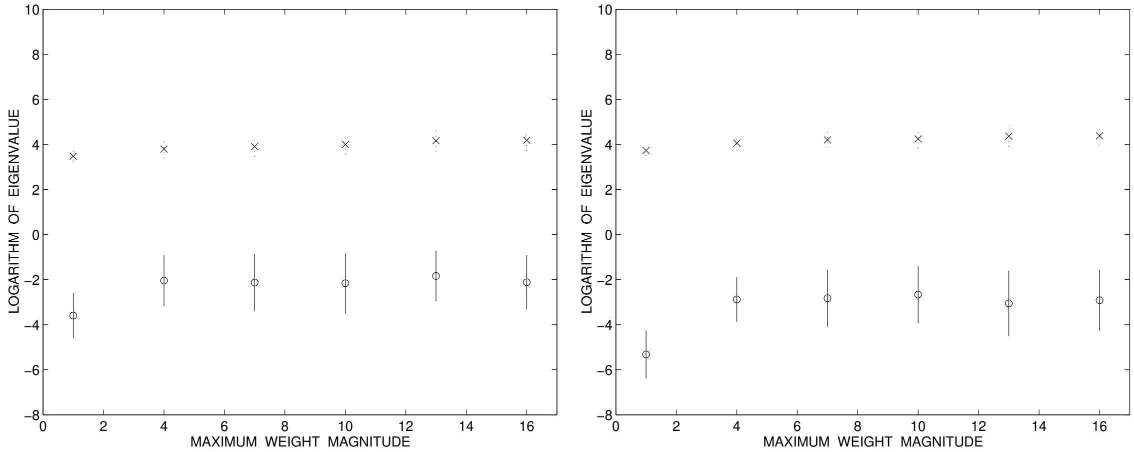


Figure 8.22: Mean values and standard deviations of the logarithm to the largest ('x') and smallest ('o') eigenvalues of the Hessian for *feed-forward* networks with random weights of increasing magnitude. Left panel: Four inputs, seven hidden units (43 weights). Right panel: Four inputs, fifteen hidden units (91 weights).

ing four external inputs, $\mathbf{x}(t) = [x(t), \ldots, x(t-3)]$, and seven and fifteen hidden units, respectively. Note that the levels of the largest eigenvalues are significantly lower than for the recurrent networks and especially note that the level of the largest eigenvalues is unaffected by the increase in the number of hidden units. The lower level of the largest eigenvalues is explained by the simpler, non-recursive structure of the partial derivatives of the feed-forward network output involving only a single term; refer to Eqs. (7.17–7.25). As the expression for each individual partial derivative is independent of the number of hidden units in the network the magnitude and thus the largest eigenvalues will remain unaffected by the increase in the number of hidden units. Furthermore, the non-recursive structure leads to less variability in the values of the derivatives which explains the decreased standard deviations compared to the recurrent networks in Figure 8.21.

This simple comparison between random recurrent and feed-forward networks has experimentally verified the observation made in section 7.2.3. that application of recurrent networks leads to increased numerical problems compared to feed-forward networks due to an increase of the largest eigenvalue of the Hessian. The differences in the smallest eigenvalues in this comparison were less pronounced due to the randomness of the networks. In order to properly illustrate the differences for the smallest eigenvalues the comparison should be made between *trained* networks; such comparison has however not been performed in the present work.

8.3 Importance of second derivative term

This chapter is concluded by an attempt to indicate the importance of the second derivative term $\mathbf{S}$ of the Hessian; refer to Eq. 4.23. The second derivative term is often neglected during both training by second-order methods as well as during analysis [DS83, SBC93, HS93] as was also the case in the experiments described above.

In [PH95] included in appendix G the relative importance of the second derivative

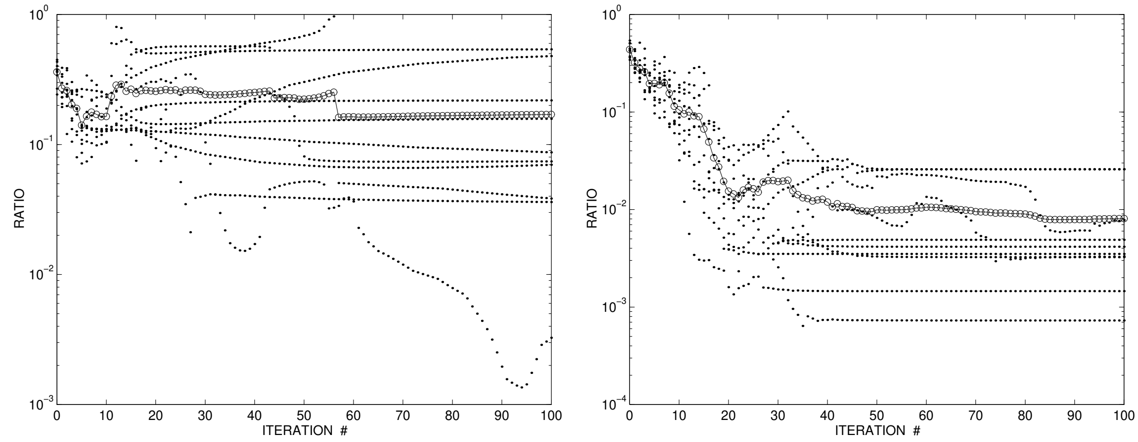


Figure 8.23: Ratios between the largest magnitude eigenvalue of the second derivative term $\mathbf{S}$ of the Hessian and the largest magnitude eigenvalue of the complete Hessian $\mathbf{J}^{\mathrm{T}}\mathbf{J} + \mathbf{S}$ as they appeared during ten training sessions on the laser series. The connected circles represent the average ratios. Left panel: Training with a small weight decay, $\alpha = 10^{-10}$. Right panel: Training with a higher weight decay, $\alpha = 10^{-1}$.

term was determined as the ratio between the largest magnitude eigenvalue of the second derivative term $\mathbf{S}$ of the Hessian and the largest eigenvalue of the complete Hessian $\mathbf{J}^{\mathrm{T}}\mathbf{J} + \mathbf{S}$. This way, the importance of the second derivative term during training on the sunspot problem was monitored as iterations progressed using two different magnitudes of weight decay. It was found that applying a weight decay significantly reduced the relative importance of the second derivative term due to the regularizing effect on the complexity of the cost function, thus justifying the Gauss-Newton approximation.

In order to support this finding a similar experiment was performed for the Santa Fe laser series using the same experimental setup as in section 8.1, i.e., one external input and three hidden units. Ten networks with random weights were trained by the damped Gauss-Newton method using a small weight decay $\alpha = 10^{-10}$ and the ratio between the eigenvalues was determined in each iteration; the second-derivative term $\mathbf{S}$ was computed from the expressions given in section 4.8.2. The resulting ratios along with the mean values are illustrated in the left panel of Figure 8.23.

The experiment was then repeated using a higher weight decay $\alpha = 10^{-1}$, the result of which is illustrated in the right panel of Figure 8.23. The average relative importance of the second derivative term is seen to be reduced, thus indicating increased justification of the Gauss-Newton approximation to the Hessian when regularizing the cost function.

Chapter 9

Recurrent network training experiments

We will now turn towards examples of recurrent network training which are more quantitative in nature. As described in the introduction to chapter 7 it generally believed to be a difficult task to train recurrent networks by first-order methods like gradient descent. Even so, gradient descent is still the most common training method of choice, possibly due to the ease of implementation. A primary object of this work has been to overcome the problems of recurrent network training by application of potentially more efficient second-order methods. In particular the damped Gauss-Newton method has been applied and it has turned out to be much more efficient than gradient descent in terms of both computation time and quality of the resulting network models as we shall see.

This chapter describes the results which have been obtained by single-input, fully recurrent networks applied to time series prediction. Section 9.1 contains a comparison between gradient descent and the damped Gauss-Newton method which illustrates the typically encountered improvement in training when using the second-order method. In section 9.2 it is investigated how the performance of a trained recurrent network depends on the number of examples used for training in terms of the *learning curve*. In particular, a learning curve is generated for both the Santa Fe laser series and the Mackey-Glass series. Once the expected performance of single-input, fully recurrent networks has been established in terms of the learning curve it is of interest to compare with the expected performance of comparable *feed-forward* networks working from an externally provided lag space. Section 9.3 describes experiments for selection of optimal lag space as well as generation of learning curves for feed-forward networks applied to the laser and the Mackey-Glass series in order to perform such a comparison. The chapter is concluded in section 9.4 by an investigation of the extent to which the single-input recurrent networks are able to simulate the dynamics which generated the chaotic time series on which they were trained.

9.1 Comparison of training methods

In the previous chapter illustrating the effects of ill-conditioning on a small example it was indicated that augmenting the cost function by a small weight decay can greatly improve the efficiency of the damped Gauss-Newton method in terms of convergence speed and quality of solution obtained. In contrast the gradient descent method was practically unaffected by the weight decay as it still converged very slowly, yielding an unsatisfactory solution. Only in the case of a prohibitively large weight decay leading to a degenerate network was gradient descent able to converge to a local minimum of the augmented cost function within reasonable computation time. These observations suggest that the gradient descent method is very sensitive to ill-conditioning and therefore *not* suited for training of recurrent networks which are inherently ill-conditioned in nature as described in chapter 7. On the other hand, once the need for a regularization term has been recognized the damped Gauss-Newton method seems to be very robust to even a relatively large condition number and is thus a better choice of training algorithm.

We will now consider a quantivative comparison between the two training methods. The first example compares the ability of the two methods to reach a local minimum of the cost function. In the next example it is illustrated how the second-order method leads to much better performing networks in much less computation time.

As in the previous chapter the problem is modeling of the first 1000 points of the Santa Fe laser series described in appendix A.1 using recurrent networks with only one external input but now using nine hidden units. As there was no feedback from the output unit back to the hidden units the networks comprised 109 weights. Five networks were generated by initializing their weights with random values drawn from a uniform distribution over the interval $[-0.3; 0.3]$. These networks were trained using gradient descent featuring the step size halving line search described in section 4.3. The cost function was augmented by a weight decay term using $\alpha = 0.02$ and the stopping criteria were set to a gradient norm $\|\mathbf{g}\|_2 < 10^{-4}$ *or* a maximum of 8000 iterations reached.

The resulting evolutions of the training errors (Normalized Mean Squared Errors defined in Eq. (A.1)) are indicated by the dotted lines in Figure 9.1. It is seen that the training errors decrease significantly during the first 5000–6000 iterations after which convergence becomes extremely slow. For all five networks training stopped after the maximally allowed 8000 iterations. If no further investigations regarding the networks are made one might draw the conclusion that the networks have converged to a local minimum of the cost function. This is however *not* the case, revealed by the gradient norm (not shown) which for all networks were $\|\mathbf{g}\|_2 \approx 1$. Progress has practically come to a halt due to the influence of ill-conditioning on the gradient descent method.

In order to clearly illustrate the difference in convergence speed between gradient descent and the damped Gauss-Newton method, the comparison between the two methods was initiated once gradient descent slowed down considerably and thus *seemingly* approached a local minimum. The networks from Figure 9.1 resulting after 6000 iterations of gradient descent were extracted and further training was applied using the damped Gauss-Newton method. The resulting evolutions of training errors are illustrated by the solid lines in Figure 9.1. After between 166–449 of damped Gauss-Newton iterations the gradient stopping criterion $\|\mathbf{g}\|_2 < 10^{-4}$ was met for all five networks, indicating closeness to local minima. Furthermore, the value of the training errors had dropped almost an order of magnitude. If given enough iterations, gradient descent would probably converge towards the same local minima as the damped Gauss-Newton and thus obtain the same low levels of error. The number of iterations needed for this however seems to be enormous, making the low levels of error practically unattainable when using gradient descent.

One might argue that the increased performance on the training set will not necessarily lead to increased performance in terms of generalization ability. Further, each iteration

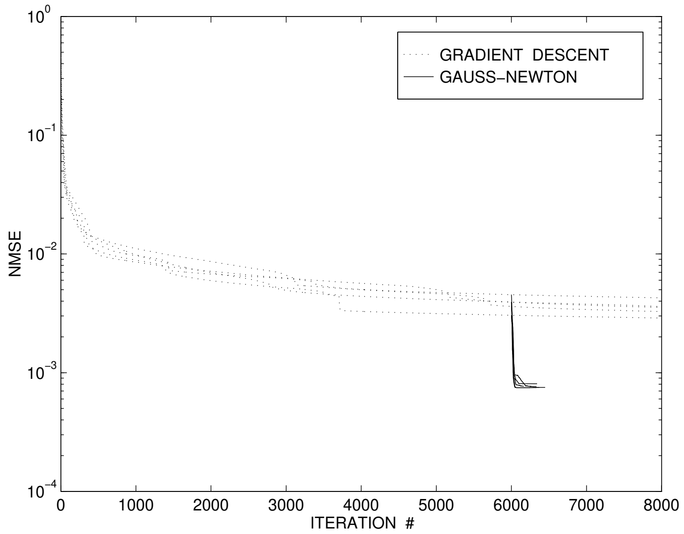


Figure 9.1: Comparison of convergence properties for gradient descent and damped Gauss-Newton. The dotted lines indicates evolution of training errors when training by gradient descent, the solid lines starting at iteration 6000 indicates a switch to the damped Gauss-Newton method.

of the second-order method comes at the expense of an increased compuational burden as it is needed to compute an approximation to the Hessian and solve a system of linear equations in each iteration as described in chapter 4. However, experience shows that the dramatic decrease in training error is generally accompanied with a decrease in estimated generalization ability as well provided that the network model does not possess excess complexity which might lead to "overfitting" as described in chapter 6. The increase in computation time in each iteration is justified by the faster convergence leading to less total computation time.

In order to provide an illustration of these statements an experiment with a setup similar to the example above was performed on the first 1000 points of the Santa Fe laser series, the following 100 points were used as a test set. Thus, each iteration of the damped Gauss-Newton method involved solving a system of linear equations of size 109 by 109. A set of six inital networks were generated by initializing their weights with values drawn from a uniform distribution over the interval $[-0.3; 0.3]$. These six networks were duplicated into a completely identical set. Performance was then compared by training one set of networks using gradient descent and training the other set by the damped Gauss-Newton method. For both sets of networks a weight decay $\alpha = 0.02$ was used. The resulting evolution of errors is shown in Figure 9.2; in the left panel is shown the resulting errors using the damped Gauss-Newton method, in the right panel using gradient descent. Using both methods the stopping criteria were set to $\|\mathbf{g}\|_2 < 10^{-4}$ or a maximum of 10000 iterations reached.

For the damped Gauss-Newton method the gradient norm stopping criterion was met in all six runs. The resulting average training error (Normalized Mean Squared Error) was $7.7 \cdot 10^{-4}$, the average test error was $4.9 \cdot 10^{-3}$. The *average* time for a complete training run was 226 iterations performed in 200 seconds. For gradient descent the gradient norm stopping criterion was never met. The average training error obtained after the maximally allowed 10000 iterations was $4.0 \cdot 10^{-3}$, the average test error was $7.8 \cdot 10^{-3}$. The average

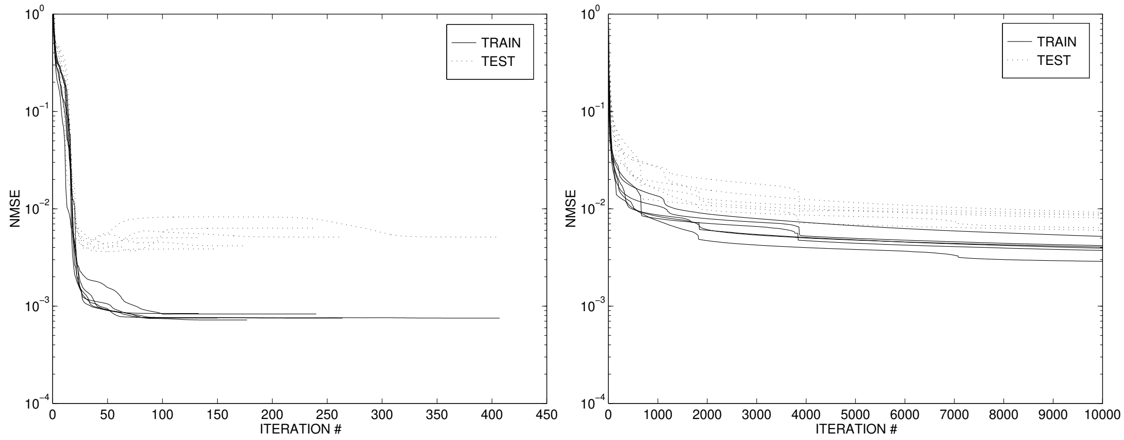


Figure 9.2: Evolution of errors when training the same initial recurrent networks on the laser series using different training methods. Left panel: The damped Gauss-Newton method. Right panel: Gradient descent with line search.

time used for obtaining these error levels was 8140 seconds. Note that the levels of both training and test errors obtained using gradient descent are much higher than the levels obtained using the damped Gauss-Newton method even though gradient descent used a factor of 50 times more iterations and a factor of 40 times more computer time. Thus, even though an iteration of the damped Gauss-Newton method is computationally more costly than an iteration of gradient descent, the additional cost per iteration is highly justified by the vastly increased convergence rate. The results of the comparison are summarized in Table 9.1.

	Gauss-Newton	Gradient descent
Final average training error	$7.7 \cdot 10^{-4}$	$4.0 \cdot 10^{-3}$
Final average test error	$4.9 \cdot 10^{-3}$	$7.8 \cdot 10^{-3}$
Average # of iterations	226	10000
Average time used per run	200 sec.	8140 sec.

Table 9.1: Summary of the results from the experiments illustrated in Figure 9.2.

By inspection of the left panel in Figure 9.2 it is seen that the networks trained by the damped Gauss-Newton method actually slightly overfits the training data as several of the test errors *increase* between iterations 30–100. This is not the case for the networks in the right panel, trained by gradient descent. Consequently, one might expect that an even more pronounced difference in performance measured on the test set would result if the model complexity was somewhat reduced.

Careful inspection of the results reveals that the difference in total computation time per iteration between gradient descent and the damped Gauss-Newton was as low as 0.07 seconds per iteration. The explanation for this very small time difference is to be found in the line search, which generally returns a smaller step size for the direction opposite the gradient than for the Gauss-Newton search direction; thus, more evaluations of the cost function are performed before a suitable stepsize is found. A slight reduction in average time per iteration could possibly be obtained from utilization of this general observation, starting from a lower initial step size when using gradient descent.

Another explanation for the small difference in computation time between the first-order and second-order methods has to do with the method used for computing the gradient. Recall from section 4.7 that the gradient could be computed using *two* approaches, the slower iterative approach (RTRL) and the faster approach involving unfolding in time (BPTT). The gradients in the experiment described above were actually computed using the slower RTRL approach. In order to investigate the improvement in speed that would result overall from the application of the BPTT approach, the experiment described above using gradient descent for training was repeated, now using BPTT. As the initial networks were the same, the resulting errors were identical to the ones reported in table 9.1. The time used for performing the 10000 iterations in each run was however reduced to 3077 seconds, or around one third of the time required when computing gradients by the RTRL approach. This overall speedup is fairly modest compared to the theoretical improvement of order $\mathcal{O}(N_u^2)$ (refer to section 4.7) in the actual gradient computation. However, the line search is requiring its fair share of the total computation time per iteration and is *not* influenced by the faster gradient computation. Overall speedup can never exceed the the fraction of the total computation time which is spent on computing the gradient (Amdahl's law, [Hay88]).

Due to the extremely slow convergence of gradient descent resulting from ill-conditioning, the damped Gauss-Newton method should be the training method of choice even for fairly large networks even though we are forced to compute the Hessian using the slow RTRL approach as described in section 4.8.1. The significant increase in the time required for obtaining the search direction for the damped Gauss-Newton method is highly justified by the superior convergence abilities leading to error levels that are practically unattainable using gradient descent. Such justification has been observed for networks with up to 300 parameters but is likely to result for even larger network structures.

9.2 Learning curves for recurrent networks

Having established the efficiency of the Gauss-Newton method for training compared to the traditionally adopted gradient descent we will now investigate the performance in terms of estimated generalization error which can be obtained from recurrent networks working from only a single external input. In chapter 6 it was described how a properly chosen model complexity is of importance for the generalization error. An equally important factor in order to obtain a small generalization error is the availability of a sufficient amount of data in order to train the model; if too little data is used for training, the model might not be able to properly capture the underlying dynamics of the series to be modeled and is furthermore likely to model the noise as well, i.e., to overfit the training data. As a result, generalization will suffer.

An important tool for investigating whether "enough" data is used for training is the *learning curve* [SSSD90, HKP91] which will be applied to recurrent networks in the following. Ideally, the learning curve expresses the average generalization error of a particular model structure over all possible training sets of a particular size T as a function of T. However, practical considerations usually lead to a more restricted definition of the learning curve. First of all, the generalization error has to be estimated using a finite amount of data; here, the generalization error is estimated empirically on a finite size test set immediately following the training set. The test set is kept fixed for all training set sizes which will "bias" the generalization error estimates for the resulting models by an equal amount, allowing for consistent ranking of models trained on different training set sizes. Furthermore, the estimated average generalization error is here obtained by averaging over different realizations of the estimated models for a *fixed* training set; that is, there is no averaging over *different* training sets of a particular size T.

Despite the restricted definition of the learning curve it is still an important tool for assessing whether the generalization error of a particular model structure may be decreased[1] if using more data for training or whether the additional training data will leave the generalization error practically unchanged and thus merely lead to an increase in computation time. A "typical" learning curve displays an initial rapid decrease in generalization error as the size of the training set is increased. For larger training set sizes the learning curve flattens out, indicating that only a small reduction in generalization error will result if increasing the training set size. It is therefore desirable to use a size of the training set at which the learning curve has flattened out. On the other hand, if the learning curve has *not* flattened out for the training set size used this is an indication that the training set used is too small and does not allow for optimal training of the model.

We will now proceed to describe the generation of learning curves for recurrent networks. Learning curves were generated for both the Santa Fe laser series as well as for the Mackey-Glass series. The series are described in appendix A.

9.2.1 RNN learning curve for the laser series

For convenience of presentation the full laser series described in appendix A.1 is illustrated in Figure 9.3 as well. In order to generate a learning curve for the Santa Fe laser series, the first 7000 points of the series corresponding to the two uppermost panels in Figure 9.3 were set aside for training and the last 3093 points corresponding to the lower panel of Figure 9.3 were used as a test set. The increasing size of the training series was obtained by extending *backwards* in time from point 7000 in order always to have the test series immediately following the training series and at the same time keeping the test series fixed. For instance, a training set of size 1000 involved training using points $x(6000)$ through $x(7000)$.

The fully recurrent network structure used to generate a learning curve for the laser series worked in "open-loop" from only a *single* input $x(t)$ in order to predict the immediately following value in the series; at each time step t the desired network network output $d(t)$ was $d(t) = x(t+1)$. As the network received only one external input the choice of lag space was eliminated. However, it is still necessary to determine a proper number of hidden units as well as the weight decay for recurrent networks as is the case for feed-forward networks.

After some initial testing it was decided to use ten hidden units as this seemed like a fair tradeoff between low model complexity and good modeling capabilities; the network thus comprised 131 weights in total. The final choice to make was that of the weight decay to use. Whereas the weight decay is often chosen from a model complexity point of view in order to improve generalization as it was described in section 6.1, the primary concern was here to choose the weight decay in such a way so as to avoid numerical problems as described in chapters 7 and 8.

[1] It is in theory always possible to reduce the generalization error further towards the noise level if using more data for training; refer to the introduction in chapter 6.

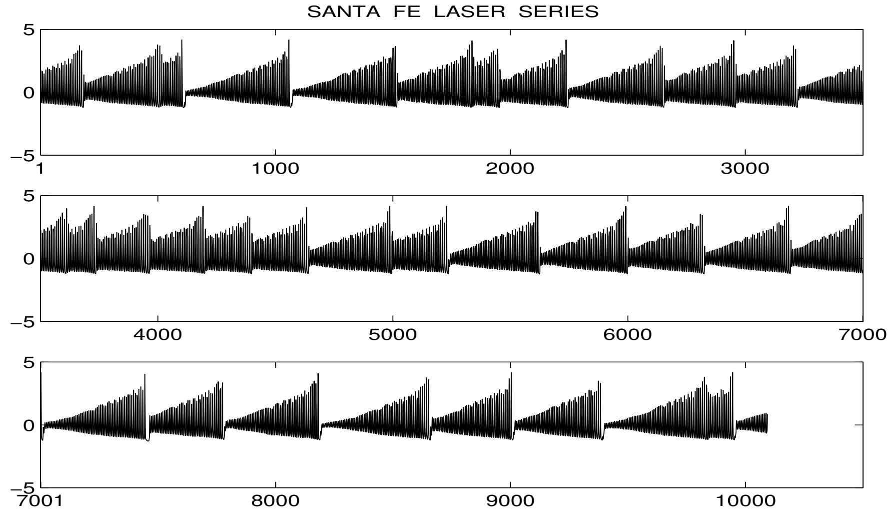


Figure 9.3: The complete Santa Fe laser series, scaled to zero mean and unit variance.

A proper value of the weight decay to use was decided upon by empirical means, i.e., by trial-and-error. Initially it was found that a large weight decay was necessary in order to avoid numerical problems during training, but using a large weight decay for *all* the weights in the network reduced the modeling capabilities of the trained networks significantly. After some testing it was found that a split-up of the weights into two groups, one containing the "feed-forward" weights which would also be present in a feed-forward network and the other group containing the "feedback" weights was appropriate; this corresponds to the regularization terms defined in Eq. (6.5). It was found that lowering the weight decay for the feed-forward weights while maintaining a larger value for the feedback weights improved the modeling capability without leading to numerical problems. This finding may be explained by the fact that the increased numerical problems for feedback networks compared to feed-forward networks tend to be caused in directions of parameter space defined by the feedback connections as described in chapter 7 and illustrated in chapter 8. This way it was decided to use the weight decays $\alpha_{\mathrm{ff}} = 0.02$ for the feed-forward weights and $\alpha_{\mathrm{rec}} = 0.4$ for the feedback weights, i.e., the constraints put on the feedback weights were much larger than the constraints put on the feed-forward weights.

The learning curve which was generated for the laser series is shown in Figure 9.4. For each of the sampled training set sizes T ten networks were trained by the damped Gauss-Newton method until the stopping criterion $\|\mathbf{g}\|_2 < 10^{-4}$ was satisfied. The generalization errors were then estimated on the test set and scaled to the Normalized Mean Squared Error (NMSE) as defined in Eq. (A.1) on page 215. Each of the resulting test set errors enters Figure 9.4 as a dot and the connected circles indicate the expected test error for the sampled training set sizes.

Initially the test error drops as the size of the training set is increased, but from training set size 2000 to 5500 the expected test error is fairly constant. The reason for this atypical behaviour of the learning curve may be explained by visual inspection of the laser series in Figure 9.3 as the "shape" of many of the collapses between the corresponding points 1500–5000 (recall that the size of the training set is increased by extending backwards from point 7000) seems atypical for the test series. The test error drops significantly when increasing the training set size from 5500 to 6000 points which might be explained by

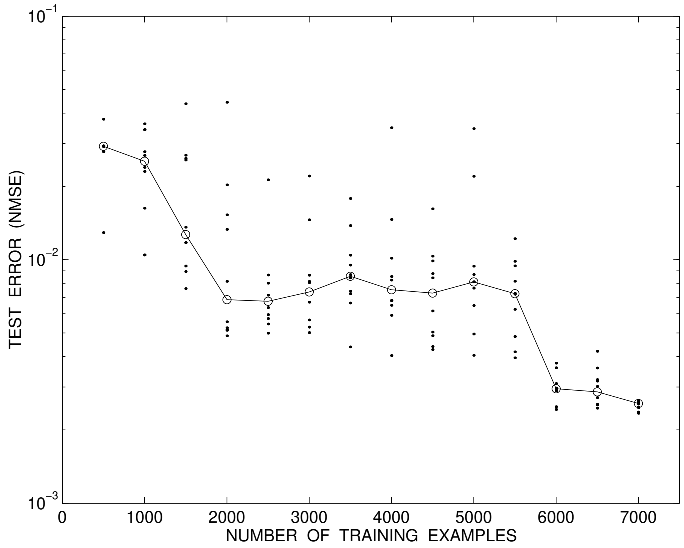


Figure 9.4: Learning curve for recurrent network having one external input and 10 hidden units (131 weights) trained on an increasing number of examples from the Santa Fe laser series. The expected normalized generalization error when training on 7000 examples is estimated to $\widehat{G}(\widehat{\mathbf{w}}) = 0.0025 \pm 0.0001$.

the fact that the training set now incorporates an additional collapse between points 1000 and 1500 very similar in shape to the ones in the test series. These observations suggest that for the laser series, the concept of an example should be conceived on several time scales: there are the pointwise examples corresponding to each single input presented to the network; but more important, there obviously exist what might be denoted as "super examples" consisting of a whole section or period of the time series, separated by the collapses. If additional "super" examples or sections when extending the training set are not similar to the sections encountered in the test series, generalization will not improve as seen from Figure 9.4.

Once more it is stressed that the behaviour of the learning curve illustrated in Figure 9.4 is atypical and is caused by the particular realization of the laser series from which the curve is generated. Had more data been available, allowing for a less restricted definition of the learning curve than described above, the learning curve would not have displayed the sudden plateau as this effect would be eliminated if averaging test errors from models trained on *different* training sets of the same size and estimating the generalization error on a larger test set.

It is noted how the variability within the models trained on the same training set decreases as the number of training examples is increased, as indicated by the decreasing spread in the resulting test errors. This corresponds to a decrease of the variance term contribution to the generalization error in Eq. (6.1). The learning curve flattens out at training set size 6000 and the variability within the resulting models is seen to be very small if using 7000 training examples. From the learning curve we may thus derive that no significant improvement in generalization error would be obtained if using more than 7000 examples for training. Finally we derive from the learning curve that one may expect a normalized generalization error of $\widehat{G}(\widehat{\mathbf{w}}) = 0.0025 \pm 0.0001$ when training a recurrent network with similar architecture and weight decay on 7000 examples from the Santa Fe laser series.

9.2.2 RNN learning curve for the Mackey-Glass series

The learning curve for the Mackey-Glass series, described in appendix A.2, was generated from the available sequence of 8500 points by setting aside the first 1500 points for training sets and using the following 7000 points as a test set in order to obtain a good estimate of the generalization ability of the resulting networks. As for the laser series, an increasing size of the training series was obtained by extending backwards in time from data point 1500 in order to evaluate networks trained on a different number of examples on the same test set, thus allowing for consistent comparison of performance.

As described in appendix A.2 it is common practice to implement a six step ahead predictor, i.e., to model $x(t+6)$ from a lag space vector $\mathbf{x}(t) = [x(t), x(t-6), \ldots]$ when using feed-forward networks. Here a fully recurrent network structure working from only one external input was used; at each time step t the network received $x(t)$ as input while the desired value for the network output was $d(t) = x(t+6)$. Once more the number of hidden units and the weight decay were decided upon by empirical means. It was decided to use eight hidden units so that the networks comprised 89 weights. It was found that a uniform weight decay for all weights in the network was appropriate, using $\alpha = \alpha_{\text{rec}} = \alpha_{\text{ff}} = 0.0125$; thus in this case it was *not* necessary to constrain the feedback weights relatively more than the feed-forward weights in order to handle numerical problems while still obtaining adequate performance from the network.

The learning curve generated for the Mackey-Glass series is illustrated in Figure 9.5. Ten networks were trained for each of the sampled training set sizes using the damped Gauss-Newton method, and the stopping criterion was once more set to $\|\mathbf{g}\|_2 < 10^{-4}$. The generalization errors were estimated on the test set, scaled to NMSE and enter Figure 9.5 as dots. The connected circles represent the expected test errors for the sampled training set sizes. The Mackey-Glass series learning curve has a more "typical" appearance than the laser series learning curve. This might be explained by increased "regularity," partly due to

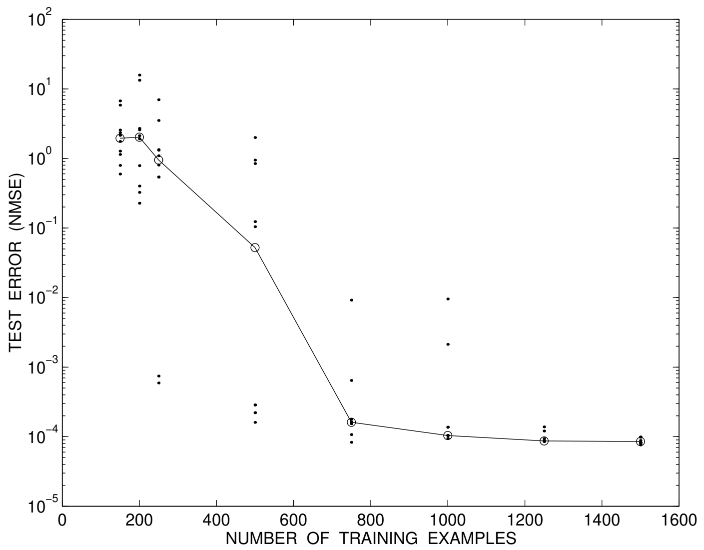


Figure 9.5: Learning curve for recurrent network having one external input and 8 hidden units (89 weights) trained on an increasing number of examples from the Mackey-Glass series. The expected normalized generalization error when training on 1500 examples is estimated to $\widehat{G}(\widehat{\mathbf{w}}) = 8.8 \cdot 10^{-5} \pm 8 \cdot 10^{-6}$.

the series being generated artificially rather than measured from a real-world experiment.

Initially the test error decreases rapidly as the training set is extended which is due to the increased amount of information available allowing for more accurate modeling of the underlying system. At a training set size of 750 examples the learning curve starts to flatten out. There is however still some variability in the resulting models; this is eliminated as the size of the training set is further increased which allows for more consistent modeling. At a training set comprising 1500 examples both the estimated expected generalization error as well as the variability seem to have reached a minimum, indicating that a further increase of the training set will yield very little improvement in performance. According to the learning curve one may expect a normalized generalization error of magnitude $\widehat{G}(\widehat{\mathbf{w}}) = 8.8 \cdot 10^{-5} \pm 8 \cdot 10^{-6}$ when training a recurrent network with similar architecture and weight decay on 1500 examples from the Mackey-Glass series.

9.3 Comparison of RNNs to feed-forward networks

A main advantage of using recurrent networks for time series prediction problems rather than feed-forward networks is their ability to work from only a single external input which has become apparent from the examples described so far. This is possible as recurrent networks are able to store a representation of previous inputs internally in the hidden units as was described in section 3.3. The ability to work from only one input relieves the modeler from the tedious and often difficult task of selecting a proper lag space as is necessary when using feed-forward networks and thus leads to a simpler and faster model selection procedure.

As single-input recurrent networks this way are more straightforward to apply it is naturally of interest to determine how the performance of the recurrent models compare to the performance of a feed-forward network working from an "optimally" chosen lag space and having a comparable number of parameters. Such a comparison will be attempted in this section by generation of learning curves for both the laser series and the Mackey-Glass series when modeling using feed-forward networks. Prior to the learning curve generation it is however necessary to determine the lag space to use.

9.3.1 Feed-forward network lag space selection

In order to determine a proper lag space to use for the feed-forward networks a pragmatic approach was adopted. A "sufficiently" large number of hidden units was chosen and networks working from an increasing number of external inputs were trained on a training set using the damped Gauss-Newton method combined with a small weight decay until the stopping criterion $\|\mathbf{g}\|_2 < 10^{-4}$ was satisfied. The error on a test set was then computed for all the trained networks and the dimension of the "optimal" lag space was then determined as the *smallest* number of inputs for which the smallest test error was obtained. Naturally, this method for lag space determination is somewhat crude as it does not allow for omission of intermediate lags.

For the feed-forward networks trained on the laser series a number of 12 hidden units was chosen. Setting the desired value of the network output at time t to $d(t) = x(t+1)$ a lag space $\mathbf{x}(t) = [x(t), x(t-1), \ldots, x(t-L+1)]$ of increasing dimension L was investigated. The training set comprised the first 7000 points of the laser series, i.e., a fairly large training set in order to reduce the effect of degrading performance as the lag space was increased. Expanding the lag space while keeping the number of hidden units fixed increases the

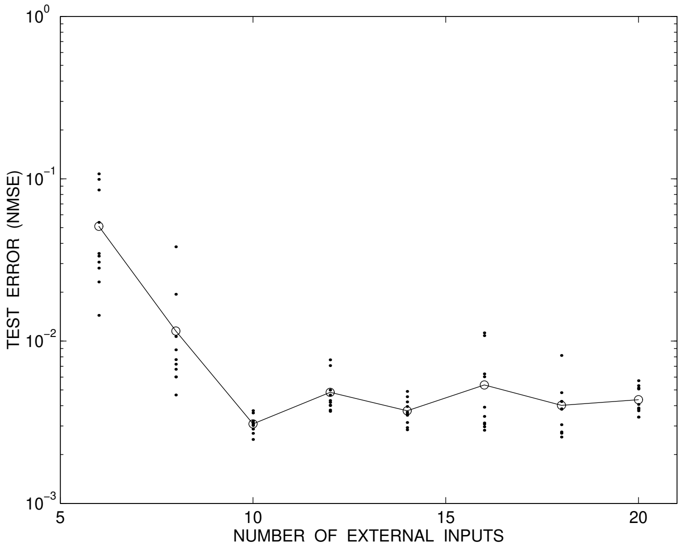


Figure 9.6: Lag space selection for feed-forward network trained on the Santa Fe laser series. The networks have 12 hidden units and training is performed on the first 7000 points of the series. The optimal number of inputs is determined to $L = 10$.

number of parameters which needs to be adapted; using too few training examples might influence the choice of proper lag space. Lag space dimensions ranging from $L = 6$ to $L = 20$ in steps of two were examined and ten networks were trained for each lag space dimension. The resulting test errors computed on the last 3093 points of the laser series are illustrated in Figure 9.6. Dots indicate the individual network errors and the connected circles denote average values. The optimal lag space was determined to be $L = 10$ as this was the smallest lag space dimension having a low error; in fact it was the lag space

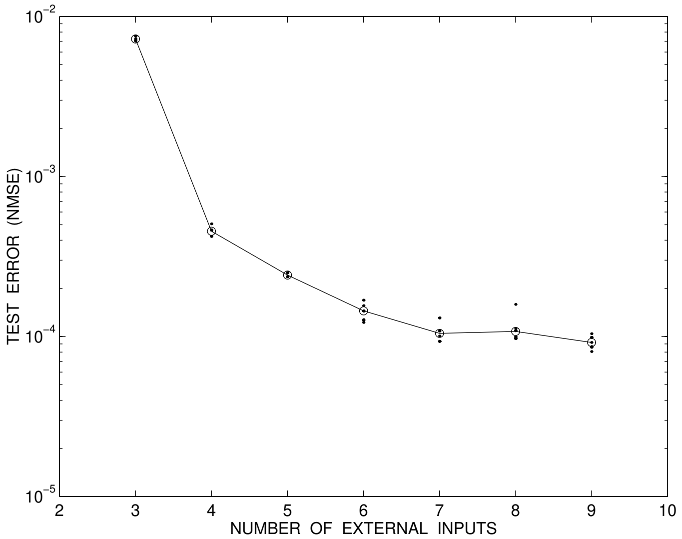


Figure 9.7: Lag space selection for feed-forward network trained on the Mackey-Glass series. The networks have 10 hidden units and training is performed on the first 1500 points of the series. The optimal number of inputs is determined to $L = 7$.

dimension for which the lowest error was obtained. The networks comprised 145 weights and the estimated generalization error was $\widehat{G}(\widehat{\mathbf{w}}) = 0.0031 \pm 0.0004$.

For the feed-forward networks trained on the Mackey-Glass series a number of ten hidden units was chosen as this number of hidden units was successfully used in [PHL96] included in appendix H and was reduced further by pruning without degrading performance. Implementing a six step ahead predictor, the desired value of the network output at time t was set to $d(t) = x(t+6)$ and the lag spaces investigated were obtained as $\mathbf{x}(t) = [x(t), x(t-6), \ldots, x(t-6(L-1))]$ for increasing L, according to standard practice; refer to appendix A.2. Lag space dimensions ranging from $L = 3$ to $L = 9$ were examined and ten networks were trained on the first 1500 points of the Mackey-Glass series for each lag space dimension. The resulting test errors on the 7000 example test set are illustrated in Figure 9.7. The performance seems to stabilize when applying seven or more external inputs to the network. Consequently the optimal lag space was determined to $L = 7$. The corresponding networks comprised 91 weights and the estimated generalization error was $\widehat{G}(\widehat{\mathbf{w}}) = 1.0 \cdot 10^{-4} \pm 1.0 \cdot 10^{-5}$.

9.3.2 Feed-forward network learning curves

After choosing the lag spaces, learning curves were generated for feed-forward networks which could be compared to the recurrent networks described above. For the feed-forward networks trained on the laser series a lag space $L = 10$ was found appropriate; as the recurrent networks used for the generation of the laser series learning curve comprised 131 weights, a number of 11 hidden units was chosen leading to 133 weights for the feed-forward networks. A learning curve was then generated by the same procedure as was adopted for the recurrent networks. Ten networks were trained for each training set size, extending the training set backwards from point 7000. The resulting learning curve is illustrated in Figure 9.8 and is seen to have a shape very similar to the recurrent network learning curve illustrated in Figure 9.4. Overall the level of the test errors are however seen to be slightly larger than for the recurrent network. The best performance is obtained when using 6500 training examples for which a generalization error of $\widehat{G}(\widehat{\mathbf{w}}) = 0.0030 \pm 0.0004$ is obtained. The performance is seen to degrade slightly when using 7000 examples for training; this might be due to "peculiarities" in this particular realization of the laser series which do not affect the recurrent network in the same way as it does the feed-forward network.

The learning curve generated for the Mackey-Glass series using feed-forward networks is illustrated in Figure 9.9. A lag space of $L = 7$ was found appropriate above and the number of hidden units was then set to ten leading to 91 weights in the feed-forward networks in order to match the 89 weights of the recurrent networks used for generation of the Mackey-Glass series learning curve. Ten networks were trained for each of the sampled training set sizes. Note how the variability in the resulting networks is much smaller than for the recurrent networks in Figure 9.5. Further note the rather slow decrease of the expected generalization error as the training set is expanded. Even so, the feed-forward networks reach a performance which is comparable to the recurrent networks; from the learning curve we see that when using 1500 examples for training of the feed-forward network one may expect a normalized generalization error of $\widehat{G}(\widehat{\mathbf{w}}) = 1.0 \cdot 10^{-4} \pm 2 \cdot 10^{-5}$. As the feed-forward learning curve has not completely flattened out at 1500 training examples it seems reasonable to belive that the performance would improve beyond the *particular* recurrent networks considered above if more examples were used for training.

From the above experiments we learn that for the problems considered here a com-

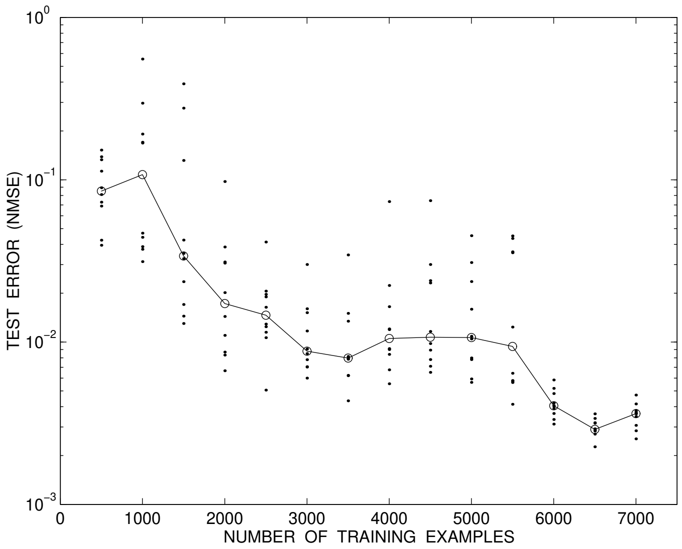


Figure 9.8: Learning curve for *feed-forward* network having 10 external inputs and 11 hidden units (133 weights) trained on an increasing number of examples from the Santa Fe laser series. The expected normalized generalization error when training on 6500 examples is estimated to $\widehat{G}(\widehat{\mathbf{w}}) = 0.0030 \pm 0.0004$, slightly more than for the recurrent networks in Figure 9.4.

parable performance may be obtained for recurrent and feed-forward networks. The performance of the recurrent networks on the laser series was slightly better than for the feed-forward networks. For the Mackey-Glass series the performances of the two model

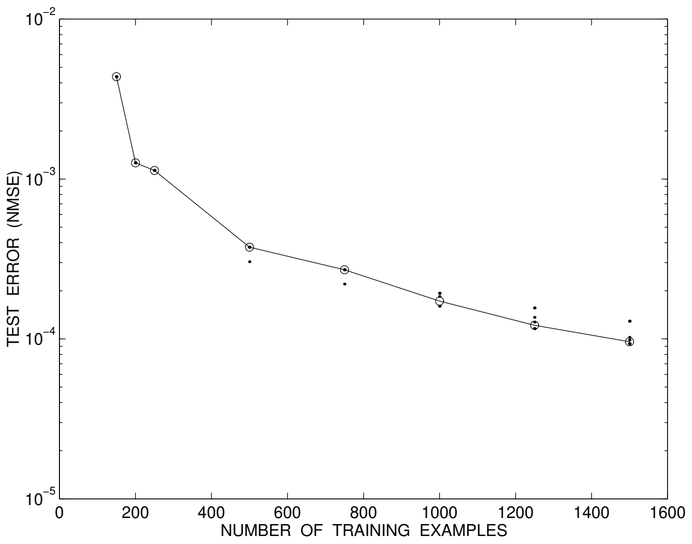


Figure 9.9: Learning curve for *feed-forward* network having 7 external inputs and 10 hidden units (91 weights) trained on an increasing number of examples from the Mackey-Glass series. The expected normalized generalization error when training on 1500 examples is estimated to $\widehat{G}(\widehat{\mathbf{w}}) = 1.0 \cdot 10^{-4} \pm 2 \cdot 10^{-5}$, which is about equal to the recurrent networks in Figure 9.5. Note the slow decrease in test error as the training set is expanded.

types were almost identical, even though it seems that the feed-forward networks would be slightly better than the recurrent networks if more examples were used for training. However, the recurrent networks still have the advantage in this comparison as the adaptive memory allows this model structure to work from only a single input, thus reducing the problems of model selection considerably. Further, the difference might be eliminated by a fine tuning of the weight decay for the recurrent networks.

9.4 Simulating the dynamics

The recurrent networks considered in this chapter have been trained to perform so-called direct prediction of the time series for a fixed time step into the future, i.e., to function in "open-loop" as illustrated in Figure 2.7. Even so it is of interest to investigate the ability of the networks to perform "closed-loop" predictions, i.e., to predict multiple time steps into the future by feeding the predicted values from the network output back to the input as it was done for feed-forward networks in e.g., [WHR90, WHR92, Sva94] and for recurrent networks in e.g., [Wul92].

As the systems underlying both the laser series and the Mackey-Glass series are chaotic in nature we know that the closed-loop iterated network output is bound to diverge from the true series at some point even in the unlikely case of a perfect model, as described in appendix A; this will also be the case in general due to the effect of accumulated prediction errors. Even though the network output diverges from the true series we may however still investigate the extend to which the network is able to *simulate* the dynamics of the underlying system, i.e., the ability to continuously generate points which lie on the attractor of the true system. This may be seen as the extent to which the network has learned the generating dynamics.

When investigating a recurrent network running in closed-loop the first step is to initialize the network dynamics. This is accomplished by "normal" open-loop iterations of the network, applying values of the true series to the external input of the network for a *sufficiently* long sequence (like e.g., the training set). Then the mode of operation is switched to closed-loop iterations, feeding the network output back to the input. When determining the extent to which the network is able to simulate the dynamics of the true system, a natural first-hand investigation is visual comparison of the series generated by the network to the true series. If obvious differences in the structure of the series are present, e.g., the network output reduces to a constant value, this may be taken as evidence that the network is *not* able to simulate the dynamics of the underlying system and further investigation is unnecessary. If the series "look alike" the next step might be to embed the network outputs in delay space and create a phase space plot. The resulting geometrical structure is then compared by visual inspection to the attractor resulting from embedding of the true series; appendix A provides illustrations of the attractors for the chaotic systems considered in this work. If the recurrent network is able to generate a structure which is similar to the true attractor one may proceed to quantize the similarity of the attractors by comparing their attractor dimensions; refer to appendix A for a description.

In the following an investigation of the recurrent network dynamics for selected networks trained on the laser series and networks trained on the Mackey-Glass series will be performed as described above.

9.4.1 Laser dynamics

All the networks resulting from the generation of the laser series learning curve illustrated in Figure 9.4 were examined for their ability to simulate the dynamics of the system which generated the series. The dynamics of each network was initialized by open-loop iterations on the respective training series and closed-loop iterations were then commenced at various points in the test set which extends beyond point 7000 in the laser series; refer to Figure 9.3.

It turned out to be very problematic for the networks to handle the "collapses" of the laser series as illustrated in Figure 9.10. Here, the dynamics of a network trained on 7000 points was initialized on the first 7100 points of the laser series after which closed-loop iterations were commenced, leading to the displayed series. It is seen that the network output is in accordance with the true series until the collapse is encountered. The network "misses" the exact location of the collapse in the true series but generates a collapse shortly after. As seen from Figure 9.10 the network does not recover from the collapse but enters a degenerate mode of constant output. The vast majority of the networks entered a similar mode of constant output after generation of the first collapse and the few networks able to "survive" a collapse were only capable of generating a few "periods" of the laser series before entering a mode of constant output as well. Consequently, examination of the closed-loop iterated network outputs revealed that *none* of the networks in the learning curve displayed in Figure 9.4 were able to properly simulate the dynamics underlying the laser series.

Examination of the network outputs revealed that the number of time steps into the future which a network performing closed-loop iterations was able to provide good predictions was highly influenced by the position in the series at which closed-loop iterations were commenced. Naturally, starting iterations just before a collapse would lead to very few accurate predictions as the network would enter a mode of constant output shortly

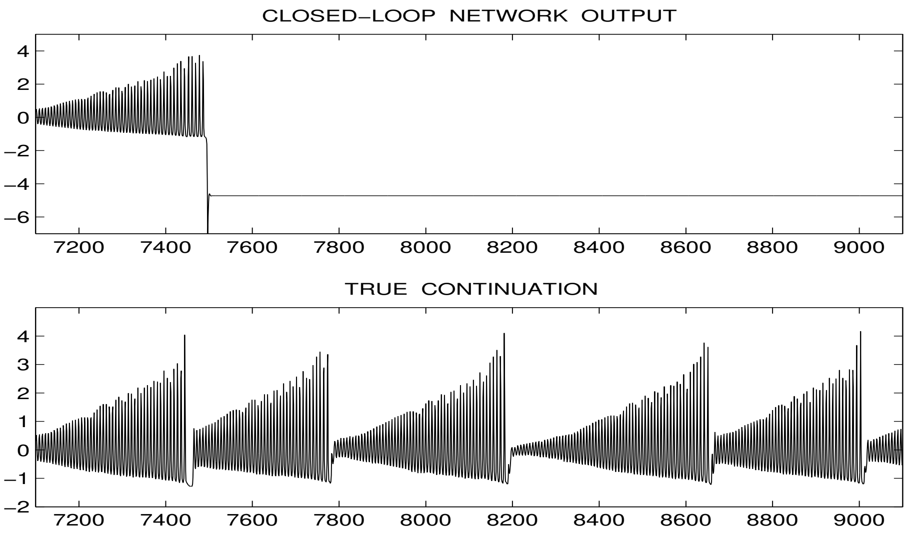


Figure 9.10: Upper panel: Closed-loop iterated outputs from a recurrent network from the laser series learning curve trained on 7000 points. The network is unable to handle a collapse of the series. Lower panel: True continuation of the laser series.

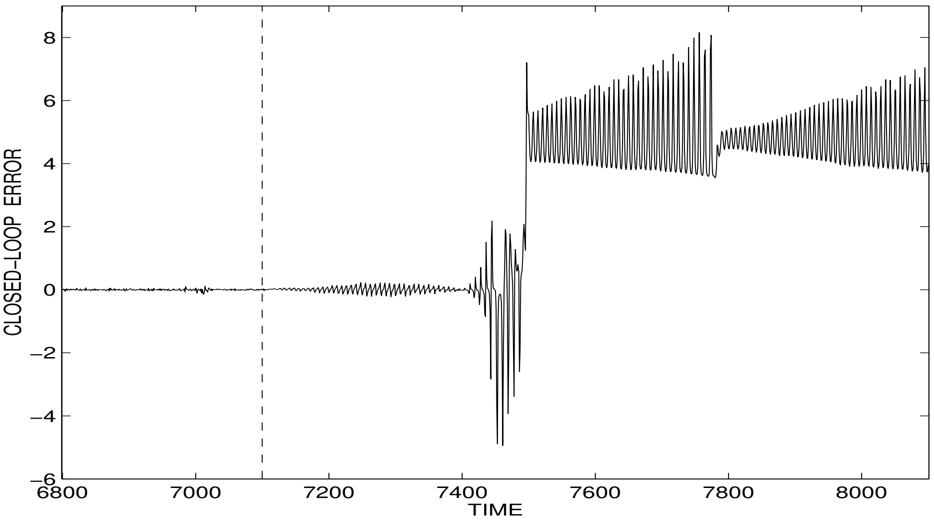


Figure 9.11: Prediction errors for the closed-loop network outputs displayed in Figure 9.10. The dashed vertical line denotes the point at which the closed-loop iterations commense, prior to this point the network is iterated in open-loop. Note the small closed-loop prediction errors for more than 300 time steps.

after. However, when starting the closed-loop iterations far from a collapse the number of iterations before the prediction error would "explode" could vary greatly by offsetting the starting point by just a few time steps; this corresponds to starting the closed-loop iterations at different points on the laser attractor illustrated in Figure A.4.

Figure 9.11 illustrates the closed-loop iterated prediction errors that resulted from the network outputs displayed in Figure 9.10. The vertical dashed line indicates the starting point of the closed-loop iterations; prior to this point the network was iterated in open-loop. It is seen that fairly accurate predictions of the laser series are provided for more than 300 iterations before the error explodes. Shortly after, the network enters the mode of constant output and the prediction errors become an offset of the true series. This illustrates that it is possible to obtain accurate long-term predictions of the laser series from the recurrent networks as long as the predictions do not extend past a collapse.

A systematic method for determination of the number of time steps one may expect reliable closed-loop iterated predictions was indicated in [WHR90, WHR92]. Here, closed-loop iterations were commenced at *each* point in a test series, allowing for estimation of the expected error after, say, n iterations. Calculating these errors for increasing n allows for determination of the expected number of iterations n' before the prediction error explodes, i.e., exceeds a specified tolerance. This number of iterations may be denoted as the *iterated prediction horizon* of the network. Such analysis may be seen as a tool for characterizing the dynamic properties of a trained recurrent network but has unfortunately not been pursued in great detail in this work.

Even though it was not possible for any of the networks of the laser series learning curve to simulate the underlying dynamics it *is* in fact possible for a single-input fully recurrent network to simulate these dynamics. This was revealed when performing a series of experiments not included in this thesis. Figure 9.12 illustrates the closed-loop

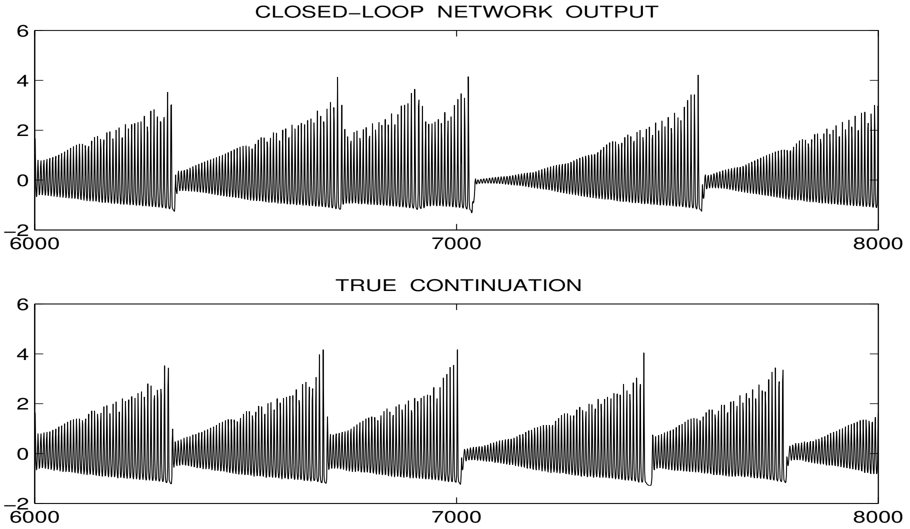


Figure 9.12: Upper panel: Closed-loop iterated outputs from a recurrent network having 14 hidden units and trained on the first 6000 points of the laser series. This network is able to simulate the dynamics which generated the laser series. Lower panel: True continuation of the laser series.

iterated output from a single-input network having 14 hidden units (239 weights), trained on the first 6000 points of the laser series. It is seen that the network is indeed capable of "surviving" the collapses. By visual comparison of the generated series to the true laser series it is seen that the "structure" of the series seems to be in accordance with

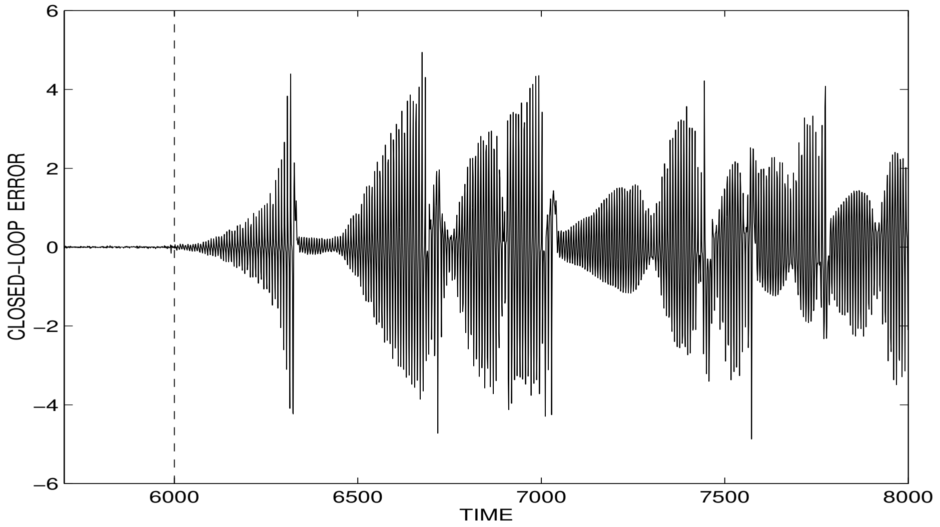


Figure 9.13: Prediction errors for the closed-loop network outputs displayed in Figure 9.12. The dashed vertical line denotes the point at which the closed-loop iterations commense, prior to this point the network is iterated in open-loop. Note the rapid increase in the prediction errors compared to Figure 9.11.

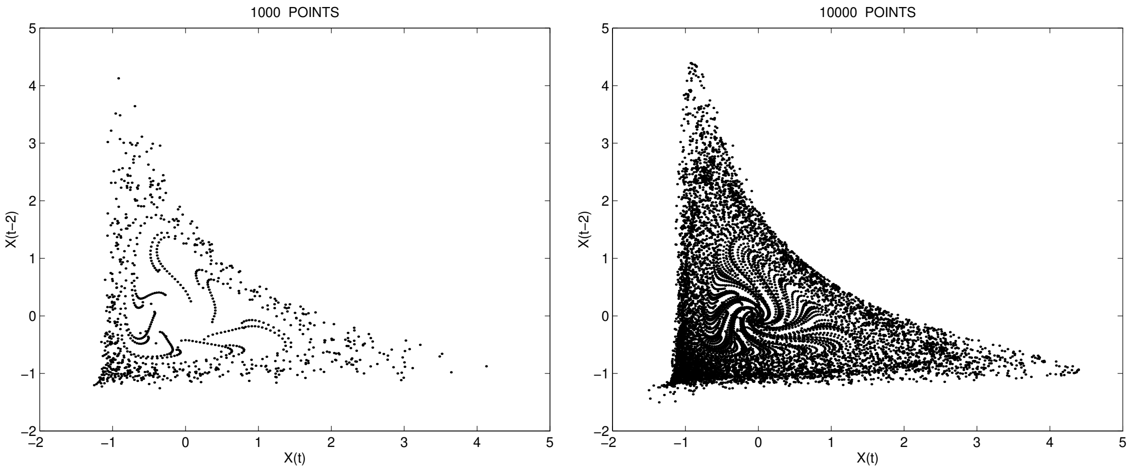


Figure 9.14: Phase space plots for the laser data series generated by a recurrent network running in closed-loop. Compare to the true attractor illustrated in Figure A.4 on page 219. Left panel: 1000 points. Right panel: 10000 points.

each other even though they are not exactly alike; Figure 9.13 displays the closed-loop prediction errors which indicates that the two series rather quickly diverge.

The series generated by the recurrent network was then embedded in two dimensions, using a delay time $\tau = 2$ as was done for the true laser series in appendix A.1.1. The resulting points were plotted in a phase space plot as illustrated in Figure 9.14; this figure should be compared to Figure A.4 on page 219 which illustrates the phase space plot of the true laser series. By visual inspection it seems that the network attractor has a structure very similar to the true attractor. During the first 1000 closed-loop iterations the network does not produce a collapse after which the oscillations start at zero as seen from Figure 9.12; this explains the "hole" in the center of the attractor in the left panel of Figure 9.14. When including 10000 points in the phase space plot as in the right panel of Figure 9.14 the "wheel with spokes"-structure however becomes clearly visible, in fact more clearly visible than in the attractor illustrated for the true series in the right panel of Figure A.4.

In order to provide a more quantitative comparison of the attractors the correlation dimension D_2 was estimated for the network attractor in the same way as it was done for the true attractor in Figure A.3 on page 218. In order to provide as unbiased a comparison as possible, 10093 points from the network output were used for the dimension estimation, the same number as for the true series. The result is illustrated in Figure 9.15. The correlation dimension seems to saturate at embedding dimension $L = 5$ and the dimension of the network attractor is estimated to $D_2 \approx 2.11$ by averaging the values obtained at $L = 5, 6, 7, 8$. This value is not quite equal to the value which was estimated for the true attractor ($D_2 \approx 2.21$) but considering the relatively few number of points from which the dimensions are estimated as well as the sampling noise present in the true series it seems fair to state that the dimensions are approximately equal and that the network has learned the underlying dynamics.

As a final remark on laser series network dynamics it is mentioned that all of the very few networks found capable of simulating the dynamics which generated the laser series had 14 hidden units, 4 more than was used for the networks when generating the learning

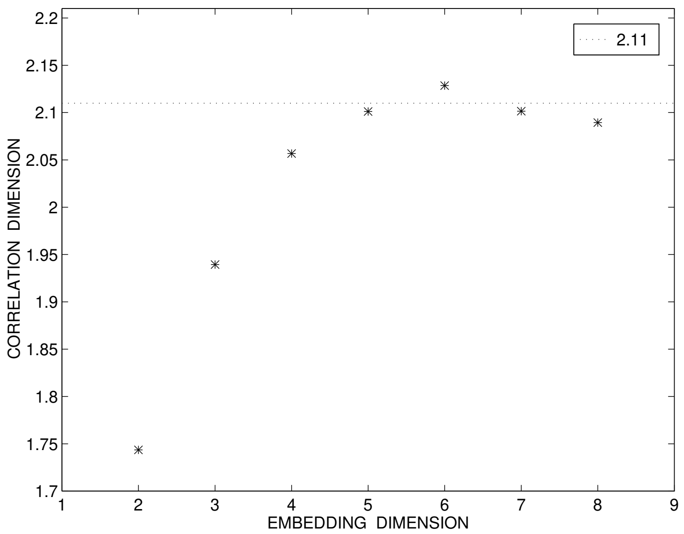


Figure 9.15: Estimation of the correlation dimension D_2 for the attractor generated by a recurrent network trained on the laser series.

curve in Figure 9.4; apparently the learning curve networks did not possess sufficient complexity. Furthermore it turned out that *every* network capable of simulation exhibited severe overtraining, i.e., the average one-step ahead prediction error on the test set was very large for all of those networks. In the experiments performed during this work *none* of the networks with a low one-step ahead test error have been found capable of simulating the dynamics generating the laser series. An explanation for this phenomenon has not been found.

9.4.2 Mackey-Glass dynamics

We now turn towards the networks resulting from the generation of the Mackey-Glass series learning curve illustrated in Figure 9.5. It turned out that learning the dynamics which generated the Mackey-Glass series is apparently an easier task than learning the dynamics of the laser series as all of the networks examined having test error $\approx 10^{-4}$ were capable of simulating the dynamics underlying the Mackey-Glass series. Figure 9.16 displays the closed-loop iterated output from one of the networks in the learning curve trained on 1500 points along with the true series; the solid vertical line at point 1500 indicates the beginning of the test set at which the closed-loop iterations were commenced. The closed-loop predictions are seen to be fairly accurate for about 200 time steps after which the true series and network generated series diverge. The corresponding closed-loop prediction errors are illustrated in Figure 9.17.

A relevant comment to make at this point is that the network has been trained to implement a six-step ahead predictor and does therefore assume no knowledge of the five values in the series immediately preceeding the value to be predicted. Consequently, when performing closed-loop iterations the network output should not be fed directly back to the input but rather pass through a delay line which delays the network output six time steps before it is applied to the input.

A series consisting of 8500 points generated by the network was embedded in three

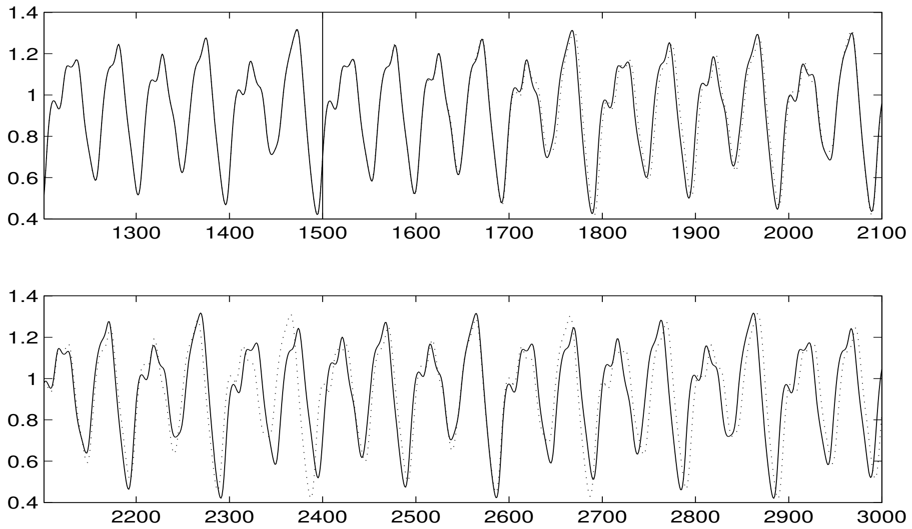


Figure 9.16: Comparison of the outputs of a recurrent network from the Mackey-Glass series learning curve trained on 1500 points when performing closed-loop iterations to the true Mackey-Glass series. The solid vertical line at time step 1500 indicates the start of the closed-loop iterations. Dotted line: Network output. Solid line: True series.

dimensions using a delay time $\tau = 6$ as was done for the true series in appendix A.2.1. The resulting points were then plotted in a phase space plot which is illustrated in Figure 9.18. Visual comparison of the network attractor to the true Mackey-Glass attractor illustrated in Figure A.7 reveals that the two structures are almost identical. In fact, only careful inspection reveals the subtle differences between the two attractors.

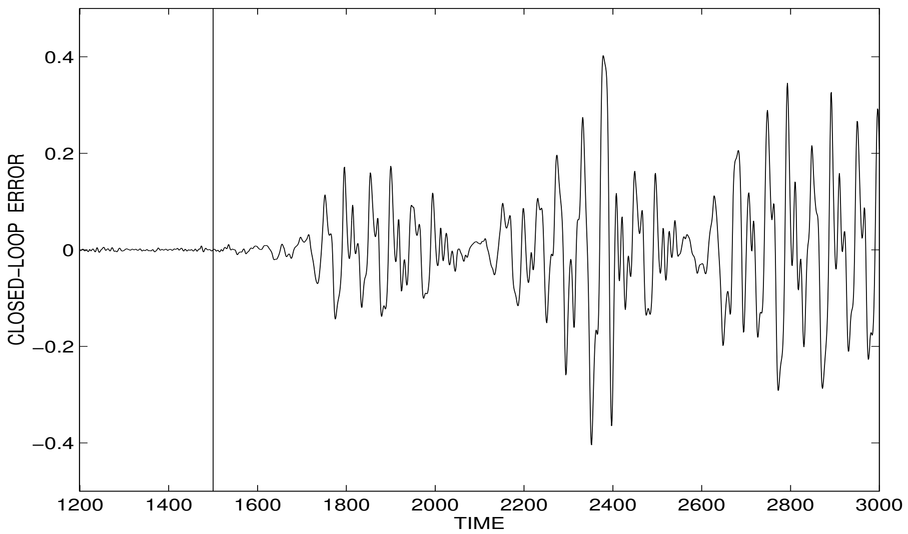


Figure 9.17: Prediction errors for the closed-loop iterated network outputs displayed in Figure 9.16. The solid vertical line denotes the point at which the closed-loop iterations commense, prior to this point the network is iterated in open-loop. Note the small prediction errors for about 200 iterations.

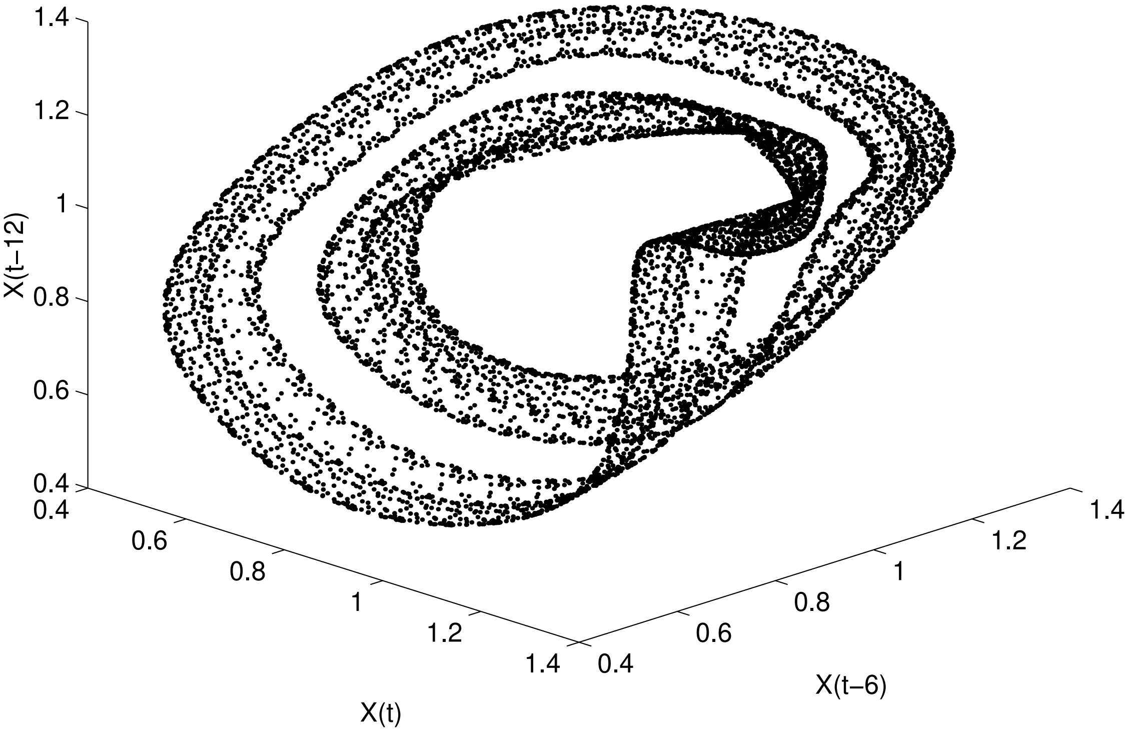


Figure 9.18: Phase space plot for the Mackey-Glass series using 8500 points generated by a recurrent network running in closed-loop. Compare to the true attractor illustrated in Figure A.7 on page 221.

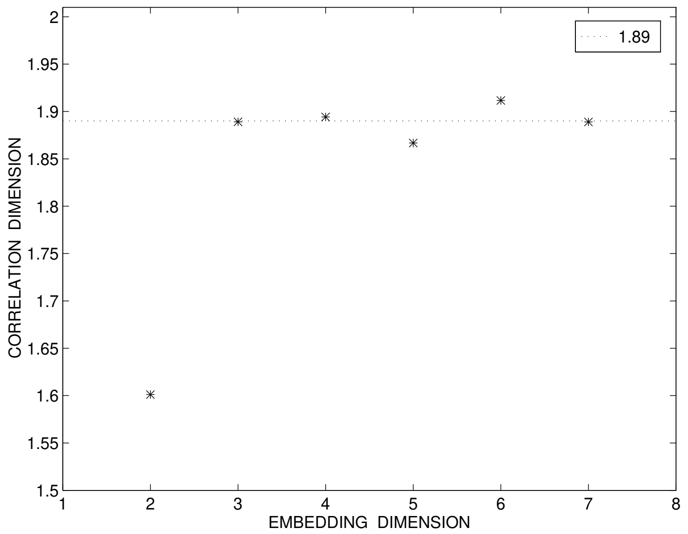


Figure 9.19: Estimation of the correlation dimension D_2 for the attractor generated by a recurrent network trained on the Mackey-Glass series.

In order to provide further comparison of the attractors the correlation dimension D_2 was estimated for the network attractor as it was done for the true Mackey-Glass attractor in Figure A.6 on page 220. As for the laser series above the network attractor dimension was estimated from the same number of points as was used for the true attractor in order to avoid a bias in the comparison due to a different number of points. Thus the network attractor dimension was estimated from an embedding of the 8500 points used to generate Figure 9.18. The result is illustrated in Figure 9.19. As for the true attractor the dimension estimate saturates at embedding dimension $L = 3$ and the network attractor correlation dimension is estimated to $D_2 \approx 1.89$ by averaging the values obtained at embedding dimensions $L = 3, 4, 5, 6, 7$. This dimension estimate is very close indeed to the dimension estimate of the true attractor ($D_2 \approx 1.92$). The small difference is probably due to the slight tendency to lacunarity at some parts of the network attractor, compared to the true attractor. The almost identical attractor dimensions combined with the visually verified identity of the two attractors may be taken as evidence that the recurrent network has indeed learned the Mackey-Glass dynamics.

Chapter 10

Illustration of recurrent network pruning

The literature is rich on examples where various pruning schemes have been applied to *feed-forward* networks (e.g., [CBD$^+$90, Tho91, GHK$^+$93, SHL93, SHLR93, HP94, HR94, HMPHL96, PHL96]) with success in terms of improved generalization ability and significantly reduced network architectures. Even though architecture optimization by pruning extends naturally to *recurrent* networks the literature is very sparse on examples. Among the examples is [GO94] where entire units are pruned away, based on a magnitude based criterion calculated from the vector of weights entering the unit. A similar method is adopted in [CFP94]. Very few attempts to employ pruning methods like OBD and OBS, based on weight saliencies defined in terms of training error (refer to chapter 6), have been reported. One of the few is [LGHK97] where saliencies are determined for each *memory order* of an output feedback network (see Figure 3.5) in a manner similar to combining several weight saliencies in the OBD pruning scheme.

In this work pruning of fully recurrent networks by the OBD and OBS schemes has been investigated. Tentative results using the OBS pruning scheme were presented in [PH95] included in appendix G. This chapter serves as an elaboration on the viability of recurrent network pruning by OBD and OBS. Section 10.1 demonstrates pruning by the OBD scheme on networks selected from the Santa Fe laser series and the Mackey-Glass series learning curves. It has been observed that saliencies computed using the OBS scheme often severely underestimate the actual change in error which results from pruning a particular weight. This leads to pruning of weights which are important to the network. In contrast, the saliency estimates computed using the OBD scheme are much more consistent with the actual change in error. In section 10.2 the quality of both OBD and OBS saliencies is assessed by comparison to the *actual* saliencies. Furthermore, the accuracy of the second-order expansion is illustrated.

10.1 Pruning by Optimal Brain Damage

Pruning of recurrent networks by OBD will now be demonstrated for selected networks from various parts of the learning curves which were described in chapter 9. Generally, the success in terms of improved generalization ability to expect from pruning depends on the *variance* of the generalization errors for the particular size of training set. The larger the variance, the larger the potential improvement from pruning as the ability to overfit is reduced, as may also be seen from Eq. 6.1. I.e., in order to benefit from pruning the potential decrease in the variance term should exceed the resulting increase in the bias term.

If training is performed using a training set size residing on the the initial part of the learning curve (small number of examples) then the potential improvement in generalization error is large due to a large variance. However, it cannot be expected that the generalization error reaches the lowest level of the learning curve as there is just too few examples available in order to properly learn the underlying dynamics.

On the other hand, if the size of the training set resides on the "flat" part of the learning curve (large number of examples) then the potential improvement in generalization error is smaller due to a small variance. In this case pruning is however a valuable tool for assessing the architecture of a minimal model still able to implement the teacher function for the problem at hand.

In order obtain a stopping criterion for the pruning procedure one might set aside data for a *validation set* [Bis95, Rip96] (different from the independent test set on which the generalization error is estimated) and then select the network having the lowest error on the validation set as the "optimal" network. In order to avoid using data for a validation set an alternative is to employ the FPE-estimate, described in section 5.3. This was done for feed-forward networks in e.g., [SHL93, HRSL94, PHL96, HMPHL96] where the generalization error was estimated by the FPE-estimate for each of the networks which resulted during pruning. By selecting the network with the lowest FPE-estimated generalization error a stopping criterion for the pruning procedure was obtained. This approach has also been attempted here for recurrent networks, even though the FPE-estimate has not been formally justified for this network type; refer to section 5.3.

These general considerations will be investigated for recurrent networks in the following. The experiments illustrate how pruning helps to utilize the available training set in the best possible way. Note that only *one* weight is pruned at a time.

10.1.1 Pruning of RNNs applied to the laser series

Pruning is here illustrated for two networks selected from the laser series learning curve displayed in Figure 9.4 on page 112. The first network is selected among the networks resulting when training on 7000 examples.

The *training curve* for the fully connected network, i.e., the evolution of the training and test errors as training iterations progress are illustrated in the upper panel of Figure 10.1. Note the rapid decrease of the errors after the training method shifts to the Gauss-Newton method after 50 initial iterations of gradient descent. The long "tail" where the errors are changing very little indicates a fine tuning of the weights in order to satisfy the stopping criterion, $\|\mathbf{g}\|_2 \leq 10^{-4}$, i.e., to get "close" to a local minimum. Verifying that a small gradient is obtained is important for the OBD pruning scheme as the saliency estimates assume that the first order term can be neglected when expanding the cost function to second order. Finally note that very little overtraining occurs due to the large size of the training set.

The evolutions of the Normalized Mean Squared Errors (refer to appendix A) as weights are pruned away are illustrated in the lower panel of Figure 10.1; the errors displayed are the ones resulting *after retraining* of the reduced networks by the damped Gauss-Newton method to a new local minimum of the cost function. The estimated generalization error on the test set exhibits a fairly modest decrease as parameters are pruned due to the large

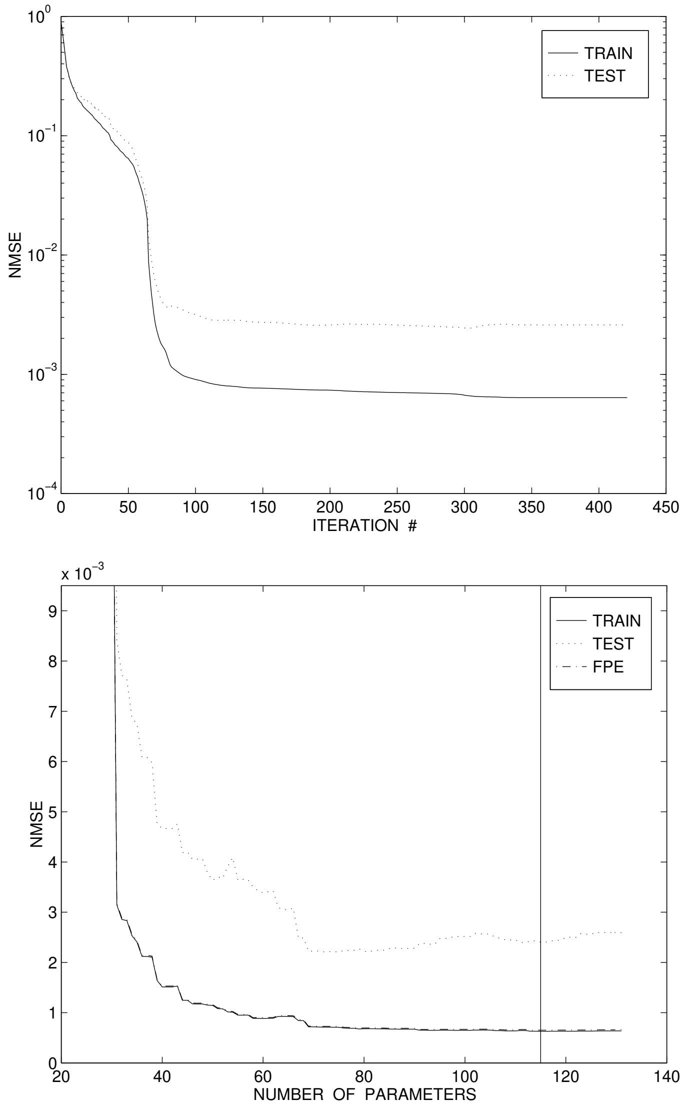


Figure 10.1: Upper panel: Training curve for fully connected network trained on 7000 examples from the laser series. Lower panel: Evolution of training, test and FPE errors during pruning. The solid vertical line indicates the minimum value of the FPE-estimate. Estimated generalization errors on test set: $\widehat{G}(\widehat{\mathbf{w}}) = 2.6 \cdot 10^{-3}$ for the fully connected network, $\widehat{G}(\widehat{\mathbf{w}}) = 2.2 \cdot 10^{-3}$ for the network having 72 weights.

number of training examples, in line with the discussion above. The test error reaches its minimum when 72 weights are left in the network, after which it increases due to a significantly increased model bias. Pruning is improving the estimated generalization error from $\widehat{G}(\widehat{\mathbf{w}}) = 2.6 \cdot 10^{-3}$ for the fully connected network to $\widehat{G}(\widehat{\mathbf{w}}) = 2.2 \cdot 10^{-3}$ for the network having 72 weights, a 15 % reduction in error and a 45 % reduction in the number of weights. Similar results were obtained for the nine other networks in the learning curve trained on 7000 points of the laser series.

The FPE-estimate defined by Eq. 5.15 is here attempted as a stopping criterion for the pruning procedure. The vertical line in the lower panel of Figure 10.1 indicates the

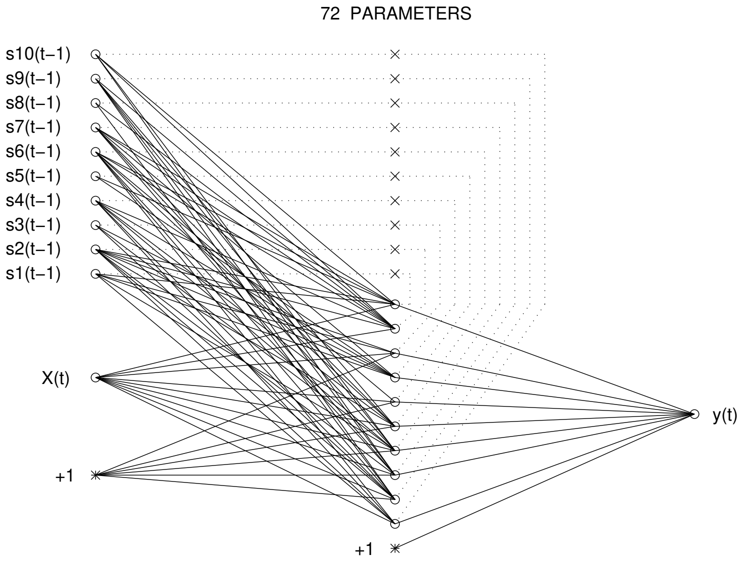


Figure 10.2: Structure of recurrent network trained on 7000 examples from the laser series and pruned to 72 weights (refer to text for interpretation).

network for which the generalization error according to FPE is minimal. Whereas FPE selects a network having 115 weights it is seen that the network with the smallest estimated generalization error on the large test set contains only 72 weights.

The FPE-estimate is in the lower panel of Figure 10.1 seen to follow the training error very closely. This is due to the large number of examples used for training. In comparison the effective number of parameters in the network is small, and the prefactor in Eq. 5.15 is therefore close to unity. This implies that the network selected by the FPE-estimate is always very close to or identical to the network having the smallest error on the training set, which is in general not the network with the best generalization ability. As the network having the smallest training error is often "close" to the fully connected network, this means that the FPE-estimate has a tendency to choosing networks containing too many weights which is also the the experience obtained in this work. Consequently, the FPE-estimate cannot be recommended as a general stopping criterion for pruning.[1]

The architecture of the network containing 72 parameters is illustrated in Figure 10.2; this network may be viewed as the smallest network still capable of accurately modeling the laser series. Solid lines in the figure indicate the remaining weights and the dotted lines indicate the feedback paths; '×' symbolizes a delay of one time step. It is seen that most of the pruned weights are feedback connections, weighting previous hidden unit outputs. However, no hidden unit has been entirely pruned away. This might be due to the fact that the entire memory of single-input recurrent networks is located in the hidden units; pruning of a whole unit may consequently hamper the network memory and thus modeling ability significantly.

It has been investigated whether the pruned network illustrated in Figure 10.2 is better capable of simulating the underlying dynamics than the fully connected network. However, the result was similar to Figure 9.10 on page 119.

[1]Note that these general observations apply to feed-forward networks as well.

The second network considered from the laser series learning curve in Figure 9.4 is selected among the networks trained on 5500 examples. The chosen network is among those having the largest error on the test set as these networks are likely to benefit the most from pruning.

The training curve for this network is illustrated in the upper panel of Figure 10.3 and reveals slight overfitting of the training set. The lower panel illustrates the evolution of errors after retraining as weights are pruned away. The network is apparently undergoing significant changes as pruning progresses even though the training error is practically unchanged. The changes are revealed by the error on the test set; the network is suddenly significantly overfitting the training series. As more weights are pruned the ability to

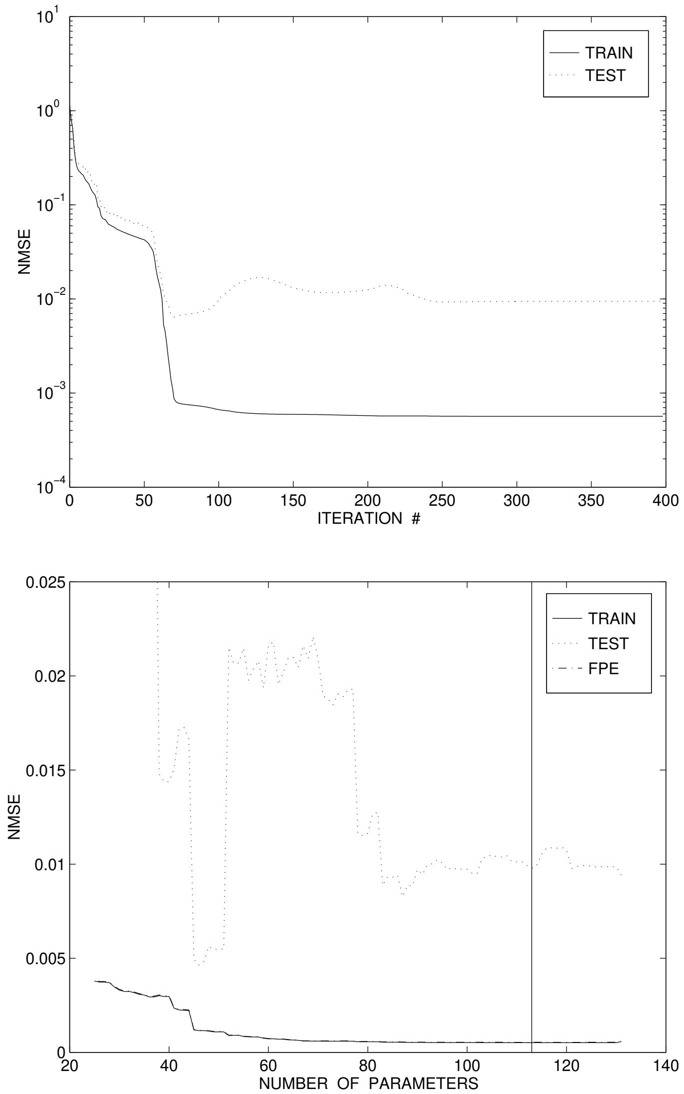


Figure 10.3: Upper panel: Training curve for fully connected network trained on 5500 examples from the laser series. Lower panel: Evolution of training, test and FPE errors during pruning. The solid vertical line indicates the minimum value of the FPE-estimate. Estimated generalization errors on test set: $\widehat{G}(\widehat{\mathbf{w}}) = 9.4 \cdot 10^{-3}$ for the fully connected network, $\widehat{G}(\widehat{\mathbf{w}}) = 4.6 \cdot 10^{-3}$ for the network having 46 weights.

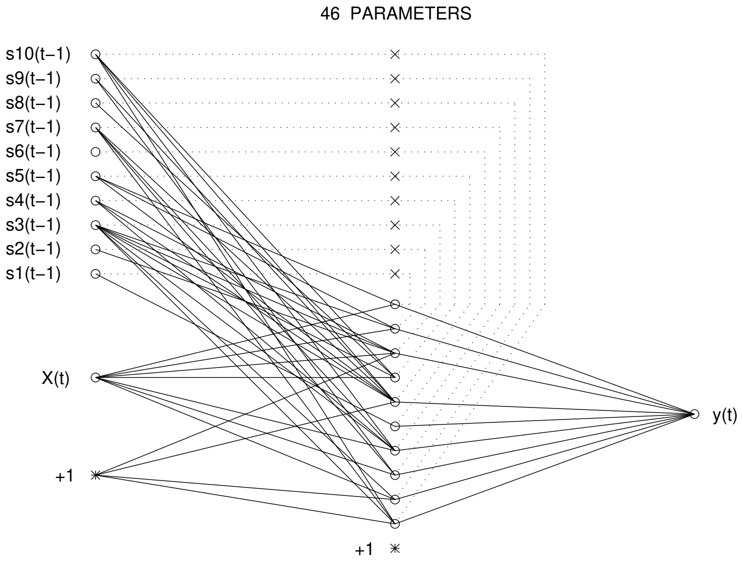


Figure 10.4: Structure of recurrent network trained on 5500 examples from the laser series and pruned to 46 weights.

overfit is eliminated which leads to a dramatic decrease in the test error. Shortly after, the test error increases again due to insufficient model complexity.

In this case pruning is improving the estimated generalization error from $\widehat{G}(\widehat{\mathbf{w}}) = 9.4 \cdot 10^{-3}$ for the fully connected network to $\widehat{G}(\widehat{\mathbf{w}}) = 4.6 \cdot 10^{-3}$ for the network having 46 weights, a 50 % reduction in error and a 65 % reduction in the number of weights. By comparison to the learning curve in Figure 9.4 it is seen that the pruned network now ranks among the best networks obtained when training on 5500 examples. Once more the FPE-estimate is seen to select a network having too many parameters (113 weights) which is close to the network having the lowest training error (123 weights).

The architecture of the network having 46 parameters is illustrated in Figure 10.4. Note the large number of input weights still present and note that *no* hidden units have been discarded even though 65 % of the parameters have been pruned away.

10.1.2 Pruning of RNNs applied to the Mackey-Glass series

We now proceed to consider pruning of networks selected from the Mackey-Glass series learning curve displayed in Figure 9.5 on page 113. The first network is selected among the networks resulting when training on 1500 examples. For these networks there is a very small variance on the resulting test errors; consequently, very little improvement should be expected from pruning. However, pruning may still provide insight into the architecture of the minimal network still able to accurately model the Mackey-Glass series.

The training curve for the selected network is illustrated in the upper panel of Figure 10.5. The training and test errors are seen to have merged completely due to the (relatively) large number of training examples used. The evolution of the errors during the pruning procedure is illustrated in the lower panel of Figure 10.5 and they too are very close throughout the course of pruning. As weights are pruned it is seen that the damped Gauss-Newton retraining method manages to converge to local minima for which

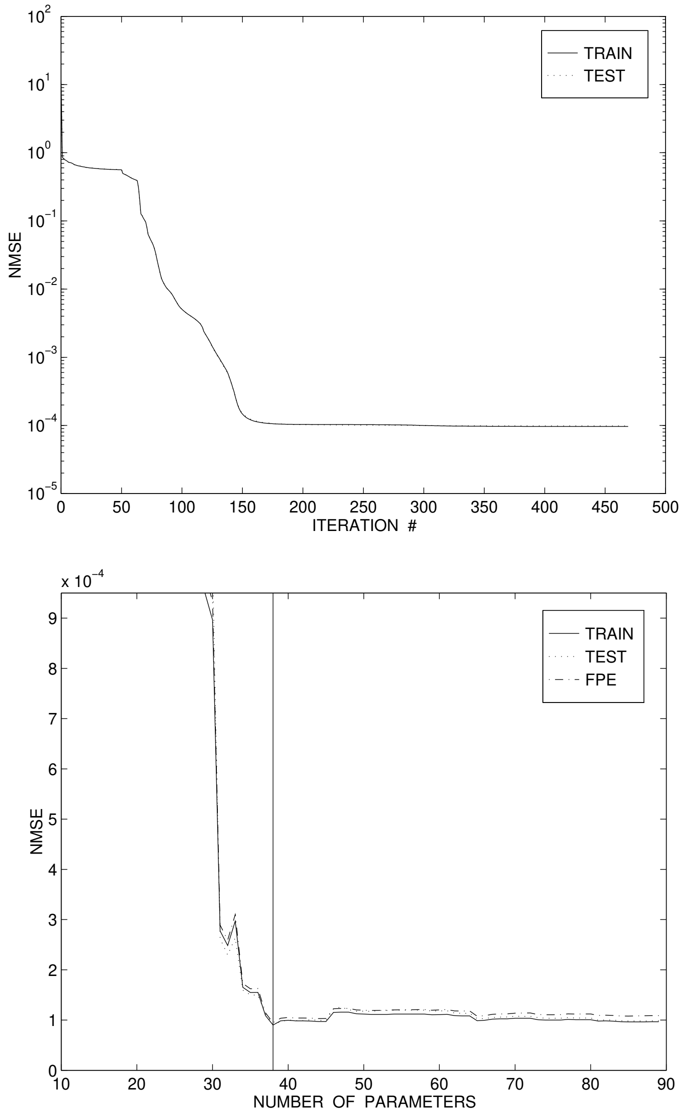


Figure 10.5: Upper panel: Training curve for fully connected network trained on 1500 examples from the Mackey-Glass series. Lower panel: Evolution of training, test and FPE errors during pruning. The solid vertical line indicates the minimum value of the FPE-estimate. Estimated generalization errors on test set: $\widehat{G}(\widehat{\mathbf{w}}) = 9.8 \cdot 10^{-5}$ for the fully connected network, $\widehat{G}(\widehat{\mathbf{w}}) = 9.1 \cdot 10^{-5}$ for the network having 38 weights.

the training error is slightly less than the initial error. The smallest training error and, consequently, the smallest test error is obtained when 38 weights are left in the network. In this case the FPE-estimate manages to point out the optimal network as it coincides with the smallest training error. The estimated generalization error is improved from $\widehat{G}(\widehat{\mathbf{w}}) = 9.8 \cdot 10^{-5}$ for the fully connected network to $\widehat{G}(\widehat{\mathbf{w}}) = 9.1 \cdot 10^{-5}$ for the network having 38 weights, a mere 7 % reduction. However, the number of weights is reduced by 57 %.

The architecture of the network having 38 weights is illustrated in Figure 10.6. Once more it is seen that nearly all the input weights are retained; only one hidden unit has

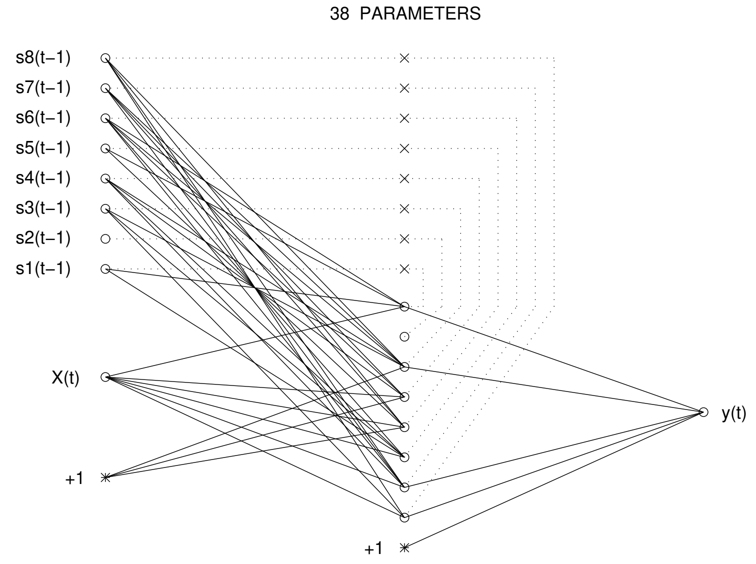


Figure 10.6: Structure of recurrent network trained on 1500 examples from the Mackey-Glass series and pruned to 38 weights.

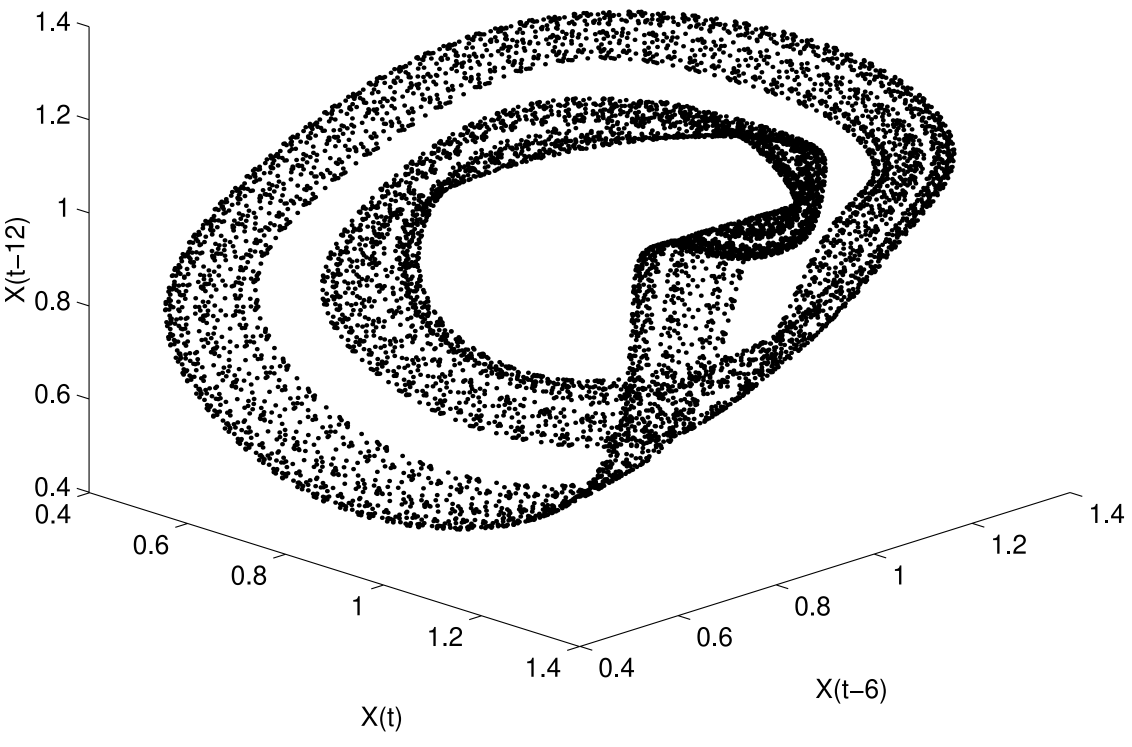


Figure 10.7: Phase space plot for the Mackey-Glass series using 8500 points generated by the network illustrated in Figure 10.6 when running in closed-loop.

been completely discarded despite the large reduction in the number of weights. Whereas the networks applied to the laser series and illustrated above sampled almost all of the hidden units in order to obtain the network output, it is noted that the network output in Figure 10.6 is obtained from relatively few of the hidden units.

The network illustrated in Figure 10.6 is indeed still capable of simulating the dynamics which generated the Mackey-Glass series. After initialization on the training set the network was operated in "closed-loop" feeding the network output back to the input as it was done for the fully connected network in section 9.4.2. This way 8500 points were generated, embedded in three dimensions and visualized in the phase space plot illustrated in Figure 10.7. Comparing to the attractor of the fully connected network in Figure 9.18 on page 125 it is noted that the pruned-network attractor exhibits less lacunarity, i.e., the "band" is more evenly filled with points. Comparing to the true attractor in Figure A.7 on page 221 it seems that the pruned-network attractor has an even greater resemblance than the attractor of the fully connected network.

The final pruning example involves a network from the Mackey-Glass learning curve in Figure 9.5 trained on only 150 examples. This training set size resides on the initial part of the learning curve where the potential improvement in generalization ability resulting from pruning is large.

When training on very few examples it is likely that the network will overfit the training data, leading to a large test error. This is also the case for the network considered here as seen from the training curve in the left panel of Figure 10.8. Initially the test error decreases along with the training error but as the network starts to overfit (shortly after application of the damped Gauss-Newton method which succeeds the initial 50 iterations of gradient descent) the test error increases dramatically.

The evolution of errors as weights are pruned away are illustrated in the right panel of Figure 10.8. Initially the test error is seen to remain fairly unchanged at a high level. As the model complexity is reduced, the ability to overfit is abruptly eliminated leading to a

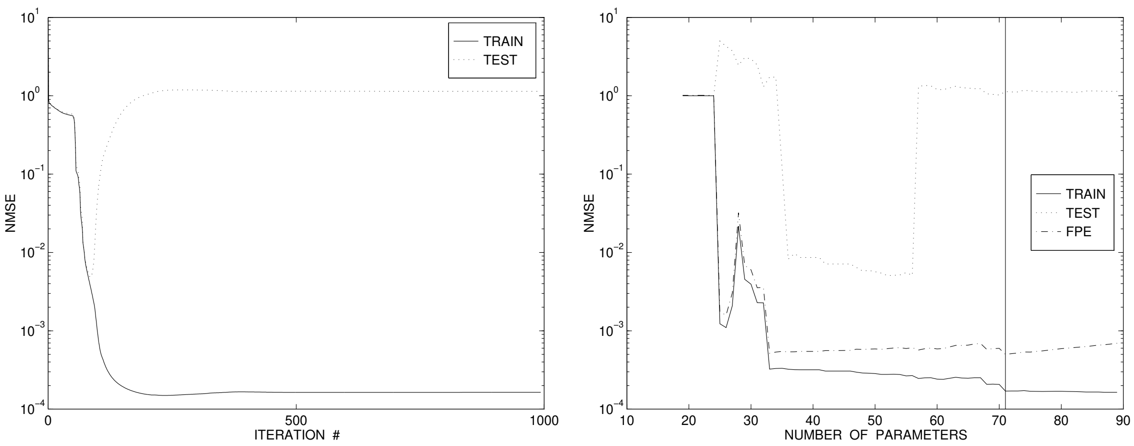


Figure 10.8: Left panel: Training curve for fully connected network trained on 1500 examples from the Mackey-Glass series. Right panel: Evolution of training, test and FPE errors during pruning. The solid vertical line indicates the minimum value of the FPE-estimate. Estimated generalization errors on test set: $\widehat{G}(\widehat{\mathbf{w}}) = 1.1$ for the fully connected network, $\widehat{G}(\widehat{\mathbf{w}}) = 5.0 \cdot 10^{-3}$ for the network having 52 weights.

significant drop in test error. The test error drops to $\widehat{G}(\widehat{\mathbf{w}}) = 5.0 \cdot 10^{-3}$ for the network having 52 parameters which is close to the lowest level $\widehat{G}(\widehat{\mathbf{w}}) = 4.8 \cdot 10^{-3}$ obtained in the left panel of Figure 10.8. Pruning has thus eliminated the capability of overfitting.

Comparing to the learning curve in Figure 9.5 it is seen that the test error obtained from pruning is much smaller than the test error for any of the networks in the learning curve trained on 150 examples. This is due to the remaining networks having severely overfitted as well. Note also that the level of the test error obtained from pruning is much larger than the smallest errors obtained in the learning curve when training on more examples. This is due to the 150 examples not being sufficient in order to properly learn the underlying dynamics.

The FPE-estimate is once more selecting a network having too many parameters and close to the network having the smallest error on the training set. The FPE-estimate is now seen to be more clearly offset from the training error, compared to the previous examples. This is due to the small number of training examples; the changes in the effective number of parameters are now significantly influencing the prefactor in Eq. 5.15 which is therefore initially larger than one. Thus, the smaller size of the training set "bias" the FPE-estimate towards selecting networks with a smaller number of weights and thus closer to the networks which have empirically been found to exhibit optimal generalization ability. However, this practical observation is not in harmony with the derivation of the FPE-estimate outlined in section 5.3 where it is assumed that the number of training examples is *large*. Thus, the application of the FPE-estimate is not properly justified in this case.

The architecture of the network with minimal test error and having 52 weights is illustrated in Figure 10.9. The characteristics are similar to the previous examples as the majority of the pruned weights are feedback connections, weighting previous hidden unit outputs. Almost all of the input weights are still remaining and no hidden units have been discarded. Note the sparse sampling of the hidden units in order to obtain the network

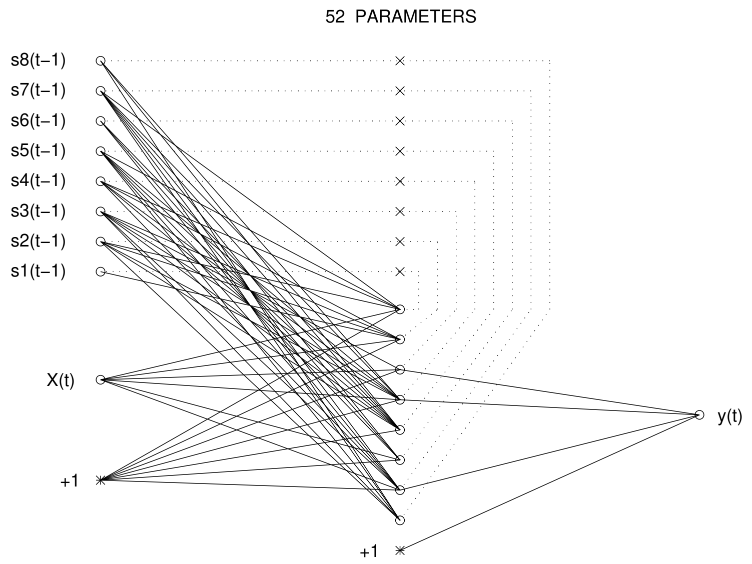


Figure 10.9: Structure of recurrent network trained on 150 examples from the Mackey-Glass series and pruned to 52 weights.

output just as in the previous example.

10.2 Saliency quality: OBD vs. OBS

We will now turn to towards an investigation of the accuracy and thereby the quality of the saliency estimates which results from the OBD and OBS pruning schemes, respectively. Such investigation was presented for *feed-forward* networks in e.g., [GHK+93] as well as in [PHL96] included in appendix H. Here it was found that the OBD saliencies were very accurate and that ranking the weights according to the estimated saliencies was consistent with ranking according to actual saliencies, obtained by setting each weight to zero in turn and computing the resulting change in error. The OBS saliencies, however, were found to highly underestimate the actual change in error resulting from the weight elimination *and* quadratic retraining of the remaining parameters. This underestimation was found to lead to a ranking of the weights according to estimated saliency which was inconsistent with ranking according to actual saliency, and thus to pruning of the wrong weights.

Similar qualitative investigations have been made for recurrent networks, showing the same results: OBD saliencies are generally very accurate, especially for the low-saliency weights in which we are interested, whereas OBS saliencies are generally much smaller than the actual saliencies, leading to elimination of important weights. The difference in accuracy found between OBD and OBS saliencies was initially indicated by a quantitative experiment in [Ped94] involving pruning of numerous recurrent networks by both OBD and OBS. It was found that, on average, OBD pruned the weight having the smallest actual saliency in 90 % of the weight eliminations whereas OBS pruned the correct weight in only 15 % of the eliminations.

During this work many experiments have been made involving pruning of recurrent networks by OBD and OBS, both methods starting from the same initial networks. The experience obtained from these experiments is that pruning by OBS rarely leads to an improvement in generalization ability whereas the results presented in the previous section have been found to be representable for the OBD pruning scheme. The reason for this difference in performance is rooted in the quality of the saliencies. In the following a qualitative comparison between OBD and OBS saliencies will be illustrated for a network trained on the Mackey-Glass series and pruned by OBD in the previous section. Saliencies for the fully connected network will be compared to actual saliencies for both OBS and OBD, and the accuracy of the second order approximation will be illustrated.

10.2.1 OBS saliencies

The OBS saliency for each weight in the *fully* connected network which was pruned in Figure 10.5 was computed using the expression in Eq. 6.26. Further, the *actual* saliency of each weight was computed by retraining the remaining weights within the quadratic approximation according to Eq. 6.21 and then determining the actual change in error. The result is illustrated in the upper panel of Figure 10.10 where actual saliency is plotted versus estimated saliency. Due to the weight decay (refer to section 9.2.2) some of the estimated saliencies were *negative*. In order to visualize these saliencies in the log-log plot as well, the *absolute* value was used and enter the upper panel of Figure 10.10 as circles.

If the saliency estimates were accurate then estimated saliency would equal actual saliency and be located on the solid line. This is seen *not* to be the case; rather, the estimated saliencies are consistently too small corresponding to an underestimation of the

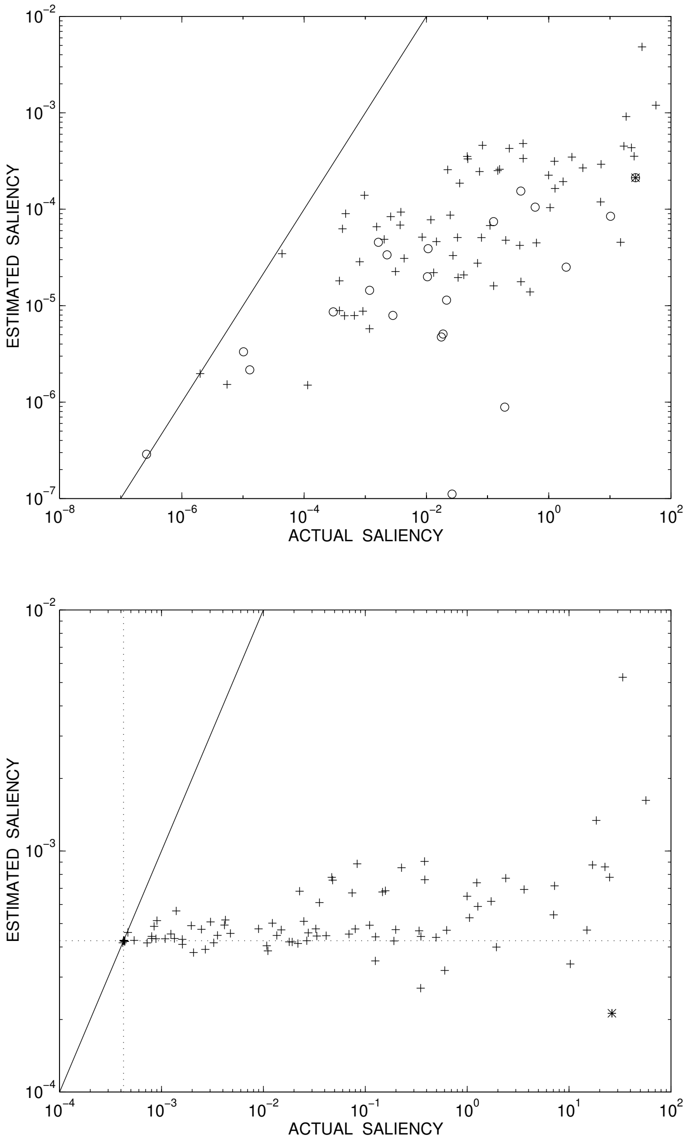


Figure 10.10: OBS saliency quality for the fully connected network of Figure 10.5. Upper panel: Actual vs. estimated saliency. Negative estimated saliencies are indicated by circles. Smallest estimated saliency is indicated by an asterisk. Lower panel: Actual vs. estimated saliencies, both coordinates offset by a constant corresponding to twice the magnitude of the largest magnitude negative saliency (see text).

actual change in error. Further, the rank ordering of the weights according to estimated saliency is not consistent with rank ordering according to actual saliency. The weight selected for pruning is the one having the largest magnitude negative saliency and is indicated by an asterisk in the figure; the corresponding actual saliency is seen to be positive and very large.

In order to more clearly illustrate the inconsistent rank ordering of the weights, each point (actual, estimated) was translated by addition of the same small constant to each coordinate. The constant was chosen as twice the magnitude of the largest magnitude negative saliency, making all estimated saliencies positive and thus more easily visualized in a log-log plot. The result is illustrated in the lower panel of Figure 10.10 where the weight selected for pruning is once more indicated by an asterisk. The dotted lines in the plot indicate the former level of zero, i.e., all points below the horizontal line had negative estimated saliency prior to the translation. From the figure the inconsistent ranking is clearly seen, i.e., the weights having the smallest estimated saliencies are not the ones with the smallest actual saliencies.

We now turn towards an examination of the quadratic approximation to the error function $E(\mathbf{w})$ from which the saliencies were computed. The OBS saliency estimate is computed as the estimated difference in error $E(\mathbf{w})$ between the retrained network and the unchanged network. Therefore the validity of the quadratic approximation will be illustrated along a line in parameter space indicated by a retraining vector $\delta\mathbf{w}_j$ determined by Eq. 6.21 and passing through the unchanged parameter vector $\widehat{\mathbf{w}}$, i.e., $\mathbf{w} = \widehat{\mathbf{w}} + c \cdot \delta\mathbf{w}_j$. The line is parametrized by c where $c = 0$ denotes the initial weights $\widehat{\mathbf{w}}$ and $c = 1$ denotes the weights for which the saliency is estimated.

The first weight to be considered in this respect is the one having the *smallest estimated* (negative) saliency. The quadratic approximation in the direction of the corresponding retraining vector is illustrated in the upper panel of Figure 10.11 along with the *true* value of the error function. Note that the approximation in this direction is not minimal at the expansion point $\widehat{\mathbf{w}}$ which is due to the weight decay used; this furthermore leads to the negative saliency. It is seen that whereas the true error displays "walls" which rise sharply, the quadratic approximation is very "flat" which explains the large discrepancy between estimated and actual saliency.

The reason for the "flat" quadratic approximation is revealed by an enlargement around $c = 0$ as illustrated in the lower panel of Figure 10.11. It is seen that the true error here displays the shape of a "bath tub" [GHK+93] with steep walls and very low curvature at the bottom. The "bath tub" results e.g., if the direction of the retraining vector $\delta\mathbf{w}$ is *almost* along the bottom of a "rain gutter"-like structure of the error function as illustrated in Figure 8.5 on page 88. Thus, the very flat quadratic approximation is a result of an ill-conditioned Hessian as described in chapter 7 and illustrated in chapter 8.

We now consider the weight having the *largest* estimated saliency in Figure 10.10. The quadratic approximation in the direction of the corresponding retraining vector for this weight is illustrated in the upper panel of Figure 10.12 along with the *true* value of the error function. Once more the quadratic approximation is seen to be very flat whereas the true error function has very steep walls, leading to a very large discrepancy between actual and estimated saliency. From the upper panel in Figure 10.12 it is clearly seen that the estimated error need not necessarily be minimal for the retrained network, $c = 1$. This is due to the retraining being performed for the weight decay *augmented* cost function $C(\mathbf{w})$ whereas the saliency is estimated as the change in the error $E(\mathbf{w})$; refer also to

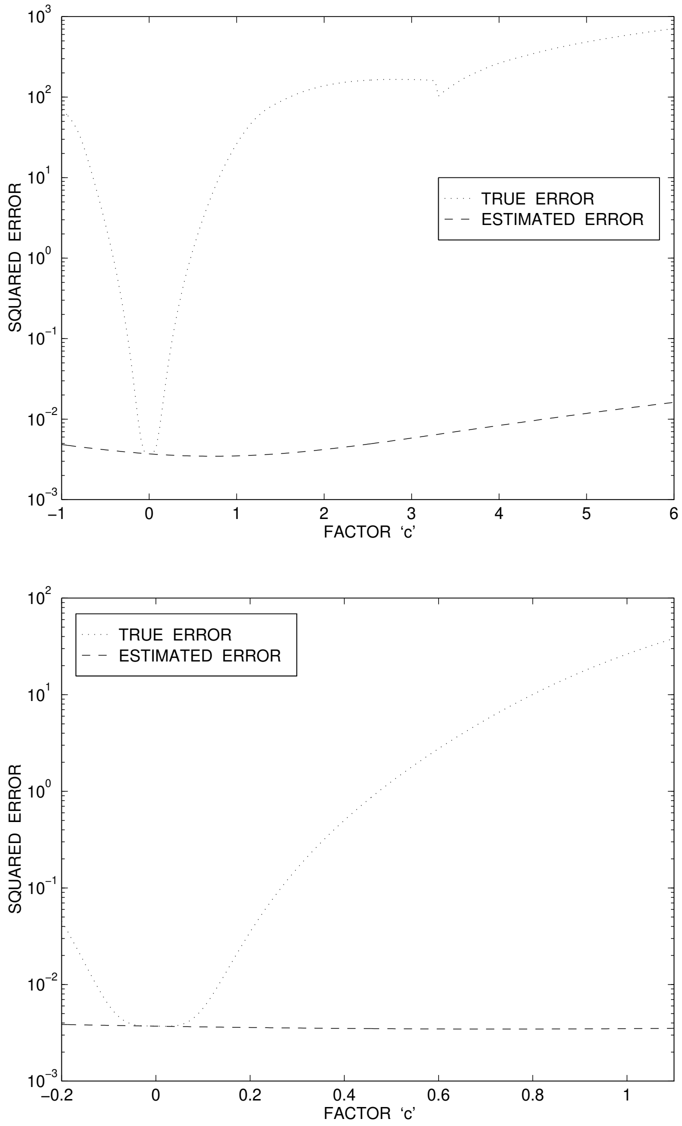


Figure 10.11: Accuracy of quadratic approximation to the squared error function $E(\mathbf{w})$. The direction in parameter space is determined by $\mathbf{w} = \widehat{\mathbf{w}} + c \cdot \delta\mathbf{w}_j$, where j indexes the weight having the *smallest* estimated saliency in Figure 10.10. Upper panel: General overview showing minimum of quadratic approximation. Right panel: Enlargement illustrating the “bath tub” shape of the true error.

section 6.2.2. The lower panel in Figure 10.12 contains a closer “zoom-in” around $c = 0$ than in the previous figure, clearly illustrating a “bath tub” shape of the cost function in this direction as well.

Generally, OBS has been found to be very sensitive to ill-conditioning of the Hessian matrix. The ill-conditioned Hessian leads to a very “flat” quadratic approximation of the cost function as illustrated above. Thus, the built-in reestimation of the remaining weights after pruning will often lead to a very large “step” in weight space in order to reach a

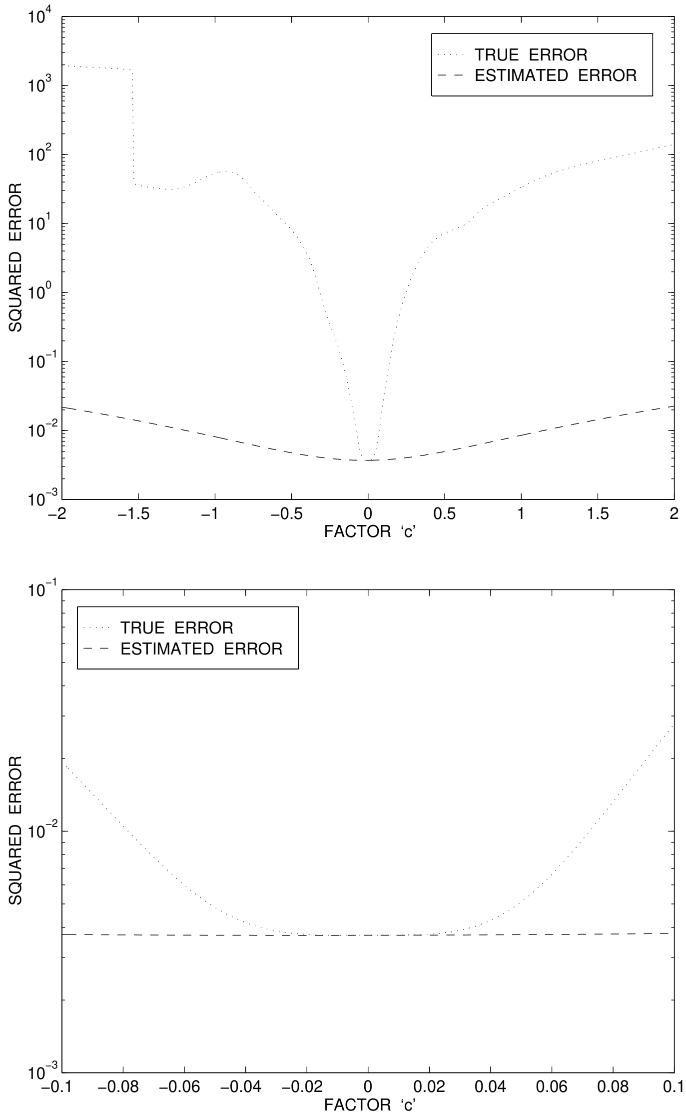


Figure 10.12: Accuracy of quadratic approximation to the squared error function $E(\mathbf{w})$. The direction in parameter space is determined by $\mathbf{w} = \widehat{\mathbf{w}} + c \cdot \delta\mathbf{w}_j$, where j indexes the weight having the *largest* estimated saliency in Figure 10.10. Upper panel: General overview showing minimum of quadratic approximation. Right panel: Enlargement, once more illustrating the "bath tub" shape of the true error.

new minimum within the quadratic approximation. The large change in the weights leads to a region in weight space where the second-order expansion is no longer valid. Thus, whereas the flat quadratic approximation leads to small *estimated* saliencies the large steps taken for reestimation lead to large *actual* saliencies. As ill-conditioning is a commonly encountered problem especially for recurrent networks (refer to chapters 7 and 8) OBS cannot be recommended for this network type.

10.2.2 OBD saliencies

The OBD saliency for each weight in the *same* fully connected network as was considered in section 10.2.1 above was then estimated according to Eq. 6.12. The actual saliencies were determined as well by setting to zero each weight in turn and computing the actual change in error. Actual saliencies were then plotted against estimated saliencies as illustrated in Figure 10.13. It is seen that the smallest estimated saliencies are very accurate and that the accuracy remains high even for fairly large-saliency weights. Furthermore it is seen that rank ordering of the weights according to estimated saliency is consistent with rank ordering according to actual saliency; thus the correct weight gets pruned.

Figure 10.14 illustrates the quadratic approximation together with the true squared error error function $E(\mathbf{w})$ along the directions indicated by the weights having the smallest and largest estimated saliency, respectively. It is seen that the approximation along these directions is *not* as "flat" as was the case along the OBS reestimation directions as illustrated above. The true error function is consequently approximated with a higher accuracy, leading to better saliency estimates.

The accuracy of the OBD saliencies has been verified in numerous investigations similar to the one described here. The explanation for the better saliencies of OBD compared to OBS seems to be that the directions where the second-order expansion is flat is typically *not* along the directions indicated by the individual weights. Rather, flat directions often occur along directions in weight space determined by combinations of several weights. This empirical observation is in line with the description of ill-conditioning in chapter 7, where it was found that ill-conditioning is essentially caused by parameters becoming redundant and thus in *combination* indicating directions of constant error.

Of importance is also that OBD does not attempt to include the effect of retraining in the saliency estimate. Consequently, the weight change is limited to the magnitude of the weight in question whereas the OBS reestimation vector might lead to large weight changes even for small magnitude weights, due to the influence of ill-conditioning. Generally, the smaller the weight changes, the more likely it is that the quadratic approximation is still

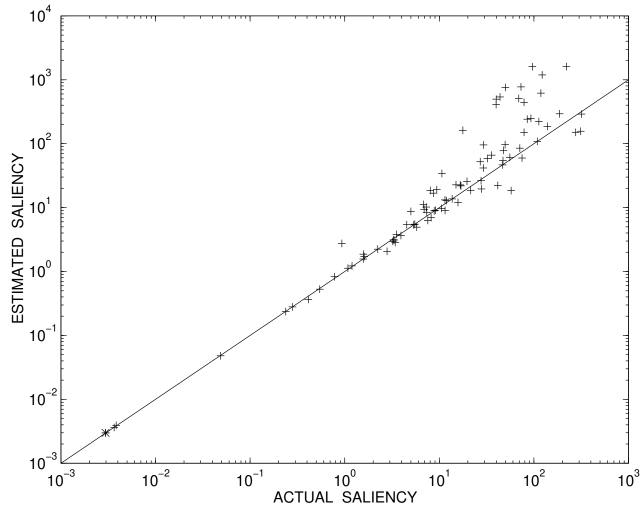


Figure 10.13: OBD saliency quality for the fully connected network of Figure 10.5: actual saliency vs. estimated saliency. Smallest estimated saliency is indicated by an asterisk.

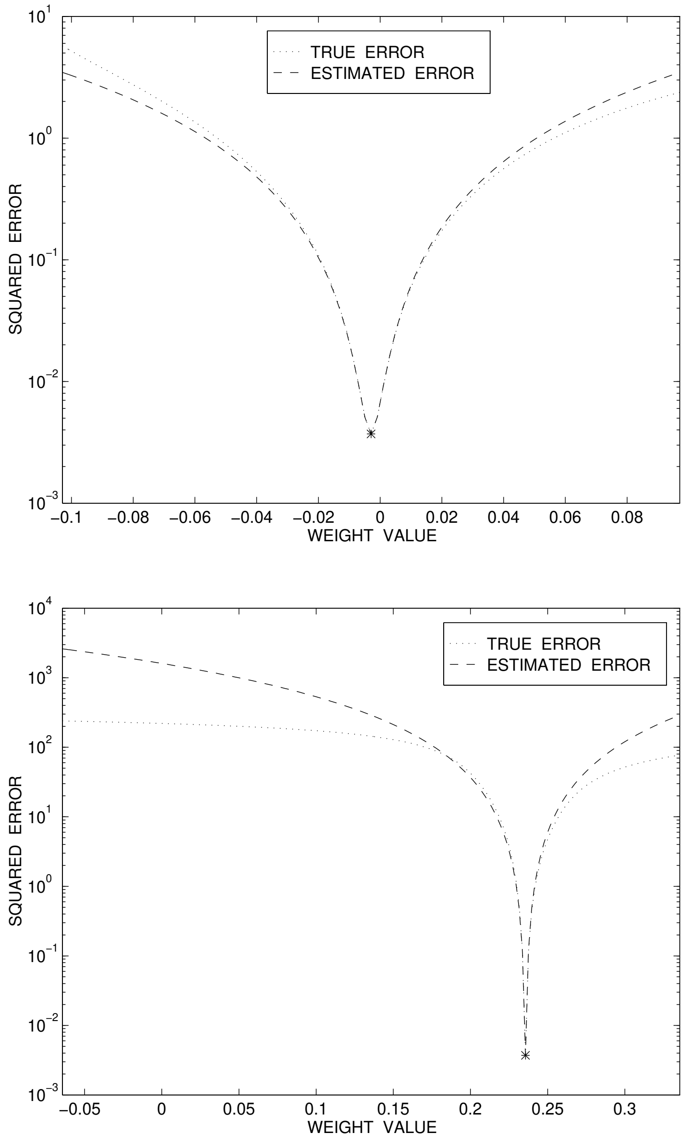


Figure 10.14: Accuracy of quadratic approximation to the squared error function $E(\mathbf{w})$. Upper panel: The direction in parameter space is determined by the weight having the *smallest* estimated saliency in Figure 10.13. Lower panel: The direction in parameter space is determined by the weight having the *largest* estimated saliency.

valid in the new point.

In this work it has been found that the saliencies obtained by OBD are very accurate not only for the quadratic cost function applied to feed-forward as well as to fully recurrent networks but also for the entropic cost function applied to novel recurrent model structures rooted in statistical physics, namely Boltzmann chains and Boltzmann zippers as will be illustrated in chapter 12.

Chapter 11

Recurrent network memory

In order to *characterize* a fully trained recurrent network one may resort to the traditional tools from non-linear dynamic systems analysis. These tools include e.g., phase space plots, measurement of fractal dimensions and measurements of Lyapunov exponents [Aba96], some of which were applied to recurrent networks in section 9.4 in order to investigate their ability to simulate the dynamics of the teacher function. However, a drawback of these tools is that they are mainly directed towards the analysis of a network when running in *closed-loop*, where the network does not receive external inputs but feeds the network output back to the input. As the recurrent networks are usually trained and intended for *open-loop* operation being driven by external inputs, the traditional tools seem inappropriate. In fact they may completely fail to provide insight if the network is incapable of functioning in a closed-loop mode of operation; this was the case for most of the recurrent networks trained on the Santa Fe laser series as described in section 9.4.1. Furthermore, the traditional tools are aimed at characterizing the solution found by a network rather than the network itself, i.e., *how* the recurrent network *implements* its solution to the modeling problem at hand.

In order to allow for proper characterization and interpretation of recurrent networks it seems necessary to create additional tools for analysis. This work has initiated the investigation of such novel tools aimed specifically at the characterization of recurrent networks. In particular, a novel operational method for assessing the *effective memory* of a recurrent network has been proposed and will be described in the following. In section 11.1 it is discussed how to conceive the notion of an *effective* memory for a recurrent network which in principle has an infinite memory. Section 11.2 contains a proposal for an operational approach towards assessing the *average* effective memory of a fully trained recurrent network and in section 11.3 the approach is slightly modified leading to a *time-local* effective memory. The chapter is concluded by section 11.4 which illustrates the proposed memory measures applied to recurrent networks trained on the Santa Fe laser series as well as the Mackey-Glass series.

11.1 The effective memory of recurrent networks

In chapters 2 and 3 it was described how feed-forward networks can accommodate dynamics by working from a lag space vector $\mathbf{x}(t) = [x(t), x(t-1), \ldots, x(t-M)]$ of previous observations from the system to be modeled. As the lag space enters the units in the first hidden layer of the feed-forward network by a linear weighting, the input to each of these units may be conceived as the outputs of linear FIR filters; refer to Figures 3.1 and 3.2. If the input to the FIR filters at present time t is denoted as $x(t)$ then the network *memory* of previous inputs $x(t-1), x(t-2), \ldots$ available to the network equals M as feed-forward networks do not posess any internal memory in the hidden units; the only memory available about previous inputs resides in the externally provided tapped delay line, extending M steps back in time from the current time step t. It is here assumed that the delay time between each of the previous observations entering the lag space equals one; naturally, if the delay time between the observations is an integer $\tau > 1$ then $\mathbf{x}(t) = [x(t), x(t-\tau), \ldots, x(t-M\cdot\tau)]$ corresponding to a "sub-sampled" tapped delay line. Considering the memory of the feed-forward network as the total time span of the lag space, the memory of the network consequently equals $M \cdot \tau$.

Whereas the memory of a feed-forward network is determined solely by the externally provided lag space, a characteristic of recurrent networks is their ability to build up an internal memory representing the "history" of previous inputs on which the predictions of future values are based. The significance of this internal memory is especially clear when using recurrent networks working from only one external input. Without the ability to create internal memory this class of networks would be practically useless. Fully recurrent networks are however highly capable of creating an internal memory which allows this network type to accurately model complex dynamics when working from only a single input as the experiments in the preceeding chapters have demonstrated.

The output of a recurrent network is based on the current and – in principle – infinitely many previous inputs which may be expressed as

$$y(t) = y\left(t|\widehat{\mathbf{w}}, x(t), x(t-1), \ldots, x(-\infty)\right) \quad , \tag{11.1}$$

where $\widehat{\mathbf{w}}$ indicates the parameter estimate resulting from training. This way, the memory of recurrent networks is neither fixed nor directly limited by the network architecture as is the case for feed-forward networks. It is however not at all clear exactly *how* a recurrent network utilizes this infinite information of the past in order to model a particular system. Interesting insight into the functionality of a recurrent network could be obtained if we could somehow characterize the way in which the network utilizes the information of the previous inputs.

One way in which to characterize the dynamics of a recurrent network is to somehow determine the *effective* memory of the network, i.e., to determine how far back in time previous inputs have *significant* influence on the network output. This problem has been addressed in the literature in the context of sequence classification problems like e.g., the so-called *latching problem* [BSF94, LHG96, GLH97]. In the latching problem it is examined how far back in time it is possible to present the trained network with information of vital importance to the classification of a sequence and still obtain a correct classification. The effective memory of the network may hence be indirectly characterized in terms of the percentage classification errors made, the so-called error rate. The lower the error rate for a particular time span between vital input and time of classification, the longer the memory of the network.

Whereas the memory of a recurrent network applied to artificial problems may be determined in terms of a set of pre-specified events easy to locate in time it is however not obvious how to assess the effective memory of a general nonlinear recurrent network applied to e.g., a real-world time series prediction problem. In this case, *measuring* the recurrent network memory poses a problem which has apparently not been addressed in the literature.

In this work a novel operational approach towards assessing the effective memory of a recurrent network applied to time series prediction has been suggested. The approach results in both an *average* memory and a *time-local* memory which will be described in the following sections. Application of these memory measures to a fully trained recurrent network will provide partial insight into the functionality and time scale of the dynamics of the network.

11.2 Measuring the average effective memory

Once a recurrent network is trained, the basic idea here is to define an integer variable M which expresses the *effective* memory length of past values of the input signal $x(t)$, in much the same way as the time span of the lag space provided to a feed-forward network expresses the memory for this network type, as described above. In order to introduce the concept of quantizing the length of the memory of a recurrent network we will initially consider the linear IIR filter (a linear recurrent network).

For a linear IIR filter the memory of previous inputs may be expressed in terms of e.g., the impulse response. The impulse response of the IIR filter may be conceived as the coefficients of an – in principle – infinite-order FIR filter, implementing a linear weighting of each element in an infinite lag space of previous inputs to the filter. The effective memory of the IIR filter may now be *defined* as the FIR-filter order M beyond which the summed weighting relative to the total weighting sum is *insignificant*, i.e., the order M beyond which the contribution to the output from values further back in time may be considered negligible. As a consequence, starting iterations of the IIR-filter at time $t - M$ which is equivalent to setting to zero all elements in the input sequence prior to time $t-M$, will yield an output at time t which is insignificantly different from the output obtained had the filter been presented with the full input sequence, in principle from time $t - \infty$.

It is not possible to fully characterize nonlinear recurrent networks using analysis tools from linear systems theory like e.g., the impulse response; thus, analytical assessment of the effective memory of a recurrent network is infeasible. However, we may still determine the memory of a recurrent network in an *operational* manner similar to what was described for the IIR filter. I.e., the effective memory is determined as the smallest M for which starting iterations at time $t - M$ with all values in the input sequence prior to this time step absent, i.e., set to zero will yield a network output

$$y(t) = y\left(t|\widehat{\mathbf{w}}, x(t), x(t-1), \ldots, x(t-M)\right) \tag{11.2}$$

for an arbitrary time t which is not *significantly* different from the output obtained with iterations commencing at time $t - \infty$ as denoted by Eq. (11.1).

As the recurrent network is intended for prediction on novel data not included in the training set it seems reasonable to decide upon the significance of the deviation in the network output caused by the limited memory of previous inputs, when using data in a separate test set. This way, the memory measure will be independent of effects particular to the training data (e.g., noise) which the network might have "discovered" during training, leading to an effective memory longer than encountered during actual application of the network. In order to determine the significance of limiting the access to previous inputs it is therefore here suggested to evaluate an estimate of the generalization error – i.e., prediction errors on a test set – using predictions based on only a *limited* number of previous inputs. This generalization error is then compared to the error obtained using all – in principle infinitely many – previous inputs.

Assuming that the test set of size V follows immediately after the training set which extends to time T, the generalization error obtained from predictions based on only the m most recent inputs is estimated by computation of

$$\widehat{G}_m(\widehat{\mathbf{w}}) = \frac{1}{V} \sum_{t=T+1}^{T+V} \left[d(t) - y\left(t | \widehat{\mathbf{w}}, x(t), x(t-1), \ldots, x(t-m)\right)\right]^2, \ m \geq 0 \tag{11.3}$$

where $d(t)$ denotes the desired output at time t; e.g., for autoregressive prediction tasks we might have $d(t) = x(t+1)$. The network output $y\left(t | \widehat{\mathbf{w}}, x(t), x(t-1), \ldots, x(t-m)\right)$ is computed for each $t \in [T+1; T+V]$ by *resetting* the hidden unit states $s_i(t-m-1)$, $i = 1, 2, \cdots, N_h$, to zero and then iterate the network from time $t-m$ until time t, using the output $y(t)$ at this time step as the prediction of $d(t)$ as outlined in Figure 11.1. Setting the hidden unit states $s_i(t-m-1)$ to zero is equivalent to *erasing* the memory of the network regarding inputs prior to time $t-m$ as the hidden unit values constitute the network memory. In the first iteration calculating $y\left(t-m | \widehat{\mathbf{w}}, x(t-m)\right)$, the network thus functions as a feed-forward network since the previous values of the hidden units – and thereby all previous external inputs – have no influence on the network output; in order to verify this the reader is referred to section 3.3.1. Then, the network gradually builds up a representation of the past in the hidden units during the next $m+1$ iterations before it makes its prediction at time t.

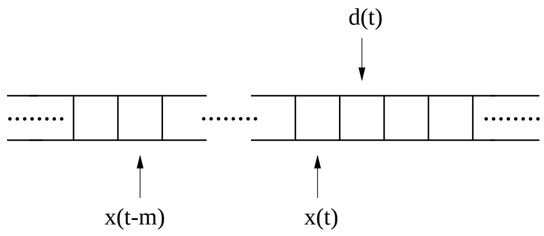


Figure 11.1: One step ahead prediction from only a limited number m of previous inputs.

The resulting limited memory generalization errors $\widehat{G}_0(\widehat{\mathbf{w}}), \widehat{G}_1(\widehat{\mathbf{w}}), \ldots$ computed from Eq. (11.3) are then compared to $\widehat{G}_\infty(\widehat{\mathbf{w}})$ denoting the error obtained from Eq. (5.8) when using all available previous inputs, i.e., no resetting of the hidden unit states at any time. The network memory M is now *defined* as,

$$M = \inf \left\{ m \,\middle|\, \forall m' \geq m, \ \frac{|\widehat{G}_{m'}(\widehat{\mathbf{w}}) - \widehat{G}_\infty(\widehat{\mathbf{w}})|}{\widehat{G}_\infty(\widehat{\mathbf{w}})} < \epsilon \right\} \tag{11.4}$$

i.e., as the integer m beyond which the limited memory generalization error is not *significantly* different from the "infinite" memory generalization error. Here, ϵ is a *small* constant indicating the level of significance. The numerical value is applied in order to eliminate fluctuation effects, as will be illustrated in section 11.4. As the effects of limiting the network memory for each prediction are averaged out across the entire test set, the memory defined by Eq. (11.4) is in fact an estimate of the *average* memory of the network. Thus the memory, M, denotes the minimal number of previous inputs beyond which additional inputs *on average* are insignificant in terms of generalization ability.

One way in which to interpret the memory measure defined by Eq. (11.4) is as an expression for the average transient to expect from the recurrent network when initiating iterations on a novel sequence of data generated by the system which has been modeled. This way, the memory measure also denotes the minimum number of time steps which on average should be iterated before the network output can be trusted. The memory M may be conceived as the dimension of the effective "lag space" implemented by the recurrent network. Naturally, working from a lag space which is not fully initialized may lead to poor predictions, just as is the case for feed-forward networks.

An alternative way in which to explain the computation of the recurrent network memory is in terms of a generalization based "pruning" procedure applied to the time-unfolded network structure. Consider computing the limited memory network output $y\left(t|\widehat{\mathbf{w}}, x(t), x(t-1), \ldots, x(t-m)\right)$. By referring to Figure 4.1 on page 41 it can be seen that the procedure described above for obtaining the limited memory network output at time t is equivalent to unfolding the recurrent network in time and then *prune* all weights in the unfolded network which reside in layers prior to time $t-m$ by setting them to zero. Computation of the the fraction in Eq. (11.4) for increasing depths of the unfolded network is in a sense equivalent to computing the (relative) saliency of the pruned weights on a test set. The network memory is then determined as the smallest unfolded network depth for which the (relative) saliencies of larger depths are negligible.

The level of significance for the relative difference between the limited memory and infinite memory generalization errors in Eq. (11.4) is expressed as a *small* constant ϵ. This constant could possibly be chosen in such a way that the resulting quantity M denotes the network memory of previous inputs beyond which the variations in generalization error are indistinguishable from the variations that would result due to different realizations of the finite size training set, when operating from an *unlimited* memory. A proper constant ϵ in this sense may possibly be determined from e.g., the derivations of the FPE generalization error estimate. This approach has however not been pursued in this work as the initial investigations have primarily concerned the behaviour of the memory measure and thus the viability of the analysis tool. Here, ϵ has been chosen "by hand".

11.3 Measuring the time-local effective memory

The memory measure defined by Eq. (11.4) determines the number of previous inputs that the network needs knowledge about in order to obtain good predictions on the average sample in the test set. Therefore the measure was interpreted as the *average* memory of the network. A recurrent network, however, is a nonlinear *dynamic* system whose internal characteristics may be highly influenced by the nature of the input series. Especially, if the input series exhibits regions of seemingly non-stationary behavior, the network dynamics including memory must clearly be affected. Such changes in dynamics are not captured by the average memory measure but we may define a *time-local* memory measure, in accordance with Eq. (11.4), using a time-local generalization error estimate

$$\widehat{G}_m(t;\widehat{\mathbf{w}}) = \frac{1}{K} \sum_{t'=t-K+1}^{t} \left[d(t') - y\left(t'|\widehat{\mathbf{w}}, x(t'), x(t'-1), \ldots, x(t'-m)\right)\right]^2, \tag{11.5}$$

where $m \geq 0$, $t > T$, and $1 \leq K \leq V$ is the size of a smaller test set. The time-local generalization error estimate at time t is thus obtained by averaging over the K most recent squared prediction errors. The time-local memory at time t is then obtained by inserting Eq. (11.5) into Eq. (11.4), comparing to the generalization error estimate $\widehat{G}_\infty(t;\widehat{\mathbf{w}})$ obtained by using an infinite memory, averaging over the K most recent squared prediction errors as well. Choosing a small K gives rise to a very noisy estimate of the generalization error; however, it will in principle provide a good resolution of changes in network memory due to the changing nature of the input series. On the other hand, increasing K improves generalization error accuracy but reduces the resolution of changes in memory. Ultimately, if K equals V the time-local memory measure obtained at time $t = T + V$ equals the average memory defined in the previous section.

11.4 Illustration of the memory measures

The memory measures suggested above will now be demonstrated on recurrent networks trained on the laser series as well as networks trained on the Mackey-Glass series, both described in appendix A. Once more it is stressed that currently, the primary objective of the memory measures is to function as tools for characterization of a recurrent network operated in "open-loop" in the same way as e.g., computation of the correlation dimension is a tool for characterization of a strange attractor and the underlying dynamics of a system running in "closed-loop"; refer to section 9.4. It will be argued below that the memory measure may be used to select between networks having a comparable estimated generalization error. The future might bring suggestions to other practically oriented applications of the proposed memory measures.

11.4.1 Memory of RNNs trained on the laser series

The memory measures will here be demonstrated for selected networks resulting from the laser series learning curve, illustrated in Figure 9.4 on page 112. The left panel of Figure 11.2 depicts the limited input generalization error $\widehat{G}_m(\widehat{\mathbf{w}})$ for increasing values of m, the number of previous inputs on which predictions are based, for one of the networks trained on 7000 examples. The horizontal dotted line indicates $\widehat{G}_\infty(\widehat{\mathbf{w}})$, i.e., the generalization error obtained without limiting the network memory. It is seen that $\widehat{G}_m(\widehat{\mathbf{w}})$ indeed converges as expected towards $\widehat{G}_\infty(\widehat{\mathbf{w}})$ as the available information of the past is increased.

By visual inspection of the figure it seems that the network memory defined by Eq. (11.4) must lie "somewhere between 110 and 250" as the two curves are collapsing onto each other in this interval. Recall that the memory was specified in terms of the absolute relative error between the limited and infinite input generalization errors, corresponding to the fraction entering Eq. (11.4). This quantity is illustrated in the right panel of Figure 11.2. The horizontal dotted line denotes an arbitrarily chosen level $\epsilon = 0.01$. According to Eq. (11.4) we now determine the memory M of the network as the number of previous samples m at which the *final* intersection between the solid and the dotted lines occurs; in this case we obtain the memory $M = 198$. Beyond this final intersection the improvement in generalization error by including more previous points for the predictions will be less than 1 % of $\widehat{G}_\infty(\widehat{\mathbf{w}})$, just as the relative difference will no longer *exceed* the 1 % significance level imposed; the influence of previous inputs beyond 198 time steps is seen to decay at an exponential rate.

From the right panel of Figure 11.2 it is clearly seen how the value M of the network memory depends on the choice of significance, ϵ in Eq. (11.4), defining the level below which we consider the two generalization errors as equivalent. Table 11.1 lists the memory

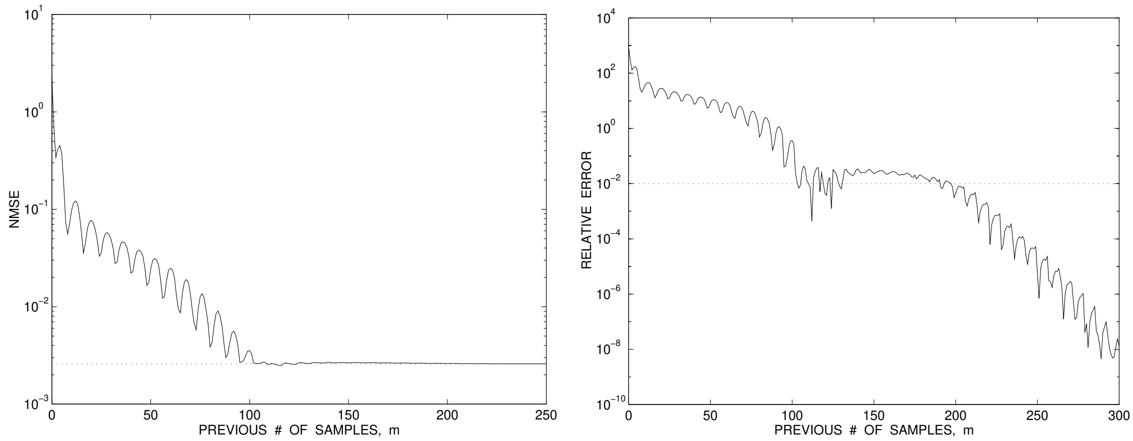


Figure 11.2: Measuring *average* memory for one of the networks from the laser series learning curve, trained on 7000 examples. Left panel: Evolution of $\widehat{G}_m(\widehat{\mathbf{w}})$ as m increases. Horizontal dotted line indicates $\widehat{G}_\infty(\widehat{\mathbf{w}})$. Right panel: Evolution of the relative error between $\widehat{G}_m(\widehat{\mathbf{w}})$ and $\widehat{G}_\infty(\widehat{\mathbf{w}})$. The dotted horizontal line denotes the threshold $\epsilon = 0.01$.

that results for different values of ϵ. The large diffences in memory resulting from relatively small changes in the threshold are seen to be due to the "plateau" which suddenly occurs for the relative error. The plateau may be interpreted in such a way that on average, the network does not utilize the information lying between approximately 140–210 time steps back in time from the current iteration. Beyond 210 time steps back the information is having influence on the network output again as seen from the now exponentially decaying relative error.

ϵ	0.05	0.025	0.01
M	102	164	198

Table 11.1: The value M of the memory for various values of ϵ for the measurements shown in Figure 11.2.

The curves in Figure 11.2 are seen to be superimposed a periodic oscillation. The period of this oscillation appears to correlate well with the period of the oscillations in the laser series; refer to Figure A.1 on page 216. The limited input generalization error $\widehat{G}_m(\widehat{\mathbf{w}})$ is at its lows when the number of previous inputs m comprises an integer number of periods of the laser series oscillations.

The time-local memory defined by Eq. (11.5) of the same network as in Figure 11.2 is examined in Figure 11.3 on the 3093 point test set following the training set. For each time step the local memory was determined by using the threshold $\epsilon = 0.01$ by averaging over the 5 and 50 most recent limited input squared prediction errors, respectively. Note how the memory in the left panel of Figure 11.3 averaging over only $K = 5$ previous errors is seen to be a very noisy quantity. As K is increased the memory measure becomes smoother. From the time-local memory measures we learn that the memory of a recurrent network

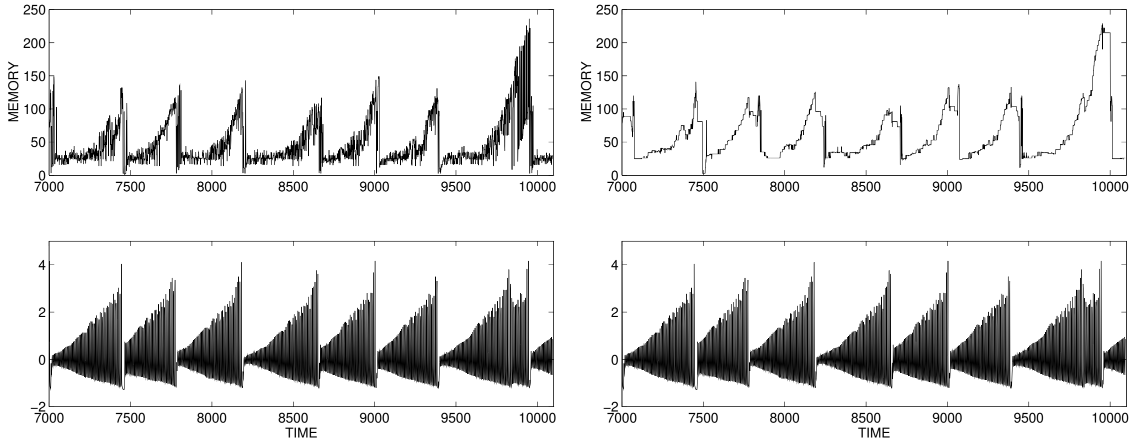


Figure 11.3: Measuring the *time-local* memory with threshold $\epsilon = 0.01$ for the laser series network from Figure 11.2 having a long average memory. Lower part of plots indicates the true series. Left panel: Using five point average, $K = 5$. Right panel: Using fifty point average, $K = 50$.

is indeed a dynamic quantity. Had the memory been static the time-local memory would lie around 200 just as for the average memory determined above. However, the memory is seen to be dependent upon and closely related to the structure of the immediately preceeding segment of the laser series.

From Figure 11.3 it is also learned that the memory of the network is maximal around a collapse of the laser series and furthermore most of the time much smaller than the 198 average memory. The reason for the large average memory is revealed by the segment between the last two collapses in the test series which has characteristics highly atypical from the rest of the series. The number of previous inputs needed for memory initialization in order to provide accurate predictions in this part of the laser series is seen to approach 250. The local-time memory examination thus provides an explanation for the plateau between previous number of inputs 140–210 encountered in the relative difference between generalization errors shown in the right panel of Figure 11.2. This interval is seen to approximately correspond to the gap between the final large memory "peak" in Figure 11.3 and the preceeding smaller peaks. Providing additional previous inputs beyond $m = 140$ time steps back does not influence prediction errors up to around time step 9800 of the laser series as the errors here are already below the specified threshold. At the same time the prediction errors on the last segment of the laser series are not influenced either as they require a much longer memory; the result is an unchanged value of $\widehat{G}_m(\widehat{\mathbf{w}})$ which leads to the plateau in relative generalization error. Not until the number of previous inputs m exceeds the memory required for the last segment does $\widehat{G}_m(\widehat{\mathbf{w}})$ decrease further.

In Figure 11.4 is illustrated the examination of the average memory of another of the networks trained on 7000 examples in the laser series learning curve shown in Figure 9.4. The first thing to note compared to Figure 11.2 is the much more rapid decay of $\widehat{G}_m(\widehat{\mathbf{w}})$ as the number of previous inputs m is increased. This leads to a much shorter memory of the network, from the right panel of Figure 11.4 it is seen that a threshold of $\epsilon = 0.01$ leads to a memory of $M = 34$. From the right panel it is furthermore seen that the value of the memory is less sensitive to the exact choice of ϵ as the curve indicating the relative

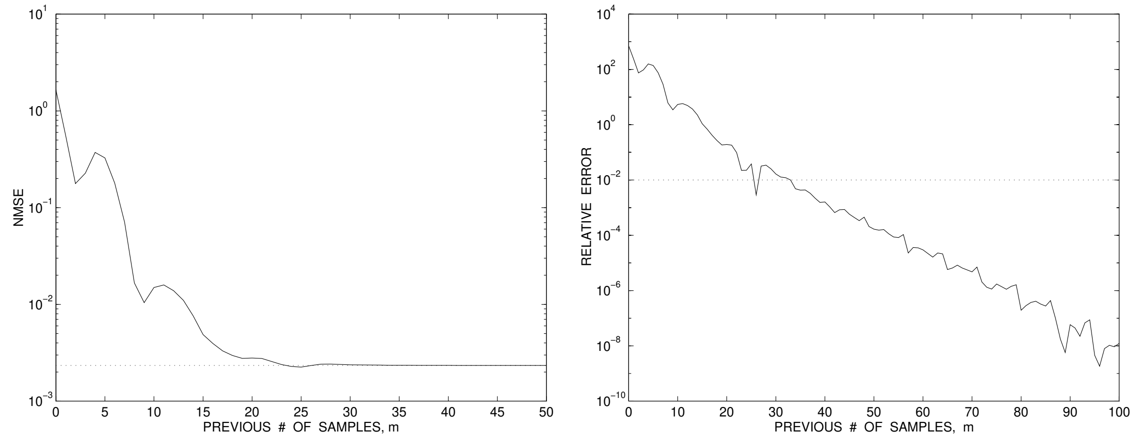


Figure 11.4: Measuring *average* memory for a typical network from the laser series learning curve, trained on 7000 examples. Left panel: Evolution of $\widehat{G}_m(\widehat{\mathbf{w}})$ as m increases. Horizontal dotted line indicates $\widehat{G}_\infty(\widehat{\mathbf{w}})$. Right panel: Evolution of the relative error between $\widehat{G}_m(\widehat{\mathbf{w}})$ and $\widehat{G}_\infty(\widehat{\mathbf{w}})$. The dotted horizontal line denotes the threshold $\epsilon = 0.01$.

errors between the generalization error estimates contains no plateaus.

Figure 11.5 illustrates the time-local memory examination for the network from Figure 11.4. Comparing to Figure 11.3 it is seen that the time-local memory still follows the structure of the laser series to some extent, but the peaks around the collapses are much less pronounced, and generally the memory is subject to fluctuations of a much smaller magnitude than in the previous example. Note that this network has no "problems" with the last segment of the test series as the memory required in this part of the laser series is no different from the memory required for the other segments of the series.

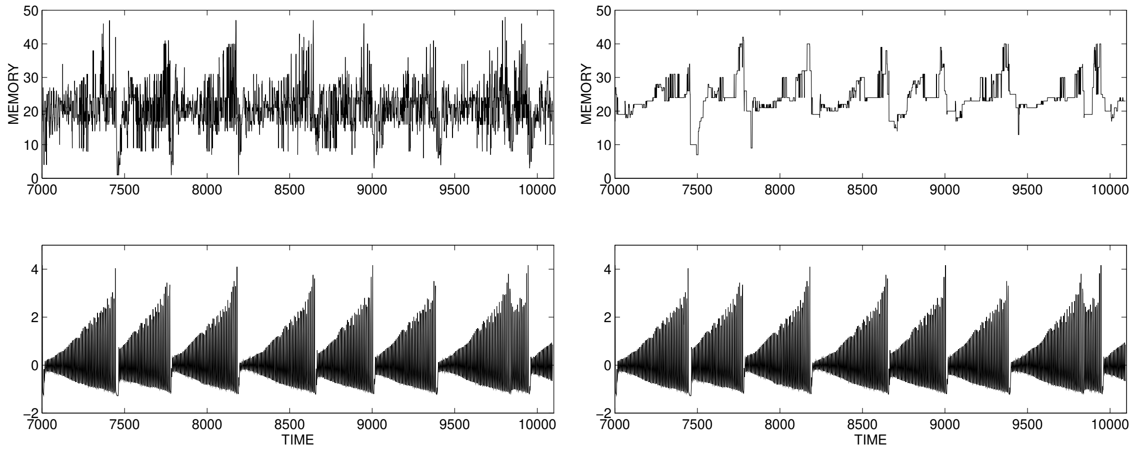


Figure 11.5: Measuring the *time-local* memory with threshold $\epsilon = 0.01$ for the laser series network from Figure 11.4 having a short average memory. Lower part of plots indicates the true series. Left panel: Using five point average, $K = 5$. Right panel: Using fifty point average, $K = 50$.

Examinations of network memory similar to the above were performed for the remaining eight networks of the laser series learning curve trained on 7000 examples. These examinations showed that the remaining networks all exhibited an average memory around $M = 30$ for the threshold $\epsilon = 0.01$, i.e., a memory similar to Figure 11.4. The network which was examined in Figure 11.2 and found to have a long average memory of $M = 198$ for the same threshold therefore appears to be rather atypical. The large difference in effective memory between networks having the exact same performance in terms of $\widehat{G}_\infty(\widehat{\mathbf{w}})$ (refer to Figure 9.4) reveals that recurrent networks may implement the dynamics of a particular problem in several widely different ways, leading to different time scales for the resulting network dynamics.

When selecting among candidate networks with comparable estimated generalization errors for a particular practical application, networks with a shorter memory seem to be preferable. The shorter memory indicates a shorter transient before the steady-state mode of operation is reached when starting iterations on a novel observation sequence, and the network thus more quickly provides accurate predictions. One might state that networks with a shorter memory are better "tuned" to the problem at hand as they are capable of utilizing the available information more efficiently.

Careful inspection of Figures 11.2 and 11.4 reveals that when limiting the number of previous inputs m the resulting generalization error $\widehat{G}_m(\widehat{\mathbf{w}})$ may actually become *lower* than $\widehat{G}_\infty(\widehat{\mathbf{w}})$. This effect is more clearly illustrated in Figure 11.6 which displays the examination of the average memory of the network in the laser series learning curve trained on 500 examples and having the smallest estimated generalization error. The phenomenon often occurs for *overtrained* networks which is also the case for the network examined here. An explanation might be that the recurrent network has adapted its memory to accomodate features like e.g., noise in the training set which are not characteristic for what will be encountered more generally in e.g., a test set. In this case, limiting the memory might act as a kind of regularization and actually improves the performance on the test set.

We now compare the typical memory of a recurrent network trained on 7000 examples of the laser series to the feed-forward network input selection curve shown in Figure 9.6 on page 115. It is seen that whereas the recurrent networks implement an effective memory of previous input values of length $M \approx 30$ the feed-forward networks need access to only ten previous values in their lag space in order to yield comparable performance. This might be explained by the difference in "resolution" [Moz93] between the memories of the two network types.

The only information available about the current dynamics of the true system is encoded into the values of the most recent observations[1]. A feed-forward network has direct access to each individual value of these most recent observations from its memory residing in the lag space; this is also denoted a *high-resolution* memory [Moz93]. In contrast, the memory of a single-input recurrent network is more of a "low-resolution" memory, holding coarser information about the most recent observations as the observation values cannot be accessed individually. Consequently, the recurrent network cannot utilize the information stored in previous observations to its full and needs to maintain a longer (or *deeper* [Moz93]) memory in order to make up for the low memory resolution and thus obtain the same information as the feed-forward network.

[1]Refer also to Takens' theorem described in section 2.2.

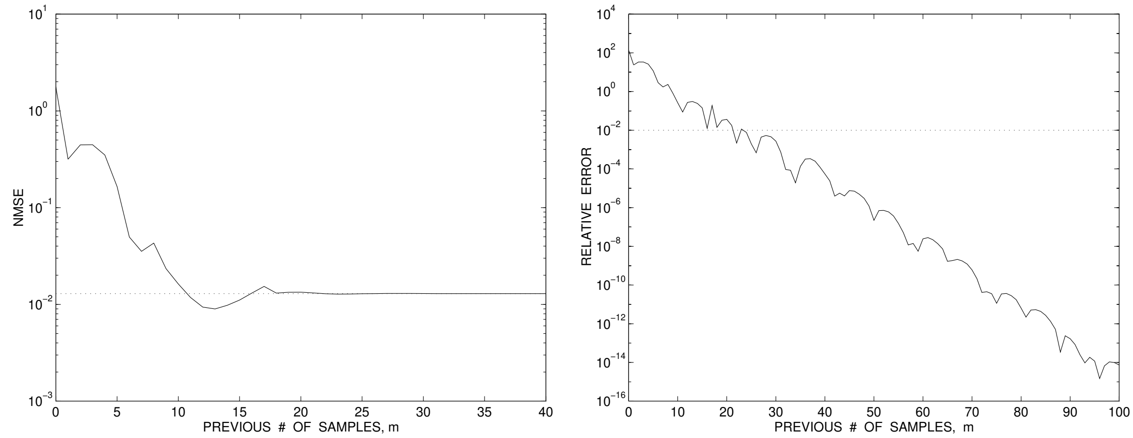


Figure 11.6: Measuring *average* memory for the network from the laser series learning curve trained on 500 examples and having the lowest generalization error. Left panel: Evolution of $\widehat{G}_m(\widehat{\mathbf{w}})$ as m increases. Horizontal dotted line indicates $\widehat{G}_\infty(\widehat{\mathbf{w}})$. Right panel: Evolution of the relative error between $\widehat{G}_m(\widehat{\mathbf{w}})$ and $\widehat{G}_\infty(\widehat{\mathbf{w}})$. The dotted horizontal line denotes the threshold $\epsilon = 0.01$.

Due to the lower resolution of the memory implemented by recurrent networks it seems reasonable to state that it will in theory always be possible to construct a feed-forward network with a lag space dimension equal to or smaller than the effective memory of the recurrent network and yielding the same performance. The effective memory of a recurrent network may hence be seen as an upper bound on the lag space dimension required by a feed-forward network in order to obtain comparable performance. In practice however, this observation might be hampered by such a feed-forward network having many more parameters than the recurrent network, thus requiring more data for training.

11.4.2 Memory of RNNs trained on the Mackey-Glass series

In order to support the presentation of the effective memory measures, these will also be briefly demonstrated for a few of the networks resulting from the Mackey-Glass series learning curve, illustrated in Figure 9.4 on page 113. Figure 11.7 illustrates examination of the average memory for one of the networks which were trained on 1250 examples and having the smallest estimated generalization error. Setting the threshold to $\epsilon = 0.01$ leads to a memory of $M = 118$ as seen from the right panel; by referring to the left panel it is verified that the curves indicating $\widehat{G}_m(\widehat{\mathbf{w}})$ and $\widehat{G}_\infty(\widehat{\mathbf{w}})$ have practically collapsed onto each other for this value. Note that the exponential decay of the relative error is fairly "well-behaved", i.e., even fairly large changes in the threshold ϵ will only lead to comparably small changes in the measured effective memory M unlike the situation in Figure 11.2.

The corresponding time-local memory measurement is illustrated in Figure 11.8 on part of the 7000 sample test set, also using a threshold $\epsilon = 0.01$. Again, the time-local memory in the left panel averaging over only $K = 5$ previous errors is seen to be a very noisy quantity. However, it clearly reveals once more that the recurrent network memory is indeed a dynamic quantity. For the Mackey-Glass problem it seems harder to relate the time-local memory directly to the structure of the series than for the laser series above,

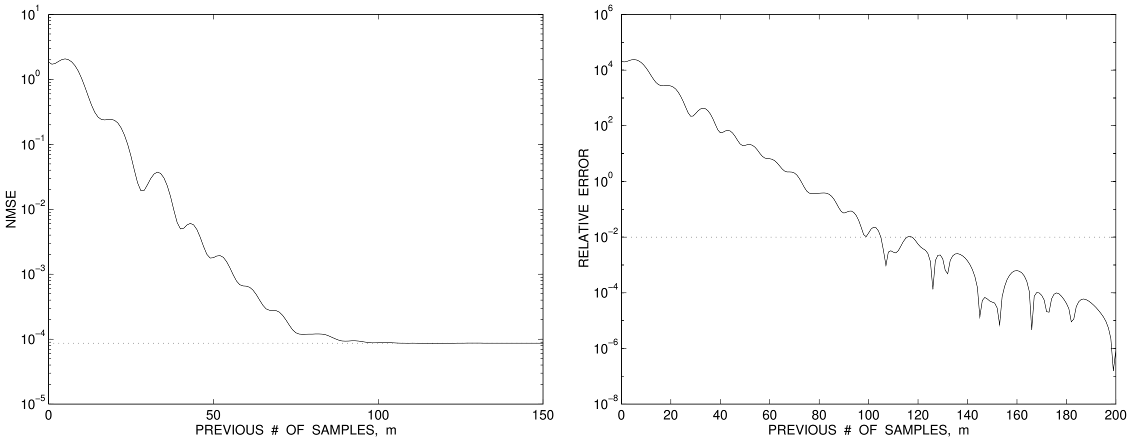


Figure 11.7: Measuring *average* memory for a network from the Mackey-Glass series learning curve, trained on 1250 examples. Left panel: Evolution of $\widehat{G}_m(\widehat{\mathbf{w}})$ as m increases. Horizontal dotted line indicates $\widehat{G}_\infty(\widehat{\mathbf{w}})$. Right panel: Evolution of the relative error between $\widehat{G}_m(\widehat{\mathbf{w}})$ and $\widehat{G}_\infty(\widehat{\mathbf{w}})$. The dotted horizontal line denotes the threshold $\epsilon = 0.01$.

but the "peaks" in memory seems to coincide with the large downwards peaks of the test series. Careful inspection of the right panel of Figure 11.8 reveals a regularity in the structure of the time-local memory as repeating patterns are present.

The average effective memory determined to $M = 118$ in Figure 11.7 was among the shortest encountered for the examined networks with comparable performance. In fact, all of the networks from the Mackey-Glass learning curve trained on 1500 examples exhibited a much longer average memory. An example of this is provided in Figure 11.9. From the level of $\widehat{G}_\infty(\widehat{\mathbf{w}})$ it is seen that the performance is almost identical to the previously examined network but the memory measurement reveals that the average effective memory has more than doubled; for $\epsilon = 0.01$ a memory of $M = 280$ is determined, as seen from

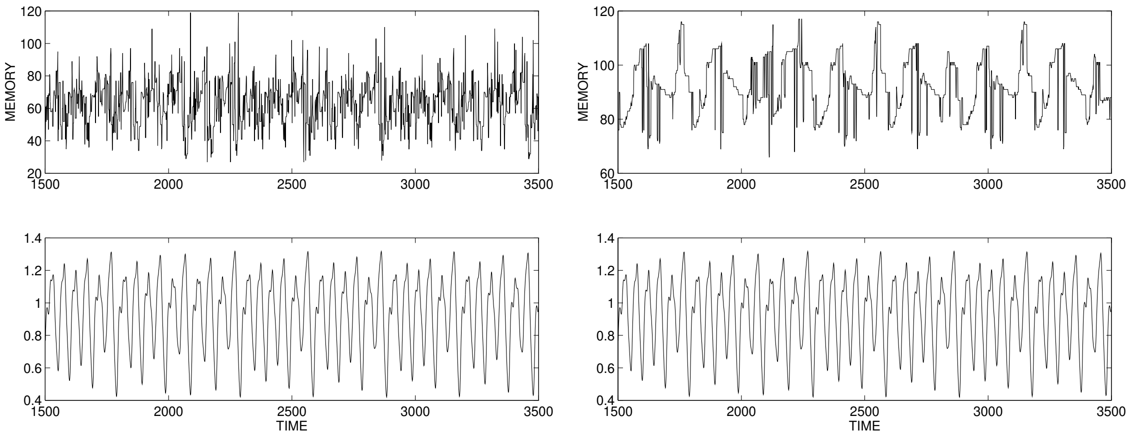


Figure 11.8: Measuring the *time-local* memory with threshold $\epsilon = 0.01$ for the Mackey-Glass series network of Figure 11.7 having a short average memory. Lower part of plots indicates the true series. Left panel: Using five point average, $K = 5$. Right panel: Using fifty point average, $K = 50$.

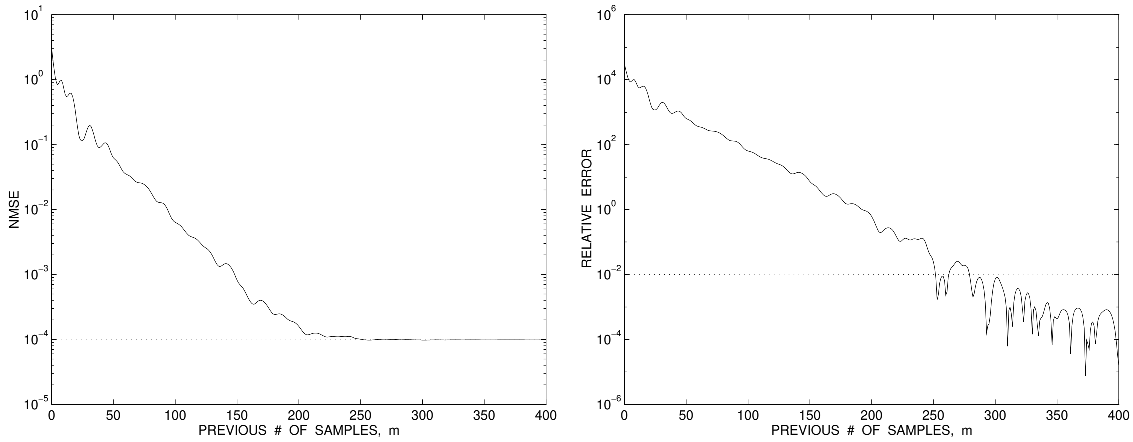


Figure 11.9: Measuring *average* memory for a network from the Mackey-Glass series learning curve, trained on 1500 examples. Left panel: Evolution of $\widehat{G}_m(\widehat{\mathbf{w}})$ as m increases. Horizontal dotted line indicates $\widehat{G}_\infty(\widehat{\mathbf{w}})$. Right panel: Evolution of the relative error between $\widehat{G}_m(\widehat{\mathbf{w}})$ and $\widehat{G}_\infty(\widehat{\mathbf{w}})$. The dotted horizontal line denotes the threshold $\epsilon = 0.01$.

the right panel. A similar memory was measured for the remaining networks also trained on 1500 examples.

A short section of the time-local memory measure is illustrated in Figure 11.10, when averaging over the five most recent errors. Comparing to the similar segment in the left panel of Figure 11.8 it is seen that the fluctuations in memory are larger in magnitude just as the overall level of the time-local memory has increased.

As was done for the laser series above we may compare the effective recurrent network memories with the memories spanned by the lag space of feed-forward networks having comparable performance in the input selection curve shown in Figure 9.7 on page 115. From this figure it is seen that between 7–9 input values are necessary in order to obtain

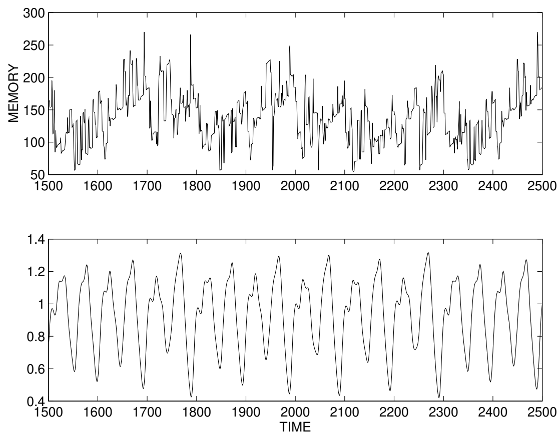


Figure 11.10: Measuring the *time-local* memory for the Mackey-Glass series network of Figure 11.9 having a long average memory, using a threshold $\epsilon = 0.01$ and a five point average, $K = 5$. Lower part of plot indicates the true series.

a performance comparable to the best recurrent networks. The networks implement a six step ahead predictor and the spacing between input values is set to $\tau = 6$. The memory of previous inputs (i.e., the time spanned by the lag space according to the definition in section 11.1 above) for the feed-forward networks are thus between 36–48. In comparison, the effective memory M of the recurrent networks were generally between 100–300; a few networks have been encountered having an effective average memory longer than 600 for the threshold $\epsilon = 0.01$. Part of an explanation for these long memories should naturally be found in the low resolution of the recurrent network memory as described above. Furthermore, whereas the feed-forward networks are sampling only every sixth element of their "memory" the recurrent networks need to store a representation of intermediate values as well.

Another part of the explanation might be the "regularity" of the Mackey-Glass series as can be seen from e.g., the lower parts of the plots in Figure 11.8, combined with the low noise which for this artificial problem origins from finite precision arithmetic only. In this case, information residing far back in time may still contribute to an explanation of the current time dynamics on *almost* equal terms as more recent observations. The recurrent networks discover these relations during training which results in the long network memories.

Chapter 12

Boltzmann Chains & Zippers: A Tutorial

So far in this thesis the problem of time series modeling has been interpreted in terms of modeling the conditional expectation of future observations given previous observations which is equivalent to prediction; refer to section 2.4. We now turn towards another and more general approach to time series modeling, namely modeling of the joint probability distribution function of the observed series. The attention is therefore directed from the fully recurrent network considered thus far and to another type of recurrent model which allows for modeling of the joint probability distribution for discrete valued time series.

The model is called the *Boltzmann chain* and an extension has been dubbed the *Boltzmann zipper*. This chapter contains a comprehensive tutorial on these models. It furthermore describes the work which has been done here in order to apply efficient second-order methods for training as well as architecture optimization by pruning to these models.

The chapter was originally written as an independent, "stand alone" tutorial and therefore briefly describes the topics of second-order training and pruning which have already been introduced in previous chapters. The tutorial follows the notation adopted in the relevant literature. It is of importance to note that the notation in this chapter is therefore inconsistent with the notation in the previous chapters. The reader is kindly asked to forgive any inconvenience this might lead to.

Two conference papers resulting from this work are included in appendices K and N. The work described in this chapter was completed during a stay at Ricoh California Research Center.

12.1 Introduction

In [SJ95] a new model called the Boltzmann chain for modeling discrete valued stochastic time series was proposed. An extended structure for modeling two simultaneously occuring and correlated stochastic time series was also introduced. It was later named the Boltzmann zipper [SL96]. The Boltzmann chain is closely related to both the Boltzmann network rooted in statistical physics and to the Hidden Markov Model (HMM) which is traditionally used for modeling stochastic time series. The Boltzmann chain combines desirable properties of Boltzmann networks, e.g., natural handling of missing data (during both training and application) with the flexibility and efficient techniques of HMMs for e.g., computing likelihoods.

This tutorial provides a detailed description of the Boltzmann chain and zipper models based on an outline of the traditional framework for stochastic modeling, HMMs and Boltzmann networks. The motivation for the tutorial is that the literature on these interesting new model types so far has been very sparse as well as very brief in nature. The tutorial includes a detailed description of methods for *reducing* the architecture of the Boltzmann chain and Boltzmann zipper. Using these methods it is possible to efficiently compute likelihoods of sequences as well as model derivatives *exactly*. Furthermore methods for conversion between HMMs and Boltzmann chains will be discussed.

Second-order optimization methods have been found capable of speeding up learning considerably compared to gradient methods for deterministic networks [Ped97]. In this tutorial, second derivatives are calculated for the entropic cost function applied to general Boltzmann networks, allowing for optimization using second-order methods. In particular the damped Gauss-Newton method is applied to Boltzmann chains and zippers.

A well-known concept in the framework of deterministic networks is algorithmic architecture optimization. A frequently used strategy is pruning where a model with excess degrees of freedom is trained and parameters eliminated according to a pruning scheme. When working with stochastic models the architecture has traditionally been chosen by hand or found by "exhaustive" search. Having derived the second derivatives for the entropic cost function, however, opens up for the application of the OBD [CDS90] and OBS [HS93] pruning schemes in the framework of general Boltzmann networks. Here, the OBD pruning scheme is applied to Boltzmann chains and Boltzmann zippers.

Both Boltzmann chains and Boltzmann zippers are applied to artificial problems as well as to a small speech recognition task. In particular, a *speechreading* system is constructed, using both audio and video information for recognition.

The tutorial is organized as follows: Section 12.2 provides a brief introduction to stochastic signal modeling and the Hidden Markov Model; the concept of "unnormalized" HMMs is also introduced. Section 12.3 reviews the traditional Boltzmann network, and second derivatives are calculated for the entropic cost function. The section is concluded by an introduction to multi state units derived from *the Potts model*, used in Boltzmann chains and zippers. Section 12.4 contains a detailed description of the Boltzmann chain and explains how exact learning is performed. The link to HMMs is explained and the section is concluded by notes and observations regarding training using second-order methods and pruning of Boltzmann chains. Boltzmann zippers are described in section 12.5 along with the reduction methods for exact learning, and some notes on training and pruning. Section 12.6 contains a description of experiments using Boltzmann chains and section 12.7 describes the experiments using Boltzmann zippers. The tutorial is summarized in section 12.8 and appendix F describes a conversion recipe for conversion from general Boltzmann chains into corresponding HMMs.

12.2 Stochastic modeling

When modeling a time series it is often assumed that the system generating the observations may be described by a stochastic process. The modeling problem is therefore cast in terms of modeling this underlying generator of the series. For prediction type problems it is common to model the conditional expectation of future observations given previous observations. Commonly adopted models for this task are e.g., linear FIR-filters and traditional feed-forward/feedback neural networks.

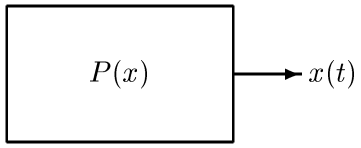


Figure 12.1: Schematic of a stochastic system. $P(\cdot)$ represents the internal stochastic process.

The most general and complete description of the stochastic generator of the series is however obtained by modeling of the joint probability distribution function of the observed series. This approach is common when working with e.g., speech recognition [Rab89] and recently [KBM$^+$95] also in biological sequence (e.g., protein and DNA sequences) modeling. The series of observations $x(t)$ (possibly vectors) can be continuous valued or they can be discrete symbols from a finite countable alphabet, e.g., $x \in \{1, 2, \ldots, m\}$.

When modeling the distribution of the series we need to choose a statistical model. Examples of these are Gaussian processes, Poisson processes and Markov processes [Rab89]. A particularly popular choice is the *Hidden* Markov model which is widely believed to be one of the best models for speech recognition. Hidden Markov Models are closely related to Boltzmann chains and are thus briefly reviewed in the following.

12.2.1 Hidden Markov Models

A Hidden Markov Model (HMM) is a stochastic model characterized by a number n of hidden states which are internal for the model. With each of the states is associated a probability distribution to account for the possible observations from the system to be modeled. These distributions can be both continuous and discrete in nature. In this context we will focus on HMMs used for modeling discrete valued time series. A comprehensive tutorial on HMMs can be found in [Rab89].

Let a time series of discrete observations be denoted $\{j_t\}_{t=1}^{L} = J_L$ where L denotes the length of the observation sequence and let the number of possible observations be m, $j_t \in \{1, \ldots, m\}$.

At each time step the HMM is in one of its n states. The initial state at time $t = 1$ is chosen according to a prior distribution π_i , $i = 1, \ldots, n$. Transitions from state i to state i' occur at each time step with probability $a_{ii'}$ according to a first-order Markov process; hence the name of the model. The Markov process thus leads to a sequence of states which we will denote $\{i_t\}_{t=1}^{L} = J_L$, $i_t \in \{1, \ldots, n\}$. With each of the states is associated a discrete probablility distribution over the m possible observations. At each time step t the model generates an observation j according to the distribution corresponding to the current state. Given that the model is in state i, observation j will occur with probability b_{ij}.

We collect the transition probabilities $a_{ii'}$ in the transition matrix A, the observation probablilities b_{ij} in the emission matrix B and the prior distribution π_i in the vector π and denote the collection of all parameters $\lambda = (A, B, \pi)$. A sequence of states $\{i_t\}$ and observations $\{j_t\}$ given the model λ is thus modeled to occur with probability

$$P(\{i_t, j_t\}_1^L|\lambda) \quad = \quad \pi_{i_1} a_{i_1 i_2} a_{i_2 i_3} \ldots a_{i_{L-1} i_L} b_{i_1 j_1} b_{i_2 j_2} \ldots b_{i_L j_L} \tag{12.1}$$

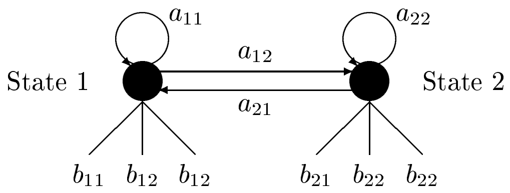


Figure 12.2: Schematic of a Hidden Markov Model with two hidden states and three possible observations in each state.

$$\begin{aligned} &= \pi_{i_1} \prod_{t=1}^{L-1} a_{i_t i_{t+1}} \prod_{t=1}^{L} b_{i_t j_t} \\ &= P(I_L, J_L|\lambda) \end{aligned}$$

In Figure 12.2 is shown a schematic of an HMM having two hidden states and modeling three possible observations.

The modeling problem now is to determine the parameter values $(a_{ii'}, b_{ij}, \pi_i)$ that maximize the likelihood of the HMM model λ generating the sequences of observations obtained from the signal source that we are trying to model. We thus want to maximize the probability

$$\begin{aligned} P(J_L|\lambda) &= \sum_{\text{all } \mathrm{I_L}} P(I_L, J_L|\lambda) && (12.2) \\ &= \sum_{\text{all } \mathrm{I_L}} P(J_L|I_L, \lambda)\, P(I_L|\lambda) && (12.3) \end{aligned}$$

where the sum is over all possible hidden state sequences able to account for the observation sequence J_L. Given n hidden states and a sequence of length L, the number of possible state sequences is n^L. This number is generally very large, and computing Eq. (12.2) by straightforward summing is computationally infeasible, even for small values of n and L.

When estimating parameters and applying the trained HMMs, *three basic problems* are of prime importance:

Problem 1. Given a sequence of observations J_L and a model λ, how do we efficiently compute the probability of the observation sequence given the model $P(J_L|\lambda)$?

Problem 2. Given the observation sequence J_L and a model λ, what is the most likely state sequence I_L having generated the observations?

problem 3. How do we determine the set of parameters $\lambda = (A, B, \pi)$ that maximizes the probability $P(J_L|\lambda)$?

A detailed discussion of the solutions to these basic problems can be found in [Rab89] and will not be repeated here. For future reference, suffice it to mention that an efficient solution to the first problem can be obtained using the *forward-backward algorithm* and the solution to problem two involves the *viterbi algorithm.* There is no known way to analytically solve for the model parameters λ which maximize the probability of the observation sequence. Thus, estimation of the parameters must be performed using an iterative procedure like the *Baum-Welch method* which is an implementation of the *EM-algorithm*, or using gradient techniques [Rab89].

12.2.2 "Unnormalized" HMMs

The parameters in an HMM are probabilities and thus subject to the following constraints:

$$\pi_i \, , \, a_{ii'} \, , \, b_{ij} \;\geq\; 0 \tag{12.4}$$

$$\sum_i \pi_i \;=\; 1 \tag{12.5}$$

$$\sum_{i'} a_{ii'} \;=\; 1 \, , \; \forall i \tag{12.6}$$

$$\sum_j b_{ij} \;=\; 1 \, , \; \forall i \tag{12.7}$$

The normalization naturally ensures that

$$\sum_{\text{all } I_L, J_L} P(I_L, J_L|\lambda) = 1 \quad ,$$

where the joint probability $P(I_L, J_L|\lambda)$ is given by Eq. (12.1).

As an alternative to this explicit normalization of the parameters for conventional HMMs we can define an "unnormalized" version of the HMM, in which the only constraint on the parameters is that they must be non-negative. In this case, in order to determine the joint probability of a particular sequence I_L of hidden states and J_L of observations for the model we need to normalize this particular "path" through the model by a normalization factor Z_L computed as the sum of *all* possible paths of length L through the model,

$$Z_L = \sum_{\text{all } I_L, J_L} \pi_{i_1} a_{i_1 i_2} a_{i_2 i_3} \ldots a_{i_{L-1} i_L} b_{i_1 j_1} b_{i_2 j_2} \ldots b_{i_L j_L} \tag{12.8}$$

Note that the normalization factor is dependent on the sequence length L. Computation of Z_L can be done in a manner similar to the forward-backward algorithm. The joint probability is now computed as

$$P(I_L, J_L|\lambda) = \frac{1}{Z_L} \cdot \pi_{i_1} a_{i_1 i_2} a_{i_2 i_3} \ldots a_{i_{L-1} i_L} b_{i_1 j_1} b_{i_2 j_2} \ldots b_{i_L j_L} \tag{12.9}$$

which may also be interpreted as a *globally* normalized HMM. As for traditional HMMs the probability of a particular observation sequence J_L originating from an "unnormalized" HMM is computed by summing over all possible hidden unit sequences like in Eq. (12.2), however, using the joint probabilities given by Eq. (12.9). We can of course still compute likelihoods for sequences of varying lengths if only we normalize with the proper normalization factors.

When using "unnormalized" HMMs we can no longer interpret the parameters as probabilities. This has the implication that it is no longer straightforward how to simulate the model timestep by timestep in an iterative manner since the prior, transition and emission parameters are now mixed together in the normalization factor; this also makes the model harder to interpret than traditional HMMs.

"Unnormalized" HMMs are not widely used in the literature, possibly due to the problems associated with interpretation and computation of the normalization factor. However, in [RK97] describing a model for speech phoneme recognition in which the transition and emission probabilities are replaced by feedforward neural networks (denoted as Hidden Neural Networks) it was found advantageous to use "unnormalized" network outputs since the normalization term cancelled out. The reason for mentioning "unnormalized" HMMs here is that they are directly related to Boltzmann chains and Boltzmann zippers which will become apparent in the following.

12.3 Boltzmann networks

Another choice of model when modeling systems generating discrete valued stochastic observations is the Boltzmann machine, or Boltzmann network. Boltzmann networks are stochastic recurrent neural networks rooted in statistical physics. The following provides an introduction to this type of neural network. Furthermore it is indicated how to train these models using second-order methods and how to perform architecture optimization using pruning schemes. Both approaches are well-known for traditional deterministic feed-forward and recurrent networks but are believed to be novel in the context of Boltzmann networks.

Boltzmann networks consist of a collection of units which are interconnected by bi-directional or *symmetric* weights, $w_{ij} = w_{ji}$ as shown in Figure 12.3. The units have no self-feedback, i.e., $w_{ii} = 0$ for all units i. The units can take on *binary* values corresponding

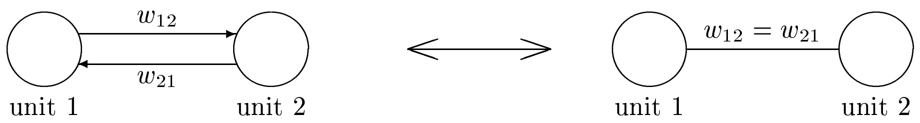


Figure 12.3: Symmetric connection between two arbitrary units in a Boltzmann network.

to "on" and "off". This has the implication that we are restricted to binary valued observations when modeling stochastic systems using traditional Boltzmann networks. In the original formulation of Boltzmann networks (e.g., [AHS85]) the units were assumed to take on values $S_i \in \{0, 1\}$. In later formulations (e.g., [PA87, HKP91]) the units take on the values $s_i \in \{-1, 1\}$, which is convenient when analyzing Boltzmann networks using theory from statistical mechanics [HKP91]. The difference is minor since we can convert between the two representations using the expression $s_i = 2S_i - 1$, thus all expressions in the following apply to both representations. The only difference is in learning, which we will return to later. For now, we adopt the notation $s_i \in \{-1, 1\}$.

Let the total number of units in the Boltzmann network be N. The units are divided into L visible units allowed to interact with the environment and $K = N - L$ hidden units that are internal to the network. In fact, Boltzmann networks can be viewed as Hopfield networks [Hop82] with hidden units. Visible and hidden units of a Boltzmann network can be arbitrarily connected as shown in Figure 12.4. The visible units may be further divided into input units and output units. The implications of such a division are only minor and will not be treated here; see e.g., [Hay94] for treatment of this case. Once the architecture of the Boltzmann network has been chosen it must remain fixed; thus, it can not be used to model observation sequences of varying lengths as e.g., the HMM.

The concatenation of the states of all units $(s_1, \ldots, s_N)$ is called the (global) state of the Boltzmann network. Since there are N units, the total network can be in any of 2^N states. Similary, the visible units can be in any of 2^L states and the hidden units in any of

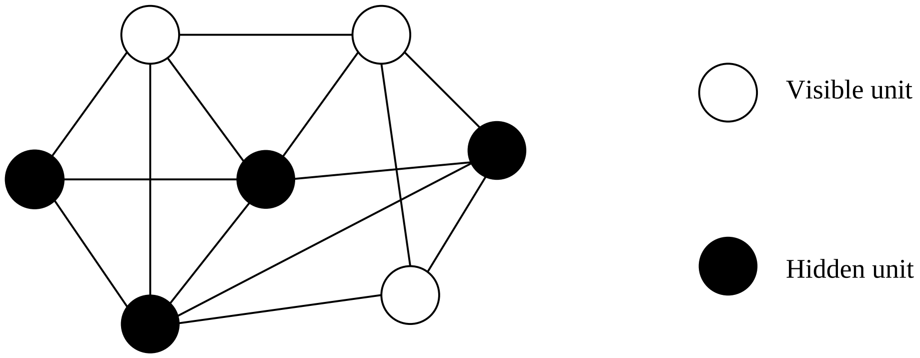


Figure 12.4: Arbitrary connected visible and hidden units in a Boltzmann network.

2^K states. For convenience, we will denote the state of the visible units as α and the state of the hidden units as β. Consequently, the state of the total network will be denoted by $\alpha\beta$.

As for Hopfield networks an *energy* is assigned to each global state $\alpha\beta$ of the network through an *energy function*

$$E_{\alpha\beta} = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}s_i s_j \tag{12.10}$$

The minima of the energy function correspond to stable configurations of the units in the Boltzmann network. The object then is to minimize the energy, finding a stable configuration well suited to the problem at hand. In order to search for the (global) minimum of the energy E for a particular network the method of *simulated annealing* is integrated into the network operation in order to avoid getting stuck in local minima of the energy. This means that the units of the Boltzmann network are *stochastic*, flipping their state according to the expression

$$\mathrm{Prob}(s_i \to -s_i) = \frac{1}{1+\exp(\Delta E_i/T)} \tag{12.11}$$

where ΔE_i is the change in energy for the network resulting from the state change of unit i. This energy change can easily be shown to be

$$\Delta E_i = E^{new} - E^{old} = 2s_i^{old}\sum_j w_{ij}s_j = 2s_i^{old}h_i \tag{12.12}$$

where h_i is the activation of unit i. The factor T in Eq. (12.11) represents the *temperature*. It controls the steepness of the slope and thus the probability of accepting a given change in energy.

When simulating a Boltzmann network we start out with a high temperature T which is graduately lowered using a simulated annealing schedule; for an introduction to simulated annealing see e.g., [AK89]. For high temperatures T the network is very likely to change state even to a state with higher energy, thus performing a coarse search of the state space. As the temperature is lowered the search becomes finer, as changes to states with higher levels of energy become less likely. Eventually, we end up in a minimum for the energy.

At each temperature the network is relaxed using Monte Carlo simulation, where units are selected at random and updated according to Eq. (12.11). This process is repeated until we reach *equilibrium*, in which the energy of the network fluctuates around a constant average value [HKP91]. At equilibrium, the probability of finding the units of the Boltzmann network in any particular global state obeys the *Boltzmann distribution*. The probability $P_{\alpha\beta}$ of finding the units in the network in any particular global state $\alpha\beta$ is therefore

$$P_{\alpha\beta} \;=\; \frac{1}{Z}\, e^{-E_{\alpha\beta}/T}, \tag{12.13}$$

where $E_{\alpha\beta}$ is the energy (dependent on the weights) when the visible units are in states α and the hidden units are in states β and Z is the normalizing partition function

$$Z = \sum_{\alpha\beta} e^{-E_{\alpha\beta}/T} \tag{12.14}$$

Note that state configurations of the units having low energy have a high probability of occuring.

When training Boltzmann networks, the object is to adjust the weights w_{ij} so as to give the states of the *visible* units a particular desired probability distribution. The desired distribution is specified by clamping (i.e., keeping the state of the visible units fixed) training patterns α onto the visible units with appropriate probabilities. The probability of finding the visible units in states α irrespective of the hidden unit states β is

$$P_{\alpha}^{-} \;=\; \sum_{\beta} P_{\alpha\beta} \;=\; \frac{1}{Z} \sum_{\beta} e^{-E_{\alpha\beta}/T}, \tag{12.15}$$

where the superscript $-$ indicates the probability of visible state α given the visible units are allowed to change freely during simulation, i.e., they are "unclamped". The superscript $+$ will denote probabilities given that the visible units are *clamped* to a desired pattern α of the states, and P_{α}^{+} is then the *desired* probability of this pattern, determined by the environment. As a measure of the difference between the probability distributions of the unclamped visible units in the Boltzmann network and the environment we use the *Kullback-Leibler* measure, or relative entropy, as our cost function:

$$H(\mathbf{w}) \;=\; \sum_{\alpha} P_{\alpha}^{+} \ln \frac{P_{\alpha}^{+}}{P_{\alpha}^{-}} \;=\; -\sum_{\alpha} P_{\alpha}^{+} \ln P_{\alpha}^{-} + \mathit{const}, \tag{12.16}$$

where *const* is a constant determined solely by the environment, and is hence independent of the weights $\mathbf{w}$. $H(\mathbf{w})$ is always positive or zero, and is zero only if $P_{\alpha}^{-} = P_{\alpha}^{+}$ for all α. When annealing the Boltzmann network by lowering the temperature T we seek to lower the energy E of the network in order to focus the probability mass around certain desired patterns. Note that this energy minimization is *separated* from the minimization of the cost function Eq. (12.16), which is an overall measure of how close we are to the target distribution.

When training Boltzmann networks by minimizing the cost function using gradient descent, we need the gradient of $H(\mathbf{w})$ wrt. the weights w_{ij},

$$\begin{aligned}
\frac{\partial H(\mathbf{w})}{\partial w_{ij}} &= -\sum_{\alpha} P_{\alpha}^{+} \frac{\partial \ln P_{\alpha}^{-}}{\partial w_{ij}} && (12.17) \\
&= -\sum_{\alpha} P_{\alpha}^{+} \left[\sum_{\beta} \frac{P_{\alpha\beta}}{P_{\alpha}^{-}} \left(-\frac{1}{T} \cdot \frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right) - \sum_{\alpha'\beta} P_{\alpha'\beta} \left(-\frac{1}{T} \cdot \frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right) \right]
\end{aligned}$$

$$
\begin{aligned}
&= -\sum_{\alpha} P_{\alpha}^{+} \left[\sum_{\beta} P_{\beta|\alpha} \left(-\frac{1}{T} \cdot \frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right) - \sum_{\alpha'\beta} P_{\alpha'\beta} \left(-\frac{1}{T} \cdot \frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right) \right] \\
&= -\sum_{\alpha} P_{\alpha}^{+} \left[\left\langle -\frac{1}{T} \cdot \frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right\rangle_{\alpha}^{+} - \left\langle -\frac{1}{T} \cdot \frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right\rangle^{-} \right] \\
&= -\sum_{\alpha} P_{\alpha}^{+} \left\langle -\frac{1}{T} \cdot \frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right\rangle_{\alpha}^{+} + \left\langle -\frac{1}{T} \cdot \frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right\rangle^{-} \\
&= -\frac{1}{T} \sum_{\alpha} P_{\alpha}^{+} < s_i s_j >_{\alpha}^{+} + \frac{1}{T} < s_i s_j >^{-} \\
&= \frac{1}{T} \left(< s_i s_j >^{-} - < s_i s_j >^{+} \right)
\end{aligned}
$$

where $< \cdots >_{\alpha}^{+}$ is the mean value given that the visible units are clamped in states α, equivalent to the *correlations* between the two units in question. $< \cdots >^{-}$ is the mean when all units are free running, and the conditional probability $P_{\beta|\alpha}$ of hidden states β given the visible states α is defined as

$$
P_{\alpha\beta} = P_{\beta|\alpha} P_{\alpha}^{-} \tag{12.18}
$$

The above expression for the gradient reveals the difference between state representation $S_i \in \{0, 1\}$ and $s_i \in \{-1, 1\}$. In the former representation only units that are both "on" contribute to the mean values entering the gradient whereas the latter representation extends the mean values to be a true correlation measure, capturing negative contributions as well. Apparently this extension has a positive effect when training Boltzmann networks [PA87].

Generally it is impossible to calculate the numerical value of the partition function since it involves summing over all the possible state configurations of the Boltzmann network, which grows exponentially with the number of units in the network. When training we are therefore not able to verify whether the value of the cost actually decreases; however, if the size of the steps taken is small enough it will. Exact calculation of the gradient is also computationally intractable for the same reason as for the cost. The values of the correlations entering the gradient can however be *estimated* by simulating the network using a Monte Carlo method as mentioned above, gathering statistics about the correlations. The combination of having to relax the network using simulated annealing and then simulate in order to gather statistics makes the implementation of Boltzmann networks very computation-intensive.

We can now formulate a recipe for training and operation of Boltzmann networks:

1. Initialize the weights in the Boltzmann network to small random values, e.g., in the range $[-0.5; 0.5]$.

2. Present every example to the network by clamping the visible units accordingly. Perform simulated annealing for a finite sequence of decreasing temperatures; at each temperature, relax the network to equilibrium by randomly updating the units according to Eq. (12.11). At the fixed final temperature T_{final} simulate the network long enough to collect statistics in order to estimate the clamped correlations $< s_i s_j >^{+}$ of Eq. (12.17).

3. Repeat the process in step 2 but now with the visible units free running, estimating the unclamped correlations $< s_i s_j >^-$ in Eq. (12.17).

4. Update the weights by taking a small step η opposite the gradient, $\Delta w_{ij} = \eta\,(< s_i s_j >^+ - < s_i s_j >^-)$.

5. Repeat from step 2 until no further changes take place in the weights $w_{ij}, \ \forall i, j$.

12.3.1 Speeding up learning

In [PA87] it was shown that tremendous speedup can be obtained in the training of Boltzmann networks if applying *mean field* theory from statistical physics. Using mean field theory it is possible to effectively calculate *approximations* of the correlations entering the gradient Eq. (12.17) by *deterministic* methods, thus avoiding the computationally expensive simulated annealing and simulation. The idea behind mean field annealing is to replace all states s_i of the units by their average values $< s_i >$ which can be calculated by iteratively solving a set of nonlinear equations. The correlations are then calculated by making the approximations $< s_i s_j > \approx < s_i > < s_j >$.

Another way to avoid the costly annealing and simulation naturally is to compute the exact value of the expected values of the correlations entering the gradient expression. As mentioned previously this is generally an infeasible task even for small networks. In [SJ94] it was however shown that for certain tree-like topologies of Boltzmann networks it is practically possible to calculate the expectation values *exactly*, without resorting to simulated annealing and Monte Carlo simulation. The key issue when calculating correlations between two units is a recursive technique to reduce the entire network to a simple, tractable subnetwork retaining the same Boltzmann distribution as the original network. The technique leads to a simpler network consisting only of the two units in question having weights and biases that leaves the correlations and partition function of the original network unchanged. Using this technique it is also possible to compute the value of the entropic cost function, opening up for more advanced optimization methods. The reduction technique furthermore opens up for the computation of visible unit state probabilities from the Boltzmann distribution Eq. (12.15) and thus for modeling of probabilistic fixed length sequences using traditional Boltzmann networks in the same way as is done for HMMs. Similar reduction techniques are central to Boltzmann chains and zippers, as will become apparent in the following.

Further speedup of the learning might be obtained if a more efficient optimization method than gradient descent is used. E.g., in [Ped97] it was shown how using a second-order method speeds up learning considerably for deterministic recurrent networks compared to gradient descent. The method used there was the damped Gauss-Newton method in which the direction of the weight change in iteration k is computed as

$$\Delta \mathbf{w}_k = -[H''(\mathbf{w}_k)]^{-1} H'(\mathbf{w}_k) \tag{12.19}$$

corresponding to the direction towards the minimum of a second-order expansion of the entropic cost function around the current iteration point; $H''(\mathbf{w}_k)$ is here a positive definite approximation to the Hessian. The term *damped* refers to the use of a line search like e.g., simple bisection in order to determine the step size η. Line search is naturally only possible if the Boltzmann network is susceptible to the reduction technique [SJ94] mentioned above as the line search involves computing the value of the entropic cost.

In order to use a second-order method like the damped Gauss-Newton for training we need the second derivatives of the entropic cost. These second derivatives are calculated as:

$$\begin{aligned}
\frac{\partial^2 H(\mathbf{w})}{\partial w_{ij}\partial w_{pq}} &= -\left[\sum_\alpha \frac{P_\alpha^+}{P_\alpha^-}\cdot\frac{\partial^2 P_\alpha^-}{\partial w_{ij}\partial w_{pq}} - \sum_\alpha \frac{\partial P_\alpha^-}{\partial w_{ij}}\cdot\frac{P_\alpha^+}{\left(P_\alpha^-\right)^2}\cdot\frac{\partial P_\alpha^-}{\partial w_{pq}}\right] \\
&= -\left[\sum_\alpha \frac{P_\alpha^+}{P_\alpha^-}\cdot\frac{\partial^2 P_\alpha^-}{\partial w_{ij}\partial w_{pq}} - \sum_\alpha \frac{\partial \ln P_\alpha^-}{\partial w_{ij}}\cdot P_\alpha^+\cdot\frac{\partial \ln P_\alpha^-}{\partial w_{pq}}\right] \\
&\approx \sum_\alpha \frac{\partial \ln P_\alpha^-}{\partial w_{ij}}\cdot P_\alpha^+\cdot\frac{\partial \ln P_\alpha^-}{\partial w_{pq}}.
\end{aligned} \tag{12.20}$$

The approximation in Eq. (12.20) is equivalent to that made in *Fisher's method of scoring* [SW89] and also corresponds to the Gauss-Newton approximation to the Hessian of a quadratic cost in which the term with second derivatives is ignored. As is the case for the quadratic cost function, the approximation becomes exact in the limit of infinitely many examples, provided the network is not underparametrized. In that limit, the parameters that minimize the entropic cost function will converge towards a set of *optimal* weights $\mathbf{w}^*$ for which $P_\alpha^- = P_\alpha^+$, $\forall\alpha$. For these weights the term in Eq. (12.20) involving second derivatives of the unclamped probabilities P_α^- reads

$$\sum_\alpha \frac{P_\alpha^+}{P_\alpha^-}\cdot\frac{\partial^2 P_\alpha^-}{\partial w_{ij}\partial w_{pq}} = \sum_\alpha \frac{\partial^2 P_\alpha^-}{\partial w_{ij}\partial w_{pq}} = \frac{\partial^2}{\partial w_{ij}\partial w_{pq}}\left(\sum_\alpha P_\alpha^-\right) = 0, \tag{12.21}$$

where in the first step we used the fact that $P_\alpha^+ = P_\alpha^-$ at $\mathbf{w} = \mathbf{w}^*$, and in the last step that $\sum_\alpha P_\alpha^- = 1$. Thus, for weights sufficiently "close" to the optimal, $\mathbf{w}^*$, the term in Eq. (12.20) involving second derivatives will be "small". Assuming that the scaling by the inverse of the final temperature, $1/T_{\text{final}}$, is incorporated into the weights, the remaining term involves terms of a form calculated from Eq. (12.17):

$$\frac{\partial \ln P_\alpha^-}{\partial w_{ij}} = \left\langle -\frac{\partial E_{\alpha\beta}}{\partial w_{ij}}\right\rangle_\alpha^+ - \left\langle -\frac{\partial E_{\alpha\beta}}{\partial w_{ij}}\right\rangle^- = < s_i s_j >_\alpha^+ - < s_i s_j >^- . \tag{12.22}$$

Thus we obtain the simplified form of the second derivatives:

$$\begin{aligned}
\frac{\partial^2 H(\mathbf{w})}{\partial w_{ij}\partial w_{pq}} &\approx \sum_\alpha \frac{\partial \ln P_\alpha^-}{\partial w_{ij}}\cdot P_\alpha^+\cdot\frac{\partial \ln P_\alpha^-}{\partial w_{pq}} \\
&= \sum_\alpha P_\alpha^+ < s_i s_j >_\alpha^+ < s_p s_q >_\alpha^+ - < s_p s_q >^- \sum_\alpha P_\alpha^+ < s_i s_j >_\alpha^+ \\
&\quad - < s_i s_j >^- \sum_\alpha P_\alpha^+ < s_p s_q >_\alpha^+ + < s_i s_j >^- < s_p s_q >^- \\
&= < s_i s_j s_p s_q >^+ - < s_p s_q >^- < s_i s_j >^+ \\
&\quad - < s_i s_j >^- < s_p s_q >^+ + < s_i s_j >^- < s_p s_q >^- .
\end{aligned} \tag{12.23}$$

We note that the approximation to the second derivatives is positive definite which is of advantage when using the second derivatives for second-order optimization. The approximation involves only terms already computed when calculating the gradient, thus implementation is straightforward and requires little computational burden beyond that needed for computing the gradient.

12.3.2 Architecture optimization

Having calculated the second derivatives of the entropic cost function opens up for the use of two well-known methods for architecture optimization, namely the *pruning* schemes Optimal Brain Damage (OBD) [CDS90] and Optimal Brain Surgeon (OBS) [HS93]. Both methods involve training fully connected networks to a local minimum of the cost function and then iteratively eliminate parameters according to *saliency*, possibly incorporating retraining in between eliminations. The rationale behind both methods is that if we iteratively remove the least salient weights according to training error we gracefully relieve the danger of overfitting, i.e., specialize the network to features present in the training set not likely to be encountered in neither a validation set nor a real-life application. Iteratively applying a pruning scheme results in a nested family of networks between which we must choose the optimal. Optimality is often defined in terms of the performance on a validation set.

Elimination of a parameter is equivalent to setting it to zero, and for both methods the saliency of a parameter is set equal to the change in training error *estimated* from a second-order expansion given the parameter in question is set to zero. Using OBD the saliency for parameter j is computed under the condition that the remaining parameters in the network remains unchanged, and is calculated as [CDS90]

$$\delta H_j^{\mathrm{OBD}} = \frac{1}{2} \frac{\partial^2 H(\mathbf{w})}{\partial w_j^2} w_j^2 \tag{12.24}$$

where it is assumed that the first-order term is small and can be ignored. Provided the second-order expansion is accurate the OBD saliency provides an upper bound on the change in training error *after* retraining of the remaining parameters, as the training error is expected to decrease by such training.

The saliencies for the OBS pruning scheme are also calculated from a second-order expansion of the cost function, again assuming first-order terms can be ignored. The difference from OBD is that saliencies for OBS take into account the effect of reestimation of the remaining parameters to a new minimum within the second-order expansion. The saliency for the jth parameter is given by [HS93]

$$\delta H_j^{\mathrm{OBS}} = \frac{1}{2} \frac{w_j^2}{\mathbf{A}_{jj}^{-1}} \tag{12.25}$$

where $\mathbf{A}$ denotes the Hessian matrix of the cost function; the denominator is thus the jth diagonal element of the inverse Hessian. The reestimation of the remaining parameters is performed according to the expression

$$\Delta \mathbf{w}_j = -\frac{\mathbf{w}^{\mathrm{T}} \mathbf{e}_j}{\mathbf{e}_j^{\mathrm{T}} \mathbf{A}^{-1} \mathbf{e}_j} \mathbf{A}^{-1} \mathbf{e}_j \tag{12.26}$$

where $\mathbf{e}_j$ denotes the jth unit vector and T denotes transpose. Provided the second-order expansion is fairly accurate the OBS saliencies will provide a tighter bound than OBD on the training error resulting from further retraining of the remaining parameters in the network which is the actual measure of interest.

As a final note on pruning we note that the above mentioned pruning schemes can be applied to *all* structures of Boltzmann networks, not just the ones for which we can compute the value of the entropic cost function. This is so since both OBD and OBS work from approximations involving only second derivatives which can be either estimated or computed exactly from Eq. (12.23) regardless of the network structure.

12.3.3 The Potts model

The binary valued units used in Boltzmann networks might seem like a severe limitation to the application of modeling discrete valued probabilistic sequences since it is common that observations can take on more than two values. One way around this problem is to assign a *group* of units to each of the observations to be modeled and then apply a coding of the possible states; i.e., a group of m binary visible units are capable of representing 2^m different states.

There is however another solution to this problem. The binary valued units traditionally used in Boltzmann networks are known as the Ising model in statistical physics [HKP91]. The statistical physics literature also treats a *multistate* model, the so-called *Potts glass* model [Wu83]. From the Potts glass we can derive multi-state units for Boltzmann networks. The resulting multistate units are also denoted *graded neurons* [PS89]. Understanding the concept of multistate units in Boltzmann networks is important for the comprehension of Boltzmann chains and Boltzmann zippers and is therefore introduced in the following.

Multistate units derived from the Potts glass model are not confined to two states only, but can be more general m-state units. With each unit we associate a *state vector* $\mathbf{S}$ of dimension m, indicating the state of the unit; if the unit is in state i, $S_i = 1$ and the rest of the elements in $\mathbf{S}$ are zero; equivalently we may write $\mathbf{S} = \mathbf{e}_i$, where $\mathbf{e}_i$ is the ith principal unit vector. In Figure 12.5 we see an example of the interconnection of two such units, each having two states in order to emphasize the difference from the binary valued Ising spin glass units illustrated in Figure 12.3. We note that these two-state units are not connected by just a single bidirectional weight as in the traditional case, but rather by a weight *matrix*. Each *state* in one unit is connected by separate bi-directional weights to every state in the other; e.g., if unit γ in Figure 12.5 is in state γ_1 and unit δ is in state δ_2 the two units are connected by weight w_{12} , which is the weight applied when calculating the energy of this particular state configuration.

The energy function for a Boltzmann network using multistate units is similar to that of a traditional Boltzmann network Eq. (12.10), and for the simple network shown in figure 12.5 it is calculated as

$$E = -\frac{1}{2}\sum_{ij} w_{ij} S_i^\gamma S_j^\delta \tag{12.27}$$

where S_i^γ is equal to one if unit γ is in state i and zero otherwise. Since each unit can be

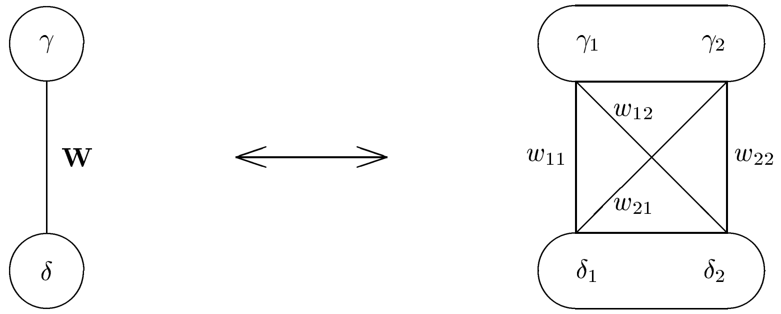


Figure 12.5: Multi state units derived from the Potts model.

in only one of m states at any given time, the energy E for each possible configuration of units is equal to minus the sum of the weights connecting the *active* states for this particular state configuration. Thus, the energy of the network in Figure 12.5 when for example in states (γ_2, δ_1) is $E = -w_{21}$.

Models using m-state Potts glass units can also be implemented in terms of traditional Boltzmann networks using binary valued Ising spin units. An m-state Potts glass unit can be thought of as a group of m $(0, 1)$ binary state units under the constraint that exactly one unit within the group is "on" at any given time,

$$\sum_i S_i = 1 \tag{12.28}$$

Every "sub unit" (state) within one group is connected to every sub unit in another group, leading to the weight matrix connections shown in Figure 12.5; there are no connections between the sub units within a group. The constraint that exactly one sub unit within a group is one at any given time can be imposed implicitly by adding a penalty term to the energy E [PS89]

$$E = -\frac{1}{2} \sum_{k \neq k'} \sum_{ij} w_{ij} S_i^k S_j^{k'} + \frac{\lambda}{2} \sum_k \left(\sum_i S_i^k - 1 \right)^2 \tag{12.29}$$

where λ is *sufficiently* large, $\lambda \to \infty$; k sums over the groups, i.e., γ and δ for the example in figure 12.5. We see that the penalty is zero if and only if exactly one sub unit within a group is "on", and very large and positive otherwise. With this interpretation of the multistate units we note that for implementation it is optional whether to be explicit about the units being multistate, or whether to be implicit about this by using a traditional Boltzmann network with a constraint added to the energy.

The partition function for a network involving multistate units is as previously calculated as the sum of exponentials to minus all the possible energy states of the network, that is, as a sum over all the possible state configurations $\alpha\beta$ of the visible and hidden units in the network Eq. (12.14), repeated here for convenience (assuming the temperature is $T = 1$):

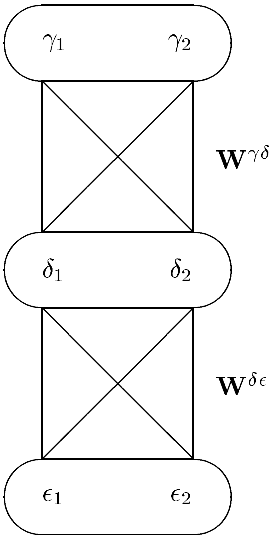
$$Z = \sum_{\alpha\beta} e^{-E_{\alpha\beta}} \tag{12.30}$$

For the simple network in figure 12.5 the partition function is calculated as

$$Z = e^{w_{11}} + e^{w_{12}} + e^{w_{21}} + e^{w_{22}} \tag{12.31}$$

In figure 12.6 is illustrated a slightly extended model now involving three two-state Potts glass units, along with all the possible (negative) energies (again assuming $T = 1$). The weights linking the active states associated with each configuration can be thought of in much the same way as the transition probabilities linking states in a particular state sequence from an HMM in that they indicate a "path" through the model.

In the following, we will often denote minus the sum of the weights connecting active units in a particular configuration a *path*. Thus, the partition function can be thought of as the sum of exponentials to every possible weight path through the network.



Possible weight paths:

$w_{11}^{\gamma\delta} + w_{11}^{\delta\epsilon}$	$w_{21}^{\gamma\delta} + w_{11}^{\delta\epsilon}$
$w_{11}^{\gamma\delta} + w_{12}^{\delta\epsilon}$	$w_{21}^{\gamma\delta} + w_{12}^{\delta\epsilon}$
$w_{12}^{\gamma\delta} + w_{21}^{\delta\epsilon}$	$w_{22}^{\gamma\delta} + w_{21}^{\delta\epsilon}$
$w_{12}^{\gamma\delta} + w_{22}^{\delta\epsilon}$	$w_{22}^{\gamma\delta} + w_{22}^{\delta\epsilon}$

Figure 12.6: Illustration of possible weight paths for partition function calculation.

12.4 Boltzmann chains

We have now described all prerequisites necessary for the introduction of Boltzmann chains. Boltzmann chains were introduced in [SJ95] and are a special type of Boltzmann networks particulary well suited for modeling discrete time series.

Boltzmann chains are Boltzmann networks using multistate units organized as shown in Figure 12.7 from which the chain-like structure is evident; hence the name of the model. The units of the Boltzmann chain are divided into L m-state visible units and L n-state hidden units where L is the length of the discrete series to be modeled. At each time step t a visible unit is connected to a corresponding hidden unit by weights B_{ij} collected in a matrix $\mathbf{B}$ of dimension $n \times m$ and between time steps the hidden units are connected by weights $A_{ii'}$ collected in a matrix $\mathbf{A}$ of dimension $n \times n$. It is important to note that the matrices $\mathbf{B}$ are *identical* for each time step t; this also applies for the matrices $\mathbf{A}$ between time steps. Thus, Boltzmann chains use extensive *weight sharing*. Besides the weights connecting the units in the Boltzmann chain the model also contains a set of *bias* weights Π_i on the first hidden unit corresponding to time step one, collected in the vector Π. The bias weights are equivalent to connections to states that are always "on".

If we denote the sequence of visible unit states as $\alpha = \{j_t\}_{t=1}^{L}$, corresponding to a configuration of the visible units, and the sequence of hidden unit states as $\beta = \{i_t\}_{t=1}^{L}$, corresponding to a configuration of the hidden units, then the energy of a particular configuration of the units in the Boltzmann chain is computed as

$$E_{\alpha\beta} = -\Pi_{i_1} - \sum_{t=1}^{L-1} A_{i_t i_{t+1}} - \sum_{t=1}^{L} B_{i_t j_t} \tag{12.32}$$

where we have summed out the explicit inclusion of the unit activations S_i^k included in Eq. (12.29), just as the factor $1/2$ has been canceled out due to the bidirectionality of the weights. In line with the description of multistate units in section 12.3.3, the energy for the Boltzmann chain corresponds to minus the sum of the weights connecting all the

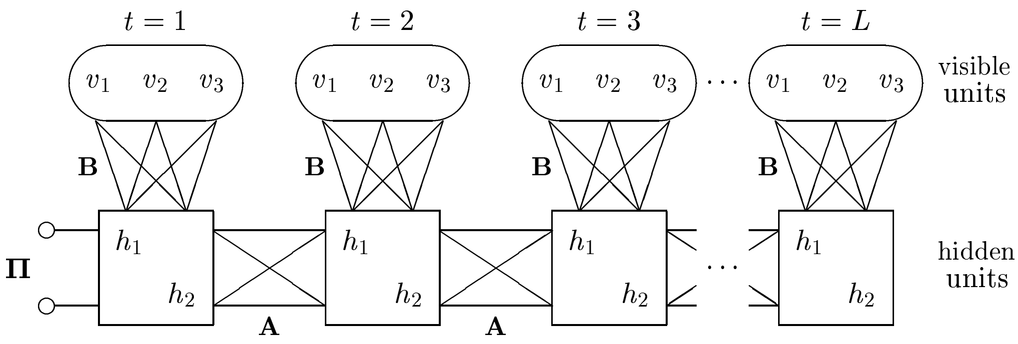


Figure 12.7: The architecture of a Boltzmann chain for modeling sequences of length L. In this example is used visible units with $m = 3$ states and hidden units with $n = 2$ states. Note that the weights connecting the units are identical at each timestep.

active states of the model for a particular state configuration $\alpha\beta$ along with minus the bias weight on the initial hidden state.

The probability of a particular state configuration of the units in a Boltzmann chain is given by the Boltzmann distribution Eq. (12.13 and 12.14). The likelihood of a particular state sequence α of the visible units is given by summing over all possible hidden unit configurations as in Eq. (12.15), repeated below:

$$P_\alpha^- = \frac{1}{Z} \sum_\beta e^{-E_{\alpha\beta}/T} = \frac{Z_\alpha}{Z}, \tag{12.33}$$

In the final expression, Z_α denotes the *clamped* partition function, that is, the partition function obtained if the visible units are clamped in states α and the sum in Eq. (12.14) thus only involves the hidden states β.

As is the case for HMMs, the modeling object is to determine the weights $(\mathbf{\Pi}, \mathbf{A}, \mathbf{B})$ that maximize the likelihood of the observed sequences used for training. *Maximizing* the likelihood of the observed training sequences is identical to *minimizing* the relative entropy Eq. (12.16) if we use the empirical distribution for the p sequences in the training set $D = \{\alpha_k | k = 1, \ldots, p\}$:

$$P_\alpha^+ = \begin{cases} \frac{1}{p} & , \quad \alpha \in D \\ 0 & , \quad \alpha \notin D \end{cases}. \tag{12.34}$$

For training Boltzmann chains using gradient methods, we need the derivatives of the relative entropy Eq. (12.16), repeated below for convenience:

$$\begin{aligned} H(\mathbf{w}) &= \sum_\alpha P_\alpha^+ \ln \frac{P_\alpha^+}{P_\alpha^-} && (12.35) \\ &= -\sum_\alpha P_\alpha^+ \ln P_\alpha^- + const && (12.36) \\ &= \ln Z - \sum_\alpha P_\alpha^+ \ln Z_\alpha + const && (12.37) \end{aligned}$$

These derivatives are obtained as follows:

$$\frac{\partial H(\mathbf{w})}{\partial w} = \frac{\partial}{\partial w}\left(\ln Z - \sum_\alpha P_\alpha^+ \ln Z_\alpha\right) \tag{12.38}$$

$$= \frac{1}{Z} \cdot \frac{\partial Z}{\partial w} - \sum_{\alpha} P_{\alpha}^{+} \frac{1}{Z_{\alpha}} \cdot \frac{\partial Z_{\alpha}}{\partial w} \tag{12.39}$$

$$= \sum_{\alpha\beta} \frac{e^{-E_{\alpha\beta}/T}}{Z \cdot T} \cdot \frac{\partial(-E_{\alpha\beta})}{\partial w} - \sum_{\alpha} P_{\alpha}^{+} \sum_{\beta} \frac{e^{-E_{\alpha\beta}/T}}{Z_{\alpha} \cdot T} \cdot \frac{\partial(-E_{\alpha\beta})}{\partial w} \tag{12.40}$$

where w represents a parameter in the network and the energy $E_{\alpha\beta}$ is given by Eq. (12.32). If w is e.g., one of the parameters in $\mathbf{A}$ connecting the hidden units, we obtain

$$\left.\frac{\partial(-E_{\alpha\beta})}{\partial w}\right|_{w=A_{pq}} = \sum_{t=1}^{L-1} \left.\frac{\partial A_{i_t i_{t+1}}}{\partial w}\right|_{w=A_{pq}} = \sum_{t=1}^{L-1} \delta_{p i_t} \delta_{q i_{t+1}} \tag{12.41}$$

where δ_{ij} stands for the Kronecker delta function and the sum over the sequence length is due to the weight sharing. Similar expressions are obtained for the parameters in $\mathbf{B}$ and Π. By insertion of these expressions in Eq. (12.40) we can write the derivatives for the parameters in the Boltzmann chain as

$$\frac{\partial H(\mathbf{w})}{\partial B_{ij}} = \frac{1}{T} \sum_{t=1}^{L} \left[\langle \delta_{i i_t} \delta_{j j_t} \rangle^{-} - \langle \delta_{i i_t} \delta_{j j_t} \rangle^{+} \right] \tag{12.42}$$

$$\frac{\partial H(\mathbf{w})}{\partial A_{ii'}} = \frac{1}{T} \sum_{t=1}^{L-1} \left[\left\langle \delta_{i i_t} \delta_{i' i_{t+1}} \right\rangle^{-} - \left\langle \delta_{i i_t} \delta_{i' i_{t+1}} \right\rangle^{+} \right] \tag{12.43}$$

$$\frac{\partial H(\mathbf{w})}{\partial \pi_i} = \frac{1}{T} \left[\langle \delta_{i i_1} \rangle^{-} - \langle \delta_{i i_1} \rangle^{+} \right] \tag{12.44}$$

where $< \cdot >^{-}$ and $< \cdot >^{+}$ denote expectations over the free and clamped Boltzmann distributions, respectively. We see that the derivatives can be obtained by estimation (or exact computation as we shall see) of the correlations between the *states* of neighboring units, similar to the expression Eq. (12.17).

12.4.1 Exact learning in Boltzmann chains

For Boltzmann chains it is possible to numerically calculate the value of the partition function Eq. (12.14), and thereby the value of both the cost function Eq. (12.37) and the correlations entering the expressions for the gradient Eqs. (12.42 – 12.44) *exactly*, using reduction techniques similar to those described in [SJ94] for traditional Boltzmann networks. It is therefore not necessary to anneal the Boltzmann chain, and in the following we therefore work from temperature-rescaled weights [SJ94, SJ95], $w^{new} = w^{old}/T_{\text{final}}$.

When calculating the partition function or the gradient, the method involves reducing the original chain into a simple structure by calculation of *effective* weights that will not change the value of the "original" partition function or correlations when calculated from the reduced structure. This structure, shown in Figure 12.8, involves only the two units between which we want to calculate correlations, and possibly a bias weight for each state in each unit. The method for reducing to this structure will be described in section 12.4.1.1 below.

The partition function is easily calculated for this structure,

$$Z = \sum_{pq} e^{-E_{pq}} = \sum_{pq} e^{b_p^{\gamma} + w_{pq} + b_q^{\delta}} \tag{12.45}$$

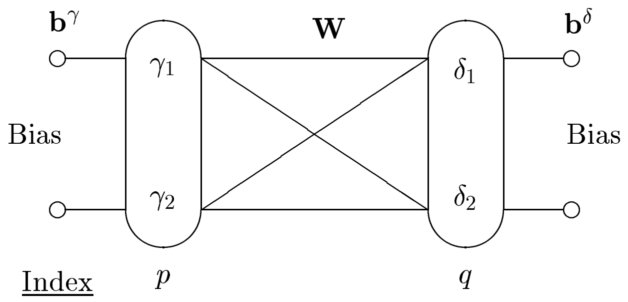


Figure 12.8: Reduced structure for calculation of partition functions and correlations for Boltzmann chains. The structure consists of two connected units with bias weights for each unit; here, each unit has two states.

It is thus straightforward to calculate the value of the entropic cost function Eq. (12.37) based on reduced models as the one in Figure 12.8. The reduced models involved in computing the cost are the ones resulting from 1) the Boltzmann chain with the visible units clamped in each pattern and 2) the Boltzmann chain with the visible units unclamped.

The derivatives of the logarithm to the partition function wrt. the weights are identical to the correlations between the connected states. The correlations for the reduced structure in Figure 12.8, $< S_i^\gamma S_j^\delta >$, where $S_i^\gamma \in \{0,1\}$ denotes state i for unit γ, are obtained as

$$\begin{aligned} \left\langle S_i^\gamma S_j^\delta \right\rangle &= \sum_{pq} \frac{e^{-E_{pq}}}{Z} \frac{\partial(-E_{pq})}{\partial w_{ij}} \\ &= \frac{e^{b_i^\gamma + w_{ij} + b_j^\delta}}{Z} \end{aligned} \tag{12.46}$$

Comparing Eq. (12.46) and Eqs. (12.42 – 12.44) we see that the gradient terms for the Boltzmann chain can be obtained as the sums of the correlations between pairs of neighbouring units. So, calculating the gradient for a Boltzmann chain lends itself naturally to the reduced structure outlined above.

12.4.1.1 Exact learning: Clipping

A technique for reducing Boltzmann chains to the simple structure shown in figure 12.8 without affecting the value of the partition function was given in [SJ95] and will be explained in the following. The technique involves the reduction of a "dangling" unit into effective bias weights on its parent, as illustrated in Figure 12.9 and was therefore originally termed "pruning" in [SJ95]. Unfortunately the term "pruning" is also widely known in association with architecture optimization which might cause some confusion. Since the method described here has nothing to do with architecture optimization, but is merely a tool for reducing a given Boltzmann network topology into an equivalent computationally tractable structure, it is here suggested to instead call the method "clipping" for obvious reasons.

The clipping procedure is illustrated by an example on a structure composed of two-state units, where it is shown how to sum over the degrees of freedom represented by the dangling unit γ in the left model of Figure 12.9, resulting in effective bias weights $\mathbf{b}^\epsilon$ on

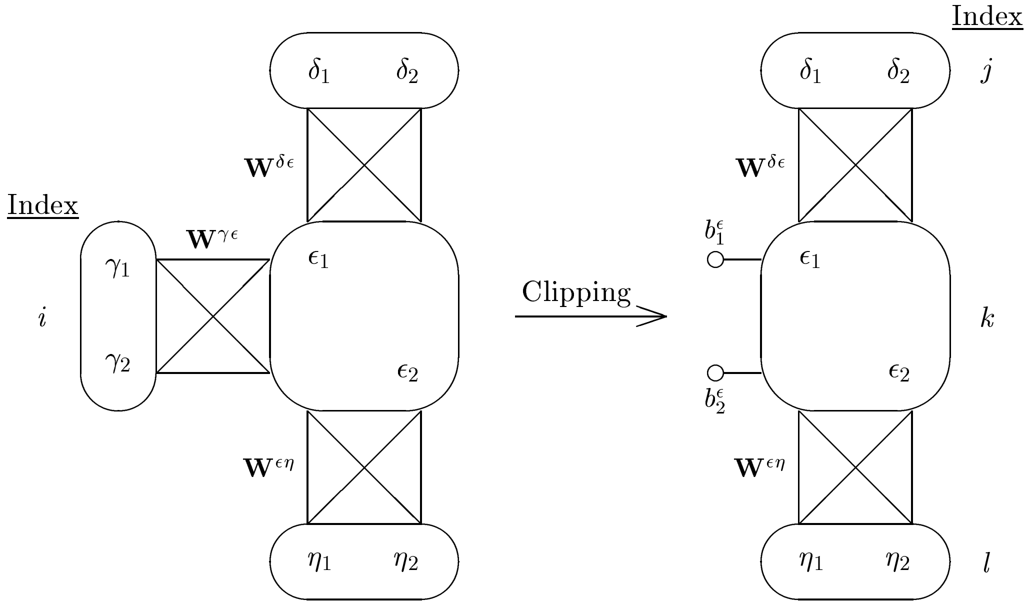


Figure 12.9: Clipping. The degrees of freedom of unit γ are summed over, yielding an effective bias $\mathbf{b}^\epsilon$ on unit ϵ.

the parent unit ϵ, as shown in the right model of Figure 12.9. The partition function Z initially derived from the structure to the left in the figure is calculated as

$$\begin{aligned} Z &= \sum_{ijkl} e^{w_{ik}^{\gamma\epsilon}+w_{jk}^{\delta\epsilon}+w_{kl}^{\epsilon\eta}} = \sum_{ijkl} e^{w_{ik}^{\gamma\epsilon}} \cdot e^{w_{jk}^{\delta\epsilon}+w_{kl}^{\epsilon\eta}} \\ &= \sum_{jkl}\left[e^{w_{jk}^{\delta\epsilon}+w_{kl}^{\epsilon\eta}} \cdot \sum_i e^{w_{ik}^{\gamma\epsilon}}\right] = \sum_{jkl} e^{w_{jk}^{\delta\epsilon}+w_{kl}^{\epsilon\eta}} \cdot e^{b_k^\epsilon} \\ &= \sum_{jkl} e^{w_{jk}^{\delta\epsilon}+w_{kl}^{\epsilon\eta}+b_k^\epsilon} \end{aligned} \tag{12.47}$$

where we see that the bias weights $\mathbf{b}^\epsilon$ on unit ϵ resulting from summing over the degrees of freedom represented by unit γ are calculated as

$$e^{b_k^\epsilon} = \sum_i e^{w_{ik}^{\gamma\epsilon}} \iff b_k^\epsilon = \ln\left(\sum_i e^{w_{ik}^{\gamma\epsilon}}\right) \tag{12.48}$$

In the example shown here there are no bias weights on unit γ which is "clipped" away; inclusion of these is however straightforward, as all that is required is to add the bias weights to the exponents on the right hand sides of Eq. (12.48).

When calculating the exact value of the entropic cost Eq. (12.37) we start by e.g., computing the value of the unclamped partition function Z, i.e., the value when the states of the visible units are allowed to change freely; refer to Figure 12.7. The procedure is as follows: First we use the clipping procedure described above to reduce all the visible units to effective bias weights on the corresponding hidden units. As the weight matrices $\mathbf{B}$ connecting the visible and hidden units are identical for each time step (refer to Figure 12.7) the biases resulting from the clipping will be identical for each hidden unit. Then, the clipping procedure is applied iteratively on the resulting reduced structure starting either from the left (forward in time) or right (backward in time) side of the structure until we arrive at a structure similar to Figure 12.8 for which the value of the partition function is easily calculated.

When calculating the value of the clamped partition function Z_α where the visible units are clamped onto the states of a particular observation sequence $\{j_t\}_{t=1}^L$ the approach is similar. However, we do not have to "clip" the visible units as the clamped visible unit states correspond to additional bias weights to the hidden units. This is because only one state is permanently "on" in a clamped visible unit, effectively turning the weights leading from the clamped state to the hidden unit state into bias weights.

The procedure for computing partition functions outlined above is similar to the forward-backward procedure used for HMMs to compute e.g., the likelihood of an observation sequence given a model [Rab89]. Clipping the Boltzmann chain from "the left" is similar to the forward procedure and clipping from "the right" is similar to the backward procedure.

When computing gradients for a Boltzmann chain we use a similar approach to that of computing partition functions. For every weight matrix in the chain we iteratively reduce to a structure involving only the two units connected by the currently considered matrix. Correlations are computed as described in section 12.4.1 above and added to the appropriate expression of Eqs. (12.42 – 12.44).

12.4.2 Link to HMMs

An interesting property of Boltzmann chains is that they can represent any first-order HMM [SJ95]. If we compare the HMM distribution Eq. (12.1) for a sequence of hidden states $\beta = \{i_t\}_{t=1}^L$ and visible symbols $\alpha = \{j_t\}_{t=1}^L$ with the Boltzmann distribution Eq. (12.13) for the Boltzmann chain Eq. (12.32), we see that the two distributions are identical if we choose the weights in the Boltzmann chain as

$$
\begin{aligned}
B_{ij} &= T \ln b_{ij} && (12.49)\\
A_{ii'} &= T \ln a_{ii'} && (12.50)\\
\Pi_i &= T \ln \pi_i && (12.51)
\end{aligned}
$$

in which case we have

$$
\begin{aligned}
P_{\alpha\beta} &= \frac{1}{Z} e^{-E_{\alpha\beta}/T} && (12.52)\\
&= \frac{1}{Z} e^{\ln \pi_{i_1} + \sum_{t=1}^{L} \ln b_{i_t j_t} + \sum_{t=1}^{L-1} \ln a_{i_t i_{t+1}}}\\
&= \pi_{i_1} a_{i_1 i_2} a_{i_2 i_3} \ldots a_{i_{L-1} i_L} b_{i_1 j_1} b_{i_2 j_2} \ldots b_{i_L j_L}\\
&= P(\{i_t, j_t\})
\end{aligned}
$$

As mentioned above, when working with Boltzmann chains we usually use temperature-rescaled weights, in which case we set $T = 1$ in the above expressions for convenience.

When modeling using HMMs or Boltzmann chains the goal is to adjust the parameters so as to maximize the probability $P(\{j_t\}) = P_\alpha^-$ of the observed training sequences $\{j_t\}_{t=1}^L = \alpha$. Learning in HMMs can thus be viewed as a special case of learning in Boltzmann chains, in which learning is performed subject to the constraints

$$\sum_i e^{\Pi_i} = 1 \tag{12.53}$$

$$\sum_j e^{B_{ij}} = 1 \tag{12.54}$$

$$\sum_{i'} e^{A_{ii'}} = 1 \tag{12.55}$$

These constrains imply $Z = 1$, as used in Eq. (12.52).

From the above, we see that any first-order HMM can be represented as a Boltzmann chain. However, not all Boltzmann chains can be represented as an HMM. The weights in the Boltzmann chain can represent arbitrary energies between $\pm\infty$, see Eq. (12.32), whereas the HMM parameters represent probabilities that are constrained to obey sum rules such as Eqs. (12.53–12.55) [SJ95]. In general, the Boltzmann chain thus has more degrees of freedom than a first-order HMM.

In [Mac96] it was shown how to transform a large class of Boltzmann chains into an equivalent HMM representation. It was shown that *if the final hidden state i_L of a Boltzmann chain is constrained to be a* particular *end state, then the distribution over sequences is identical to that of a hidden Markov model.* Thus, if we force the *final* hidden state i_L of a Boltzmann chain to be the same for all sequences, it is possible to represent the Boltzmann chain as an HMM also having a fixed end state.

The transformation from a Boltzmann chain to an HMM is done by manipulating the weights in the exponent $-E_{\alpha\beta}$ Eq. (12.32) of the Boltzmann distribution Eq. (12.13) by adding and subtracting terms to the weights in such a way that the probability distribution is left unchanged. The additional terms are then solved for, enabling the transformation to HMM representation. Appendix F includes a detailed explanation of the transformation.

Whereas traditional HMMs are special cases of the Boltzmann chain it is interesting to note that there is a direct one to one correspondence between *any* Boltzmann chain and the "unnormalized" HMM described in section 12.2.2. When converting from a Boltzmann chain into an equivalent "unnormalized" HMM we simply compute the HMM parameters as

$$b_{ij} = e^{B_{ij}} \tag{12.56}$$

$$a_{ii'} = e^{A_{ii'}} \tag{12.57}$$

$$\pi_i = e^{\Pi_i} \tag{12.58}$$

where it is assumed that $T = 1$. When converting the other way around we use Eqs. (12.49–12.51). In this case, by comparing the partition functions for the two models Eq. (12.14) to Eq. (12.8) and the joint probabilities Eq. (12.52) to Eq. (12.9) we see that the distributions for the "unnormalized" HMM and the Boltzmann chain are identical.

Figure 12.10 illustrates the relationship between the Boltzmann chain and various HMM variants as described above. The example in the figure involves models with two hidden states and observations having three states. Note how the Boltzmann chain corresponds to the HMM unfolded in time.

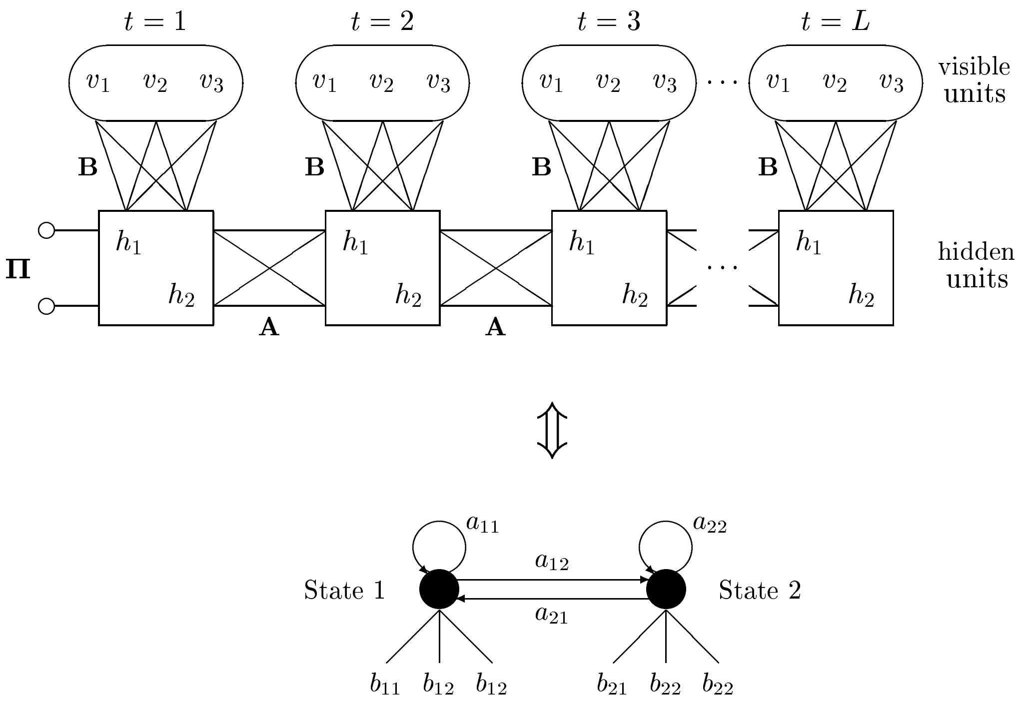


Figure 12.10: Relationship between a Boltzmann chain and a corresponding HMM.

12.4.3 Notes on training chains

Training of Boltzmann chains by minimization of the entropic cost function on a set of training sequences can naturally be performed using gradient descent as indicated in section 12.4. In section 12.4.1 it was shown that it is fairly straightforward to compute the exact value of the gradient and thus also the exact value of the second derivatives Eq. (12.23) and of the entropic cost function. Thus, we can apply the effective damped Gauss-Newton second-order optimization method involving a line search, described in section 12.3.1, to the training of Boltzmann chains.

The Gauss-Newton method involves solving a system of linear equations in each iteration. Of prime importance to the solution of this system of equations is naturally that the Hessian matrix is not singular. This will be the case if rows/colums are linearly dependent and will arise if the entropic cost function is constant in certain directions in parameter space. Unfortunately Boltzmann chains have a built-in rank deficiency in the Hessian due to this problem. For any parametrization of the Boltzmann chain we can add an arbitrary constant c to all elements of any of the parameter matrices $(\mathbf{\Pi}, \mathbf{A}, \mathbf{B})$ without affecting the value of the entropic cost function. Adding the constant c to e.g., the $\mathbf{B}$-matrix means that the energy of *any* state configuration $\alpha\beta$ Eq. (12.32) will be calculated as

$$\begin{aligned} E_{\alpha\beta} &= -\Pi_{i_1} - \sum_{t=1}^{L}(B_{i_t j_t} + c) - \sum_{t=1}^{L-1} A_{i_t i_{t+1}} \\ &= -\Pi_{i_1} - \sum_{t=1}^{L} B_{i_t j_t} - \sum_{t=1}^{L-1} A_{i_t i_{t+1}} - Lc \end{aligned} \quad (12.59)$$

When applying exponentials for calculation of the partition function Eq. (12.14) the constant thus corresponds to multiplication by the constant e^{Lc} to each factor of the sum.

When computing likelihoods P_α^- Eq. (12.33) the constant cancels out and the value of the cost function Eq. (12.37) is left unchanged.

In order to handle this rank-deficiency problem for Boltzmann chains we need to augment the entropic cost function with a *regularization term*. A simple yet highly effective regularizer is the simple quadratic weight decay and we thus arrive at the augmented cost function,

$$C(\mathbf{w}) = H(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w} \tag{12.60}$$

where λ is a small positive constant. The regularization ensures that the parameters can no longer be changed along any direction in parameter space without affecting the total cost function. When calculating the Hessian of the augmented cost function the effect of the weight decay term is addition of λ to the diagonal, and thereby incrementation of all eigenvalues by λ. Weight decay thus removes the rank-deficiency and generally makes the Hessian more well-conditioned. Ill-conditioning as described here is also a problem for deterministic recurrent networks which is illustrated in [Ped97].

The name "weight decay" for the regularizer is due to the bias towards zero that is introduced for all parameters. For deterministic networks it is well known that weight decay has a "smoothing" effect on the surface of the cost function; see e.g., [PH95] for a graphical illustration of this effect. The "encouragement" of the parameters to attain the same value will also have a smoothing effect on the Boltzmann distribution since uniformly valued parameters will yield a uniform distribution; augmenting the entropic cost with a simple weight decay term thus biases the Boltzmann distribution towards a uniform distribution.

As a final comment on training Boltzmann chains it should be noted that we are not confined to modeling sequences of fixed length only. As the parameters $\mathbf{A}$ and $\mathbf{B}$ are identical in each time step, the length of the chain can be varied according to the length of the sequence in question. This is the case during both training and application and is similar to the approach adopted when modeling sequences of different lengths using HMMs. The only implication of varying sequence lengths is that when computing likelihoods according to Eq. (12.33) the value of the normalizing partition function Z will depend on the sequence length, and the final separation of terms in the expression for the entropic cost Eq. (12.37) is then no longer valid.

12.4.4 Notes on pruning chains

Architecture optimization as described in section 12.3.2 is naturally also an option when working with Boltzmann chains. The essence of the methods reviewed is to remove degrees of freedom not necessary to the network in order to relieve the danger of overfitting. Removal of degrees of freedom is obtained by fixing parameters at some value. Usually, a "neutral" value is chosen which ensures that the pruned parameters no longer influence the model, which corresponds to the parameters being absent from the model.

The neutral value used by the pruning schemes OBD and OBS reviewed in section 12.3.2 is zero. When pruning Boltzmann networks, setting a parameter equal to zero means that it no longer has any influence on the energy Eq. (12.10) of the network state configuration and thus does no longer influence the model. The function of weights in a traditional Boltzmann network is to let the units influence each other so as to put constraints onto the possible state configurations of the network. Removing weights by pruning them to the value zero loosens these constraints and will thus have a smoothing effect on the distribution of state configurations. In the extreme case when all weights are pruned away the units of the Boltzmann network are completely disconnected in which case all state configurations become equally likely.

Absent parameters in an HMM can also be thought of as being set to the value zero as all state sequences involving the absent parameters will then occur with probability zero and will therefore not contribute when computing likelihoods. Examples of "pruned" HMMs are the so-called left-to-right models often used in speech recognition [Rab89] in which transitions are only possible to states having a number which is greater than or equal to the current; all other transitions have the probability zero. Setting parameters to zero in an HMM is equivalent to putting ultimate constraints on the possible state sequences that can occur and thus has the exact opposite effect of pruning parameters in a traditional Boltzmann network.

Since Boltzmann chains have an interpretation as both Boltzmann networks and as HMMs it seems that pruning parameters can have *two* meanings, namely imposing *less* constraints onto the possible state configurations and imposing *more* constraints onto the possible configurations. When setting a pruned and thus absent parameter to the value zero in the Boltzmann chain, the parameter will no longer provide a contribution to the energy Eq. (12.32) of the chain for state configurations including states connected by the parameter; refer also to Figure 12.7. In the extreme case when all parameters have been pruned to zero we end up with a Boltzmann chain in which all state configurations are equally likely. Pruning parameters to the value zero thus has the effect of loosening the constraints imposed on the possible state configurations, corresponding to the Boltzmann network interpretation.

From the above it is clear that pruning a parameter in a Boltzmann chain to the value zero does *not* result in the effect that particular state configurations or sequences involving the pruned parameter will occur with probability zero according to the Boltzmann distribution Eq. (12.13). When interpreting the Boltzmann chain as an HMM, pruning parameters to the value zero does therefore not correspond to setting probabilities to zero in the corresponding HMM as *all* state sequences can still occur with non-zero probability. A "value" for a Boltzmann chain parameter that corresponds to the probability zero in an HMM is "$-\infty$" as can be seen directly from Eqs. (12.56–12.58). Setting a parameter in a Boltzmann chain to a large negative value equivalent to $-\infty$ has the effect that state sequences involving this parameter will have infinite energy as seen from Eq. (12.32). When calculating the probability of an infinite energy state configuration from the Boltzmann distribution Eq. (12.13) we see that it will occur with probability zero, as desired.

When pruning Boltzmann chains we are consequently faced with the choice between two values for the pruned parameters. Either we must choose to prune to the value zero which corresponds to putting less constraints onto the possible state sequences, thus biasing the Boltzmann distribution towards a smoother uniform distribution. Alternatively we prune to $-\infty$ which corresponds to putting ultimate constraints onto the possible state sequences, thus biasing towards a peaked, non-smooth distribution. Which choice, zero or $-\infty$, that is the "right" one will depend on the application. E.g., in speech recognition there is strong empirical justification for using HMMs like the left-to-right model [Rab89] in which certain hidden state sequences cannot occur. In this case it seems most appropriate to prune Boltzmann chain parameters to $-\infty$ leading to zero transition probabilities if interpreted as HMMs.

An advantage when pruning Boltzmann chain parameters to $-\infty$ is that the approach provides a way in which to actually prune traditional Hidden Markov Models. A fully trained HMM can be converted into a Boltzmann chain or, alternatively, the initial training can be performed on the Boltzmann chain as well. Pruning and further retraining is then performed on the Boltzmann chain and the optimal chain is converted into a corresponding HMM. The necessary constraint on the HMMs is that they must have a mandatory end state so that conversion can be performed using Mackays recipe described in section F. This is however a commonly applied constraint in the HMM literature [Mac96]. The reason for not performing pruning on the HMM directly is primarily due to the constraint that the probabilities in an HMM must sum to one at any given time. Setting a parameter to zero in an HMM leads to the problem of how to adjust the remaining parameters in the HMM so that the constraint is still satisfied. This is not a problem for Boltzmann chains as the normalization is handled indirectly; changing a single Boltzmann chain parameter may lead to a change in several HMM parameters when converting using Mackays recipe.

A disadvantage when pruning Boltzmann chain parameters to $-\infty$ is that we can no longer *estimate* the change in cost when pruning a parameter, that is, compute saliencies from expressions like Eq. (12.24) and Eq. (12.25) as the second-order expansion does not extend all the way to $-\infty$. Instead we have to determine the saliencies by setting each parameter equivalent to $-\infty$ in turn and *compute* the actual change in the entropic cost function. This leads to an extra computational burden but also to very accurate saliencies.

Another disadvantage when pruning parameters to $-\infty$ is that successful conversion from any Boltzmann chain to a corresponding HMM using Mackays recipe described in section F is no longer guaranteed to be possible. This is so since the elements of the matrix in Eq. (F.10) can now obtain the value zero thus violating the requirements for the Perron-Frobenius theorem which can no longer guarantee that a positive eigenvalue with positive associated eigenvector will exist. This can lead to problems but experience from this work indicates that this is seldomly a problem in practice. When problems do arise it is most often due to degenerate models for which it is obvious that they could be implemented as an equivalent model using less parameters.

A practical consideration when pruning Boltzmann chain parameters to $-\infty$ is that we should not prune the parameters $\mathbf{B}$ connecting the visible unit states with the hidden unit states. This is due to the pragmatic observation that during the architecture optimization visible-to-hidden parameters unnecessary to "explain" the training set get pruned away. This may actually lead to the model adapting itself to peculiarities in the training set not present in a test set in such a way that certain sequences in the test set are modeled to occur with probability zero; this leads to an infinite test error as can be seen from Eq. (12.33). Not pruning the visible-to-hidden parameters means that any *observation* sequence will always be modeled with non-zero probability as every observation can occur with non-zero probability in every state. Drawing the analogy to HMMs this is equivalent to only putting restrictions on the possible *transition* probabilities and thus the possible *hidden* state sequences while leaving emission probabilities unrestricted, which is customary as seen in [Rab89].

Other practical considerations concern the implementation when pruning parameters of the Boltzmann chain to $-\infty$. Firstly, from Eq. (12.48) we see that setting parameters equivalent to $-\infty$ might actually lead to the application of the logarithm ln to the value zero when clipping. This should of course be handled in some appropriate way, possibly by having ln returning the large negative value used for $-\infty$ in this case. Secondly, parameters pruned to $-\infty$ should of course not be considered when computing the value of the weight decay term in Eq. (12.60).

12.5 Boltzmann zippers

When modeling stochastic signals it is sometimes desirable to build a unified model from two simultaneously occuring and correlated observation sequences. Often the two sequences will have variations on disparate time scales, i.e., observations occur at different time rates. This is an important problem in speech recognition [JR91] where e.g., spectral parameters varying on a time scale of 10 msec are combined with the signal energy varying on a time scale of 100 msec. It is also a problem of great relevance to *speechreading* [Sto97, HSP96] in which both audio and video information is used to recognize speech.

One way in which to build a unified model from the two simultaneous sequences is simply to form the Cartesian product of their observation state spaces and then model as usual, e.g., using an HMM or a Boltzmann chain. This way of integrating the observation sequences is called *early integration* [HSP96]. Theoretically this simple method is very powerful since it indeed allows the model to capture any correlation that might exist between the observations from the two sequences. In practice however, early integration poses several problems. Forming the Cartesian product of the observation state spaces will lead to an explosion in the number of possible states and thus in the number of parameters that must be fitted from the training data; early integration models therefore lead to increased demand for training data. Another problem is that of different time scales; depending on the time scale chosen for the model, early integration might lead to an oversampling of the slowest sequence and thus redundancy or an undersampling of the fastest sequence and thereby loss of information.

Another approach is to build two separate models, one for each of the two observation sequences. The model outputs in the form of sequence likelihoods are then combined in some manner by e.g., multiplication of the likelihoods (i.e., assuming independence between the two sequences!) or by some weighting scheme expressing the reliability or relative importance of each sequence [Hen96]. This strategy is called *late integration* [HSP96]. The advantages of late integration is that we avoid the curse of dimensionality problem associated with forming the Cartesian product of state spaces. Furthermore, since the two models work independently of one another, problems with oversampling or undersampling can be avoided. This indepencence might however also be a disadvantage since possibly relevant timing information regarding correlation between observations is lost and can not be made use of.

The *Boltzmann zipper* is a model type that allows for *intermediate integration* of observation sequences on disparate time scales without the need for forming the intractable Cartesian product of the state spaces. The structure of a Boltzmann zipper is illustrated in Figure 12.11 and is seen to be composed of two parallel Boltzmann chains with hidden units connected by cross-connection weights $\mathbf{C}$; hence the name of the model [SL96]. If the sequences to be modeled are on disparate time scales we distinguish between the chains for each sequence as the *fast* and the *slow* chain. Each of the hidden units in the slow chain is connected to the corresponding hidden units in the fast chain as seen in Figure 12.11. Note that the two chains have separate parameter sets and that the parameters $\mathbf{C}$ are identical for each cross-connection; if the number of hidden states in the fast chain is n_f and the number of hidden states in the slow chain is n_s, the dimension of $\mathbf{C}$ is thus $n_f \times n_s$.

If we denote the sequences of visible unit and hidden unit states for the fast chain as $\alpha^f = \{j_t^f\}_{t=1}^{Lf}$ and $\beta^f = \{i_t^f\}_{t=1}^{Lf}$, respectively, and the sequences of visible unit and hidden unit states for the slow chain as $\alpha^s = \{j_t^s\}_{t=1}^{L^s}$ and $\beta^s = \{i_t^s\}_{t=1}^{L^s}$, respectively, the energy

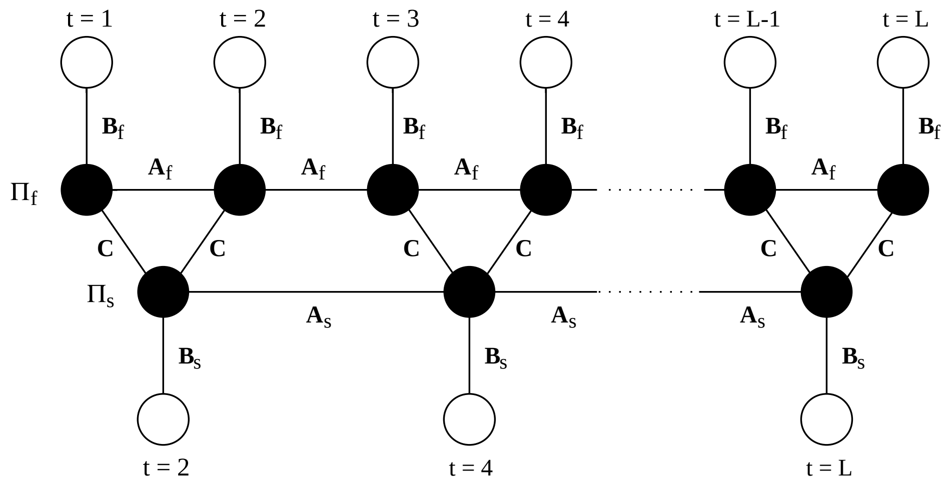


Figure 12.11: Structure of the Boltzmann zipper. Here, the time scales have a 2:1 disparity. The upper chain modeling the faster sequence is denoted the *fast chain*, the lower chain is denoted the *slow chain*.

of a particular configuration of the units in the Boltzmann zipper can be expressed as

$$\begin{aligned} E_{\alpha^f \beta^f \alpha^s \beta^s} &= -\Pi^f_{i_1^f} - \sum_{t=1}^{L^f-1} A^f_{i_t^f i_{t+1}^f} - \sum_{t=1}^{L^f} B^f_{i_t^f j_t^f} \\ &\quad - \sum_{l=1}^{L^s} \sum_{k=lr-r+1}^{lr} C_{i_k^f i_l^s} - \Pi^s_{i_1^s} - \sum_{t=1}^{L^s-1} A^s_{i_t^s i_{t+1}^s} - \sum_{t=1}^{L^s} B^s_{i_t^s j_t^s} \end{aligned} \tag{12.61}$$

where L^f is the length of the fast chain, L^s is the length of the slow chain and r is the ratio between the time scales, $r = L^f/L^s$ (must be integer). As for the previously described Boltzmann network structures the goal when training Boltzmann zippers is to make the distribution of the visible units,

$$P^-_{\alpha^f \alpha^s} = \frac{1}{Z} \sum_{\beta^f \beta^s} e^{-E_{\alpha^f \beta^f \alpha^s \beta^s}} = \frac{Z_{\alpha^f \alpha^s}}{Z}\,, \tag{12.62}$$

here working from temperature rescaled weights, resemble the distribution specified by the environment by minimization of the entropic cost function Eq. (12.37). The first derivatives for the entropic cost function wrt. the parameters $(\mathbf{\Pi}^f, \mathbf{A}^f, \mathbf{B}^f)$ and $(\mathbf{\Pi}^s, \mathbf{A}^s, \mathbf{B}^s)$ are computed by formulas similar to Eqs. (12.42–12.44). The derivatives wrt. the cross-connections are similary computed as

$$\frac{\partial H(\mathbf{w})}{\partial C_{ii'}} = \sum_{l=1}^{L^s} \sum_{k=lr-r+1}^{lr} \left[\left\langle \delta_{ii_k^f} \delta_{i'i_l^s} \right\rangle^- - \left\langle \delta_{ii_k^f} \delta_{i'i_l^s} \right\rangle^+ \right] \tag{12.63}$$

This expression states that the gradient is computed by summing the correlations between the states of all the pairs of hidden units that are connected by the cross-connections $\mathbf{C}$. $< \cdot >^+$ denotes summing averages with the visible units of the Boltzmann zipper clamped in the desired training patterns, $< \cdot >^-$ denotes summing averages with the visible units unclamped or free running, as described previously.

12.5.1 Exact learning in Boltzmann zippers

As is the case for Boltzmann chains it is possible to compute the exact values of both the entropic cost function and the gradient for Boltzmann zippers. The approach is similar to the description given in section 12.4.1, but two additional techniques for reduction of the zipper structure are called for: decimation and joining. Below follows a description of the two additional reduction techniques; a graphical illustration of how to iteratively apply clipping, decimation and joining for the reduction of a Boltzmann network can be found in [SJ94].

12.5.1.1 Exact learning: Decimation

Decimation [SJ94, SJ95] is a technique for reducing Boltzmann networks like zippers by summing out, or decimating, degrees of freedom represented by the middle unit in a structure of three units connected in series, as illustrated in Figure 12.12. Though not directly connected, the units γ and ϵ in the left network have an effective interaction that is mediated through the weight matrices $\mathbf{W}^{\gamma\delta}$ and $\mathbf{W}^{\delta\epsilon}$ by the middle unit δ. This interaction can be represented by a set of *effective weights* $\mathbf{W}^{\gamma\epsilon}$ as shown in the right network of Figure 12.12 by summing over the degrees of freedom represented by unit δ. In the figure, "Index" labels the units in the following.

Consider the network to the left in Figure 12.12. The partition function Z derived for this network is

$$Z = \sum_{ijk} e^{w_{ij}^{\gamma\delta} + w_{jk}^{\delta\epsilon}} = \sum_{ik}\sum_{j} e^{w_{ij}^{\gamma\delta} + w_{jk}^{\delta\epsilon}} = \sum_{ik} e^{w_{ik}^{\gamma\epsilon}} \tag{12.64}$$

where we see that the weights $w_{ik}^{\gamma\epsilon}$ resulting from summing over the degrees of freedom for unit δ are calculated as

$$e^{w_{ik}^{\gamma\epsilon}} = \sum_{j} e^{w_{ij}^{\gamma\delta} + w_{jk}^{\delta\epsilon}} \iff w_{ik}^{\gamma\epsilon} = \ln\left(\sum_{j} e^{w_{ij}^{\gamma\delta} + w_{jk}^{\delta\epsilon}}\right) . \tag{12.65}$$

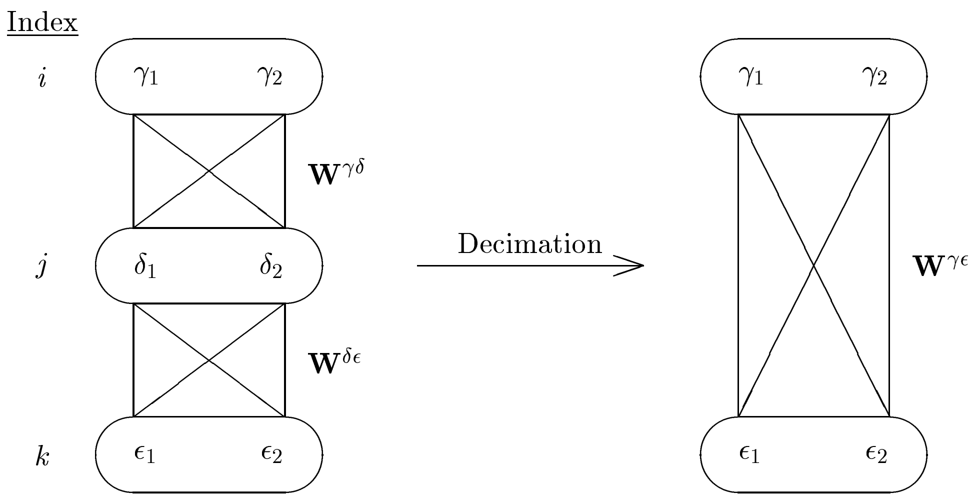


Figure 12.12: Decimation. Unit δ is decimated away and the units γ and ϵ are connected by the resulting weights $\mathbf{W}^{\gamma\epsilon}$.

As an example, consider the weight $w_{12}^{\gamma\epsilon}$. This weight is obtained by summing over all the possible weight "paths" from state γ_1 to state ϵ_2:

$$\begin{aligned} e^{w_{12}^{\gamma\epsilon}} &= e^{w_{11}^{\gamma\delta}+w_{12}^{\delta\epsilon}} + e^{w_{12}^{\gamma\delta}+w_{22}^{\delta\epsilon}} \\ \Updownarrow & \\ w_{12}^{\gamma\epsilon} &= \ln\left(e^{w_{11}^{\gamma\delta}+w_{12}^{\delta\epsilon}} + e^{w_{12}^{\gamma\delta}+w_{22}^{\delta\epsilon}}\right) \end{aligned} \tag{12.66}$$

From the above example we see that using the effective weights $\mathbf{W}^{\gamma\epsilon}$ when calculating correlations and partition functions, possibly for a larger structure in which the substructure in Figure 12.12 is embedded, will not change the numerical value of these quantities.

12.5.1.2 Exact learning: Joining

Joining refers to the situation where we have two sets of parallel weights connecting two units, as shown in the left network of Figure 12.13. When calculating correlations or partition functions it is easily seen that the two sets of parallel weights have the same effect as a single set of weights equal to the sum of the parallel sets,

$$\mathbf{W}^{\gamma\delta} = \mathbf{U}^{\gamma\delta} + \mathbf{V}^{\gamma\delta} \tag{12.67}$$

as shown to the right on Figure 12.13.

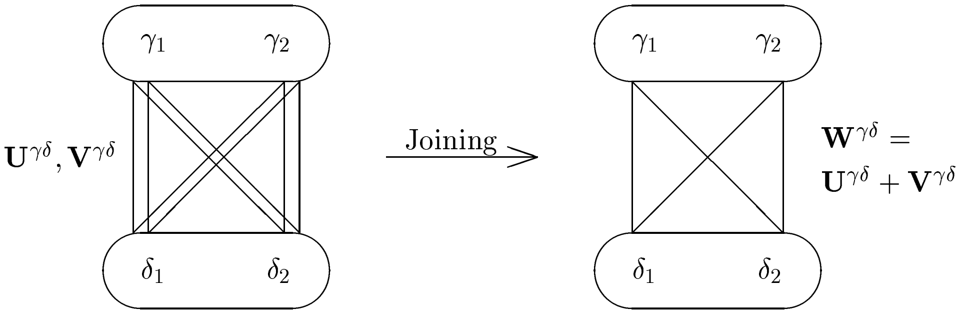


Figure 12.13: Joining. The two sets of weights between units γ and δ have the same effect as the sum of the weights.

12.5.2 Notes on Boltzmann zippers

The Boltzmann zipper can be viewed as an extension to the Boltzmann chain and the notes made in sections 12.4.3 and 12.4.4 regarding training using second-order methods and pruning of chains naturally apply to zippers as well. A note to add on training is the suggestion in [SJ95] to train the *fast* and the *slow* chains separately before combining them in the zipper structure of Figure 12.11 and applying further training to the full structure. This approach might reduce the training time.

Whereas the separate Boltzmann chains in special cases can be converted into corresponding traditional HMMs and vice versa, there is no direct isomorphism between the parameters of the complete Boltzmann zipper structure and a corresponding HMM structure. At any given time step the Boltzmann zipper is always in *two* hidden states which is the prime reason for the lack of conversion possibilities into corresponding HMMs.

However, we may still conceptually apprehend a Boltzmann zipper as two interconnected HMMs.

Through the inter-connection weights $\mathbf{C}$ the hidden unit states of the fast and the slow chain can influence each other. If all the weights $\mathbf{C}$ are identical (e.g., they are all pruned to the value zero), the net influence of the C's will be the same for all possible hidden unit state sequences $\beta^f\beta^s$. When computing likelihoods Eq. (12.62) the terms including the C's will cancel out and in this case the Boltzmann zipper is equivalent to two separate Boltzmann chains having their likelihoods multiplied (late integration). By increasing the numerical value of some C's and decreasing the value of others we can increase the joint likelihoods of certain hidden unit state configurations and decrease the likelihoods of others and thus capture correlations between the observations of the sequences to be modeled (intermediate integration). Thus, when training from C's with the initial value zero we graduately transform the Boltzmann zipper from a late integration model into an intermediate integration model.

Pruning cross-connection weights in a Boltzmann zipper to the value zero will have the effect of loosening the constraints between the hidden unit state sequences of the fast and the slow chain. Pruning cross-connections to the value zero thus biases the complete model towards a late integration strategy. On the other hand one might prune the values of the cross-connections to $-\infty$ as discussed for Boltzmann chains in section 12.4.4. This will put ultimate constraints on the hidden unit state sequences that can occur simultaneously as any joint state sequences involving states combined with a cross-connection $-\infty$ weight will occur with probability zero. This strategy might be advantageous if there is believed to be a highly constrained underlying structure responsible for the correlations between the two sequences to be modeled. Such a constrained structure should be viewed in line with e.g., left-to-right HMMs in which there are restrictions to the order in which hidden unit states can occur.

Finally, a few practical notes concerning the reduction techniques decimation and joining when pruning parameters to $-\infty$. The same note that was made about clipping applies to decimation. If the argument to ln in Eq. (12.65) is zero, the large negative value used as the equivalent to $-\infty$ should be returned, as there is then no valid "path" between the two states connected by the resulting effective weight; refer to Figure 12.12. When joining it is obvious that if one or both of the weights to be joined are equivalent to $-\infty$ the combination should also be $-\infty$ as the energy of a state configuration involving a $-\infty$ connection is infinite; refer to Figure 12.13.

12.6 Experiments using Boltzmann chains

This section contains a description of the experiments performed involving Boltzmann chains. The first class of experiments is based on an artificial problem, namely the identification of a "teacher" Hidden Markov Model (HMM). The second class of experiments centers around the construction of a small isolated-word speech recognition system, including a *speechreading* system incorporating video information.

12.6.1 Identification of an HMM

As described in previous sections it is possible to convert between Boltzmann chains and Hidden Markov Models, especially if the final hidden state is constrained to be a particular end state. In this experiment the relationship between Boltzmann chains and HMMs is

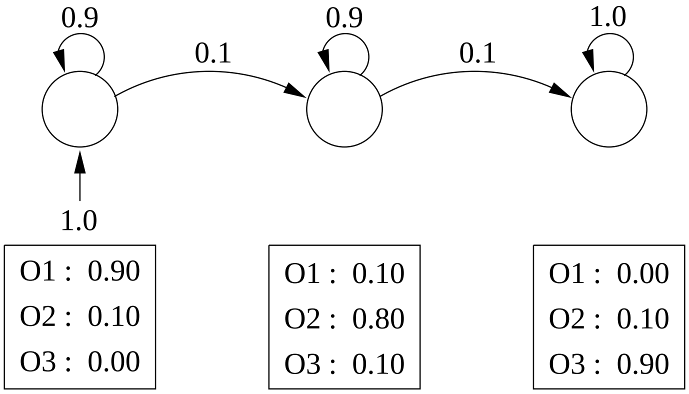


Figure 12.14: Topology of the teacher HMM. The numbers at the arrows indicate transition probabilities and the numbers in the boxes indicate the observation probabilities.

experimentally verified by using a Boltzmann chain to identify the structure of an HMM from observation sequences generated by the HMM.

The underlying "teacher" HMM is illustrated in Figure 12.14. The HMM has a left-to-right structure with three hidden states and three observations in each state. 200 sequences of length 40 were generated from this HMM and all of them terminated in the rightmost hidden state in order to satisfy the criterion for exact conversion between the Boltzmann chain and the HMM. The sequences were divided into a training set and a separate test set, each containing 100 sequences. In Figure 12.15 is shown four examples of the generated sequences.

12.6.1.1 Training

The Boltzmann chain used for identification of the teacher HMM shown in Figure 12.14 had visible units with three states and hidden units with three states corresponding to

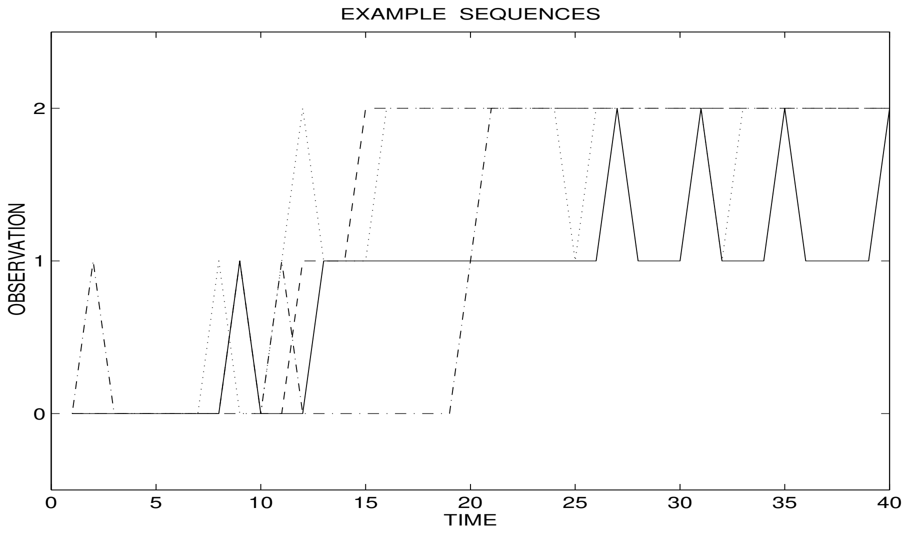


Figure 12.15: Four examples of the generated sequences from the teacher HMM.

the topology of the teacher HMM; the final hidden state was forced to be a particular end state. The chain was trained by minimizing the relative entropy cost function on the training sequences; the constant term in Eq. (12.37) was *ignored* and is not included in any of the illustrations in the following. Minimization was attempted using both gradient descent and the second-order Gauss-Newton method in order to investigate the difference in performance between the two methods. Both methods were combined with a simple linesearch algorithm and in order to ensure numerical stability (refer to section 12.4.3) the entropic cost function was augmented by a small quadratic weight decay, $\lambda = 10^{-3}$ in Eq. (12.60). In both cases the stopping criterion was set to either a maximum number of 5000 iterations reached or the Euclidean norm of the gradient sufficiently small, $\| \cdot \|_2 < 10^{-4}$, indicating *closeness* to a (local) minimum.

In the left panel of Figure 12.16 is shown a typical training curve, i.e., trace of the relative entropy (*excluding* the weight decay term) during training when using gradient descent. Training is performed in "batch mode", i.e., the weights are not updated until all sequences have been presented to the chain. It is seen that convergence is initially fairly fast but then a plateau is reached from which convergence is extremely slow. After 5000 iterations the maximum number of iterations was reached still without satisfying the gradient norm stopping criterion.

In the right panel of Figure 12.16 is shown a typical training curve when using the Gauss-Newton method; the initial 25 iterations were however performed using gradient descent in order to get "close" to a (local) minimum before initiating the second-order method. Convergence is very fast and training stops after about 150 iterations after which the gradient stopping criterion is satisfied. Note that the level of cost obtained is about 20 % lower than the level obtained when using gradient descent and in fact *equal to the entropic cost obtained from the teacher HMM.* The difference in the level of cost reached when using gradient descent resp. Gauss-Newton was quite persistent. In ten runs using different initial weights, only once did gradient descent manage to reach the level shown in the right panel of Figure 12.16; using the very same initial weights, the Gauss-Newton method managed to reach that level in all ten runs.

Figure 12.17 illustrates the evolution of the Euclidean norm (the 2-norm) of the gradient corresponding to the training curves shown in Figure 12.16. Note that the final

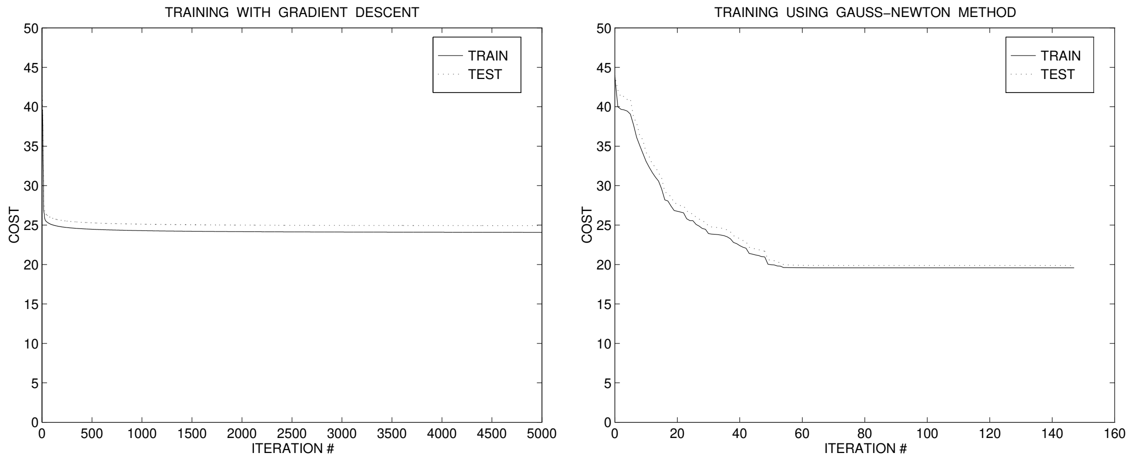


Figure 12.16: Left panel: Training using gradient descent. Right panel: Training using the Gauss-Newton method.

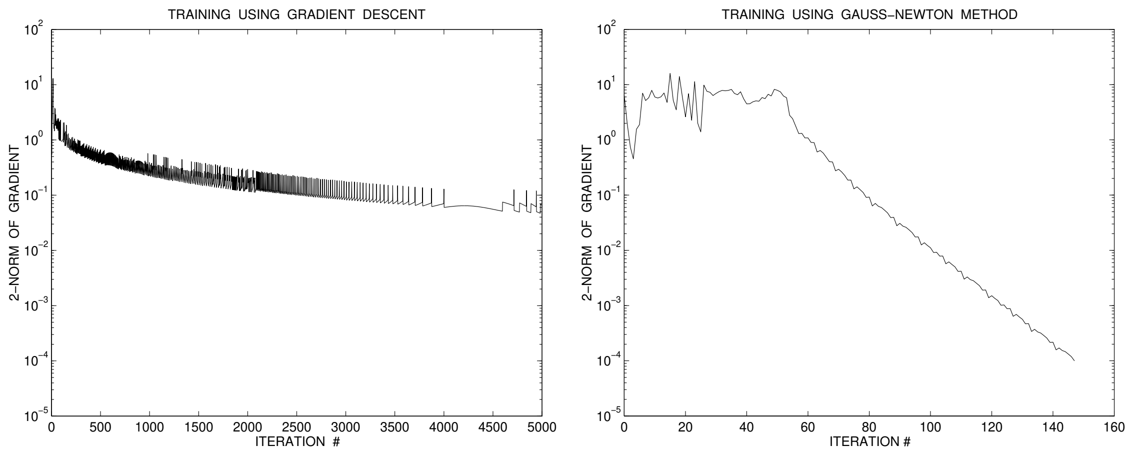


Figure 12.17: Evolution of the gradient norm. Left panel: Training using gradient descent. Right panel: Training using the Gauss-Newton method.

gradient norm using gradient descent, shown in the left panel of Figure 12.17, is three orders of magnitude *larger* than the final gradient norm when using the Gauss-Newton method (right panel). The final chain when using gradient descent thus seems to be further away from a (local) minimum than the chain trained with the Gauss-Newton method even though a factor of 30 times more iterations was used. Given enough iterations the error as well as the gradient norm would probably reach the levels obtained by the Gauss-Newton method when training using gradient descent, but at the cost of much more computation time.

The training session shown in the left panel of Figure 12.16 using gradient descent took 4508 seconds on a 90 MHz Pentium, averaging 0.9 seconds per iteration. The training session using the Gauss-Newton method shown in the right panel of Figure 12.16 took 182 seconds, averaging 1.2 seconds per iteration. When using the Gauss-Newton method it is necessary to solve a linear system of equations in each iteration, scaling as $\mathcal{O}(n^3)$ operations, n being the number of parameters in the model. Even so, the increased computational burden in each iteration is highly justified as the convergence rate using the Gauss-Newton method is usually much higher than when training using gradient descent,

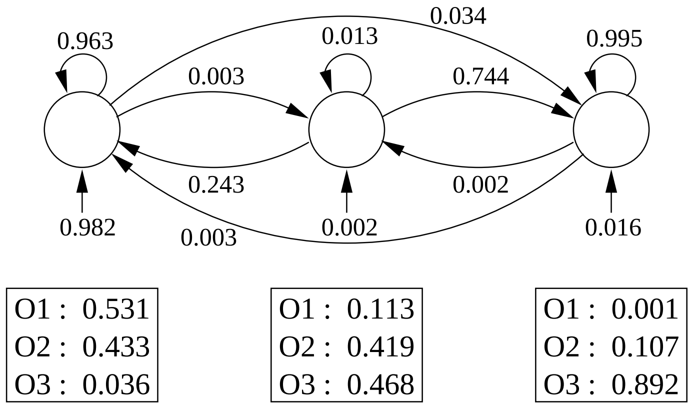


Figure 12.18: Boltzmann chain trained by gradient descent converted into the corresponding HMM.

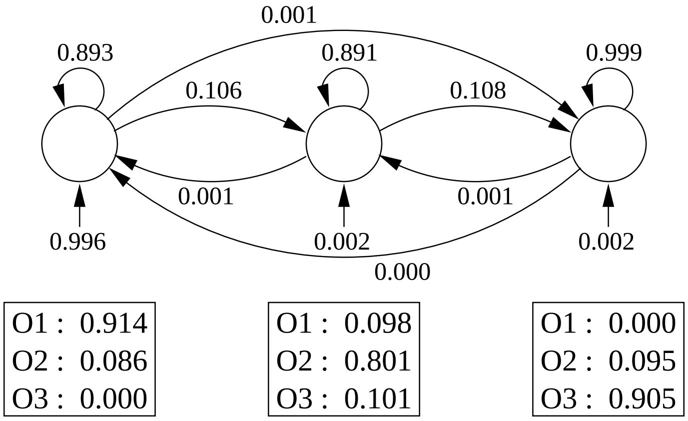


Figure 12.19: Boltzmann chain trained by the Gauss-Newton method converted into the corresponding HMM.

thus reducing the total training time.

It turned out that within the limits set on the training time the choice of training method was crucial for identification of the teacher HMM. In Figure 12.18 is shown the Boltzmann chain resulting from the training using gradient descent converted into the corresponding HMM using Mackays recipe. It is seen that only the prior for the initial hidden state and the end state transition/observation probabilities bears resemblance to the parameters of the teacher HMM.

In Figure 12.19 is shown the Boltzmann chain resulting from training using the Gauss-Newton method converted into the corresponding HMM. It is seen that the transition and observation probabilities for this model are all very close to the corresponding parameters in the teacher HMM. This was the case for the chains resulting from all ten runs using the Gauss-Newton method; only one chain trained by gradient descent came as close, the one that reached the same level of error as the chains trained by the Gauss-Newton method.

Of great importance to the distribution implemented by the trained Boltzmann chain is naturally the relative magnitude of the parameters in the model; the more uniform the weights within a group $\mathbf{\Pi}$, $\mathbf{A}$ or $\mathbf{B}$, the more uniform the distribution. In Figure 12.20 is shown the evolution of the parameters when training using gradient descent. The weigths are seen to grow a lot in the first few iterations after which the weight changes slow down considerably, due to the poor convergence when using gradient descent.

In Figure 12.21 is shown the evolution when training using the Gauss-Newton method. Recall that in the first 25 iterations gradient descent was used; during these iterations the weight changes are fairly "well-behaved". When the Gauss-Newton method is initiated the weights undergo a dramatic change which leads to a larger spread in the magnitudes. During the last 90 or so iterations the weight changes are seen to be very small; the weights are close to a (local) minimum and the Gauss-Newton method is performing a fine tuning in order to satisfy the stopping criterion (refer to right panel of Figure 12.17).

By comparison of Figure 12.20 and Figure 12.21 it can be seen that the spread of weight magnitudes is larger after training with the Gauss-Newton method than after training with gradient descent. The larger spread makes the resulting Boltzmann distribution more focused around the training (and test) sequences from the teacher HMM and leads to the 20 % lower error seen in Figure 12.16.

From the Boltzmann distribution for a Boltzmann chain Eqs. (12.32 and 12.33) we see

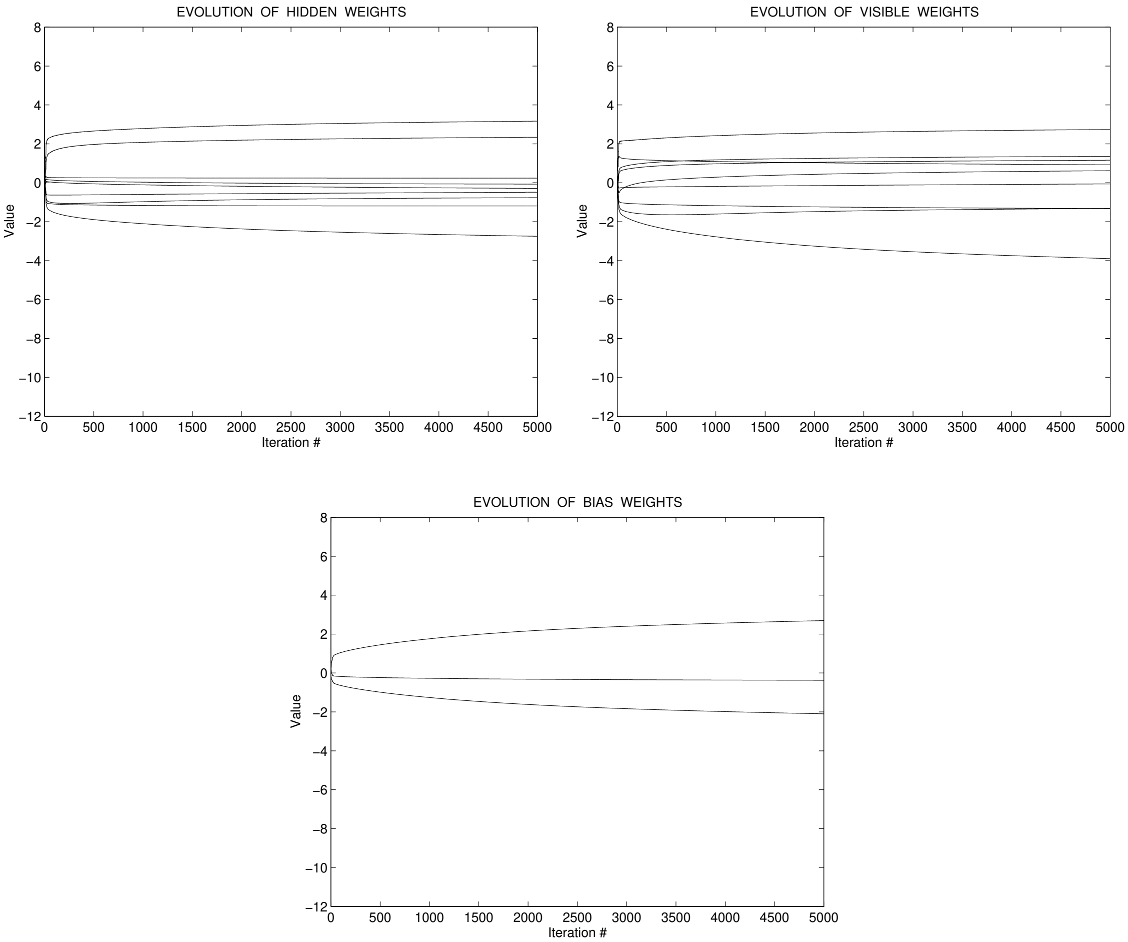


Figure 12.20: Training using gradient descent. Upper left panel: Evolution of hidden-to-hidden unit parameters **A**. Upper right panel: Evolution of hidden-to-visible unit parameters **B**. Lower panel: Evolution of hidden unit initial bias parameters $\mathbf{\Pi}$.

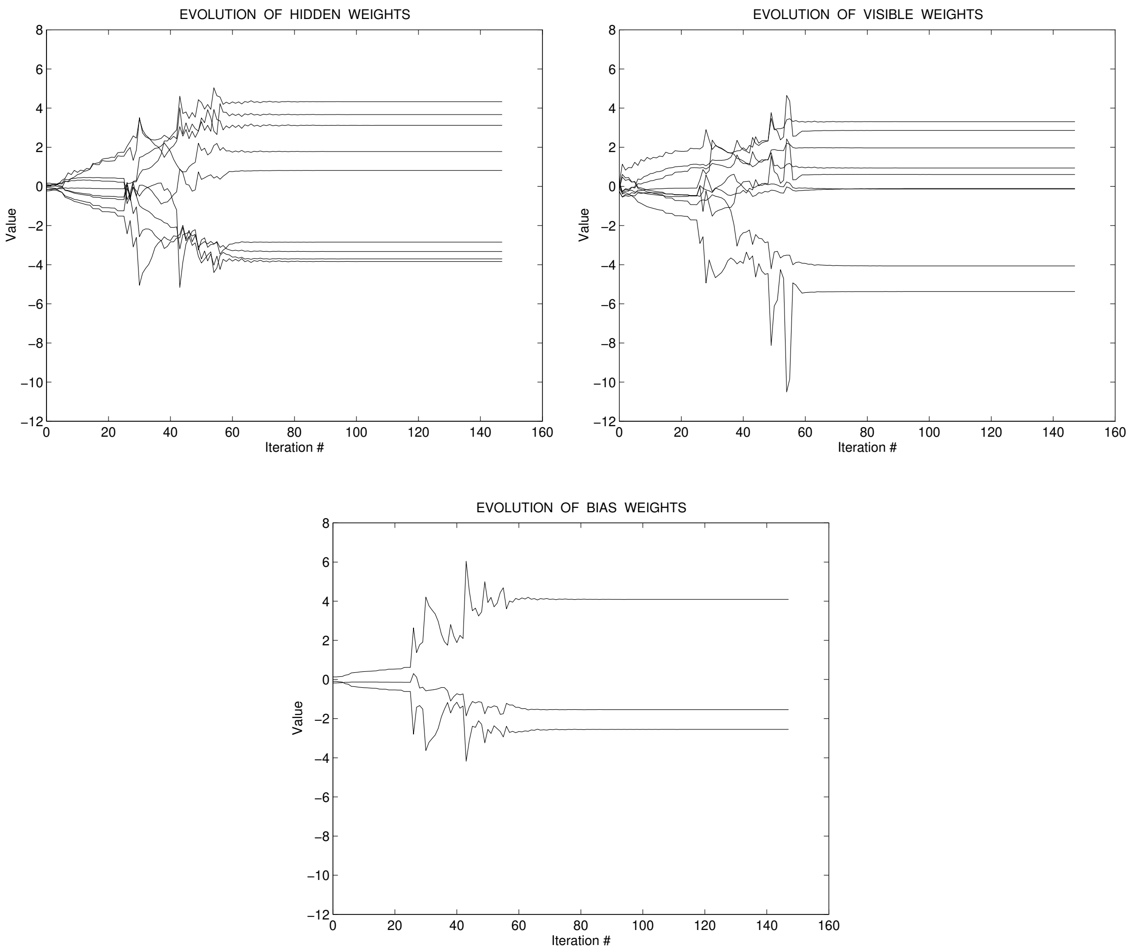


Figure 12.21: Training using the Gauss-Newton method. Upper left panel: Evolution of hidden-to-hidden unit parameters $\mathbf{A}$. Upper right panel: Evolution of hidden-to-visible unit parameters $\mathbf{B}$. Lower panel: Evolution of hidden unit initial bias parameters $\mathbf{\Pi}$.

that sequences involving a weight with a relatively large magnitude will have a relatively large probability of occuring; thus, a weight with a relatively large magnitude may be compared to a large prior, emission or transition probability in an HMM. In fact, the three largest magnitude weights in the upper left panel of Figure 12.21 corresponds to the three self-transitions in the corresponding HMM shown in Figure 12.19. The following two weights when ranking according to magnitude correspond to the left-to-right transitions of Figure 12.19 and the last weights having significantly lower magnitude correspond to the superfluous transitions in the HMM when comparing to the teacher HMM. In the upper right panel of Figure 12.21 the two lowest magnitude weights correspond to the observation probabilities equivalent to zero, and the largest magnitude weight in the lower panel of Figure 12.21 corresponds to the prior probability of the leftmost state in the HMM shown in Figure 12.19.

In Figure 12.22 is shown the evolution of the numerical value of the unclamed partition function Z for the Boltzmann chain. The partition function is seen to grow to extremely large values, 10^{90}–10^{115} in this case. Such levels (and higher) are very common when working with Boltzmann chains. The explanation for the large values lies in the terms

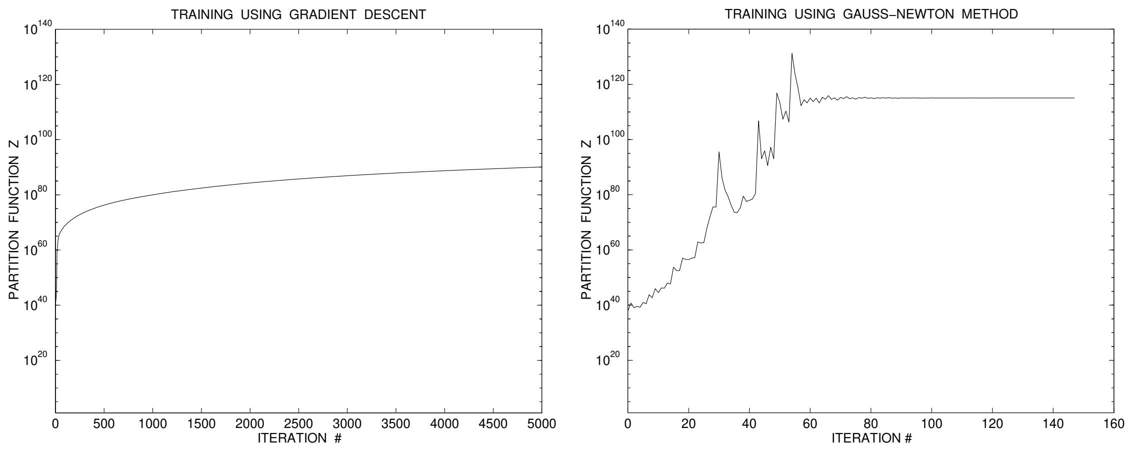


Figure 12.22: Evolution of the unclamped partition function Z. Left panel: Training using gradient descent. Right panel: Training using the Gauss-Newton method.

which enters the partition function Eq. (12.14), namely the sum of exponentials of minus the energy of all possible state configurations of the chain. If some weights are large and positive many of the exponents will be large and positive which leads to the explosion in the numerical value of the partition function. In order to avoid numerical problems that will otherwise occur when computing e.g., likelihoods Eq. (12.15) we need to *scale* the parameters of the Boltzmann chain before computation in order to reduce the partition function to a more numerically manageable level. Scaling due to numerical problems is routinely applied when working with HMMs, see e.g., [Rab89], and similar techniques can be applied to Boltzmann chains by subtraction of appropriate quantities from the parameter matrices $\mathbf{\Pi}$, $\mathbf{A}$ and $\mathbf{B}$.

12.6.1.2 Pruning to $-\infty$

In order to examine the possibilities for architecture optimization, Boltzmann chains having too many hidden unit states compared to the teacher HMM were trained using the Gauss-Newton method. Parameters were then pruned using the OBD pruning scheme outlined in section 12.3.2; after a parameter was pruned the reduced model was retrained. In section 12.4.4 it was discussed how we have to choose between setting pruned parameters to the value zero or setting them equivalent to $-\infty$. For the present problem of identifying an HMM it seemed most appropriate to impose more constraints on the model through pruning, i.e., letting pruned parameters correspond to a zero probability in the corresponding HMM. Parameters were thus pruned to a value equivalent with $-\infty$; here the value -10^{50} was used. As a consequence of this choice the parameters $\mathbf{B}$ connecting the hidden unit and visible unit states were not pruned, due to the problem with specific adaption to the training sequences sometimes resulting in test sequences with zero probability, leading to an infinite test error as described in section 12.4.4. Furthermore, pruning to such a value naturally means that we cannot estimate the saliencies according to Eq. (12.24) but have to set the parameters to -10^{50} in turn and calculate the resulting changes in cost.

In figure 12.23 is shown the evolution of the relative entropy for the training and test set as the parameters in the chain were pruned; note that the initial level of the cost is comparable to the level obtained in the right panel of Figure 12.16. We note that the

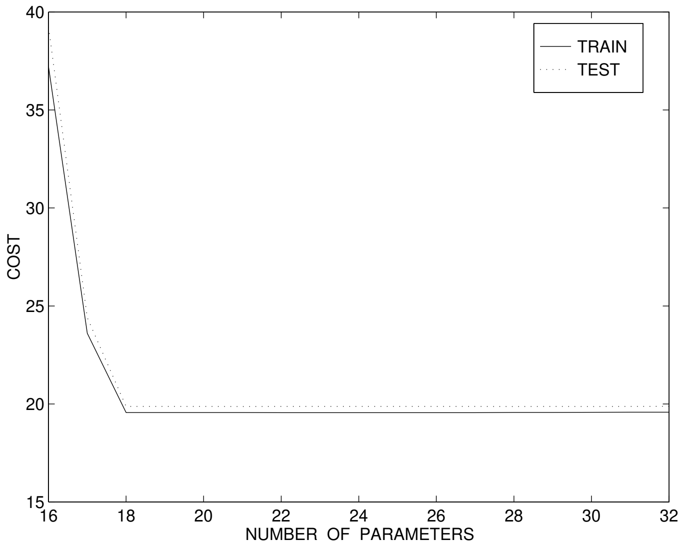


Figure 12.23: Evolution of the relative entropy for the training and test set as the parameters of a chain initially having four hidden unit states are pruned to $-\infty$.

error remains unchanged even though many parameters are pruned. The explanation for this is that only parameters superfluous to the implementation of the teacher HMM are pruned and the retraining thus brings performance back to the same low level as before the pruning.

In Figure 12.24 is shown the "optimal" Boltzmann chain having 18 parameters converted into the corresponding HMM using Mackays recipe described in section F. Optimal is here defined as the smallest chain still able to account for the data. HMM probabilities of the converted chain with the value zero are omitted in Figure 12.24 and we note that one of the hidden unit states of the chain has been completely pruned away. Furthermore the remaining non-zero probabilites of the converted chain bear close resemblance to the probabilities of the teacher HMM.

A detailed count of the parameters of the optimal HMM shown in Figure 12.24 reveals that the number of actual parameters in the model is 15 rather than the 18 reported from Figure 12.23. The reason for this "bias" is that the emission parameters associated with

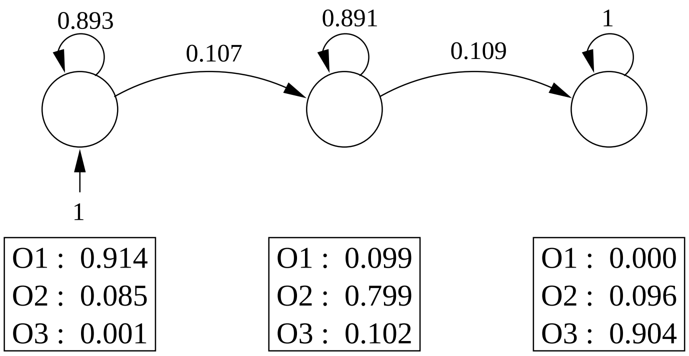


Figure 12.24: "Optimal" Boltzmann chain initially having four hidden unit states converted into corresponding HMM after pruning parameters to $-\infty$.

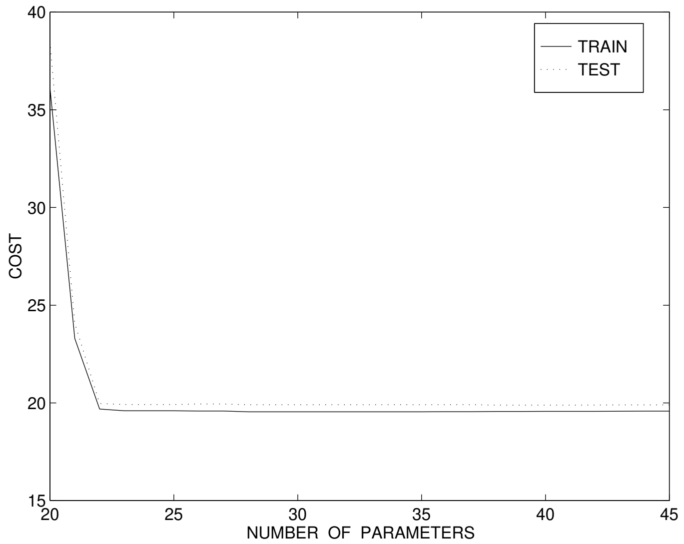


Figure 12.25: Evolution of the relative entropy for the training and test set as the parameters of a chain initially having five hidden unit states are pruned to $-\infty$.

the pruned hidden unit state were determined as still being actual parameters in the model by the parameter count mechanism employed.

It was then attempted to train and prune a Boltzmann chain having *five* hidden unit states and the evolution of the cost when pruning is shown in Figure 12.25. Again we note that the initial level of the cost is the same as obtained in the right panel of Figure 12.16 and that the level of the cost remains unchanged as completely superfluous parameters are being pruned.

In Figure 12.26 is shown the "optimal" chain having 22 parameters converted into the corresponding HMM. We note that once more a single hidden unit state has been discarded. The resulting HMM thus has *four* hidden states compared to three for the teacher HMM. The leftmost and rightmost states of the HMM in Figure 12.26 are fairly good models of the leftmost and rightmost states of the teacher HMM (Figure 12.14) but the middle state of the teacher HMM is modeled by a construction involving *two* hidden unit states. In this

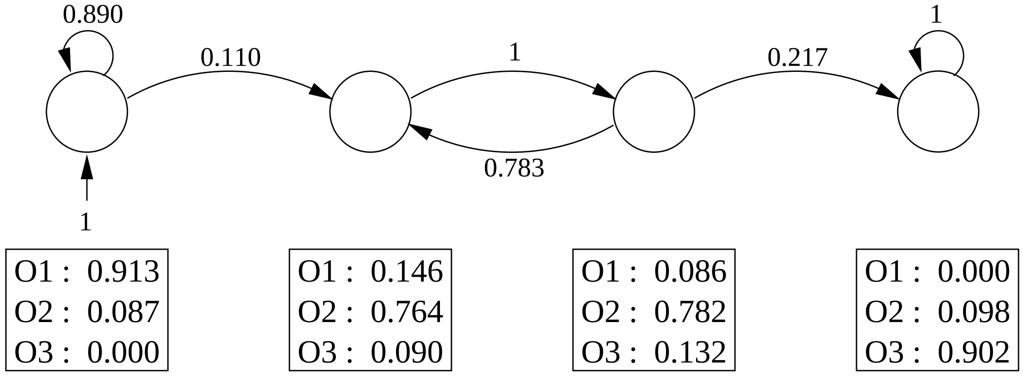


Figure 12.26: "Optimal" Boltzmann chain initially having five hidden unit states converted into corresponding HMM after pruning parameters to $-\infty$.

case it is not straightforward to assess whether the Boltzmann chain actually implements the teacher HMM or not. One way of assessing statistical equivalence between to models is to compare the log-likelihoods obtained on sequences generated by one of the models, using e.g., the entropic cost function [Rab89]. As the converted Boltzmann chain shown in Figure 12.26 has the same level of cost as was obtained for the teacher HMM it thus seems reasonable to believe that the two models are in fact statistically equivalent.

12.6.1.3 Pruning to the value zero

It was then examined what the effect of pruning parameters to the value zero instead of $-\infty$ would be. A Boltzmann chain with four hidden unit states having the same initial weight values as the chain from Figure 12.23 was pruned by setting parameters to zero; again, the hidden-to-visible unit weights $\mathbf{B}$ were *not* pruned. The evolution of the cost is shown in the left panel of Figure 12.27. Comparing Figure 12.23 and Figure 12.27 the evolution of the costs are very similar. The only difference is in the area where performance starts to degrade. When pruning weights to the value zero the performance is seen to degrade more gracefully than when pruning to $-\infty$. In this case the reason seems fairly obvious, as setting any more transition probabilities to zero in Figure 12.19 (i.e., pruning Boltzmann chain parameters to $-\infty$) will have a great impact on the modeling capabilities. On the other hand, pruning parameters to the value zero does *not* lead to zero transition probabilities for the corresponding HMM parameters as discussed in section 12.4.4; they are still in "the ball game" so to speak. The actual values of the corresponding HMM transition probabilities for the pruned parameters will depend upon the remaining free parameters in the chain. The consequences of pruning an "important" parameter to the value zero can therefore be accounted for to some extent by manipulation of the remaining free parameters in such a way that the pruned parameter approximates its "old" HMM transition probability interpretation; thus, pruning a Boltzmann chain parameter to zero need not be as fatal to the modeling capabilities as pruning to $-\infty$, resulting in a more graceful degradation of performance.

The right panel of Figure 12.27 illustrates the quality of the OBD saliencies. For the

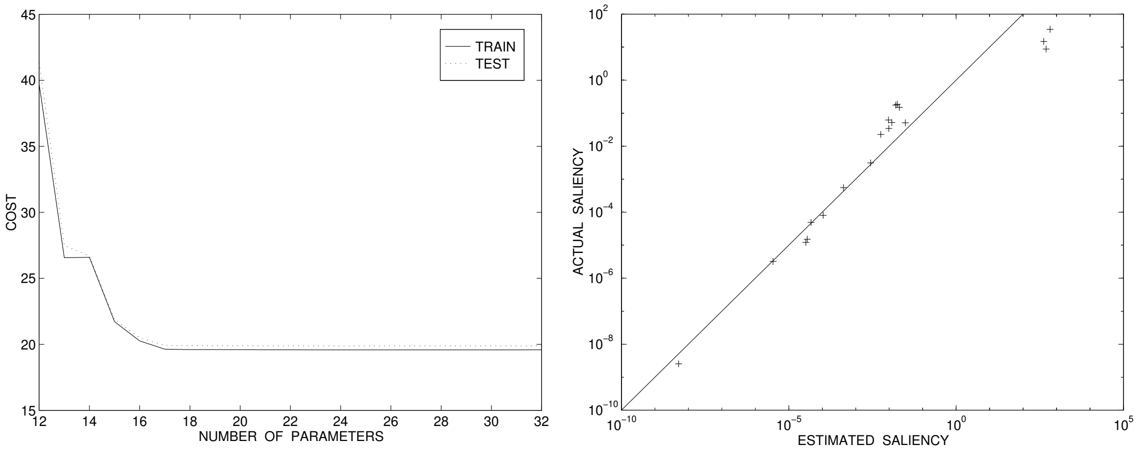


Figure 12.27: Left panel: Evolution of the relative entropy cost function as the parameters of a Boltzmann chain initially having four hidden unit states are pruned to the value zero. Right panel: Estimated versus actual saliencies for the fully connected Boltzmann chain.

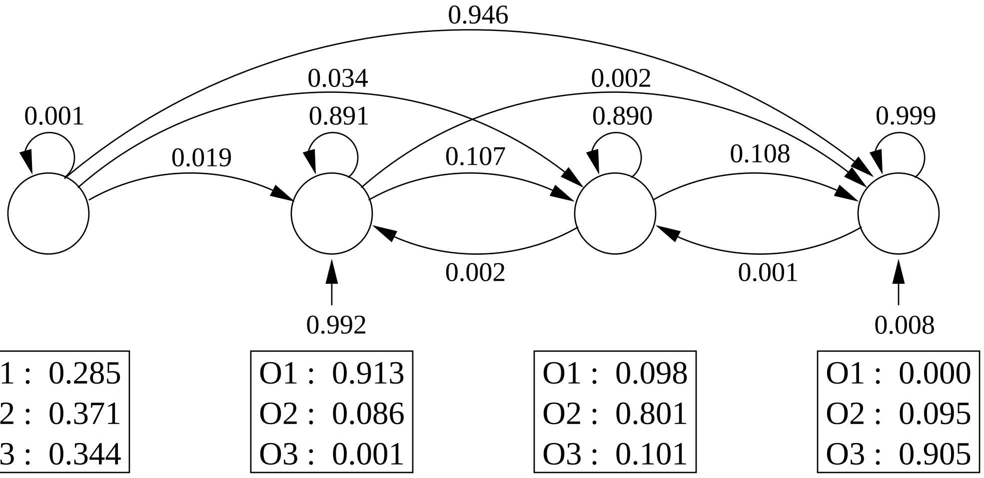


Figure 12.28: "Optimal" Boltzmann chain initially having four hidden unit states converted into corresponding HMM after pruning parameters to the value zero.

fully connected Boltzmann chain saliencies were estimated for parameters $\mathbf{\Pi}$ and $\mathbf{A}$ using Eq. (12.24) and plotted against the actual saliencies computed by setting the parameters to zero in turn and calculating the resulting change in training error. It is seen that the estimated saliencies approximates the actual saliencies very well indeed, especially for the low saliency parameters. It is of importance to note that the rank ordering of the low saliency parameters according to estimated saliency is consistent with rank ordering according to actual saliency; and thus the correct parameters are selected for pruning.

The model from the left panel of Figure 12.27 having 18 free parameters was converted into the corresponding HMM, again using Mackays recipe. The resulting (fully connected) HMM is illustrated in Figure 12.28; however, only transition probabilities of magnitude greater than or equal to 0.001 are shown. This model should be compared to the corresponding converted Boltzmann chain when pruning to $-\infty$, illustrated in Figure 12.24. Looking at Figure 12.28 we note that the leftmost state has a very small probability of being entered and can be regarded as being pruned away. The remaining rightmost three states are seen to have transition probabilities close to the corresponding transitions in Figure 12.24; consequently, the superfluous transitions are very small. In this experiment we thus end up with very similar models even if using two different concepts of pruning.

12.6.2 Speech recognition

This section describes the results obtained when trying to create a small speech recognition system using Boltzmann chains. The system considered here is based on isolated word recognition meaning that a model is trained specifically on each of the words to be recognized. When performing recognition on a new utterance it is presented to each of the models within the recognition system and the utterance is recognized as the word associated with the model having the maximum likelihood of the new utterance. For an introduction to speech recognition systems see e.g., [Rab89].

In addition to do recognition using audio information only it will here be attemted to incorporate visual information from the talker into the model as well, in which case the recognizer is denoted a *speechreading* system [HSP96]. The visual information might include features describing the shape of the lips, positioning of the tounge, chin and jaw etc. Video information is incorporated into the system in the hope that it will improve the recognition rate on e.g., similar sounding utterances like `/me` and `/knee`. Incorporating video information is however especially advantageous in *noisy* environments where it may lead to a dramatic improvement in recognition rates since audio-only systems are likely to perform poorly in noisy conditions. An informal introduciton to the concept of speechreading can be found in [Sto97] and an in-dept description of the video processing system that was used here to extract video information is found in [HSP96, Hen96].

For the recognition system considered here the task was to properly classify the three nonsense utterances `/asklee/`, `/asklaa/` and `/askluu/`. Twenty repetitions of each utterance were recorded (including video information) from a single talker in a low-noise environment. Each of the three sets of utterances was divided into a set of 15 utterances for training and five utterances for testing. The audio part of each utterance consisted of a sequence of 48 feature vectors, each feature vector consisting of 12 log mel power spectrum coefficients [Hen96]. The feature vectors were converted into discrete observations using the LBG algorithm [Gra84]. A codebook for vector quantization having eight entries was generated for each of the three sets of 15 training utterances. Then, both the training and test sequences for each utterance `/asklee/`, `/asklaa/` and `/askluu/` were quantized using appropriate codebooks. The small number of codebook entries used here was the result of a tradeoff. Generally, the more entries there are in the codebook, the better the representation of the original data (the smaller the distortion). A large number of codebook entries however leads to a large number of observations and thus parameters that need to be estimated from the training data. Thus, the larger the codebook, the more data is needed for training. Since the number of utterances available for this experiment was extremely limited, a number of eight codebook vectors was found to be appropriate.

The video information for each utterance consisted of a sequence of 24 feature vectors; thus the video information was obtained at half the speed of the audio information. Each vector was a collection of ten visual parameters representing height and width of the mouth opening, thickness of the upper lip etc.; refer to [Hen96] for details. As for the audio information, three codebooks were generated using the video information of the three sets of 15 training utterances each, again using the LBG algorithm. The number of codebook entries were set to four; both training and test sequences were quantized using appropriate codebooks.

The first recognition system considered was based on audio information only. For each of the three utterances `/asklee/`, `/asklaa/` and `/askluu/` a Boltzmann chain having eight hidden unit states was trained on the corresponding training sequences using the second-order damped Gauss-Newton method described in section 12.3.1. As in the previous example, the final hidden unit state was constrained to be a particular one in order to allow for conversion into HMMs using Mackays recipe. Then, pruning was performed on each Boltzmann chain using the OBD pruning scheme. The weights were pruned to $-\infty$ in order to put constraints into the model and this way allow for comparison between the resulting chains and the left-to-right structure which has been found empirically to be the optimal structure of an HMM for speech recognition [Rab89]. As before, the value -10^{50} was used as a "substitute" for $-\infty$ and the saliencies were calculated by actually setting the parameter in question to -10^{50} and calculate the resulting change in cost. The weights $\mathbf{B}$ were once again left untouched due to the complications described in section 12.4.4 that were otherwise likely to arise.

In Figure 12.29 is shown the evolution of the entropic cost (excluding the constant

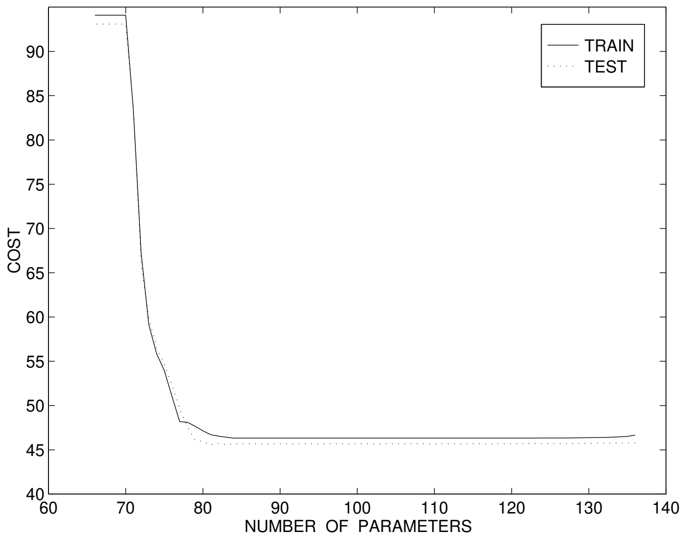


Figure 12.29: Evolution of the relative entropy cost function for the model trained on the `/asklee/` data as the parameters of a Boltzmann chain initially having eight hidden unit states are pruned to the value $-\infty$.

term of Eq. (12.37)) for the model trained on the `/asklee/` data as the parameters were pruned away. It is seen that the cost remains unchanged until 80 parameters are left in the model, below which the cost starts to increase dramatically.

The "optimal" model, chosen as the smallest model still able to account for the data, had 80 parameters. The parameters leading to and from three of the hidden unit states were completely pruned away and the model was thus reduced to having five hidden unit states. In Figure 12.30 the model is illustrated after conversion to a corresponding HMM using Mackays recipe; for convenience, only the transition probabilities are illustrated. The HMM is seen to show slight signs towards the implementation of a left-to-right model. The rightmost state in Figure 12.30 is apparently very likely to model both the beginning as well as the end (which it was forced to) of an utterance which corresponds to silence.

The evolution of the cost for the models trained on the utterances `/asklaa/` and

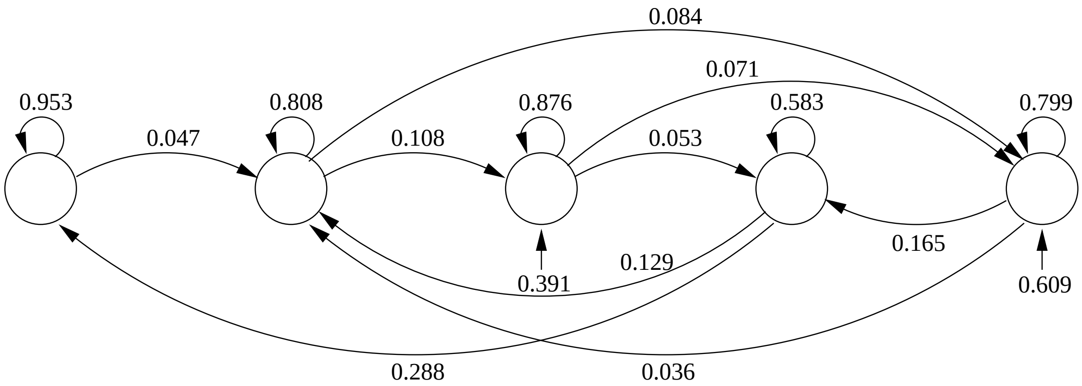


Figure 12.30: Structure of the corresponding HMM for the "optimal" chain trained on the `/asklee/` data when pruning parameters to $-\infty$.

`/askluu/` was similar to Figure 12.29. The number of hidden unit states for the chosen models (just before the dramatic increase in training error) was six for the `/asklaa/` model and seven for the `/askluu/` model, even though the total number of parameters was comparable to the optimal `/asklee/` model. The three chosen models were then combined into a recognizer and each presented with all the test utterances (all quantized using the codebook associated with the model to which they were presented). Likelihoods were computed and each test utterance was classified as being the utterance associated with the model having maximum likelihood. This way, the recognizer made only one error on the 15 test utterances.

In order to improve the recognition, utilization of video information was incorporated into the model as well. Three Boltzmann chains each having four hidden unit states were trained on the video features alone for the utterances `/asklee/`, `/asklaa/` and `/askluu/` and pruned in the same way as the audio models. The models were slightly reduced but no hidden unit states were completely pruned away. Using the resulting video models a recognizer was created which made seven errors on the 15 test utterances; recognition based on video information only is a difficult task.

The audio and video models were then combined using a late integration strategy. The log-likelihoods of the acoustic models were multiplied with a constant λ and the log-likelihoods of the video models were multiplied with $(1-\lambda)$. After some experiments $\lambda = 0.3$ was found to be optimal and the complete speechreading system made zero errors on the 15 test utterances.

12.6.2.1 Pruning weights to the value zero

As for the experiment identifying a teacher HMM it was also attempted to prune the weights in the Boltzmann chains used for modeling speech to the value zero. The left panel of Figure 12.31 shows the evolution of the entropic cost for the model trained on the `/asklee/` data as the parameters were pruned to zero. The initial weights for the model were the same as the initial weights for the model used in Figure 12.29. The evolution of the errors in Figure 12.29 and in the left panel of Figure 12.31 is seen to be very similar. As in the previous example, the only differences occur in the area where performance starts to degrade. When pruning weights to the value zero the performance is again seen to degrade more gracefully than when pruning to $-\infty$. This is due to the fact that the restrictions imposed on the model are less radical when setting parameters to zero than when setting them equivalent to $-\infty$. As long as there are plenty of parameters in the model the effects of either pruning approach are easily handled by the remaining parameters after retraining. When model resources get scarce though, the less restricted model of Figure 12.31 is still capable of indirectly compensating for the lost degrees of freedom as discussed in section 12.6.1.3.

Once more the estimated saliencies were compared to the actual saliencies obtained by setting the parameters to zero in turn and computing the change in cost. The right panel of Figure 12.31 illustrates the quality of the estimated saliencies by plotting them against the actual saliencies for parameters $\mathbf{\Pi}$ and $\mathbf{A}$ of the fully connected Boltzmann chain. It is seen that the estimated saliencies approximates the actual saliencies very well indeed, and as before especially for the low saliency parameters. Again the rank ordering of the low saliency parameters according to estimated saliency is consistent with rank ordering according to actual saliency leading to pruning of the correct parameters.

The model from the left panel of Figure 12.31 having 80 free parameters was converted

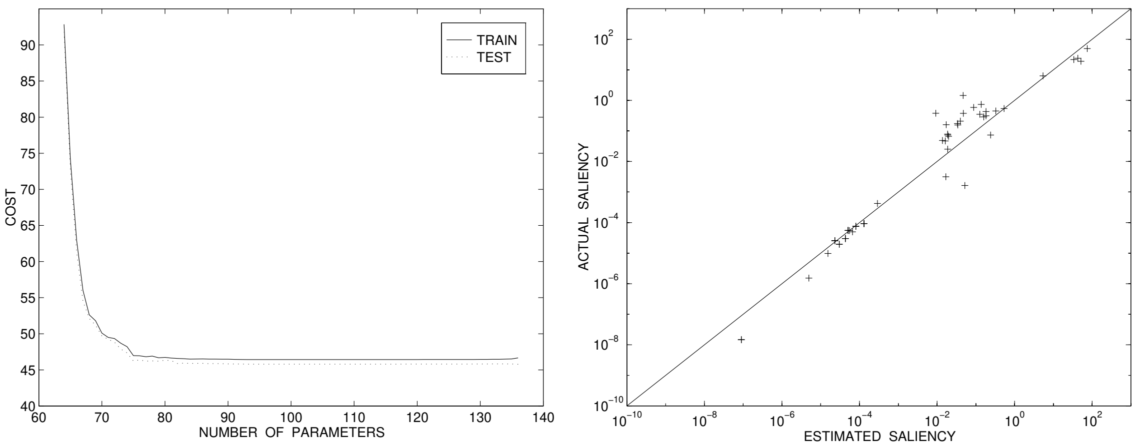


Figure 12.31: Left panel: Evolution of the relative entropy cost function for the model trained on the `/asklee/` data as the parameters of a Boltzmann chain initially having eight hidden unit states are pruned to the value zero. Right panel: Estimated versus actual saliencies for the fully connected Boltzmann chain.

into an HMM using Mackays recipe, for comparison with the resulting HMM when pruning to $-\infty$. The structure is illustrated in Figure 12.32, again omitting the emission probabilities for convenience of presentation. Since the weights were pruned to the value zero the corresponding HMM had non-zero transition probabilites only and thus no hidden states were pruned away. However, it turned out that the transition probabilities leading to and from three of the hidden states were extremely small and could thus be regarded as being pruned away. In Figure 12.32, only transition probabilities of magnitude 0.001 or greater are included. The resulting HMM is seen to show great similarity to the structure of the HMM in Figure 12.30; in fact, several transition probabilities are close to being identical in the two figures.

It was then investigated which weights that actually get pruned away. One might suspect that the weight to be pruned in each turn is simply the weight in the Boltzmann chain with the lowest value or, alternatively, the weight for which the corresponding transition

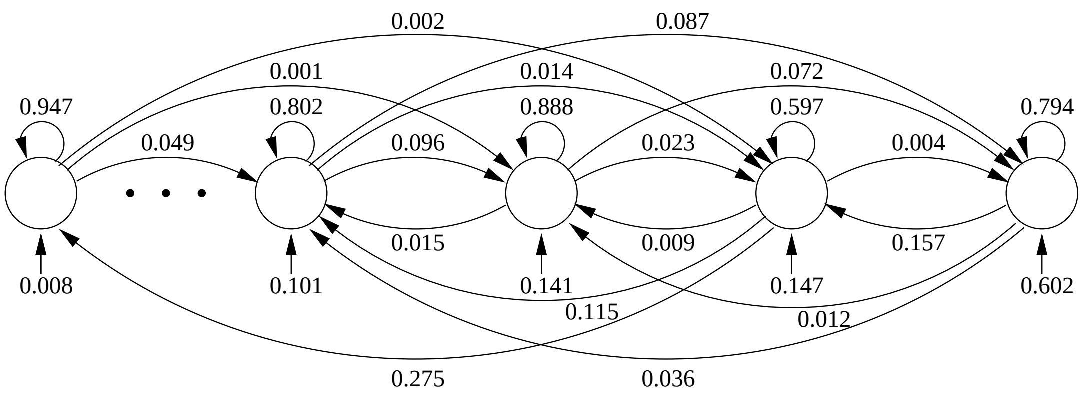


Figure 12.32: Structure of the corresponding HMM for the "optimal" chain trained on the `/asklee/` data when pruning parameters to the value zero.

probability in an equivalent HMM is the smallest. This was investigated for both the example described in section 12.6.1 identifying a teacher HMM as well as for the speech recognition problem described in the present section. The investigation incorporated examples where pruning was performed using both the value zero as well as $-\infty$. In all examples it turned out that there was no consistent pattern as to which weight that got pruned away. This was the case when looking at both weight magnitudes and when converting into HMM transition probabilities. In some cases it was even the largest weight and the largest transition probability that got pruned away; this was however mostly the case when an entire hidden unit state was pruned away. The only "rule" that could be deducted from the investigation was that when pruning parameters to the value zero it was more likely to actually prune the weight closest in magnitude to zero than when pruning to $-\infty$.

12.7 Experiments using Boltzmann zippers

This section contains a description of the experiments performed using Boltzmann zippers; the experiments focused on two problems. The first problem was artificial and involved determination of the nature of the correlation between two simple correlated HMMs. The second problem was once more the construction of a speechreading system based on both audio and video information.

12.7.1 Correlated HMMs

The first problem on which Boltzmann zippers were used involved identification of the correlation between two hidden unit states of two HMMs generating observations on disparate time scales.

Synthetic observation sequences were generated by two left-to-right HMMs. In order to generate observations on two different time scales, a *fast* model was iterated twice as fast as a *slow* model. The fast model had three hidden states and three observation symbols, the slow model had two hidden states and two observation symbols. The rightmost hidden states in the model were "connected" as illustrated in Figure 12.33, increasing the probability of making transitions to the rightmost hidden state in one model if the other model was already in its rightmost state. Thus, the fast HMM was able to make direct transitions from its leftmost to its rightmost state *if* the slow HMM was already in its rightmost state. The correlation/connection was obtained by having two separate sets of

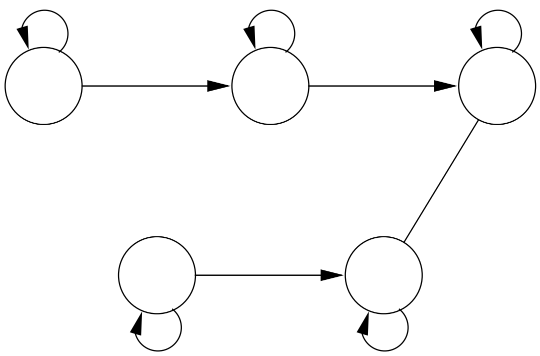


Figure 12.33: Schematic of two "connected" left-to-right HMMs.

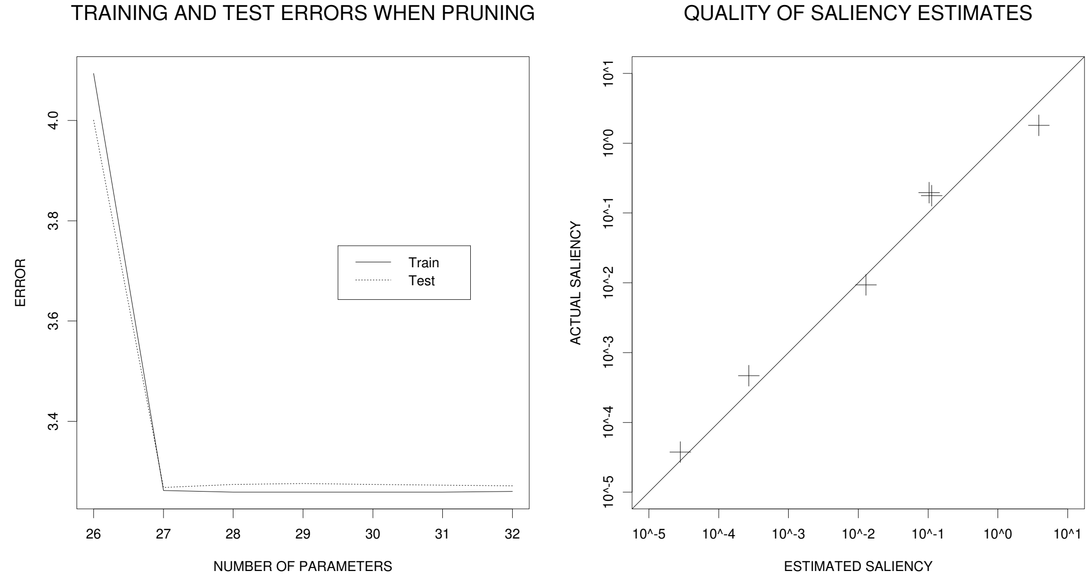


Figure 12.34: Left panel: Cross-entropy error on training and independent test sets (150 patterns each) versus the number of bidirectional weights between units. Pruning proceeds from right to left, and at the extreme left (26 weights), all cross connections have been removed. Right panel: The true saliences in a full network versus the saliencies estimated to second order. Note the excellent agreement over six orders of magnitude.

transition probabilities for each of the HMMs, the "original" left-to-right transitions and an additional set of transitions with direct and increased probability connections to the rightmost hidden state. Whenever either the fast or the slow HMM entered its rightmost hidden state, the other model switched to the additional set of transition probabilities, thus providing the correlation between the two rightmost states.

Three hundred pairwise sequences of lengths 14 and 7 (respectively) were generated, and a fully connected Boltzmann zipper having the same number of hidden and visible unit states as the underlying HMM models was trained, using 150 examples for training and 150 examples as a separate test set. The Boltzmann zipper did not feature the hidden unit bias weights $\mathbf{\Pi}^f$ and $\mathbf{\Pi}^s$ and the initial model thus had 32 parameters. It was trained by gradient descent followed by the damped Gauss-Newton method. In order to ensure numerical stability (see section 12.4.3), the entropic cost function was augmented by a small quadratic weight decay term.

In order to investigate the utilization of the cross-connections $\mathbf{C}$ only these six parameters were pruned using the OBD pruning scheme. It is not clear how the cross-connections relate to the transition probabilities in HMMs. Pruning all cross-connections to zero leads to the equivalent of two independent models having their likelihoods multiplied, whereas pruning them all to $-\infty$ leads to a degenerated model. Since the aim was to investigate whether the cross-connections would have a significant impact on the performance or whether two separate models would do just as well it was chosen to prune the cross-connection weights to the value zero. After each weight elimination the remaining parameters in the zipper were trained using the Gauss-Newton method.

In the left panel of Figure 12.34 is shown the results of pruning. Note that the errors are left almost unchanged until the final cross-connection is pruned, after which the errors increase significantly. This is consistent with the model that generated the data and indicates that the zipper has indeed captured the correlation between the two underlying HMMs well. This is emphasized by the fact that the test error using only one cross-connection is slightly lower than when using more cross-connections.

In 12 pruning experiments such as just described it was found that in all 12 cases the value of the entropic cost was fairly constant until only a single cross-connection weight was left, after which the value of the entropic cost rose dramatically. In all 12 cases the pruned zippers displayed improved generalization ability compared to the unpruned zippers, and of these it was found that 9 displayed best generalization using only a single cross-connection weight, as expected.

In the right panel of Figure 12.34 is illustrated the quality of the saliency estimates. The estimated saliencies for the fully connected Boltzmann zipper are plotted versus actual saliencies computed by setting parameters to zero in turn and calculating the resulting change in training error. Note that the estimates are in excellent agreement to the actual saliencies even over six orders of magnitude.

12.7.2 Speechreading

This section contains the results obtained when using Boltzmann zippers to create a small speechreading system. As described in section 12.6.2, a speechreading system recognizes speech based on both acoustic and visual information.

As described in section 12.5 the Boltzmann zipper allows for intermediate integration of the two observation sequences (possibly on disparate time scales) from which the model is built. It is thus possible for the Boltzmann zipper to "discover" correlations between observations in the two sequences and utilize this correlation to e.g., increase the likelihood of being in hidden unit state a in the fast chain if the connected hidden unit in the slow chain is in state b; refer to Figure 12.11.

Intermediate integration is advantageous when building speechreading systems due to a phenomenon known as *coarticulation*. Coarticulation [Sto97] means the influence of one utterance on the sound of another. Coarticulation is also the term for the shaping/rounding of the lips in a particular way in anticipation of a succeeding sound. As an example, take the utterances `/asklee/` and `/askluu/`. When pronouncing `/asklee/` the lips are pulled back during the `/skl/` sound in anticipation of the `/ee/` sound that follows; when pronouncing `/askluu/` the lips are rounded during the `/skl/` sound in anticipation of the following `/uu/` sound. By intermediate integration of the audio and video information we can think of e.g., the hidden units in the acoustic chain being biased towards the state (or states) modeling the `/uu/` sound of `/askluu/` once the hidden units are more likely to be in the state (or states) modeling lip rounding. Being able to utilize coarticulation this way is naturally most effectful in noisy environments in which the quality of the acoustic information suffers.

The speech recognition problem considered here is the same as described in section 12.6.2, i.e., constructing an isolated word recognizer for the three nonsense utterances `/asklee/`, `/asklaa/` and `/askluu/`. The original aim was to create an intermediate integration speechreading system and see whether it would be able to outperform the corresponding late integration system, thus verifying the theoretical advantages of intermediate integration. However, the late integration system constructed in section 12.6.2 was able to correctly classify all the utterances in the test sets. Therefore a significantly larger and perhaps also more noise influenced data set needs to be available in order to render the advantage of intermediate integration possible.

Even though it was not possible to outperform the late integration system it was still attempted to see if Boltzmann zippers implementing an intermediate integration system could do just as well. The structure chosen for the Boltzmann zippers was highly influenced by the results obtained using Boltzmann chains. Pruning of the chains trained on the acoustic information in section 12.6.2 led to models for the three utterances with a comparable number of parameters. Of these, the smallest number of hidden unit states left in the chain was five, thus the number of hidden unit states for the fast chain receiving audio information (see Figure 12.11) was set to five. The chains trained on the video information in section 12.6.2 initially had four hidden unit states. None of these were pruned away but sufficiently many parameters were pruned away that it was decided to try with only three hidden unit states in the slow chain of the zipper. The motivation for the reduction of the slow chain was mainly to keep the number of parameters in the model as small as possible due to the limited number of training sequences.

The Boltzmann zippers were trained from scratch (thus *not* utilizing the already optimized Boltzmann chains from section 12.6.2) using gradient descent followed by the damped Gauss-Newton method; the entropic cost was augmented by a small weigth decay term. Then the Boltzmann zippers were pruned using the OBD pruning scheme. Parameters were pruned to the value -10^{50} equivalent to $-\infty$. Boltzmann zippers can not be converted into corresponding HMMs (and therefore the final states were not constrained to be particular end states) but it seemed reasonable to prune to $-\infty$ after all since any "path" through the zipper involving a weight set to $-\infty$ will occur with probability zero as was found appropriate for Boltzmann chains/HMMs modeling speech. As was the case when pruning chains the weights connecting the hidden and visible units were not pruned.

In Figure 12.35 is shown the evolution of the entropic cost (again *excluding* the regularization term) for the zipper trained on the `/asklee/` data as the parameters were

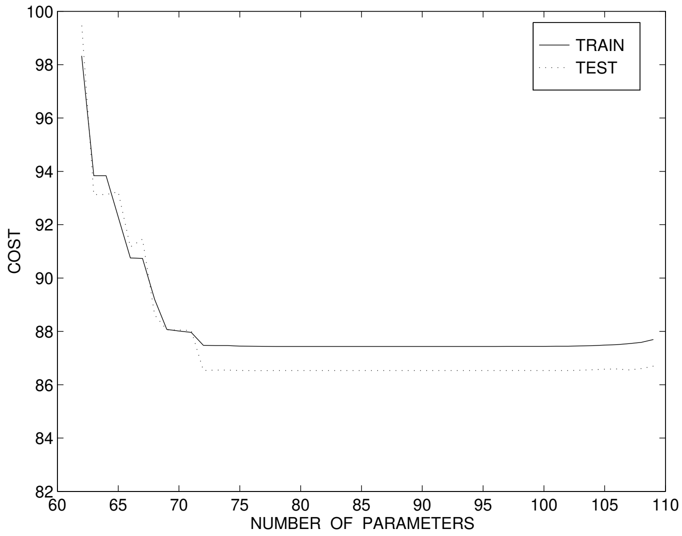


Figure 12.35: Evolution of the relative entropy cost function for the model trained on the `/asklee/` data as the parameters of a Boltzmann zipper initially having five fast and three slow hidden unit states are pruned to $-\infty$.

pruned away. It is seen that the cost also in this case remains almost unchanged until 73 parameters are left in the model, below which the cost starts to increase dramatically. The model having 73 parameters is thus the "optimal" model as it is the smallest model still able to account for the data. Inspection of the resulting model revealed that two hidden unit states in the fast chain as well as one hidden unit state in the slow chain were completely pruned away. The optimal Boltzmann zipper thus had only three hidden unit states for the fast chain and two hidden unit states for the slow chain. Due to the "biased" parameter count mechanism explained previously we should subtract the hidden-to-visible parameters corresponding to the pruned states and we arrive at 53 active parameters for the optimal model. Compared to the late integration system described in section 12.6.2 the optimal Boltzmann zipper thus contains far fewer parameters. E.g., the combined late integration model for the `/asklee/` data contained 90 parameters in total, 56 for the chain modeling the audio information and 34 for the chain modeling the video information.

Similar results were obtained for the Boltzmann zippers optimized for the `/asklaa/` and `/askluu/` data. The three zippers were then combined into a recognition system and presented with the 15 test utterances. As before, the audio and video features of the test utterances were quantized using the codebooks associated with the model to which they were presented. For each utterance the likelihood was computed for each of the models and a test utterance was recognized as the utterance associated with the model having maximum likelihood. This way, the Boltzmann zipper speechreading system made zero errors on the 15 test utterances.

12.7.2.1 Pruning weights to the value zero

As for the previous experiments it was also attempted to prune the Boltzmann zipper parameters to the value zero. The left panel of Figure 12.36 shows the evolution of the entropic cost for the Boltzmann zipper trained on the `/asklee/` data as the parameters were pruned to zero. The initial weights for the zipper were the same as the initial weights for the zipper used in Figure 12.35. We note that both the training and test error actually

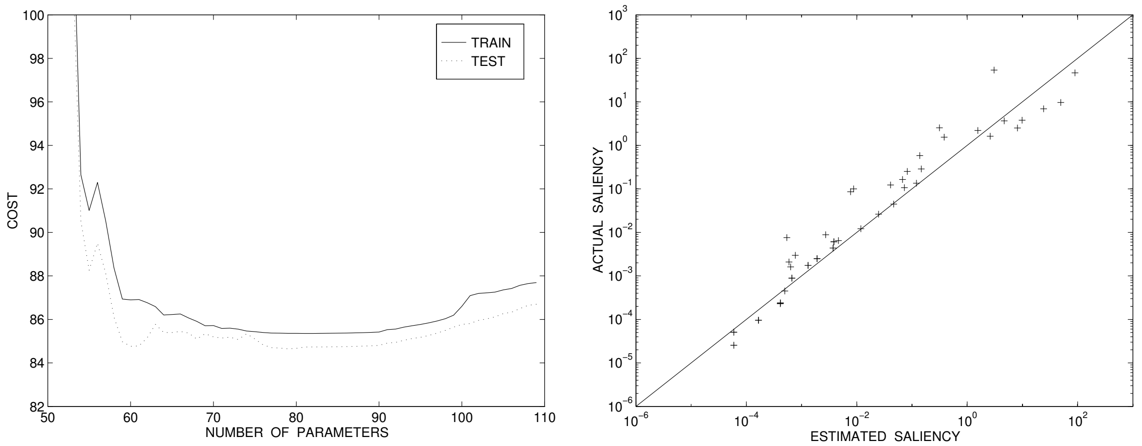


Figure 12.36: Left panel: Evolution of the relative entropy cost function for the model trained on the `/asklee/` data as the parameters of a Boltzmann zipper initially having five fast and three slow hidden unit states are pruned to the value zero. Right panel: Estimated versus actual saliencies for the fully connected Boltzmann zipper.

decreased as parameters were removed. This was caused by the Gauss-Newton algorithm being able to locate lower-lying local minima during retraining of the reduced model.

The interesting part of the curve is around 60 free parameters left in the model. The training error has begun to increase again due to the restrictions imposed by pruning. However, the test error suddenly drops to the lowest level obtained. This is explained by the zipper being sufficiently gracefully restricted and thereby reducing the possibilities of overfitting the training sequences.

In the right panel of Figure 12.36 is illustrated the quality of the estimated saliencies for the fully connected Boltzmann zipper. As previously, the estimated saliencies are plotted versus actual saliencies computed by setting parameters to zero in turn and calculating the resulting change in training error. As seen for the chains there is rather good correspondance between the estimated and actual saliencies, especially for the low-saliency parameters. Once more it is seen that for the low-saliency parameters rank ordering according to estimated saliency is fairly consistent with ranking according to actual saliency, thus ensuring that the correct parameter is chosen for pruning.

12.8 Summary

This tutorial has provided an extensive description of Boltzmann chains and zippers as well as an outline of Hidden Markov Models and Boltzmann networks, the traditional framework for stochastic time series modeling to which chains and zippers should be compared. A positive definite approximation to the second derivatives of the entropic cost function was derived, equivalent to the well-known Gauss-Newton approximation to the second derivatives of the quadratic cost function.

The second derivatives led to the introduction of second-order methods for training of general Boltzmann networks. The damped Gauss-Newton method was applied to both Boltzmann chains and zippers and was found to significantly speed up learning, compared to "batch mode" gradient descent with line search. Central to the successful application of second-order methods is however proper regularization of the Hessian matrix. It was shown how the Hessian matrix for the entropic cost function applied to Boltzmann chains and zippers (and in fact also to *general* Boltzmann networks) is inherently rank-deficient due to the invariance of the Boltzmann distribution to changes in the overall *level* of the parameters. It was suggested to handle this problem by augmenting the entropic cost function by a quadratic weight decay term. The weight decay "drags" the parameters towards a uniform value, zero, and biases the Boltzmann distribution towards a uniform distribution; the weight decay thus has a smoothing effect on the Boltzmann distribution.

Second derivatives for the entropic cost function furthermore opened up for the introduction of algorithmic architecture optimization in the framework of general Boltzmann networks, in particular the pruning schemes Optimal Brain Damage and Optimal Brain Surgeon. It was discussed how we can now adopt *two* strategies when removing degrees of freedom from Boltzmann networks, namely freezing parameters at a fixed *finite* value or setting them equivalent to $-\infty$. Pruning parameters to a fixed finite value biases the Boltzmann distribution towards a smoother, uniform distribution and can thus be considered as an extreme form of weight decay; the value of pruned parameters should naturally match the value to which parameters are "dragged" by the weight decay, zero in this case. On the other hand, setting parameters equivalent to $-\infty$ biases the Boltzmann distribution towards a more sharply "peaked" distribution as certain state configurations will then occur with probability zero. This approach is consistent with the design of HMMs, where certain transition probabilities are set to zero. Both strategies can thus be theoretically justified and the choice must depend on the application in question.

Both pruning strategies were applied to Boltzmann chains and zippers using the OBD pruning scheme. The experiments showed no significant difference between the two approaches as long as resources in the models were plenty, as the models were able to "recover" from the pruning of completely superfluous parameters by retraining of the remaining parameters. The real difference was revealed in the fairly narrow interval where the model undergoes the transition from having excess degrees of freedom to having too few. Pruning to $-\infty$ led to very abrupt changes in performance as the restrictions imposed on the model by this pruning strategy are very extreme, whereas pruning to the finite value zero led to a more graceful degradation of performance on the training data. The graceful degradation is preferable as the overfitting capabilities of the model are then *slowly* reduced, thereby increasing the probability of improving generalization; abrupt changes in performance on the training data are most likely to lead to decreased generalization ability as well. Successful pruning is a fine balancing act between having slightly too many and slightly too few degrees of freedom.

Both Boltzmann chains and Boltzmann zippers were applied to a small real-world speechreading application in order to compare the late integration strategy to the theoretically more appealing intermediate integration strategy. By combining second-order methods for training and architecture optimization by pruning, optimally trained architectures were obtained for both model types. In both cases the resulting speechreading systems were able to correctly classify all utterances in a test set. The good results using both integration strategies unfortunately render a meaningful comparison impossible. The dataset available was too small, a larger and possibly more noise influenced dataset is needed for proper comparison. As the field of speechreading is still somewhat in its infancy, no large standard databases of speechreading data exist yet. Hopefully the near future will bring large speechreading databases, comparable to the TIMIT database for audio-only based speech recognition, enabling reliable comparisons between different model types and integration strategies.

Chapter 13

Conclusion

This thesis has dealt with optimization of recurrent network structures applied to time series modeling. Training has been viewed as a nonlinear unconstrained function optimization problem and architecture optimization has been obtained by pruning. In particular the thesis has focused on fully recurrent networks composed of a single external input, one layer of nonlinear hidden units and a linear output unit. The networks have been applied to time series prediction problems.

The problem of training fully recurrent networks was analyzed from a numerical point of view. It was found that the training problem was rooted in ill-conditioning of the Hessian matrix which leads to slow convergence for many training algorithms. The causes of ill-conditioning was found to be rooted in the modular structure of the networks leading to a very small eigenvalue in the likely case that the outputs from two units become highly correlated. Further, the high connectivity between the hidden units in a fully recurrent network was found to cause a large eigenvalue in the case of large magnitude weights.

The analysis of the training problem also explained why training is usually more difficult for recurrent networks than it is for feed-forward networks, as recurrent networks are more disposed to ill-conditioning. The higher connectivity between recurrent network units was found to lead to many more parameters affected by redundancy in case of correlated hidden unit outputs; it was further found that the magnitude of the largest eigenvalue grows with the number of hidden units whereas it remains unchanged for feed-forward networks.

In order to overcome the problem of ill-conditioning the cost function was augmented by a simple quadratic weight decay term. The simple regularizer was found to be very effective as it bounded both the smallest eigenvalue as well as the largest eigenvalue of the Hessian.

Training of recurrent networks was performed using the damped Gauss-Newton method, involving the *full* Hessian matrix. Once the need for a small regularization term had been recognized this second-order method was found to be much more efficient than gradient descent in terms of both quality of obtained solution and computation time required. The improvement in training was illustrated qualitatively as well as quantitatively for the problem of predicting the laser series.

Learning curves were generated for the laser and Mackey-Glass series in order to assess the performance to expect from the applied recurrent networks. A comparison was made to similar learning curves for *feed-forward* networks and the two model types were found to have comparable performance; the small differences could possibly be eliminated by a fine tuning of the employed weight decays.

It was then examined whether single-input recurrent networks are capable of simulating the dynamics underlying the chaotic laser and Mackey-Glass series. This was found to be possible provided the networks had sufficiently many hidden units.

A theoretical definition of the generalization error for dynamic systems like recurrent networks was formulated. It was found that in order to minimize the bias introduced by transient effects when estimating the generalization error, the test series should follow immediately after the training series on which iterations should be initiated.

In order to improve generalization ability the OBD pruning scheme was applied to recurrent networks selected from the learning curves. It was found that architecture optimization by pruning is indeed a viable approach for fully recurrent networks as the OBD scheme resulted in a significant reduction of the estimated generalization error. As anticipated, the largest reductions were obtained for the networks chosen from "the middle" of the learning curves.

Akaike's Final Prediction Error estimate was tentatively applied as a stopping criterion for the pruning procedure in order to empirically assess the relevance of this analytical generalization error estimate for recurrent networks. It was found that the estimator consistently chose an architecture "close" to the one having the smallest training error which is generally not the architecture with the best generalization ability. Consequently, the FPE-estimate cannot be recommended in the present context of time series prediction using nonlinear recurrent networks. Instead, the choice of optimal network architecture has to be made from the error on a validation set.

The quality of the saliency estimates obtained from both the OBD and OBS pruning schemes was examined. It was found that OBD provides very accurate estimates of the change in error resulting from pruning a weight. However, OBS was found to severely underestimate the actual change in error if pruning a weight, leading to inconsistent ranking and thus pruning of important weights. The underestimation was found to be caused by ill-conditioning of the Hessian; as ill-conditioning is a commonly encountered problem for recurrent networks OBS cannot be recommended for this network type.

A novel operational tool for examination of the internal *memory* of recurrent networks running in open-loop was suggested. The tool allows for assessment of both the *average effective* memory as well as the *time-local effective* memory of a recurrent network. The effective memory should conceptually be compared to the depth of the lag space for a feed-forward network, and thereby represents the time scale of the dynamics in the recurrent network.

The suggested memory measures were applied to selected networks from both the laser and Mackey-Glass series learning curves. The time-local memory measure revealed that the internal memory of a recurrent network is indeed a dynamic quantity; in contrast, the "memory" of a feed-forward network is fixed. The average memory revealed that there can be a large difference between the memories of recurrent networks having a similar estimated generalization error; this corresponds to different implementations of the underlying dynamics.

The average effective memory may also be seen as the average length of the transient when initiating network iterations on a novel segment of the time series. Consequently, this memory measure represents the average number of iterations before predictions from the network are reliable. It is desirable that this number is small, and the memory measure may thus be used to choose between recurrent networks having comparable estimated generalization errors.

This thesis has also dealt with the most general approach towards time series modeling, namely modeling of the joint probability distribution function of observed, discrete valued time series. In particular, two recent recurrent network structures denoted the Boltzmann chain and the Boltzmann zipper, respectively, were considered and a comprehensive tutorial was provided. The derivation of an approximation to the second derivatives of the entropic cost function lead to the successful application of second-order training by the damped Gauss-Newton method as well as pruning by OBD to chains and zippers. It was discussed how *two* strategies can be adopted when pruning these models, namely freezing parameters at a fixed, finite value or setting them equivalent to $-\infty$. Pruning to a finite value (e.g., zero) biases the Boltzmann distribution towards a smooth uniform distribution; on the other hand, pruning to $-\infty$ biases the Boltzmann distribution towards a sharply "peaked" distribution. Experiments on Boltzmann chains and zippers encompassed both artificial problems as well as the construction of a small speechreading system.

Suggestions to future work

The introduction of second-order methods for training and architecture optimization by pruning has made fully recurrent networks a highly applicable model type for time series prediction. Single-input recurrent networks are easier to use than feed-forward networks as it is not necessary to determine an externally provided lag space. However, there is still a choice of significant importance left to be made by the user, namely the choice of a proper weight decay[1]. This choice should ideally be taken care of by an automated, adaptive procedure selecting the weight decay(s) as small as possible from a *numerical* point of view[2] in order to handle the effects of ill-conditioning. It was briefly attempted during this work to device such adaptive procedure based on ad hoc principles. The attempts were however not successful and the problem remains an extremely challenging task of the future.

The novel memory measure introduced in this work needs to be applied to additional problems in order to examine its properties further and thereby hopefully establish its relevance as a tool for characterization and interpretation of recurrent networks. Furthermore a formally justified criterion for selection of the threshold ϵ, indicating when the difference between the estimated generalization errors is negligible, should be developed and tested.

The examination of recurrent network memory may be taken further by attempting to determine the *structure* of the memory, i.e., the relative importance of each of the input values $x(t), x(t-1), \ldots$ to the present network output $y(t)$. Such analysis has been dubbed the "profile" of the recurrent network memory. Several approaches have been attempted towards determination of the memory profile. The attempts have included examination of generalization ability with $x(t-i)$ consistently set to zero when predicting $\widehat{x}(t+1) = y(t)$, setting $x(t-i) = \widehat{x}(t-i)$ when predicting $\widehat{x}(t+1)$ as well as making a first-order approximation to the network output in each iteration, obtained by differentiation

[1]This is the case for feed-forward networks as well.

[2]A proper model complexity is obtained by pruning.

of the network output $y(t)$ wrt. the current and previous inputs. However, none of these approaches have lead to clear, interpretable results. Future work might lead to a more appropriate way in which to determine the memory profile.

Finally, preliminary experiments have indicated that the output of a recurrent network is less sensitive to noise pertubations of the input series than the output of a feed-forward network trained on the same problem to a comparable performance. It would be interesting to investigate this in more detail.

Appendix A

Data set descriptions

This appendix contains a description of the two data sets on which the experiments reported in this thesis are carried out. Characteristic for both series is that they are chaotic in nature, relatively many data points are available and both are wellknown benchmark series in the neural network community. A consequence of the chaotic nature of the systems from which the series are generated is the property of sensitive dependence on initial conditions [PC89]. This means that iterating the system from two different but nearby sets of initial conditions will lead to diverging observation sequences which eventually become uncorrelated, no matter how small the difference in initial conditions. As this is a property of the generating system itself one cannot hope for long term predictions over an arbitrary time step no matter how accurate the model, as even the slightest error possibly due to finite precision arithmetic will eventually scale up to most significant decimal place. However, accurate short term predictions are still feasible provided the time step for the prediction does not extend too far into the future.

Before introducing the series it is appropriate to describe the error measure in terms of which results are reported in this work. The performance of a model on a set $\mathcal{S}$ of data $\{x_t\}, t \in \mathcal{S}$ is evaluated in terms of the *Normalized Mean Squared Error* according to standard practice [WG93] , defined as

$$\mathrm{NMSE}(\mathcal{S}) = \frac{1}{\widehat{\sigma}^2_{\mathcal{S}}} \frac{1}{|\mathcal{S}|} \sum_{t \in \mathcal{S}} (x_t - \widehat{x}_t)^2 \tag{A.1}$$

where $\widehat{\sigma}^2_{\mathcal{S}}$ denotes the sample variance of the values ("targets") in the set $\mathcal{S}$, $|\mathcal{S}|$ is the size of the set and $\widehat{x}_t$ is the prediction from the model to be evaluated. Division by $|\mathcal{S}|$ makes the error measure independent of the size of the set and normalization by $\widehat{\sigma}^2_{\mathcal{S}}$ removes the dependence on the dynamic range of the data. This normalization implies that if the estimated mean of the data is used as predictor, a value of NMSE=1 is obtained. The NMSE error measure has previously been denoted as the Average Relative Variance (ARV) in the literature [WHR90, WHR92].

A.1 The Santa Fe laser series

The Santa Fe laser series was part of the Santa Fe time series prediction competition described in [WG93] where it was the series receiving the most attention. The series is apparently relieving the renowed sunspot data series as the benchmark series of choice in the neural network community, a welcomed renewal.

Originally, 1000 points of the series were released and an object of the time series prediction competition was to provide a prediction of the following 100 points. Even though originally a multi-step ahead prediction task, this problem is gaining significance as a one-step ahead prediction problem as well which is also the problem considered in this work. In Figure A.1 is illustrated the original 1000 point training set followed by the 100 point test set which are separated by a vertical dotted line.

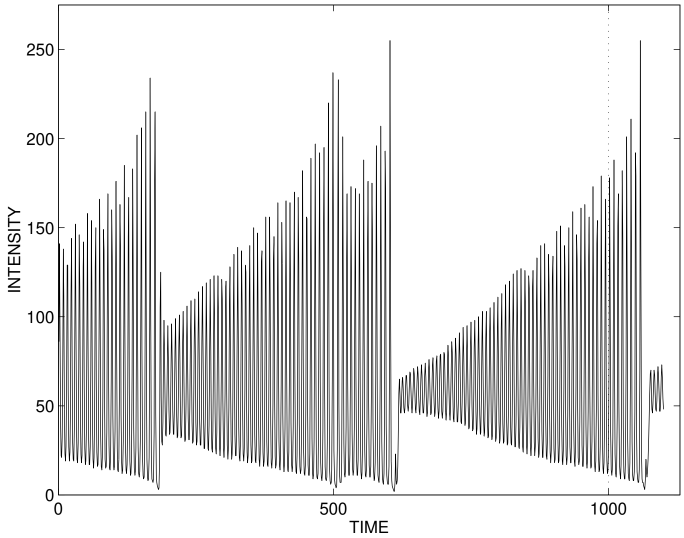


Figure A.1: Originally provided Santa Fe laser series.

The laser series was obtained as the measured intensity in an NH_3 far-infrared (FIR) laser, which exhibit spontaneous periodic and chaotic pulsation. The dynamics of the intensity has been found to be described well by a set of three coupled nonlinear ordinary "Lorenz like" differential equations similar to the so-called Lorenz equations. The measured intensity of the laser was sampled at an interval of 80 ns, using an 8 bit resolution as may be read from Figure A.1. A thorough description of the Lorenz-like chaos in NH_3-FIR lasers as well as a numerical analysis of experimental data can be found in [Hüb93].

After the Santa Fe time series competition had finished a somewhat larger data set was released, comprising the original 1000 point as well as the following 9093 points, 10093 points in total. The full series is illustrated in Figure A.2, however scaled to zero mean and unit variance as is common practice and performed prior to the experiments in this work as well. So far there has been no common agreement in the literature regarding how to divide the extended series into a training set and a test set, as usually only the original shorter series is considered. In the experiments performed on the extended series in this work the first 7000 points of the series have been set aside for training and the following 3093 points have been used as a test set.

A.1.1 Attractor dimension for the laser series

Characteristic for chaotic systems is that as iterations progress, the orbit/trajectory of the internal state vector displays "chaotic" behaviour which from a practical point of view may be defined as a steady-state behavior that is not an equilibrium point, not periodic, and not quasi-periodic as there is no widely accepted strict definition of chaos [PC89]. The limit set of a chaotic system, i.e., the geometrical object in state space to which chaotic

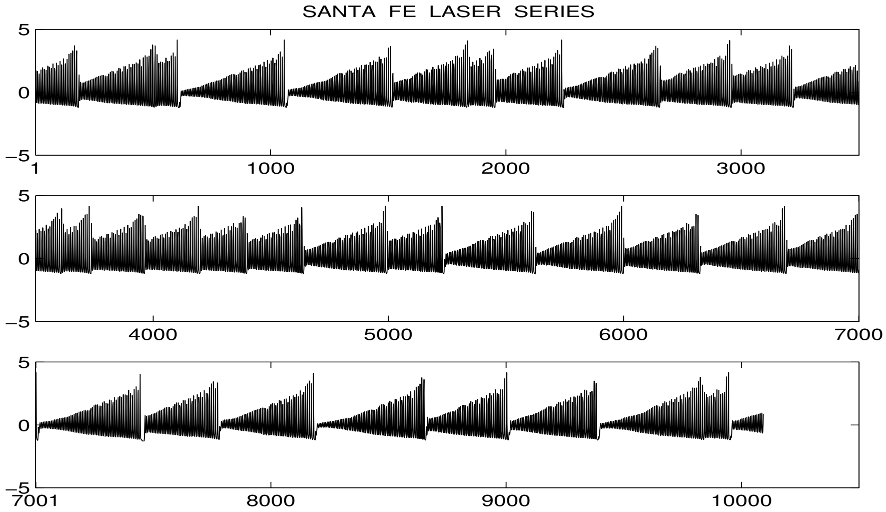


Figure A.2: The complete laser series, scaled to zero mean and unit variance.

orbits/trajectories of the state of the system are attracted is called a *strange attractor*. As of yet there is no generally agreed upon definition of a strange attractor [PC89], as it is not a simple geometrical object like a circle or a torus. A characteristic of a strange attractor is that it is a quite complex geometrical structure which possesses a *fractal* dimension [PC89], i.e., a dimension which is non-integer.

A main effort in the literature attempting to characterize chaotic dynamic systems has been put into the definition and determination of the dimension of the strange attractor. Usually it is not possible to access the state vector internal to the system directly, but a consequence of Takens theorem outlined in section 2.2 is that the orbit which is traced out by a *delay coordinate embedding* of time-delayed measured outputs from the system (a lag space vector of dimension L) preserves geometrical invariants of the system. E.g., the embedding has an attractor with the same dimension as that of the system itself, provided sufficiently many delay coordinates L are used [CEFG91, PS93].

Many definitions on how to determine the fractal dimension of a strange attractor have been suggested in the literature, see e.g., [PC89] for an overview. The dimensions considered here are a set of numbers D_q known as *generalized dimensions* [PS93], obtained by a box counting technique. Consider an embedding in L-dimensional delay space. If the system attractor is covered by L-dimensional boxes having side length l in units of an arbitrary side length l_0, then

$$D_q \equiv \frac{1}{q-1} \limsup_{l \to 0} \frac{\log\left(\sum_i p_i^q\right)}{\log(l/l_0)} \quad , \tag{A.2}$$

where p_i is the *measure* (i.e., the fraction) of the attractor in box i and the sum is only over occupied boxes. In practice, the limit is not calculable with finite data but it is usual to assume that for small, finite l/l_0 the sum will scale as $(l/l_0)^{-D_q}$. For finite data the measure in the ith box is estimated as $p_i = n_i/n$, where n_i is the number of points in the ith box and n is the total number of points. The generalized dimension D_q is then estimated as the slope of a straight-line region on a log-log plot of the sum in Eq. (A.2) vs. the relative box side length l/l_0, obtained for a range of decreasing box side lengths l.

The dimension D_0 is denoted the *capacity* dimension, or simply the box-counting dimension, D_1 is denoted the *information* dimension and D_2 is denoted the *correlation* dimension; while D_q is defined for all real q [PS93] the interest is usually confined to these three dimensions. In this work an extremely fast and highly recommendable box counting algorithm for estimation of D_q originally suggested in [LT89] and elaborated upon in [PS93] has been implemented; refer to [PS93] for a detailed description of the algorithm.

In practice when determining the dimension of a strange attractor the dimension defined by Eq. (A.2) is estimated for a series of increasing embedding dimensions L. Once L is sufficiently high D_q will saturate at the attractor dimension of the system. This is illustrated for the 10093 point laser series in Figure A.3; in particular, the correlation dimension D_2 has been considered here in order to compare with results obtained in the literature. Further, for the laser data the embedding was performed using a delay time $\tau = 2$ between the data points in order to be consistent with [PS93]. From Figure A.2 it is seen that the correlation dimension saturates at embedding dimension $L = 6$ and is estimated to $D_2 \approx 2.21$ by averaging the values obtained at $L = 6, 7, 8$. These results are equivalent to what was obtained and illustrated for the laser series in [PS93].

As in [PS93] the attractor is illustrated in a phase space plot of dimension two in order to visualize the nature of the attractor, even though we cannot expect the 2.2–dimensional attractor to be embedded in less than five dimensions according to Takens theorem. Figure A.4 shows the phase space plots for both the initial 1000 points as well as the full set of 10000 points, as the structure of the attractor shows more clearly using fewer points, possibly due to the measurement noise introduced by the 8-bit sampling. Note the “wheel”-like structure of the attractor with “spokes” emanating from the center.

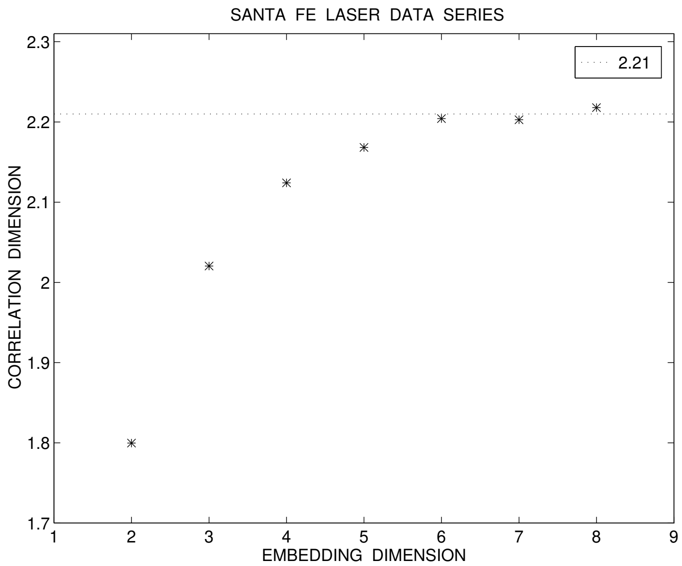


Figure A.3: Estimation of the correlation dimension D_2 for the laser series using an increasing embedding dimension. The series is embedded using a delay time $\tau = 2$.

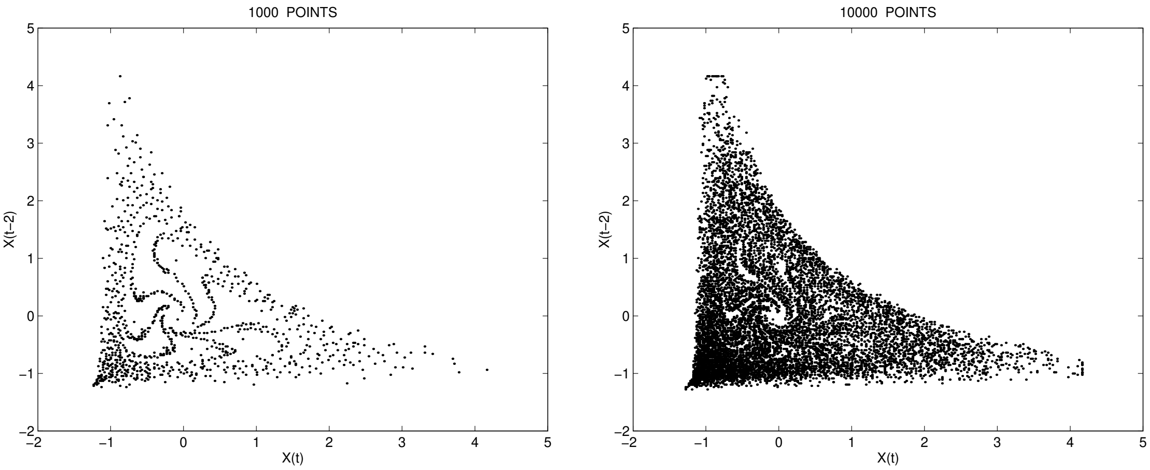


Figure A.4: Phase space plots for the laser series. Left panel: 1000 data points. Right panel: 10000 data points.

A.2 The Mackey-Glass series

The Mackey-Glass time series is a chaotic time series obtained by integration of a time-delay differential equation,

$$\frac{dx(t)}{dt} = -b\,x(t) + a\,\frac{x(t-\tau)}{a + x(t-\tau)^c} \quad , \tag{A.3}$$

where a, b and c are constants and τ is the delay parameter. The equation was originally proposed in [MG77] as a model of white blood cell production. It is common practice to take the constants as $a = 0.2$, $b = 0.1$ and $c = 10$. The nature of the behaviour of the equation is determined by the delay τ, for $\tau > 16.8$ the equation has a chaotic (strange) attractor; the series used in this work were obtained using $\tau = 17$ and a sampling period of $\Delta t = 1$. The data set that was available for this work comprised 8500 data points, extracted after the iterates had settled on the attractor; Figure A.5 illustrates a section of the data.

For the Mackey-Glass series it is common practice (e.g., [SUH90, MD89] and [SHLR93, HRSL94]) to implement a *six* step ahead predictor, i.e., in the case of modeling using a feed-forward network, the prediction $\widehat{x}(t)$ of $x(t)$ is obtained from a lag space $\mathbf{x}(t) = [x(t-6), x(t-12), \ldots]$ thus using a delay time of $\tau = 6$ between samples.

A.2.1 Attractor dimension for the Mackey-Glass series

As for the laser series the attractor correlation dimension D_2 defined by Eq. (A.2) was estimated on the 8500 point Mackey-Glass series by use of the efficient box counting algorithm adopted from [PS93]. Figure A.6 illustrates the correlation dimension estimates obtained for increasing embedding dimension L, using a delay time $\tau = 6$ between data points. It is seen that the dimension estimate saturates at embedding dimension $L = 3$ and is etimated to $D_2 \approx 1.92$ by averaging the values obtained at $L = 3$–6. In [GP83] the correlation dimension was estimated to $D_2 \approx 1.95$ and in [Wul92] the result $D_2 \approx 1.96$ was obtained. The small difference between these values and the value obtained in Figure A.6

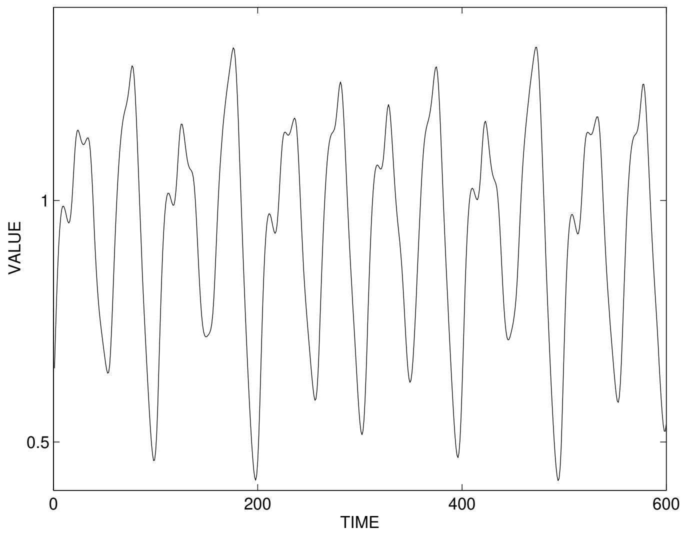


Figure A.5: A section of the Mackey-Glass chaotic time series.

is probably due to the smaller data set used here and the fact that the estimates in [GP83] and [Wul92] were not obtained by use of a box counting technique but rather by use of a correlation sum technique; refer to page 369 in [PS93] for a description of various techniques. Even if the dimension estimate obtained here might be slightly biased the main purpose has not been to accurately determine the dimension of the attractor but rather to provide a reference towards which the attractors generated by network models can be compared, using the same estimation tool.

In Figure A.7 the attractor for the Mackey-Glass series is illustrated in a phase space plot, using the 8500 point data set embedded in three dimensions using a delay time $\tau = 6$. Note the beautifully curled band-like structure.

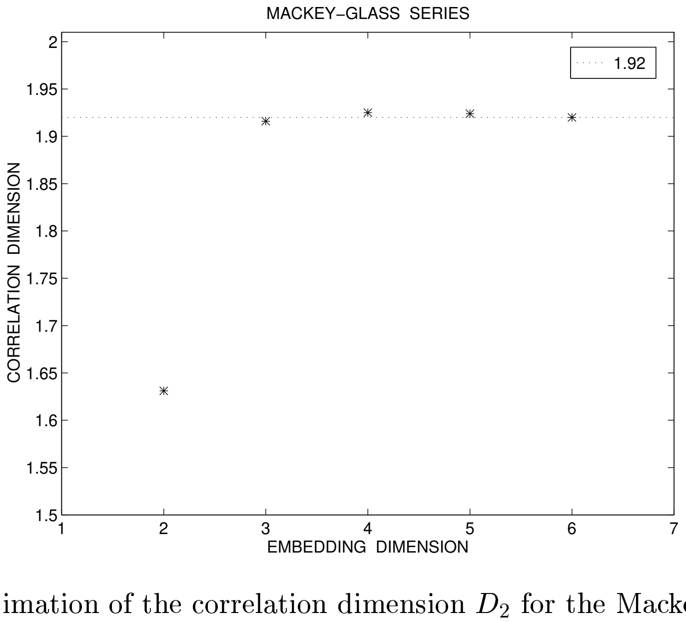


Figure A.6: Estimation of the correlation dimension D_2 for the Mackey-Glass series.

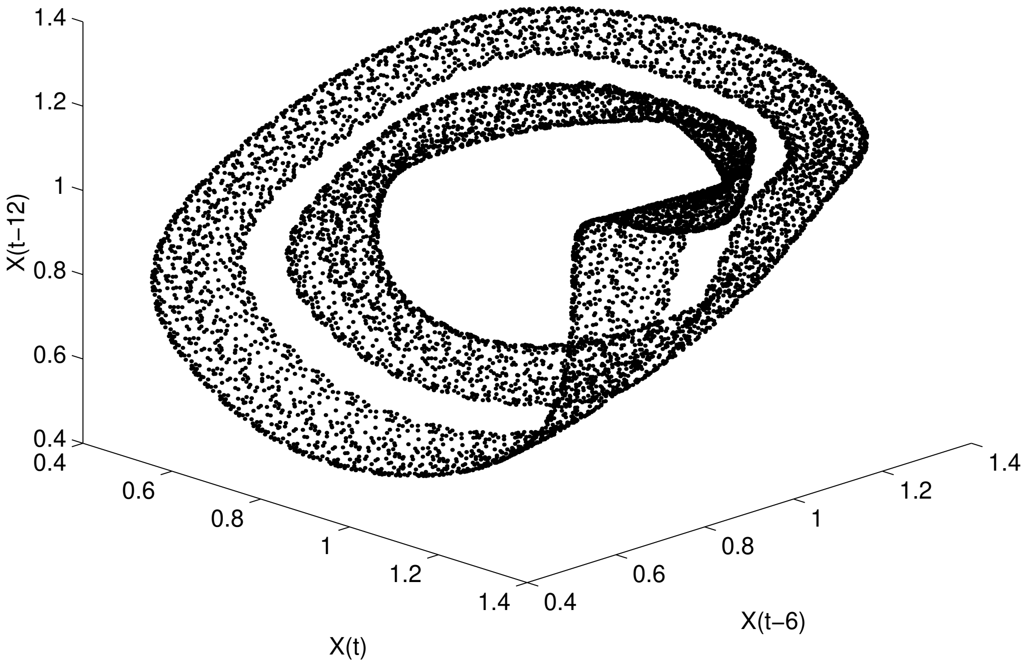


Figure A.7: Phase space plot for the Mackey-Glass data using 8500 data points.

Appendix B

Layered vs. non-layered update of recurrent networks

As described in section 3.3.1 the recurrent networks considered in the present work are *layered*, i.e., the units are divided into a number of sequential layers. In each iteration of operation the units in the network are updated layer by layer, feeding the outputs from preceeding layers to the inputs of the following layers in the same way as is done for feed-forward networks; this update may also be comprehended as an asynchronous update as the units are *not* updated simultaneously. In the present work the recurrent networks considered are restricted to a single hidden layer followed by an output unit but the layered concept may be extended to an arbitrary number of hidden layers as illustrated in e.g., [Ped94].

In numerous presentations, e.g., [WZ89, RF91, KZM94], the update of the units in the recurrent networks considered is performed in a synchronous manner, i.e., all units in the network are updated simultaneously in each iteration of operation. This updating scheme leads to *non*-layered networks meaning that all the units in the network, including designated output units, can be perceived as belonging to the same layer. This non-layered construction will however lead to reduced modeling capability of the resulting recurrent networks, compared to the layered construction. This will be demonstrated in the following.

For convenience of presentation we will focus on recurrent networks receiving a single external input $x(t)$ and having a single output $y(t)$; for the layered networks a single hidden layer will be assumed. However, the results of the following analysis will apply generally.

Whether working from a layered or non-layered construction, the outputs $s_j(t)$ from the N_h hidden units which are *not* producing externally available outputs are computed as

$$s_i(t) = f\left[\sum_{j=1}^{N_h} w_{ij}s_j(t-1) + w_{io}y(t-1) + w_{ix}x(t) + w_{ib}\right] \quad , \tag{B.1}$$

i.e., based on "internal" information from the previous time step $t-1$ only as well as external information for the present time step t. Here, w_{ij} is the weight *to* hidden unit i *from* unit j, w_{io} weights the previous output unit output, w_{ix} weights the external input and w_{ib} is the bias weight for unit i.

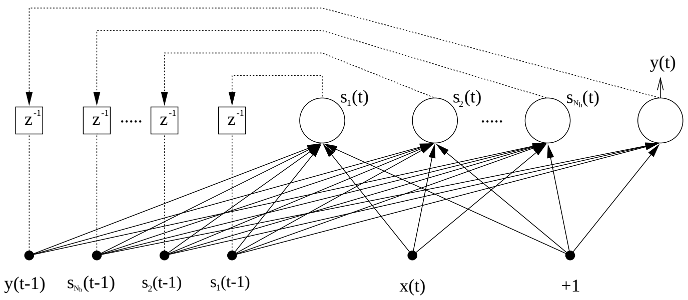


Figure B.1: *Non*-layered recurrent network construction.

Non-layered recurrent networks seem to be inspired from synchronous state machines [Koh78] from the field of *Finite State Automata* as the output unit is updated (or "clocked") at the same time as the hidden units, thus at time step t also being updated from the states of the units (the "state vector") from the previous time step $t-1$. Using the non-layered construction, the output unit output at time t is thus computed as

$$y(t) = \sum_{i=1}^{N_h} w_{oi} s_i(t-1) + w_{ob} \tag{B.2}$$

where w_{oi} is the weight to the output unit from hidden unit i and w_{ob} is the output unit output. Figure B.1 illustrates a non-layered recurrent network construction.

Layered recurrent networks on the other hand are inspired from feed-forward networks as the output of the output unit at time step t is based on the hidden unit outputs at the same time step t, which are computed prior to computing the output unit output. Using the layered construction, the output unit output at time t is therefore computed as

$$y(t) = \sum_{i=1}^{N_h} w_{oi} s_i(t) + w_{ob} \tag{B.3}$$

Figure B.2 illustrates a layered construction of a recurrent network which is otherwise similar to Figure B.1.

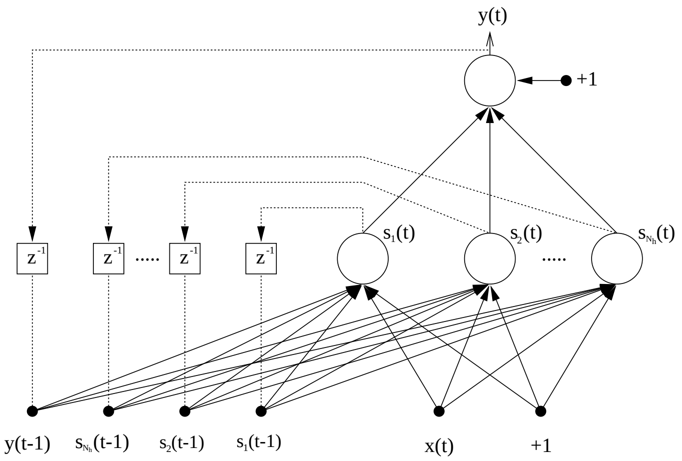


Figure B.2: Layered recurrent network construction.

When modeling using recurrent networks it is necessary to specify an initial value for the units in the network when starting iterations at the first time step. Assuming that the first time step where the network produces an output is $t = 1$ it is customary [WZ89] to set the previous values of the hidden units and the output unit to zero, i.e., $s_i(0) = 0$, $i = 1, \dots N_h$ and $y(0) = 0$.

If modeling using a *non-layered* recurrent network it is obvious from the initial value of the hidden units inserted into Eq. (B.2) that the initial output from the network at time $t = 1$ will be identical to the output unit bias w_{ob} which will not be appropriate for most modeling tasks. Furthermore, as the output $y(t)$ at time step t is based on the hidden unit outputs $s_i(t-1)$ from the previous time step, it is evident from Eqs. (B.1– B.2) that the output $y(t)$ is *not* based on the most recent external input $x(t)$, but rather the external input $x(t-1)$ from the previous time step. This will be inappropriate for most modeling tasks as well.

When modeling using a *layered* recurrent network however, it is straightforward to see from the initial unit values inserted into Eq. (B.1) and Eq. (B.3) that the initial output from the network at time $t = 1$ is identical to the output from a simple one-input feed-forward network. Furthermore, as the output $y(t)$ at time step t is based on the hidden unit outputs $s_i(t)$ from the *same* time step, it is evident from Eq. (B.1) inserted into Eq. (B.3) that the output $y(t)$ is indeed based on the most recent external input $x(t)$ which seems appropriate for most modeling tasks.

It should be evident from the observations described above that a *non-layered* construction of the units in a recurrent network will lead to reduced modeling capability for most tasks whereas the modeling abilities of a *layered* network are more appropriate. In the following the differences between the two updating schemes will be illustrated for a time series prediction task.

B.1 Simple example for comparison

The differences between a *layered* update and a *non-layered* update of the units in a recurrent network will now be illustrated for simple one hidden unit networks applied to a one-step ahead time series prediction problem. As above, the networks receive only a single external input $x(t)$ and the network output $y(t)$ at time step t corresponds to the prediction $\widehat{x}(t+1)$ of the input series in the following time step.

For the *non-layered* recurrent network, the output at time step t is calculated as

$$\begin{aligned} \widehat{x}(t+1) &= y(t) \\ &= w_{o1} s_1(t-1) + w_{ob} \\ &= w_{o1} f\left[w_{11} s_1(t-2) + w_{1o} y(t-2) + w_{1x} x(t-1) + w_{1b}\right] + w_{ob} \end{aligned} \tag{B.4}$$

as seen from Eqs. (B.1– B.2). Inserting initial values at time $t = 0$ we obtain an inital prediction as

$$\widehat{x}(2) = y(1) = w_{ob} \tag{B.5}$$

which is likely to be a very poor prediction. If using feedback connections from the output unit back to the hidden units as in this example, the poor initial prediction is fed back into the network and used for future predictions; this might introduce a significant error in the following predictions as well. By inspection of Eq. (B.4) we furthermore note that the predictions $\widehat{x}(t+1)$ are *not* based on the immediately preceeding value $x(t)$ but rather on $x(t-1)$ and earlier through the feedback. The non-layered recurrent network is thus implementing a *two*-step ahead predictor and *not* a one-step ahead predictor as intended.

For the *layered* recurrent network the output at time step t is calculated as

$$\begin{aligned} \widehat{x}(t+1) &= y(t) \\ &= w_{o1}s_1(t) + w_{ob} \\ &= w_{o1}f\left[w_{11}s_1(t-1) + w_{1o}y(t-1) + w_{1x}x(t) + w_{1b}\right] + w_{ob} \end{aligned} \tag{B.6}$$

as seen from Eq. (B.1) and Eq. (B.3). Inserting initial values at time $t=0$ into this equation we obtain an inital prediction as

$$\widehat{x}(2) = y(1) = w_{o1}f\left[w_{1x}x(1) + w_{1b}\right] + w_{ob} \tag{B.7}$$

which is recognized as the output from a simple one-input feed-forward network. This prediction is likely to be far better than the output unit bias in its own. By inspection of Eq. (B.6) we note that the predictions $\widehat{x}(t+1)$ are indeed based on values $x(t)$ and earlier, implementing a one-step ahead preditor as desired.

One may argue that if the output unit of the non-layered network in the example above received the external input $x(t)$ in line with the hidden units it would implement a one-step ahead predictor as well. This is indeed so but the modeling capabilities of the *non-layered* network would in this case still be reduced compared to the *layered* network. This is so since the model output $y(t)$ would be linear in $x(t)$ and non-linear only in $x(t-1)$ and earlier external inputs. In contrast, the *layered* network is non-linear in $x(t)$ and earlier inputs, allowing for implementation of more complex output functions, making the layered construction the preferable choice when modeling using recurrent networks.

Appendix C

Iterative computation of the inverse Hessian

This appendix contains an analysis of the computational complexity of a standard method for iterative computation of the inverse Hessian of e.g., the quadratic cost function. The method involves the *matrix inversion lemma* and is central to the application of second order methods for online training of adaptive models [Lju87], which would otherwise be computationally intractable.

Iterative computation of the inverse Hessian may also be applied to offline model adaption in order to avoid e.g., solving a linear system of equations when training using the Gauss-Newton method. Also, when performing architecture optimization using the OBS pruning scheme, a seemingly computationally costly matrix inversion may be avoided if the inverse is computed by the iterative method, as suggested in [HS93]. However, careful accounting of the computational costs reveals that the iterative method will generally require *more* operations than offline computation of the Hessian followed by a matrix inversion. This will be detailed in the following.

In order to iteratively compute the inverse Hessian it is required that the Hessian matrix itself is computed incrementally from a recursive expression of the form,

$$\mathbf{H}_t = \mathbf{H}_{t-1} + u_t \cdot \mathbf{g}_t \, \mathbf{g}_t^{\mathrm{T}} \;\;,\;\; t = 1, \ldots, T \;\;,\;\; \mathbf{H}_0 = \mathbf{0} \;\; . \tag{C.1}$$

Here, $\mathbf{H}_t$ denotes the Hessian of the cost function computed from the first example up to example t, T denotes the total number of training examples, u_t is a scalar and $\mathbf{g}_t$ is a vector. If the Hessian $\mathbf{H}$ to be computed is the Gauss-Newton approximation to the second derivatives of the quadratic cost function then the vector $\mathbf{g}_t$ denotes the partial derivatives of the model output for example t and the scalar u_t is constantly one; refer to Eq. (4.22). Iterative computation of the inverse Hessian may also be applied to the approximation of the second derivatives of the entropic cost function for Boltzmann networks Eq. (12.23). In this case $\mathbf{g}_t$ denotes the partial derivatives of the log likelihood of the model generating example t and u_t denotes the "target" probability for example t.

Provided that the Hessian can be brought on a form equivalent to Eq. (C.1) it is possible to compute the inverse from a standard lemma for matrix inversion,

$$\left[\mathbf{A} + \mathbf{BCD}\right]^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{B}\left[\mathbf{C}^{-1} + \mathbf{D}\mathbf{A}^{-1}\mathbf{B}\right]^{-1}\mathbf{D}\mathbf{A}^{-1} \;\; . \tag{C.2}$$

Letting $\mathbf{A} = \mathbf{H}_{t-1}$, $\mathbf{B} = \mathbf{D}^{\mathrm{T}} = \mathbf{g}_t$ and $\mathbf{C} = u_t$ we obtain

$$\mathbf{H}_t^{-1} = \mathbf{H}_{t-1}^{-1} - \frac{\mathbf{H}_{t-1}^{-1}\,\mathbf{g}_t\,\mathbf{g}_t^{\mathrm{T}}\,\mathbf{H}_{t-1}^{-1}}{u_t^{-1} + \mathbf{g}_t^{\mathrm{T}}\,\mathbf{H}_{t-1}^{-1}\,\mathbf{g}_t} \;\;,\;\; t = 1, \ldots, T \;\;,\;\; \mathbf{H}_0^{-1} = \alpha^{-1}\mathbf{I} \;\;. \tag{C.3}$$

In order to "get the iterations going" we need to start from an initial inverse $\mathbf{H}_0^{-1}$ different from zero. If the cost function is augmented by a simple weight decay Eq. (7.28) the initial inverse comes natural as the inverse of the second derivatives of the weight decay term, $\alpha^{-1}\mathbf{I}$, where $\mathbf{I}$ denotes the identity matrix. Otherwise, α must be set to a *small* value, thus introducing a small error in the inverse.

We will now analyze the number of operations required to compute the inverse Hessian from Eq. (C.3). For simplicity the analysis will focus on the number of multiplications and divisions entering the fraction as these operations are computationally more costly than additions and subtractions [Hay88]. As the total number of additions and subtractions will be approximately equal to the total number of multiplications and divisions, the computing time required for the latter operation types will thus be dominating. One may argue that the subtraction in Eq. (C.3) will amount to a substantial amount of computation which ought to be included in the analysis. However, the resulting count of subtractions is to be compared to Eq. (C.1) involving an equivalent number of additions. When making the comparison the omitted computations will cancel out and thus not influence the comparison, justifying the focus on multiplications and divisions.

In the analysis the symmetry of the Hessian will be taken into account, i.e., $\mathbf{H}^{-1}\,\mathbf{g} = (\mathbf{g}^{\mathrm{T}}\,\mathbf{H}^{-1})^{\mathrm{T}}$. This way, the most efficient way of computing the fraction entering Eq. (C.3) is to initially compute the vector $\mathbf{H}_{t-1}^{-1}\,\mathbf{g}_t = (\mathbf{g}_t^{\mathrm{T}}\,\mathbf{H}_{t-1}^{-1})^{\mathrm{T}} = \mathbf{v}_1$. From the resulting vector $\mathbf{v}_1$ we compute the term $\mathbf{g}_t^{\mathrm{T}}\,\mathbf{H}_{t-1}^{-1}\,\mathbf{g}_t$ as the dot product $\mathbf{v}_1^{\mathrm{T}} \cdot \mathbf{g}_t$ and calculate the denominator which is denoted *denom*. From the vector $\mathbf{v}_1$ we form a vector $\mathbf{v}_2$ as $\mathbf{v}_2 = \mathbf{H}_{t-1}^{-1}\,\mathbf{g}_t/\mathit{denom}$ and we finally compute the fraction as the outer product between $\mathbf{v}_1$ and $\mathbf{v}_2$, naturally utilizing the symmetry of the resulting matrix appropriately.

If we let n denote the dimensionality of the Hessian corresponding to the number of parameters in the model, the operations needed to form the fraction in each iteration of Eq. (C.3) are distributed as follows:

$$\begin{array}{lcl} \mathbf{H}_{t-1}^{-1}\,\mathbf{g}_t \;,\;\; \mathbf{g}_t^{\mathrm{T}}\,\mathbf{H}_{t-1}^{-1} & = & n^2 \\ \mathbf{g}_t^{\mathrm{T}}\,(\mathbf{H}_{t-1}^{-1}\,\mathbf{g}_t) & = & n \\ \mathbf{H}_{t-1}^{-1}\,\mathbf{g}_t \,/\, \mathit{denom} & = & n \\ (\mathbf{H}_{t-1}^{-1}\mathbf{g}_t \,/\, \mathit{denom})\,(\mathbf{g}_t^{\mathrm{T}}\,\mathbf{H}_{t-1}^{-1}) & = & n(n+1)/2 \end{array} \tag{C.4}$$

In total the iterative computation of the inverse Hessian matrix $\mathbf{H}^{-1}$ amounts to $T(\frac{3}{2}n^2 + \frac{5}{2}n)$, where T as previously denotes the total number of training examples. This number is to be compared to the operations count for computing the Hessian $\mathbf{H}$ from Eq. (C.1), followed by a matrix inversion. Assuming that the constant u_t is different from unity the total number of operations required for computing the Hessian $\mathbf{H}$ amounts to $T(n + n(n+1)/2) = T(\frac{1}{2}n^2 + \frac{3}{2}n)$, again utilizing symmetry. Add to this the cost of inverting the Hessian matrix by e.g., Gauss-Jordan elimination which costs approximately n^3 operations [PFTV92].

Comparing the operations counts for the two approaches towards obtaining the inverse

Hessian matrix, we see that the iterative method will be preferable provided that

$$\begin{aligned} & n^3 + T(\tfrac{1}{2}n^2 + \tfrac{3}{2}n) > T(\tfrac{3}{2}n^2 + \tfrac{5}{2}n) \\ \Updownarrow \quad & \\ & n > \tfrac{1}{2}T\left(1 + \sqrt{1 + 4/T}\right) \approx T \end{aligned} \tag{C.5}$$

i.e., if the number of parameters in the model exceeds the number of training examples. This will generally *not* be the case when training adaptive models, as training in this case forms an ill-posed problem. Rather, the number of parameters will generally be significantly *less than* the number of training examples in order to obtain better determined parameters and thus improved generalization ability.

The iterative method for computing the inverse Hessian is sometimes promoted based on its cost scaling as $\mathcal{O}(n^2)$ whereas the traditional matrix inversion scales as $\mathcal{O}(n^3)$. This argument is perfectly sound in the framework of online methods, but for offline computations the number of parameters n is no longer the dominating factor; the number of training examples T is. Therefore the iterative method will generally require more computations than the "standard" method and is therefore not recommendable for offline learning.

Appendix D

Eigenvalue analysis in terms of the Jacobian

This appendix provides the mathematical prerequisites for an analysis of the condition number of a Hessian matrix as well as the magnitudes of the smallest and largest eigenvalues in terms of columns of the Jacobian matrix. The material presented in this appendix is an elaboration on material presented in [SBC93]. It will be demonstrated that an increasing collinearity between columns of the Jacobian leads to a decrease of the *smallest* Hessian eigenvalue and that an increasing disparity between the lengths of the Jacobian columns leads to an increase of the *largest* Hessian eigenvalue.

We start by formulating

Corollary D.1 *Let* $\mathbf{J} = [\mathbf{j}_1, \ldots, \mathbf{j}_n]$ *be a column partitioning of* $\mathbf{J} \in \Re^{m \times n}$, $m \geq n$. *If* $\mathbf{J}_r = [\mathbf{j}_1, \ldots, \mathbf{j}_r]$ *then for* $r = 1, \ldots, n-1$

$$\sigma_1(\mathbf{J}_{r+1}) \geq \sigma_1(\mathbf{J}_r) \geq \sigma_2(\mathbf{J}_{r+1}) \geq \ldots \geq \sigma_r(\mathbf{J}_{r+1}) \geq \sigma_r(\mathbf{J}_r) \geq \sigma_{r+1}(\mathbf{J}_{r+1}) \ .$$

Proof. This corresponds to Corollary 8.6.3 in [GL96]. □

The Corollary says that by adding a column to a matrix, the largest singular value increases and the smallest singular value is diminished. This leads to the following Proposition,

Proposition D.1 *Let* $\mathbf{A}$ *be a submatrix of* $\mathbf{J}$ *consisting of columns of* $\mathbf{J}$. *Then* $\kappa(\mathbf{A}) \leq \kappa(\mathbf{J})$.

Proof. This follows from repeated application of Corollary D.1. □

The effects of Corollary D.1 and Proposition D.1 are the possibilities of investigating the singular values and thus the condition number of a matrix $\mathbf{J}$ in terms of a submatrix $\mathbf{A}$. From Propostition D.1 it immediately follows that

$$\kappa(\mathbf{H}) \ = \ \kappa(\mathbf{J}^{\mathrm{T}}\mathbf{J}) \ = \ \kappa^2(\mathbf{J}) \ \ \geq \ \ \kappa^2(\mathbf{A}) \ = \kappa(\mathbf{A}^{\mathrm{T}}\mathbf{A}) \tag{D.1}$$

stating that the condition number of a matrix $\mathbf{H} = \mathbf{J}^{\mathrm{T}}\mathbf{J}$ can be examined in terms of a submatrix $\mathbf{A}$ consisting of columns of $\mathbf{J}$. We now formulate

Proposition D.2 *Let* $\mathbf{A} = [\mathbf{x}\ \mathbf{y}] \in \Re^{m\times 2}$ *and suppose that the angle* θ *between* $\mathbf{x}$ *and* $\mathbf{y}$ *is small so that* $\cos\theta \approx 1 - \frac{\theta^2}{2}$. *Then*

$$\kappa^2(\mathbf{A}) \quad > \quad \frac{1}{\theta^2(4-\theta^2)} \cdot \left(\frac{\|\mathbf{x}\|^2}{\|\mathbf{y}\|^2} + \frac{\|\mathbf{y}\|^2}{\|\mathbf{x}\|^2} + 2 \right) \quad .$$

Proof. The square of the condition number of $\mathbf{A}$ is obtained as the condition number of $\mathbf{G} = \mathbf{A}^{\mathrm{T}}\mathbf{A}$. The eigenvalues of $\mathbf{G}$ are non-negative and are obtained by solving

$$\lambda^2 - \lambda\left(\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y}\right) - \left(\mathbf{y}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{x}^{\mathrm{T}}\mathbf{y} - \mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}\right) \quad = \quad 0 \tag{D.2}$$

The solutions are obtained as

$$\begin{aligned} \lambda &= \frac{1}{2}\left[\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y} \pm \sqrt{(\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y})^2 + 4\left((\mathbf{x}^{\mathrm{T}}\mathbf{y})^2 - \mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}\right)}\right] && \text{(D.3)} \\ &= \frac{1}{2}\left[\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y} \pm \sqrt{(\mathbf{x}^{\mathrm{T}}\mathbf{x} - \mathbf{y}^{\mathrm{T}}\mathbf{y})^2 + 4(\mathbf{x}^{\mathrm{T}}\mathbf{y})^2}\right] && \text{(D.4)} \end{aligned}$$

The condition number of $\mathbf{G}$ is calculated as $\kappa(\mathbf{G}) = \lambda_{\max}/\lambda_{\min}$ and we obtain

$$\begin{aligned} \kappa(\mathbf{G}) &= \frac{\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y} + \sqrt{(\mathbf{x}^{\mathrm{T}}\mathbf{x} - \mathbf{y}^{\mathrm{T}}\mathbf{y})^2 + 4(\mathbf{x}^{\mathrm{T}}\mathbf{y})^2}}{\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y} - \sqrt{(\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y})^2 + 4\left((\mathbf{x}^{\mathrm{T}}\mathbf{y})^2 - \mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}\right)}} && \text{(D.5)} \\ &= \frac{\left[\ \mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y} + \sqrt{(\mathbf{x}^{\mathrm{T}}\mathbf{x} - \mathbf{y}^{\mathrm{T}}\mathbf{y})^2 + 4(\mathbf{x}^{\mathrm{T}}\mathbf{y})^2}\ \right]^2}{4\left[\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y} - (\mathbf{x}^{\mathrm{T}}\mathbf{y})^2\right]} && \text{(D.6)} \\ &> \frac{[\ \mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y}\ ]^2}{4\left[\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y} - (\mathbf{x}^{\mathrm{T}}\mathbf{y})^2\right]} && \text{(D.7)} \end{aligned}$$

where the square root is neglected for simplicity. Since we assume that $\cos\theta = \frac{\mathbf{x}^{\mathrm{T}}\mathbf{y}}{\|\mathbf{x}\|\cdot\|\mathbf{y}\|} = 1 - \frac{\theta^2}{2}$ we have $(\mathbf{x}^{\mathrm{T}}\mathbf{y})^2 = (1-\frac{\theta^2}{2})^2\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}$. Inserting into (D.7) leads to

$$\kappa(\mathbf{G}) \quad > \quad \frac{[\ \mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y}\ ]^2}{4\left[1 - (1-\frac{\theta^2}{2})^2\right]\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}} \quad = \quad \frac{1}{\theta^2(4-\theta^2)} \cdot \left(\frac{\|\mathbf{x}\|^2}{\|\mathbf{y}\|^2} + \frac{\|\mathbf{y}\|^2}{\|\mathbf{x}\|^2} + 2 \right) \quad . \qquad \square$$

Proposition D.2 combined with (D.1) puts a lower bound on the condition number of a matrix $\mathbf{H} = \mathbf{J}^{\mathrm{T}}\mathbf{J}$. Taking $\mathbf{A}$ in Proposition D.2 to be composed of two columns of $\mathbf{J}$ we learn that the condition number will grow large if two (or more) columns of $\mathbf{J}$ grow nearly collinear, leading to a small θ. Furthermore, we learn that the condition number will grow if the length of some columns of $\mathbf{J}$ grow large compared to the length of other columns, i.e., if a large column length disparity arise.

The proposition in itself does however not inform us about how these situations will affect the smallest and the largest eigenvalues of $\mathbf{H}$ *individually* as the proposition relates to their ratio only. We may however investigate these eigenvalues in terms the submatrix $\mathbf{A}$ as well since

$$\begin{aligned} \lambda_{\max}(\mathbf{H}) &\geq \lambda_{\max}(\mathbf{A}^{\mathrm{T}}\mathbf{A}) && \text{(D.8)} \\ \lambda_{\min}(\mathbf{H}) &\leq \lambda_{\min}(\mathbf{A}^{\mathrm{T}}\mathbf{A}) && \text{(D.9)} \end{aligned}$$

which follows from the fact that the eigenvalues of $\mathbf{H}$ are obtained as the square of the singular values of $\mathbf{J}$ (refer to section 7.1) combined with Corollary D.1.

In order to determine the effects of column collinearity and column length disparity on the individual eigenvalues we must investigate (D.3–D.4) during these situations. We will first consider the situation where θ becomes small, corresponding to two columns $\mathbf{A}$ of $\mathbf{J}$ being nearly collinear. In order to focus on the effects of a decreasing θ only we will assume that the lengths of the columns are equivalent, $\mathbf{x}^{\mathrm{T}}\mathbf{x} \approx \mathbf{y}^{\mathrm{T}}\mathbf{y}$. In this case the square root entering Eq. (D.4) may be calculated as

$$\begin{aligned} & \left[(\mathbf{x}^{\mathrm{T}}\mathbf{x} - \mathbf{y}^{\mathrm{T}}\mathbf{y})^2 + 4(\mathbf{x}^{\mathrm{T}}\mathbf{y})^2\right]^{\frac{1}{2}} && \text{(D.10)} \\ = & \left[(\mathbf{x}^{\mathrm{T}}\mathbf{x})^2 - 2\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y} + (\mathbf{y}^{\mathrm{T}}\mathbf{y})^2 + 4(1-\frac{\theta^2}{2})^2\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}\right]^{\frac{1}{2}} && \text{(D.11)} \end{aligned}$$

where we used $(\mathbf{x}^{\mathrm{T}}\mathbf{y})^2 = (1-\frac{\theta^2}{2})^2\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}$. Using $\mathbf{x}^{\mathrm{T}}\mathbf{x} \approx \mathbf{y}^{\mathrm{T}}\mathbf{y}$ and collecting terms we obtain

$$\begin{aligned} & \left[\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}\left(\frac{(\mathbf{x}^{\mathrm{T}}\mathbf{x})^2}{\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}} + \frac{(\mathbf{y}^{\mathrm{T}}\mathbf{y})^2}{\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}} - 2 + 4(1-\frac{\theta^2}{2})^2\right)\right]^{\frac{1}{2}} && \text{(D.12)} \\ \approx & \left[4\,\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}(1-\frac{\theta^2}{2})^2\right]^{\frac{1}{2}} && \text{(D.13)} \\ \approx & \;\mathbf{x}^{\mathrm{T}}\mathbf{x}(2-\theta^2) && \text{(D.14)} \end{aligned}$$

Inserting this result into (D.4) we obtain

$$\begin{aligned} \lambda_{\max} & \approx \frac{1}{2}\left[\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y} + \mathbf{x}^{\mathrm{T}}\mathbf{x}(2-\theta^2)\right] && \text{(D.15)} \\ & \approx 2\,\mathbf{x}^{\mathrm{T}}\mathbf{x} && \text{(D.16)} \end{aligned}$$

and

$$\begin{aligned} \lambda_{\min} & \approx \frac{1}{2}\left[\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y} - \mathbf{x}^{\mathrm{T}}\mathbf{x}(2-\theta^2)\right] && \text{(D.17)} \\ & \approx \frac{\theta^2}{2}\,\mathbf{x}^{\mathrm{T}}\mathbf{x} && \text{(D.18)} \end{aligned}$$

From these results we learn that the overall scale of the eigenvalues is determined by $\|\mathbf{x}\|^2 \approx \|\mathbf{y}\|^2$ and that the smallest eigenvalue scales as $\frac{\theta^2}{2}$. Thus, assuming constant and comparable column lengths in $\mathbf{J}$ and increasing collinearity, the smallest eigenvalue of $\mathbf{H}$ will decrease as $\frac{\theta^2}{2}$ while the largest eigenvalue will remain unchanged.

The second situation that we learned from Proposition D.2 would lead to a growing condition number of $\mathbf{H} = \mathbf{J}^{\mathrm{T}}\mathbf{J}$ is when an increasing disparity between the lengths of some columns of $\mathbf{J}$ arise This situation corresponds to e.g., $\mathbf{x}^{\mathrm{T}}\mathbf{x} \gg \mathbf{y}^{\mathrm{T}}\mathbf{y}$. In this case the square root entering (D.3) may be calculated as

$$\begin{aligned} & \sqrt{(\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y})^2 + 4\left((\mathbf{x}^{\mathrm{T}}\mathbf{y})^2 - \mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}\right)} && \text{(D.19)} \\ = & \sqrt{(\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y})^2 + 4\left(\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}\cos^2\theta - \mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}\right)} && \text{(D.20)} \\ = & \sqrt{(\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y})^2 + 4\mathbf{x}^{\mathrm{T}}\mathbf{x}\cdot\mathbf{y}^{\mathrm{T}}\mathbf{y}\,(\cos^2\theta - 1)} && \text{(D.21)} \end{aligned}$$

where we used $\cos^2\theta = \frac{(\mathbf{x}^T\mathbf{y})^2}{\mathbf{x}^T\mathbf{x}\cdot\mathbf{y}^T\mathbf{y}}$. Using $\cos^2\theta + \sin^2\theta = 1$ we obtain

$$\begin{aligned} & \sqrt{(\mathbf{x}^T\mathbf{x}+\mathbf{y}^T\mathbf{y})^2 - 4\mathbf{x}^T\mathbf{x}\cdot\mathbf{y}^T\mathbf{y}\sin^2\theta} && \text{(D.22)} \\ = \; & (\mathbf{x}^T\mathbf{x}+\mathbf{y}^T\mathbf{y})\sqrt{1 - \frac{4\mathbf{x}^T\mathbf{x}\cdot\mathbf{y}^T\mathbf{y}\sin^2\theta}{(\mathbf{x}^T\mathbf{x}+\mathbf{y}^T\mathbf{y})^2}}\;. && \text{(D.23)} \end{aligned}$$

As we are assuming $\mathbf{x}^T\mathbf{x} \gg \mathbf{y}^T\mathbf{y}$ the fraction within the square root will be small, and we may use the approximation $\sqrt{1+x} \approx 1 + \frac{x}{2}$ to obtain

$$\begin{aligned} & (\mathbf{x}^T\mathbf{x}+\mathbf{y}^T\mathbf{y})\left(1 - \frac{2\mathbf{x}^T\mathbf{x}\cdot\mathbf{y}^T\mathbf{y}\sin^2\theta}{(\mathbf{x}^T\mathbf{x}+\mathbf{y}^T\mathbf{y})^2}\right) && \text{(D.24)} \\ = \; & \mathbf{x}^T\mathbf{x}+\mathbf{y}^T\mathbf{y} - \frac{2\mathbf{x}^T\mathbf{x}\cdot\mathbf{y}^T\mathbf{y}\sin^2\theta}{(\mathbf{x}^T\mathbf{x}+\mathbf{y}^T\mathbf{y})} && \text{(D.25)} \\ \approx \; & \mathbf{x}^T\mathbf{x}+\mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{y}\left(\theta^2 - \frac{\theta^4}{3}\right) && \text{(D.26)} \end{aligned}$$

in which we once more used $\mathbf{x}^T\mathbf{x} \gg \mathbf{y}^T\mathbf{y}$ as well as the expansion, $\sin^2\theta \approx \theta^2 - \frac{\theta^4}{3}$. Inserting into (D.3) we obtain

$$\begin{aligned} \lambda_{\max} & \approx \frac{1}{2}\left[\mathbf{x}^T\mathbf{x}+\mathbf{y}^T\mathbf{y}+\mathbf{x}^T\mathbf{x}+\mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{y}\left(\theta^2 - \frac{\theta^4}{3}\right)\right] && \text{(D.27)} \\ & \approx \mathbf{x}^T\mathbf{x} && \text{(D.28)} \end{aligned}$$

and

$$\begin{aligned} \lambda_{\min} & \approx \frac{1}{2}\left[\mathbf{x}^T\mathbf{x}+\mathbf{y}^T\mathbf{y}-\mathbf{x}^T\mathbf{x}-\mathbf{y}^T\mathbf{y} + 2\mathbf{y}^T\mathbf{y}\left(\theta^2 - \frac{\theta^4}{3}\right)\right] && \text{(D.29)} \\ & = \mathbf{y}^T\mathbf{y}\left(\theta^2 - \frac{\theta^4}{3}\right) && \text{(D.30)} \\ & \approx \mathbf{y}^T\mathbf{y}\theta^2 && \text{(D.31)} \end{aligned}$$

From these results we learn that in the case of a large difference between column lengths, $\mathbf{x}^T\mathbf{x} \gg \mathbf{y}^T\mathbf{y}$, the largest eigenvalue is determined by the largest magnitude vector $\mathbf{x}$ and the smallest eigenvalue is determined by the smallest magnitude vector $\mathbf{y}$. Thus, other factors being equal, growth in the length of a column in $\mathbf{J}$ will lead to a growth in the largest eigenvalue of $\mathbf{H}$ while the smallest eigenvalue will remain unchanged. Further, if at the same time the length of the shorter column and possibly the angle between the two columns decrease, then the smallest eigenvalue will decrease as well.

The analysis in this appendix relates in a natural way to a Jacobian matrix $\mathbf{J}$ and the corresponding (Gauss-Newton approximation to the) Hessian matrix $\mathbf{H} = \mathbf{J}^T\mathbf{J}$. To summarize the analysis it has been demonstrated how near collinearity between some columns of the Jacobian will lead to small eigenvalues of the Hessian while the largest eigenvalue remains unaffected. Furthermore it has been demonstrated how growth in the length of some columns of the Jacobian compared to the length of other columns will lead to an increase of the largest eigenvalue of the Hessian only. The resulting effect of these situations is naturally an increasing condition number of the Hessian.

Appendix E

Perturbation analysis

This appendix contains an analysis of the influence of the condition number of a square matrix on the solution when solving a system of linear equations in the presence of errors. The analysis is focused around the influence on the solution when the matrix or the right-hand side are encumbered with small errors, possibly due to finite precision arithmetic resulting in accumulated rounding errors during the formation of these quantities.

The condition number of a (non-singular) square matrix $\mathbf{A}$ is defined generally as

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \, \|\mathbf{A}^{-1}\| \tag{E.1}$$

where $\|\cdot\|$ denotes a matrix norm induced from a corresponding vector norm. In the case of the matrix norm being induced from the l_2-norm, a.k.a. the Euclidean norm, and the matrix $\mathbf{A}$ being symmetric, this definition coincides with the definition of the condition number defined in terms of an SVD decomposition for a general matrix, as described in section 7.1. The norm of a matrix multiplied by a vector satisfies

$$\|\mathbf{A}\mathbf{x}\| \leq \|\mathbf{A}\| \, \|\mathbf{x}\| \tag{E.2}$$

which will be made use of in the following. Consider the system of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{E.3}$$

where $\mathbf{A}$ is assumed to be square and non-singular. The *exact* solution to this system is given by

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{E.4}$$

We will now analyse the effects of small perturbations of $\mathbf{A}$ and $\mathbf{b}$ in terms of the condition number. Such perturbations might arise due to e.g., finite precision arithmetic and accumulated rounding errors during the computation of these quantities. In order to simplify the analysis we will consider perturbations of the lefthand side $\mathbf{A}$ and the righthand side $\mathbf{b}$ separately. The subsequent perturbation analysis is adopted from [GMW81, DS83].

Suppose that the righthand side of Eq. (E.3) is perturbed to $\mathbf{b} + \delta\mathbf{b}$ and the exact solution to the perturbed system is $\mathbf{x} + \delta\mathbf{x}$, i.e.,

$$\begin{aligned} &\mathbf{A}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \\ \Updownarrow\quad& \\ &\mathbf{A}\delta\mathbf{x} = \delta\mathbf{b} \end{aligned} \tag{E.5}$$

where "δ" denotes a small change. The solution to the reduced system is obtained as

$$\delta\mathbf{x} = \mathbf{A}^{-1}\delta\mathbf{b} \ . \tag{E.6}$$

By use of Eq. (E.2) and corresponding vector and matrix norms we obtain

$$\|\delta\mathbf{x}\| \leq \|\mathbf{A}^{-1}\| \, \|\delta\mathbf{b}\| \ . \tag{E.7}$$

From the "original" system $\mathbf{Ax} = \mathbf{b}$ we similary obtain

$$\|\mathbf{b}\| \leq \|\mathbf{A}\| \, \|\mathbf{x}\| \quad \Leftrightarrow \quad \frac{1}{\|\mathbf{x}\|} \leq \frac{\|\mathbf{A}\|}{\|\mathbf{b}\|} \ . \tag{E.8}$$

By combination of the inequalities in (E.7) and (E.8) we obtain

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|\mathbf{A}\| \, \|\mathbf{A}^{-1}\| \, \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} = \kappa(\mathbf{A}) \, \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} \tag{E.9}$$

where $\|\delta\mathbf{x}\|/\|\mathbf{x}\|$ is the *relative* error of the solution.

Similary, if instead the lefthand side of Eq. (E.3) is perturbed by $\delta\mathbf{A}$ then the perturbed system may be written as

$$\begin{aligned} & (\mathbf{A} + \delta\mathbf{A})(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} \\ \Updownarrow \\ & \mathbf{A}\delta\mathbf{x} + \delta\mathbf{A}(\mathbf{x} + \delta\mathbf{x}) = 0 \\ \Updownarrow \\ & \delta\mathbf{x} = -\mathbf{A}^{-1}\delta\mathbf{A}(\mathbf{x} + \delta\mathbf{x}) \end{aligned} \tag{E.10}$$

where "δ" once more denotes small changes. Two applications of (E.2) using corresponding vector and matrix norms lead to

$$\|\delta\mathbf{x}\| \leq \|\mathbf{A}^{-1}\| \, \|\delta\mathbf{A}(\mathbf{x} + \delta\mathbf{x})\| \leq \|\mathbf{A}^{-1}\| \, \|\delta\mathbf{A}\| \, \|(\mathbf{x} + \delta\mathbf{x})\| \tag{E.11}$$

which may be rearranged as

$$\frac{\|\delta\mathbf{x}\|}{\|(\mathbf{x} + \delta\mathbf{x})\|} \leq \|\mathbf{A}^{-1}\| \, \|\mathbf{A}\| \, \frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} \ , \tag{E.12}$$

from which we obtain

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \approx \frac{\|\delta\mathbf{x}\|}{\|(\mathbf{x} + \delta\mathbf{x})\|} \leq \|\mathbf{A}^{-1}\| \, \|\mathbf{A}\| \, \frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} = \kappa(\mathbf{A}) \, \frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} \ . \tag{E.13}$$

In both cases, the relative change in the solution $\mathbf{x}$ to Eq. (E.3) due to perturbations of the data is bounded by the relative change in the data multiplied by the condition number of $\mathbf{A}$. The results indicate that if the condition number is "large", the solution may be changed substantially by even small changes in the data. The condition number of the matrix $\mathbf{A}$ is therefore a measure of the *sensitivity* of the system Eq. (E.3) to changes in the data, in terms of the ratio of the maximum to the minimum *stretch* induced by $\mathbf{A}$. In the case of a large condition number the system of linear equations is said to be *ill-conditioned.*

Note that the perturbation analysis above has concerned the *exact* solution of a system of linear equations and therefore addresses an inherent characteristic of the mathematical problem. In practice the system is however solved using finite precision arithmetic which may in itself lead to inaccuracies in the obtained solution. The nature of the inaccuracies introduced by the solver were analyzed in [Wil63] where it was shown that when using finite precision arithmetic to solve a system $\mathbf{Ax} = \mathbf{b}$, the inaccurate solution $\widetilde{\mathbf{x}}$ obtained is equivalent to the exact solution to a perturbed system, $(\mathbf{A} + \delta\mathbf{A})\widetilde{\mathbf{x}} = \mathbf{b}$. This analysis is referred to as the Wilkinson backward error analysis in the literature. The analysis shows that $\|\delta\mathbf{A}\|/\|\mathbf{A}\|$ in (E.13) is limited to a constant k times the machine precision ϵ (defined below) so that the relative error of the exact solution $\mathbf{x}$ is bounded by $\kappa(\mathbf{A}) \cdot k \cdot \epsilon$; hence, the condition number of the matrix is once more found to play a decisive rôle.

From the analyses above we learn that in order for the obtained solution to be considered trustworthy it is required that

$$\kappa(\mathbf{A}) \frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} \ll 1 \quad , \tag{E.14}$$

and it is consequently of interest to determine in practice when the system should be considered too ill-conditioned for this to hold. In order to determine a bound on the condition number it is necessary to take the floating point formats in which real numbers are stored in computers into account; the reader is referred to [Gol91] for an excellent introduction to floating point formats and floating point arithmetic. A key measure of any floating point format is the *machine precision*. The machine precision, or the machine *epsilon* ϵ, is defined as the largest relative rounding error with which a real number is stored in a finite precision floating point representation. For the widely adopted IEEE 754 64-bit floating point standard using *base* $\beta = 2$ the *significand* is 52 bits wide and the machine epsilon is obtained as $\epsilon = 2^{-52} \approx 2.22 \cdot 10^{-16}$.

From the definition of the machine precision combined with the expressions (E.9) and (E.13) we learn that if $\kappa(\mathbf{A}) \geq \epsilon^{-1}$ then the computed solution to a system of linear equations is likely to be entirely unreliable as imprecision in the data corresponding to even the smallest representable relative error ϵ may blow up to dominate the most significant digit of the solution, leading to a relative error of order unity. In [DS83] it is however stated that it is "generally felt" (in the optimization community) that the solution *may* not be trustworthy already if $\kappa(\mathbf{A}) > \epsilon^{-\frac{1}{2}}$. The statement is not mathematically justified in terms of a rigorous systems analysis but rather seems to serve as a "rule of thumb" based on experience as to when attention should be paid to the condition number. For the IEEE 64-bit floating point representation the rule of thumb reads

$$\kappa(\mathbf{A}) > 6.7 \cdot 10^{7} \quad . \tag{E.15}$$

The perturbation analysis provided in this appendix has been a "worst case" analysis providing upper bounds on the effect of the condition number on small errors in the data when solving a system of linear equations. The upper bounds apply in general but may turn out to be too pessimistic for some problems. The *exact* influence of the condition number is determined by the particular structure of the problem at hand and cannot be accounted for in general terms.

Appendix F

Conversion of Boltzmann chains to HMMs

This appendix contains the method given in [Mac96] for conversion of a Boltzmann chain into an equivalent HMM. In order to perform this conversion the final hidden unit state of the Boltzmann chain must be constrained to be a *particular* end state i_L. Naturally, the resulting HMM must obey the same constraint.

For convenience the Boltzmann distribution is repeated below for a Boltzmann chain modeling sequences of length L. The probability of finding the chain in visible states $\alpha = \{j_t\}_{t=1}^L$ and hidden states $\beta = \{i_t\}_{t=1}^L$ is given by

$$P_{\alpha\beta} = P\left(\{i_t, j_t\}_1^L\right) = \frac{1}{Z} \exp\left(\Pi_{i_1} + \sum_{t=1}^{L-1} A_{i_t i_{t+1}} + \sum_{t=1}^{L} B_{i_t j_t}\right) \tag{F.1}$$

where Z is the partition function Eq. (12.14). For sequences of length L, the joint distribution of the sequences for an HMM is

$$P\left(\{i_t, j_t\}_1^L\right) = \pi_{i_1} \prod_{t=1}^{L-1} a_{i_t i_{t+1}} \prod_{t=1}^{L} b_{i_t j_t} \tag{F.2}$$

where π_i is the prior probability for the initial hidden state, $a_{ii'}$ are the transition probabilities between hidden states and b_{ij} are the observation probabilities. These probabilities satisfies the constraints

$$\sum_i \pi_i = 1 \tag{F.3}$$

$$\sum_{i'} a_{ii'} = 1 \;,\; \forall i \tag{F.4}$$

$$\sum_j b_{ij} = 1 \;,\; \forall i \tag{F.5}$$

with indexes i, i' running over the n hidden states, and index j running over the m visible states.

We now seek to manipulate the terms in the exponents for the Boltzmann distribution Eq. (F.1) to allow for conversion to the parameters of the HMM distribution Eq. (F.2) without changing the distribution itself. In the following we use quantities α_{i_t} and β_{i_t}. These should not be confused with the notation α and β (no subscript) previously used to denote visible and hidden unit state sequences.

We note that the distribution Eq. (F.1) is left unchanged if we subtract arbitrary constants μ, ν from the energy functions entering as exponents in Eq. (F.1), since this is equivalent to multiplying both the numerator and the denominator Z with the same constants, leaving the fraction unchanged. The distribution is also unchanged if we add arbitrary terms β_{i_t} to every appearance of $B_{i_t j_t}$, provided we also subtract β_{i_t} again from the corresponding appearance of $A_{i_t i_{t+1}}$. Similary, we can add terms $\alpha_{i_{t+1}}$ to every appearance of $A_{i_t i_{t+1}}$ without affecting the distribution, provided we subtract the same term $\alpha_{i_{t+1}}$ from the following term $A_{i_{t+1} i_{t+2}}$. By performing these manipulations, we may rewrite Eq. (F.1) unchanged as

$$\begin{aligned} P\left(\{i_t, j_t\}_1^L\right) \quad = & \\ & \frac{1}{Z}\exp\left((\Pi_{i_1} - \alpha_{i_1} - \mu) + \sum_{t=1}^{L-1}(A_{i_t i_{t+1}} + \alpha_{i_t} - \alpha_{i_{t+1}} + \beta_{i_t} - \nu)\right. \\ & \left. + \sum_{t=1}^{L}(B_{i_t j_t} - \beta_{i_t}) + \alpha_{i_L} + \beta_{i_L}\right) \end{aligned} \tag{F.6}$$

where $\mu, \nu, \{\alpha_i, \beta_i\}_1^n$ are arbitrary quantities. This Boltzmann distribution is equivalent to an HMM distribution Eq. (F.2) if the following requirements are met [Mac96]:

- The quantities $\pi_i \equiv \exp(\Pi_i - \alpha_i - \mu)$, $a_{ii'} \equiv \exp(A_{ii'} + \alpha_i - \alpha_{i'} + \beta_i - \nu)$ and $b_{ij} \equiv= \exp(B_{ij} - \beta_i)$ satisfy the normalizing constraints Eqs. (F.3 – F.5).
- The trailing term $\alpha_{i_L} + \beta_{i_L}$ can be treated as a constant, so that they do not change the probability distribution Eq. (F.6) depending on the state of the last hidden unit in the chain. If we fix the last hidden unit state i_L to a *particular* end state, say $i_L = n$, the trailing term will be the same for all possible sequences and can thus be treated as a constant.

The problem now is to find a solution over $\mu, \nu, \{\alpha_i, \beta_i\}$ so that the normalization requirements Eqs. (F.3 – F.5) are met. For β_i we find

$$\begin{aligned} & \sum_j b_{ij} && = 1 \\ \Updownarrow & \\ & \sum_j e^{B_{ij} - \beta_i} && = 1 \\ \Updownarrow & \\ & \ln\left(e^{-\beta_i}\sum_j e^{B_{ij}}\right) && = 0 \\ \Updownarrow & \\ & \beta_i = \ln\left(\sum_j e^{B_{ij}}\right) \end{aligned} \tag{F.7}$$

From the above, we see that the conditions that $\{a_{ii'}\}$ and ν must satisfy is

$$\sum_{i'} \exp(A_{ii'} + \alpha_i - \alpha_{i'} + \beta_i - \nu) = 1 \ , \ \forall i \tag{F.8}$$

which can be rearranged into the system of equations

$$\sum_{i'} \left[\exp\left(W^{\alpha}_{ii'} + \beta_i\right)\right] \left[\exp\left(-\alpha_{i'}\right)\right] = \exp(\nu) \left[\exp\left(-\alpha_i\right)\right] \ , \ \forall i \tag{F.9}$$

These equations can be recognized as an eigenvector/eigenvalue equation for the matrix $M_{ii'} \equiv \exp\left(A_{ii'} + \beta_i\right)$, with $\lambda \equiv \exp(\nu)$ being the eigenvalue and $v_i \equiv \exp\left(-\alpha_i\right)$ being the eigenvector:

$$\mathbf{M}\,\mathbf{v} = \lambda\,\mathbf{v} \tag{F.10}$$

We note that all the elements of $\mathbf{M}$ are positive; thus, the above eigenproblem has a solution according to the Perron-Frobenius theorem [Sen73], which states that a positive matrix (a matrix in which all elements $M_{ii'}$ are positive) has a positive eigenvector with a positive eigenvalue. Therefore, a solution for $\{\alpha_i\}$ and ν exists and is straightforward to determine from the positive eigenvalue and vector.

Finally, the constant μ is determined as

$$\begin{aligned}
\sum_i \pi_i \quad &= \quad 1 \\
\Updownarrow \qquad & \\
\sum_i e^{\Pi_i - \alpha_i - \mu} \quad &= \quad 1 \\
\Updownarrow \qquad & \\
\ln\left[e^{-\mu} \sum_i e^{\Pi_i - \alpha_i}\right] \quad &= \quad 0 \\
\Updownarrow \qquad & \\
\mu = \ln\left[\sum_i e^{\Pi_i - \alpha_i}\right] &
\end{aligned} \tag{F.11}$$

From the above we see that if we have a prior over the final hidden unit state of the Boltzmann chain, we can convert any (fully connected) Boltzmann chain into an equivalent HMM which must naturally have the same prior over the final hidden state. Constraining the final hidden state to a particular state is a commonly applied constraint in the HMM literature [Mac96], especially in speech recognition applications [Rab89]. When implementing Boltzmann chains this constraint can be imposed by only including the hidden unit weights $A_{i_{L-1}n}$ leading *to* a particular final state $i_L = n$ between the hidden units at timestep $L-1$ and L (refer to Figure 12.7) and thus only sum over these weights when reducing the chain structure using clipping.

As a final note, it might seem to be of advantage if the exponents entering the distribution of the Boltzmann chain could be manipulated in a way similar to the one employed above so that the result is a prior on a particular *initial* state. Unfortunately, such manipulation does not seem possible.

Appendix G

NIPS*94 contribution

This appendix contains the paper "Recurrent Networks: Second Order Properties and Pruning," presented at the 1994 Neural Information Processing Systems conference. This paper contains a description of the problems of synchronous update of the units in each iteration of a recurrent network as well as a derivation of the second derivatives of the network output. The regularizing effects when augmenting the quadratic cost function by a weight decay term are illustrated in terms of the complexity of the cost function surface as well as the importance of the second derivative term which is neglected when employing the Gauss-Newton approximation to the Hessian. Finally, a recurrent network trained on the sunspot prediction problem is pruned using the OBS pruning scheme and the optimal network structure is illustrated.

Reference for the paper: [PH95].

Recurrent Networks: Second Order Properties and Pruning

Morten With Pedersen and **Lars Kai Hansen**
CONNECT, Electronics Institute
Technical University of Denmark B349
DK-2800 Lyngby, DENMARK
emails: with,lkhansen@ei.dtu.dk

Abstract

Second order properties of cost functions for recurrent networks are investigated. We analyze a layered fully recurrent architecture, the virtue of this architecture is that it features the conventional feedforward architecture as a special case. A detailed description of recursive computation of the *full Hessian* of the network cost function is provided. We discuss the possibility of invoking simplifying approximations of the Hessian and show how weight decays *iron* the cost function and thereby greatly assist training. We present tentative pruning results, using Hassibi et al.'s *Optimal Brain Surgeon*, demonstrating that recurrent networks can construct an efficient internal memory.

1 LEARNING IN RECURRENT NETWORKS

Time series processing is an important application area for neural networks and numerous architectures have been suggested, see e.g. (Weigend and Gershenfeld, 94). The most general structure is a fully recurrent network and it may be adapted using *Real Time Recurrent Learning* (RTRL) suggested by (Williams and Zipser, 89). By invoking a recurrent network, the *length* of the network memory can be adapted to the given time series, while it is fixed for the conventional lag-space net (Weigend et al., 90). In forecasting, however, feedforward architectures remain the most popular structures; only few applications are reported based on the Williams&Zipser approach. The main difficulties experienced using RTRL are slow convergence and lack of generalization. Analogous problems in feedforward nets are solved using second order methods for training and pruning (LeCun et al., 90; Hassibi et al., 92; Svarer et al., 93). Also, regularization by weight decay significantly improves training and generalization. In this work we initiate the investigation of second order properties for RTRL; a detailed calculation scheme for the cost function Hessian is presented, the importance of weight decay is demonstrated, and preliminary pruning results using Hassibi et al.'s Optimal Brain Surgeon (OBS) are presented. We find that the recurrent network discards the available lag space and constructs its own efficient internal memory.

1.1 REAL TIME RECURRENT LEARNING

The fully connected feedback nets studied by Williams&Zipser operate like a state machine, computing the outputs from the internal units according to a state vector $\mathbf{z}(t)$ containing *previous* external inputs and internal unit outputs. Let $\mathbf{x}(t)$ denote a vector containing the external inputs to the net at time t, and let $\mathbf{y}(t)$ denote a vector containing the outputs of the units in the net. We now arrange the indices on $\mathbf{x}$ and $\mathbf{y}$ so that the elements of $\mathbf{z}(t)$ can be defined as

$$z_k(t) = \begin{cases} x_k(t) & , \quad k \in I \\ y_k(t) & , \quad k \in U \end{cases}$$

where I denotes the set of indices for which z_k is an input, and U denotes the set of indices for which z_k is the output of a unit in the net. Thresholds are implemented using an input permanently clamped to unity. The k'th unit in the net is now updated according to

$$y_k(t+1) = f_k[s_k(t)] = f_k\left[\sum_{j\in I} w_{kj}x_j(t) + \sum_{j\in U} w_{kj}y_j(t)\right] = f_k\left[\sum_{j\in I\cup U} w_{kj}z_j(t)\right]$$

where w_{kj} denotes the weight to unit k from input/unit j and $f_k(\cdot)$ is the activation function of the k'th unit.

When used for time series prediction, the input vector (excluding threshold) is usually defined as $\mathbf{x}(t) = [x(t), \ldots, x(t-L+1)]$ where L denotes the dimension of the *lag space*. One of the units in the net is designated to be the output unit y_o, and its activating function f_o is often chosen to be linear in order to allow for arbitrary dynamical range. The prediction of $x(t+1)$ is $\hat{x}(t+1) = f_o[s_o(t)]$. Also, if the first prediction is at $t = 1$, the first example is presented at $t = 0$ and we set $\mathbf{y}(0) = \mathbf{0}$. We analyse here a modification of the standard Williams&Zipser construction that is appropriate for forecasting purposes. The studied architecture is *layered*. Firstly, we remove the external inputs from the linear output unit in order to prevent the network from getting trapped in a linear mode. The output then reads

$$\hat{x}(t+1) = y_o(t+1) = \sum_{j\in U} w_{oj}y_j(t) + w_{\text{thres},o} \tag{1}$$

Since $\mathbf{y}(0) = \mathbf{0}$ we obtain a first prediction yielding $\hat{x}(1) = w_{\text{thres},o}$ which is likely to be a poor prediction, and thereby introducing a significant error that is fed back into the network and used in future predictions. Secondly, when pruning a fully recurrent feedback net we would like the net to be able to reduce to a simple two-layer feedforward net if necessary. Note that this is *not* possible with the conventional Williams&Zipser update rule, since it doesn't include a layered feedforward net as a special case. In a layered feedforward net the output unit is disconnected from the external inputs; in this case, cf. (1) we see that $\hat{x}(t+1)$ is based on the internal 'hidden' unit outputs $y_k(t)$ which are calculated on the basis of $\mathbf{z}(t-1)$ and thereby $\mathbf{x}(t-1)$. Hence, besides the startup problems, we also get a two-step ahead predictor using the standard architecture.

In order to avoid the problems with the conventional Williams&Zipser update scheme we use a layered updating scheme inspired by traditional feedforward nets, in which we distinguish between hidden layer units and the output unit. At time t, the hidden units work from the input vector $\mathbf{z}^h(t)$

$$z_k^h(t) = \begin{cases} x_k(t-1) & , \quad k \in I \\ y_k^h(t-1) & , \quad k \in U \\ y^o(t-1) & , \quad k = O \end{cases}$$

where I denotes the input indices, U denotes the hidden layer units and O the output unit. Further, we use superscripts h and o to distinguish between hidden unit and output units. The activation of the hidden units is calculated according to

$$y_k^h(t) = f_k^h[s_k^h(t)] = f_k^h\left[\sum_{j \in I \cup U \cup O} w_{kj} z_j^h(t)\right] \quad , \quad k \in U \tag{2}$$

The hidden unit outputs are forwarded to the output unit, which then sees the input vector $\mathbf{z}_k^o(t)$

$$z_k^o(t) = \begin{cases} y_k^h(t) & , \quad k \in U \\ y^o(t-1) & , \quad k = O \end{cases}$$

and is updated according to

$$y^o(t) = f^o[s^o(t)] = f^o\left[\sum_{j \in U \cup O} w_{oj} z_j^o(t)\right] \tag{3}$$

The cost function is defined as $C = E + \mathbf{w}^T \mathbf{R} \mathbf{w}$. $\mathbf{R}$ is a regularization matrix, $\mathbf{w}$ is the concatenated set of parameters, and the sum of squared errors is

$$E = \frac{1}{2} \sum_{t=1}^{T} [e(t)]^2 \quad , \quad e(t) = x(t) - y^o(t), \tag{4}$$

where T is the size of the training set series. RTRL is based on gradient descent in the cost function, here we investigate accelerated training using Newton methods. For that we need to compute first and second derivatives of the cost function. The essential difficulty is to determine derivatives of the sum of squared errors:

$$\frac{\partial E}{\partial w_{ij}} = -\sum_{t=1}^{T} e(t) \frac{\partial y^o(t)}{\partial w_{ij}} \tag{5}$$

The derivative of the output unit is computed as

$$\frac{\partial y^o(t)}{\partial w_{ij}} = \frac{\partial f^o[s^o(t)]}{\partial s^o(t)} \cdot \frac{\partial s^o(t)}{\partial w_{ij}} \tag{6}$$

where

$$\frac{\partial s^o(t)}{\partial w_{ij}} = \delta_{oi} z_j^o(t) + \sum_{j' \in U} w_{oj'} \frac{\partial y_{j'}^h(t)}{\partial w_{ij}} + w_{oo} \frac{\partial y^o(t-1)}{\partial w_{ij}} \tag{7}$$

where δ_{jk} is the Kronecker delta. This expression contains the derivative of the hidden units

$$\frac{\partial y_k^h(t)}{\partial w_{ij}} = \frac{\partial f_k^h[s_k^h(t)]}{\partial s_k^h(t)} \cdot \frac{\partial s_k^h(t)}{\partial w_{ij}} \quad , \quad k \in U \tag{8}$$

where

$$\frac{\partial s_k^h(t)}{\partial w_{ij}} = \delta_{ki} z_j^h(t) + \sum_{j' \in U} w_{kj'} \frac{\partial y_{j'}^h(t-1)}{\partial w_{ij}} + w_{ko} \frac{\partial y^o(t-1)}{\partial w_{ij}} \tag{9}$$

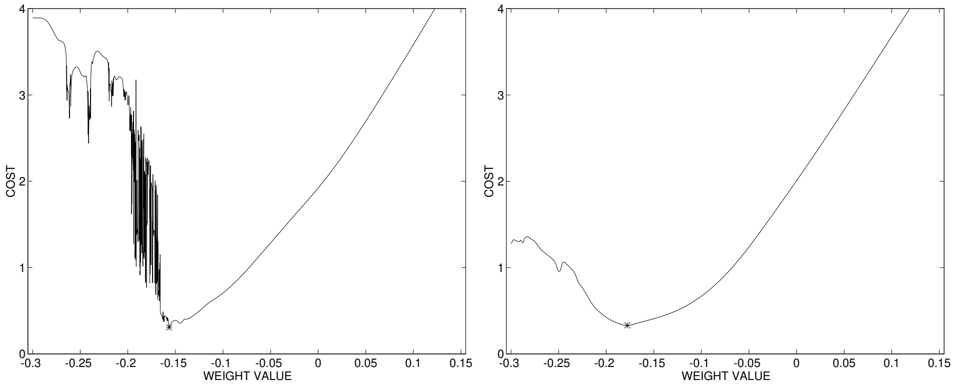


Figure 1: Cost function dependence of a weight connecting two hidden units for the sunspot benchmark series. Left panel: Cost function with small weight decay, the (local) optimum chosen is marked by an asterix. Right panel: The same slice through the cost function but here retrained with higher weight decay.

The complexity of the training problem for the recurrent net using RTRL is demonstrated in figure 1. The important role of weight decay (we have used a simple weight decay $\mathbf{R} = \alpha \mathbf{1}$) in controlling the complexity of the cost function is evident in the right panel of figure 1. The example studied is the sunspot benchmark problem (see e.g. (Weigend et al., 90) for a definition). First, we trained a network with the small weight decay and recorded the left panel result. Secondly, the network was retrained with increased weight decay and the particular weight connecting two hidden units was varied to produce the right panel result. In both cases all other weights remained fixed at their optimal values for the given weight decay. In addition to the complexity visible in these one-parameter slices of the cost function, the cost function is highly anisotropic in weight space and consequently the network Hessian is ill-conditioned. Hence, gradient descent is hampered by slow convergence.

2 SECOND ORDER PROPERTIES OF THE COST FUNCTION

To improve training by use of Newton methods and for use in OBS-pruning we compute the second derivative of the error functional:

$$\frac{\partial^2 E}{\partial w_{ij}\partial w_{pq}} = -\sum_{t=1}^{T}\left[e(t)\frac{\partial^2 y^o(t)}{\partial w_{ij}\partial w_{pq}} - \frac{\partial y^o(t)}{\partial w_{ij}}\cdot\frac{\partial y^o(t)}{\partial w_{pq}}\right] \tag{10}$$

The second derivative of the output is

$$\frac{\partial^2 y^o(t)}{\partial w_{ij}\partial w_{pq}} = \frac{\partial^2 f^o[s^o(t)]}{\partial s^o(t)^2}\cdot\frac{\partial s^o(t)}{\partial w_{ij}}\cdot\frac{\partial s^o(t)}{\partial w_{pq}} + \frac{\partial f^o[s^o(t)]}{\partial s^o(t)}\cdot\frac{\partial^2 s^o(t)}{\partial w_{ij}\partial w_{pq}} \tag{11}$$

with

$$\frac{\partial^2 s^o(t)}{\partial w_{ij}\partial w_{pq}} = \delta_{oi}\frac{\partial z_j^o(t)}{\partial w_{pq}} + \sum_{j'\in U} w_{oj'}\frac{\partial^2 y_{j'}^h(t)}{\partial w_{ij}\partial w_{pq}} + w_{oo}\frac{\partial^2 y^o(t-1)}{\partial w_{ij}\partial w_{pq}} + \delta_{op}\frac{\partial z_q^o(t)}{\partial w_{ij}} \tag{12}$$

This expression contains the second derivative of the hidden unit outputs

$$\frac{\partial^2 y_k^h(t)}{\partial w_{ij}\partial w_{pq}} = \frac{\partial^2 f_k^h[s_k^h(t)]}{\partial s_k^h(t)^2}\cdot\frac{\partial s_k^h(t)}{\partial w_{ij}}\cdot\frac{\partial s_k^h(t)}{\partial w_{pq}} + \frac{\partial f_k^h[s_k^h(t)]}{\partial s_k^h(t)}\cdot\frac{\partial^2 s_k^h(t)}{\partial w_{ij}\partial w_{pq}} \tag{13}$$

with

$$\frac{\partial^2 s_k^h(t)}{\partial w_{ij}\partial w_{pq}} = \delta_{ki}\frac{\partial z_j^h(t)}{\partial w_{pq}} + \sum_{j'\in U} w_{kj'}\frac{\partial^2 y_{j'}^h(t-1)}{\partial w_{ij}\partial w_{pq}} + w_{ko}\frac{\partial^2 y^o(t-1)}{\partial w_{ij}\partial w_{pq}} + \delta_{kp}\frac{\partial z_q^h(t)}{\partial w_{ij}} \tag{14}$$

Recursion in the five index quantity (14) imposes a significant computational burden; in fact the first term of the Hessian in (10), involving the second derivative, is often neglected for computational convenience (LeCun et al., 90). Here we start by analyzing the significance of this term during training. We train a layered architecture to predict the sunspot benchmark problem. In figure 2 the ratio between the largest eigenvalue of the second derivative term in (10) and the largest eigenvalue of the full Hessian is shown. The ratio is presented for two different magnitudes of weight decay. In line with our observations above the second order properties of the "ironed" cost function are manageable, and we can simplify the Hessian calculation by neglecting the second derivative term in (10), i.e., apply the Gauss-Newton approximation.

3 PRUNING BY THE OPTIMAL BRAIN SURGEON

Pruning of recurrent networks has been pursued by (Giles and Omlin, 94) using a heuristic pruning technique, and significant improvement in generalization for a sequence recognition problem was demonstrated. Two pruning schemes are based on systematic estimation of weight *saliency*: the Optimal Brain Damage (OBD) scheme of (LeCun et al., 90) and OBS by (Hassibi et al., 93). OBD is based on the diagonal approximation of the Hessian and is very robust for forecasting (Svarer et al., 93). If an estimate of the full Hessian is available OBS can be used

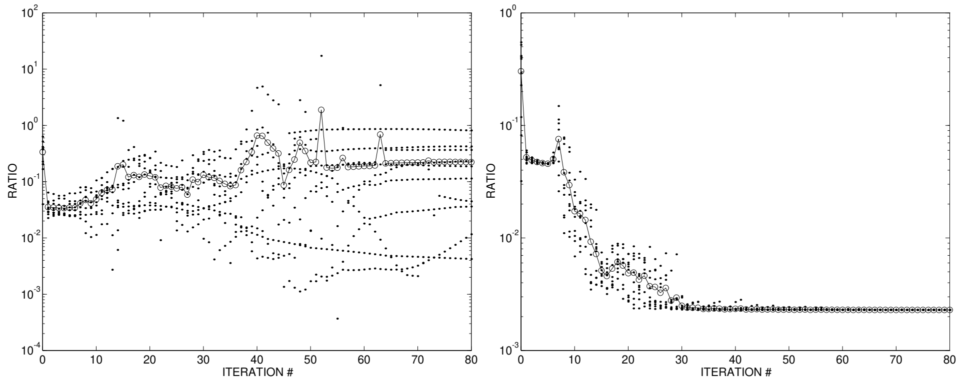


Figure 2: Ratio between the largest magnitude eigenvalue of the second derivative term of the Hessian (c.f. equation (10)) and the largest magnitude eigenvalue of the complete Hessian as they appeared during ten training sessions. The connected circles represent the average ratio. Left panel: Training with small weight decay. Right panel: Training with a high weight decay.

for estimation of saliencies incorporating *linear* retraining. In (Hansen and With Pedersen, 94) OBS was generalized to incorporate weight decays; we use these modifications in our experiments. Note that OBS in its standard form only allows for one weight to be eliminated at a time. The result of a pruning session is a nested family of networks. In order to select the optimal network within the family it was suggested in (Svarer et al., 93.) to use the estimated test error. In particular we use Akaike's Final Prediction Error (Akaike, 69) to estimate the network test error $\widehat{E}_{\text{test}} = ((T+N)/(T-N)) \cdot 2E/T$ [1], and N is the number of parameters in the network. In figure 3 we show the results of such a pruning session on the sunspot data starting from a (4-4-1) network architecture. The recurrent network was trained using a *damped* Gauss-Newton scheme. Note that the training error increases as weights are eliminated, while the test error and the estimated test error both pass through shallow minima showing that generalization is slightly improved by pruning. In fact, by retraining the optimal architecture with reduced weight decay both training and test errors are decreased in line with the observations in (Svarer et al., 93). It is interesting to observe that the network, though starting with access to a lag-space of four delay units, has lost three of the delayed inputs; hence, rely solely on its internal memory, as seen in the right panel of figure 3. To further illustrate the memory properties of the optimal network, we show in figure 4 the network response to a unit impulse. It is interesting that the response of the network extends for approximately 12 time steps corresponding to the "period" of the sunspot series.

[1] The use of Akaike's estimate is not well justified for a feedback net, test error estimates for feedback models is a topic of current research.

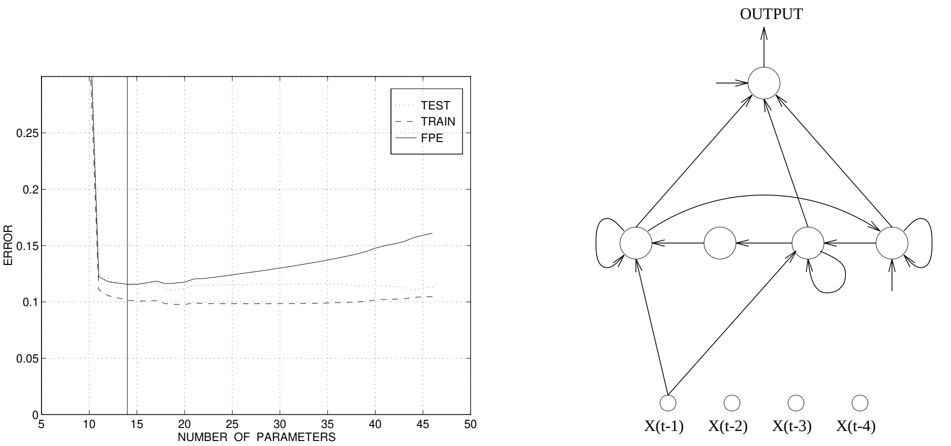


Figure 3: Left panel: OBS pruning of a (4-4-1) recurrent network trained on sunspot benchmark. Development of training error, test error, and Akaike estimated test error (FPE). Right panel: Architecture of the FPE-optimal network. Note that the network discards the available lag space and solely predicts from internal memory.

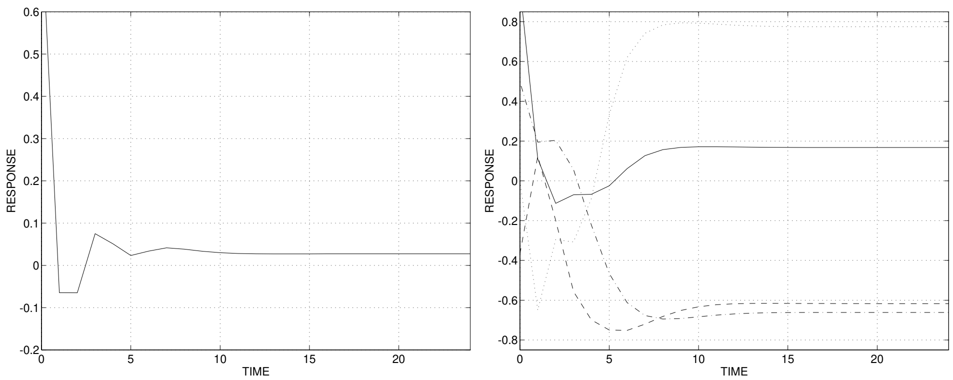


Figure 4: Left panel: Output of the pruned network after a unit impulse input at $t = 0$. The internal memory is about 12 time units long which is, in fact, roughly the period of the sunspot series. Right panel: Activity of the four hidden units in the pruned network after a unit impulse at time $t = 0$.

4 CONCLUSION

A layered recurrent architecture, which has a feedforward net as a special case, has been investigated. A scheme for recursive estimation of the Hessian of the fully recurrent neural net is devised. It's been shown that weight decay plays a decisive role when adapting recurrent networks. Further, it is shown that the second order information may be used to train and prune a recurrent network and in this process the network may discard the available lag space. The network builds an efficient internal memory extending beyond the lag space that was originally available.

Acknowledgments

We thank Jan Larsen, Sara Solla, and Claus Svarer for useful discussions, and Lee Giles for providing us with a preprint of (Giles and Omlin, 94). We thank the anonymous reviewers for valuable comments on the manuscript. This research is supported by the Danish Natural Science and Technical Research Councils through the Computational Neural Network Center (CONNECT).

References

H. Akaike: *Fitting Autoregressive Models for Prediction.* Ann. Inst. Stat. Mat. **21**, 243-247, (1969).

Y. Le Cun, J.S. Denker, and S.A. Solla: *Optimal Brain Damage.* In Advances in Neural Information Processing Systems 2, (Ed. D.S. Touretzsky) Morgan Kaufmann, 598-605, (1990).

C.L. Giles and C.W. Omlin: *Pruning of Recurrent Neural Networks for Improved Generalization Performance.* IEEE Transactions on Neural Networks, to appear. Preprint NEC Research Institute (1994).

L.K. Hansen and M. With Pedersen: *Controlled Growth of Cascade Correlation Nets*, International Conference on Artificial Neural Networks ICANN'94 Sorrento. (Eds. M. Marinaro and P.G. Morasso) Springer, 797-801, (1994).

B. Hassibi, D. G. Stork, and G. J. Wolff, *Optimal Brain Surgeon and General Network Pruning*, in Proceedings of the 1993 IEEE International Conference on Neural Networks, San Francisco (Eds. E.H. Ruspini et al.) IEEE, 293-299 (1993).

C. Svarer, L.K. Hansen, and J. Larsen: *On Design and Evaluation of Tapped Delay Line Networks*, In Proceedings of the 1993 IEEE International Conference on Neural Networks, San Francisco, (Eds. E.H. Ruspini et al.) 46-51, (1993).

A.S. Weigend, B.A. Huberman, and D.E. Rumelhart: *Predicting the future: A Connectionist Approach.* Int. J. of Neural Systems **3**, 193-209 (1990).

A.S. Weigend and N.A. Gershenfeld, Eds.: *Times Series Prediction: Forecasting the Future and Understanding the Past.* Redwood City, CA: Addison-Wesley (1994).

R.J. Williams and D. Zipser: *A Learning Algorithm for Continually Running Fully Recurrent Neural Networks*, Neural Computation **1**, 270-280, (1989).

Appendix H

NIPS*95 contribution

This appendix contains the paper "Pruning with generalization based saliencies: γOBD, γOBS," presented at the 1995 Neural Information Processing Systems conference. In this paper it is suggested to generally define the saliency of a weight in terms of estimated change in *generalization* error if the weight is pruned away, instead of the traditional change in training error. In particular, the approach is exemplified by the FPE generalization error estimate, combined with the pruning schemes OBD and OBS; the resulting pruning schemes are named γOBD and γOBS, respectively. Furthermore, attention is directed towards the problem of so-called nuisance parameters when pruning, and it is described how this problem can be handled for the OBS, γOBS pruning schemes by invoking Schur's matrix inversion lemma. γOBD and γOBS are applied to a feed-forward network trained on the Mackey-Glass time series and it is illustrated how the saliency obtained by an OBS-type pruning scheme often severely underestimates the actual change in error if a parameter is pruned away.

Reference for the paper: [PHL96].

Pruning with generalization based weight saliencies: γOBD, γOBS

Morten With Pedersen
Lars Kai Hansen
Jan Larsen
CONNECT, Electronics Institute
Technical University of Denmark B349
DK-2800 Lyngby, DENMARK
emails: with,lkhansen,jlarsen@ei.dtu.dk

Abstract

The purpose of most architecture optimization schemes is to improve generalization. In this presentation we suggest to estimate the weight saliency as the associated change in generalization error if the weight is pruned. We detail the implementation of both an $O(N)$-storage scheme extending OBD, as well as an $O(N^2)$ scheme extending OBS. We illustrate the viability of the approach on prediction of a chaotic time series.

1 BACKGROUND

Optimization of feed-forward neural networks by pruning is a well-established tool, used in many practical applications. By careful fine tuning of the network architecture we may improve generalization, decrease the amount of computation, and facilitate interpretation.

The two most widely used schemes for pruning of feed-forward nets are: Optimal Brain Damage (OBD) due to (LeCun et al., 90) and the Optimal Brain Surgeon (OBS) (Hassibi et al., 93). Both schemes are based on weight ranking according to *saliency* defined as the change in *training error* when the particular weight is pruned. In OBD the saliency is estimated as the direct change in training error, i.e., without retraining of the remaining weights, while the OBS scheme includes retraining in a local quadratic approximation. The *rationale* of both methods is that if the least significant weights (according to training error) are deleted, we gracefully relieve the danger of overfitting. However, in both cases one clearly needs a *stop criterion*. As both schemes aim at minimal generalization error an estimator for this quantity is needed. The most obvious candidate estimate is a *test error* estimated on a validation set. Validation sets, unfortunately, are notoriously very noisy (see, e.g., the discussion in Weigend et al., 1990). Hence, an attractive alternative is to estimate the test error by statistical means, e.g., Akaike's FPE (Akaike, 69). For regression type problems such a pruning stop criterion was suggested in (Svarer et al., 93).

However, why not let the saliency itself reflect the possible improvement in test error? This is the idea that we explore in this contribution.

2 GENERALIZATION IN REGULARIZED NEURAL NETWORKS

The basic asymptotic estimate of the generalization error was derived by Akaike (Akaike, 1969); the so-called Final Prediction Error (FPE). The use of FPE-theory for neural net learning has been pioneered by Moody (see e.g. (Moody, 91)), who derived estimators for the average generalization error in regularized networks.

Our network is a feed-forward architecture with n_I input units, n_H hidden sigmoid units and a single linear output unit, appropriate for scalar function approximation. The initial network is fully connected between layers and implements a non-linear mapping from input space $\mathbf{x}(k)$ to the real axis: $\widehat{y}(k) = F_{\mathbf{u}}(\mathbf{x}(k))$, where $\mathbf{u} = [\mathbf{w}, \mathbf{W}]$ is the N-dimensional weight vector and $\widehat{y}(k)$ is the prediction of the target output $y(k)$. The particular family of non-linear mappings considered can be written as:

$$F_{\mathbf{u}}(\mathbf{x}(k)) = \sum_{j=1}^{n_H} W_j \tanh\left(\sum_{i=1}^{n_I} w_{ji} x_i(k) + w_{j0}\right) + W_0, \tag{1}$$

W_j are the hidden-to-output weights while w_{ij} connect the input and hidden units.

We use the sum of squared errors to measure the network performance

$$E_{\text{train}} = \frac{1}{p}\sum_{k=1}^{p}\left[y(k) - F_{\mathbf{u}}(\mathbf{x}(k))\right]^2, \tag{2}$$

where p is the number of training examples. To ensure numerical stability and to assist the pruning procedure we augment the cost function with a regularization term.[1] The resulting cost function reads

$$E = E_{\text{train}} + \frac{1}{2}\mathbf{u}^T\mathbf{R}\mathbf{u} \tag{3}$$

The main source of uncertainty in learning is the shortage of training data. Fitting the network from a finite set of noisy examples means that the noise in these particular examples will be fitted as well and when presented with a new test example the network will make an error which is larger than the error of the "optimal network" trained on an infinite training set. By careful control of the fitting capabilities, e.g., by pruning, such overfitting may be reduced.

The generalization error is defined as the average squared error on an example from the example distribution function $P(\mathbf{x}, y)$. The examples are modeled by a *teacher* network with weights $\mathbf{u}^*$, degraded by additive noise: $y(k) = F_{\mathbf{u}^*}(\mathbf{x}(k)) + \nu(k)$. The noise samples $\nu(k)$ are independent identically distributed variables with finite, but unknown variance σ^2. Further, we assume that the noise terms are independent of the corresponding inputs. The quantity of interest for model optimization is the training set average of the generalization error, viz., the average over an ensemble

[1] $\mathbf{R}$ will be a positive definite diagonal matrix.

of networks in which each network is provided with its individual training set. This averaged generalization error is estimated by

$$\widehat{E}_{\text{test}} = \left(1 + \frac{N_{\text{eff}}}{p}\right) \sigma^2 + O\left((1/p)^2\right), \tag{4}$$

with the effective number of parameters being $N_{\text{eff}} = \text{tr}(\mathbf{HJ}^{-1}\mathbf{HJ}^{-1})$ (Larsen and Hansen, 94). The Hessian, $\mathbf{H}$, is the second derivative matrix of the training error with respect to the weights and thresholds, while $\mathbf{J}$ is the regularized Hessian: $\mathbf{J} = \mathbf{H} + \mathbf{R}$. An asymptotically unbiased estimator of the noise level is provided by: $\sigma^2 = E_{\text{train}}/(1 - N_{\text{eff}}/p)$. Inserting, we get

$$\widehat{E}_{\text{test}} = \frac{p + N_{\text{eff}}}{p - N_{\text{eff}}} E_{\text{train}} \approx \left(1 + \frac{2N_{\text{eff}}}{p}\right) E_{\text{train}}. \tag{5}$$

While OBD and OBS are based on estimates of the change in E_{train} we see that in order to obtain saliencies that estimate the change in generalization we must generally take the prefactor into account. We note that if the network is *not* regularized $N_{\text{eff}} = \text{tr}(\mathbf{HJ}^{-1}\mathbf{HJ}^{-1}) = \text{tr}(\mathbf{1}) = N$, in which case the prefactor is only a function of the total number of weights. In this case ranking according to training error saliency is equivalent to ranking according to generalization error.

However, in the generic case of a regularized network this is no more true ($N_{\text{eff}} < N$), and we need to evaluate the change in the prefactor, i.e., in the effective number of parameters, associated with pruning a weight. Denoting the generalization based saliency of weight u_l as $E_{\text{test},l}$, we find

$$\delta E_{\text{test},l} \approx \delta E_{\text{train},l} - \frac{2\left(N_{\text{eff}} - N_{\text{eff},l}\right)}{p} E_{\text{train}} \tag{6}$$

Where the number of parameters after pruning of weight l is $N_{\text{eff},l}$, and $\delta E_{\text{train},l}$ is the training error based saliency.

To proceed we outline two implementations, the major difference being the computational complexity involved. In the first, which is an elaboration on the OBD scheme, the storage complexity is proportional to the number of weights and thresholds (N), while in the second scheme the complexity scales with N^2, and is a generalization of the OBS. To emphasize that we use the generalization error for ranking of weights we use the prefix γ: γOBD and γOBS.

3 γOBD: AN $O(N)$ IMPLEMENTATION

Our $O(N)$ simulator is based on *batch mode*, second order pseudo-Gauss Newton optimization which is described in (Svarer et al., 93). The scheme, being based on the diagonal approximation for the Hessian, requires storage of a number of variables scaling linearly with the number of parameters N. As in (Le Cun et al., 90) we approximate the second derivative matrix by the positive semi-definite expression:

$$\frac{\partial^2 E_{\text{train}}}{\partial u_j^2} \approx \frac{2}{p} \sum_{k=1}^{p} \left(\frac{\partial F_{\mathbf{u}}(\mathbf{x}(k))}{\partial u_j}\right)^2. \tag{7}$$

In the diagonal approximation we find

$$N_{\text{eff}} = \sum_{j=1}^{N} \left(\frac{\lambda_j}{\lambda_j + \alpha_j/p}\right)^2, \tag{8}$$

where $\lambda_j \equiv \partial^2 E_{\text{train}} / \partial u_j^2$. Further, α_j/p are the weight decay parameters (diagonal elements of the regularization matrix $\mathbf{R}$).

The OBD method proposed by (Le Cun et al., 90) was successfully applied to reduce large networks for recognition of handwritten digits. The basic idea is to estimate the increase in the *training error* when deleting weights. Expanding the training error to second order in the pruned weight magnitude it is found that

$$\delta E_{\text{train},l} = \left(\frac{\alpha_l}{p} + \frac{1}{2} \frac{\partial^2 E_{\text{train}}}{\partial u_l^2} \right) u_l^2. \tag{9}$$

This estimate takes into account that the weight decay terms force the weights to depart from the minimum of the training set error. The first derivative of the training error is non-zero, hence, the first term in (9). Computationally, we note that the diagonal Hessian terms are reused from the pseudo Gauss-Newton training scheme.

Using (6) and the diagonal form of N_{eff}, we find the following approximative expression for generalization saliency (γOBD):

$$\delta E_{\text{test},l} \approx \delta E_{\text{train},l} - \frac{2}{p} \left(\frac{\lambda_l}{\lambda_l + \alpha_l/p} \right)^2 E_{\text{train}} \tag{10}$$

From this expression we learn that of two weights inducing similar changes in training error we should delete the one which has the largest ratio of training error curvature (λ) to weight decay, i.e., the weight which has been *least* influenced by weight decay. However, from a computational point of view we also want to reduce the number of parameters as far as possible; so we might in fact accept to delete weights with small positive generalization saliency (in particular considering the amount of approximation involved in the estimates).

4 γOBS: AN O(N^2) IMPLEMENTATION

In the Optimal Brain Surgeon (Hassibi et al., 92) the increase in training error is estimated including the effects of quadratic retraining. This allows for pruning of more general degrees of freedom, e.g., situations where the training error induces linear constraints among two or more weights. The price to be paid is that we need to operate with the full $N \times N$ Hessian matrix of second derivatives. The O(N^2) simulator, hence, is based on full Gauss Newton optimization. When eliminating the l'th weight retraining is determined by

$$\delta \mathbf{u}_l = -\frac{u_l}{(\mathbf{J}^{-1})_{ll}} \mathbf{J}^{-1} \mathbf{e}_l \tag{11}$$

where $\mathbf{e}_l$ is the l'th unit vector. We need to modify the OBS saliencies when working from a weight decay regularized cost function. The modified saliencies were given in (Hansen and With, 94)[2]

$$\delta E_{train,l} = \frac{1}{2} \frac{u_l^2}{(\mathbf{J}^{-1})_{ll}} + \frac{\alpha}{p} \left(\frac{u_l (\mathbf{e}_l^{\mathrm{T}} \mathbf{J}^{-1} \mathbf{u})}{(\mathbf{J}^{-1})_{ll}} - \frac{1}{2} \frac{u_l^2 (\mathbf{J}^{-2})_{ll}}{((\mathbf{J}^{-1})_{ll})^2} \right) \tag{12}$$

Whether using the generalization based γOBS or standard OBS, we want to point to an important aspect of OBS that seems not to be generally appreciated, namely the

[2]The expression is for the case of all weight decays being equal, see (Hansen and With, 94) for the general expression.

problem of "nuisance" parameters (White, 89), (Larsen, 93). When eliminating an output weight u_o, all the weights to the corresponding hidden unit are in effect also pruned away. Such a situation is well-known in the statistics literature on model selection where such "ghost" input weights are known as nuisance parameters. It is important to remove these parameters from the network function before estimating the saliency $\delta E_{train,o}$ and the resulting effective number of parameters N_{eff}, as they would otherwise give "spurious" contributions to these estimates. Applying OBS without taking this fact into consideration often results in sudden jumps in the level of the network error due to pruning of an important weight based on a corrupted saliency estimate. Removing the superfluous weights from the weight vector $\mathbf{u}$ and the corresponding rows and columns in $\mathbf{J}$ to form the reduced (regularized) Hessian $\mathbf{J}_1$ is straightforward, but it is computationally expensive to invert each of the resulting (sub-)matrices $\mathbf{J}_1$ for use in (11) and (12). This cost can be considerably reduced by rearranging the rows and columns of $\mathbf{J}$ as

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_1 & \mathbf{J}_2 \\ \mathbf{J}_3 & \mathbf{J}_4 \end{bmatrix} \quad \rightarrow \quad \mathbf{J}^{-1} = \begin{bmatrix} (\mathbf{J}^{-1})_1 & (\mathbf{J}^{-1})_2 \\ (\mathbf{J}^{-1})_3 & (\mathbf{J}^{-1})_4 \end{bmatrix} \tag{13}$$

where $\mathbf{J}_2$, $\mathbf{J}_3$ and $\mathbf{J}_4$ are the rows and columns corresponding to the nuisance parameters. Using a standard lemma for partitioned matrices, we obtain

$$(\mathbf{J}_1)^{-1} = (\mathbf{J}^{-1})_1 - (\mathbf{J}^{-1})_2[(\mathbf{J}^{-1})_4]^{-1}(\mathbf{J}^{-1})_3 \tag{14}$$

which only calls for inversion of the (small) submatrix $(\mathbf{J}^{-1})_4$. In (Hassibi et al., 93) it was argued that one might save on computation by using an iterative scheme for calculation of the inverse Hessian $\mathbf{J}^{-1}$. However, since standard matrix inversion is an $\mathrm{O}(N^3)$ operation while the iterative scheme scales as $\mathrm{O}(pN^2)$, a detailed count shows that that it is only beneficial to use the iterative scheme in the atypical case $N > p/2$.

5 EXPERIMENT

We will illustrate the viability of the proposed methods on a standard problem of nonlinear dynamics viz. the Mackey-Glass chaotic time series. The series is generated by integration of the differential equation

$$\frac{dz(t)}{dt} = -bz(t) + a\frac{z(t-\tau)}{1+z(t-\tau)^{10}} \tag{15}$$

where the constants are $a = 0.2$, $b = 0.1$ and $\tau = 17$. The series is resampled with sampling period 1 according to standard practice. The network configuration is $n_I = 6$, $n_H = 10$ and we train to implement a six step ahead prediction. That is, $\mathbf{x}(k) = [z(k-6), z(k-12), \cdots, z(k-6n_I)]$ and $y(k) = z(k)$. In Fig. 1 we show pruning scenarios based on the two different implementations. The training errors, test errors and FPE errors are plotted for a training set size of 250 examples, the test set comprises 8500 examples. In the left panel we show the results of pruning according to γOBD and similarly in the right panel we show the results of pruning as it occurred using γOBS. In this example we do not find significant improvement in performance by use of γOBS.

To illustrate the ability of the estimators for predicting the effects of pruning on the test error we plot in figure 2 the estimated test errors versus the actual test errors after pruning. In the OBD case this means the test error resulting from pruning the parameters without retraining, while in the OBS case it means the test error following pruning and retraining in the quadratic approximation. We note that the γOBD estimates of the test error approximately equal the actual

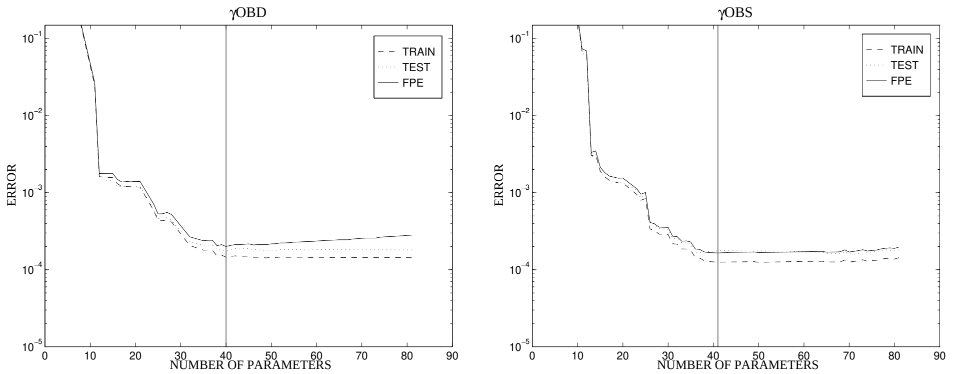


Figure 1: The evolution of training and test errors during pruning for the Mackey-Glass time series for a training set of size 250. In the left panel is shown pruning by γOBD, while in the right we show pruning by γOBS. The vertical solid line indicates the network for which the *estimated* test error is minimal.

test error, offset by a constant corresponding to the FPE-offset in the left panel of figure 1. The most important feature of this plot is that ranking according to the estimated test error is consistent with ranking according to the actual test error. In the right panel of figure 2, however, we see that γOBS highly underestimates the actual errors resulting from the quadratic retraining. It is not clear how the ranking inconsistencies affect the overall performance of γOBS. The weight selected for pruning (indicated by a circle) is clearly not the optimal according to the actual test error. However, as depicted in the figure, after full Gauss-Newton retraining for 20 epochs the measured actual test error is comparable to the estimated value (retraining is indicated by the arrow). Hence, one may say that γOBS “recovers” after retraining, while the initial estimate based on quadratic retraining is rather poor.

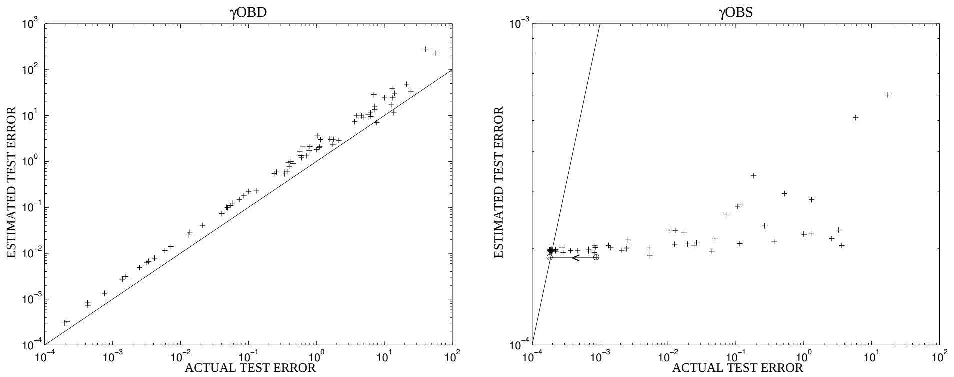


Figure 2: Left panel: Estimated test errors for fully connected network using γOBD and the actual test errors computed by actual deletion of the weight and computing the test error on the 8500 members test set. Right panel: Errors for fully connected network using γOBS. The weight selected for pruning is indicated by a circle, the result of further retraining is indicated by an arrow.

6 CONCLUSION

Since a main objective of pruning algorithms is to improve generalization we suggest that weight saliencies are estimated from the test error rather than the training error. We have shown how this might be carried out for scalar function approximation, in which case we have a rather simple test error estimate (based on Akaike's FPE). We provided implementation details for a scheme of linear complexity, γOBD, which is the generalization of OBD and a scheme of quadratic complexity γOBS which is the generalization of OBS. Furthermore, we provided a way to significantly reduce the computational overhead involved in the handling of nuisance parameters. An application within time series prediction showed the viability of the suggested approach.

Acknowledgements

We thank Peter Magnus Nørgaard for valuable discussions. This research is supported by the Danish Natural Science and Technical Research Councils through the Computational Neural Network Center (CONNECT). JL acknowledge the Radioparts Foundation for financial support.

References

H. Akaike: *Fitting Autoregressive Models for Prediction.* Ann. Inst. Stat. Mat. **21**, 243–247, (1969).

Y. Le Cun, J.S. Denker, and S.A. Solla: *Optimal Brain Damage.* In Advances in Neural Information Processing Systems 2, Morgan Kaufman, 598–605, (1990).

L.K. Hansen and M. With Petersen: *Controlled Growth of Cascade Correlation Nets.* Proceedings of ICANN'94 International Conference on Neural Networks, Sorrento, Italy, 1994. Eds. M. Marinaro and P.G. Morasso, 797–800, (1994).

B. Hassibi, D. G. Stork, and G. J. Wolff: *Optimal Brain Surgeon and General Network Pruning*, in Proceedings of the 1993 IEEE International Conference on Neural Networks, San Francisco (Eds. E.H. Ruspini et al.) 293–299, (1993).

J. Larsen: *Design of Neural Network Filters.* Ph.D. Thesis, Electronics Institute, Technical University of Denmark, (1993).

J. Larsen and L.K. Hansen: *Generalization Performance of Regularized Neural Network Models.* "Neural Networks for Signal Processing IV" Proceedings of the IEEE Workshop, Eds. J. Vlontzos et al., IEEE Service Center, Piscataway NJ, 42–51, (1994).

J.E. Moody: *Note on Generalization, Regularization and Architecture Selection in Nonlinear Systems.* In Neural Networks For Signal Processing; Proceedings of the 1991 IEEE-SP Workshop, (Eds. B.H. Juang, S.Y. Kung, and C. Kamm), IEEE Service Center, 1–10, (1991).

C. Svarer, L.K. Hansen, and J. Larsen: *On Design and Evaluation of Tapped Delay Line Networks*, In Proceedings of the 1993 IEEE International Conference on Neural Networks, San Francisco, (Eds. E.H. Ruspini et al.) 46–51, (1993).

A.S. Weigend, B.A. Huberman, and D.E. Rumelhart: *Prediction the future: A Connectionist Approach.* Int. J. of Neural Systems **3**, 193-209, (1990).

H. White: *Learning in Artificial Neural Networks: A Statistical Perspective.* Neural Computation **1**, 425–464, (1989).

Appendix I

ICFMHB'96 contribution

This appendix contains the abstract "A Concordance Correlation Coefficient for Reproducibility of Spatial Activation Patterns" presented at Second International Conference on Functional Mapping of the Human Brain. Various data analysis methods were applied to a set of fMRI time series from a 16×16 voxel area of the brain of a person performing a simple finger-tapping motor task, in order to locate the activated areas in the brain. The application of artificial neural networks for this task was based on the pragmatic consideration that the lower the prediction error on a *test* set for a particular voxel, the less likely it would be that the signal for the voxel was only "background noise", thus the more likely it would be that the area in the brain corresponding to the particular voxel was activated. This approach appeared to correlate rather well with other data analysis methods applied.

Reference for the paper: [LHP$^+$96].

A Concordance Correlation Coefficient for Reproducibility of Spatial Activation Patterns

N. Lange[1,2], L. K. Hansen[3], M. W. Pedersen[3], R. L. Savoy[4], S. C. Strother[5]

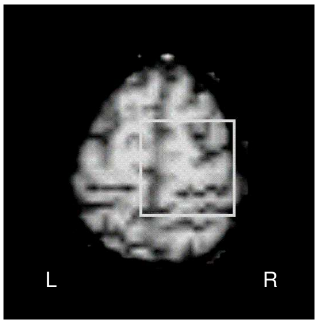
[1]National Institutes of Health, Bethesda, MD, USA; [2]Brain Imaging Center, McLean Hospital, Belmont, MA, USA; [3]Technical University of Denmark, Lyngby, Denmark; [4]MGH-NMR Center, Charlestown, MA, USA; [5]University of Minnesota and VA Medical Center, Minneapolis, MN, USA

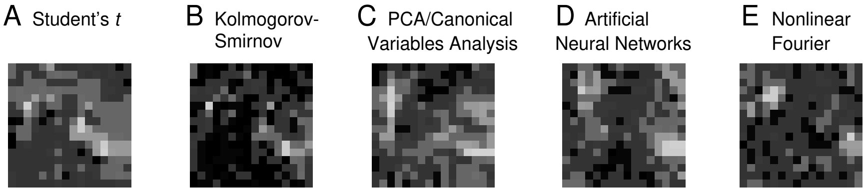
Objective. We develop and apply a quantitative measure of image reproducibility for the comparison of spatial activation patterns derived from fMRI time series or PET datasets. An fMRI experiment involving a simple motor task and five different statistical models demonstrates the measure's utility.

Method. For spatial activation patterns I_1 and I_2, the concordance correlation coefficient for reproducibility (1) is defined as:

$$\rho_c = 1 - \frac{E\left[(I_1 - I_2)^2\right]}{\sigma_1^2 + \sigma_2^2 + (\mu_1 - \mu_2)^2} = \frac{2\rho\sigma_1\sigma_2}{\sigma_1^2 + \sigma_2^2 + (\mu_1 - \mu_2)^2}. \quad [1]$$

In [1], images I_1, I_2 have means μ_1, μ_2, standard deviations σ_1, σ_2 and Pearson product-moment correlation coefficient ρ. The paired *t*-test, intraclass correlation coefficient, coefficient of variation and simple least-squares analyses can all fail in a variety of actual situations (1). The statistic ρ_c provides a more direct measure of agreement using the line of identity as the point of departure and equals ρ if and only if $\mu_1 = \mu_2$ and $\sigma_1 = \sigma_2$, so that $-1 \leq -|\rho| \leq \rho_c \leq |\rho| \leq 1$. We have investigated ρ_c for five voxel-by-voxel data analytic strategies of varying complexity applied to fMRI time series collected from a healthy female subject (see Figure caption): (A) Student's *t*; (B) Kolmogorov-Smirnov statistic [2]; (C) principal components/canonical variables [3,4]; (D) an artificial neural network [5]; (E) a nonlinear Fourier method that allows spatially varying shapes and scales for estimated hemodynamic response functions modeled as gamma densities [6].





	A	B	C	D	E
A		.48	.45	.34	.22
B	.72		.28	.33	.20
C	.48	.47		.26	.19
D	.35	.44	.28		.43
E	.25	.23	.24	.46	

Figure 1. fMRI activation images of a single subject performing a left-handed finger-to-thumb opposition task. Multiple runs of 72 2.5-second whole-brain echo planar images were aligned, and a 16 x 16 voxel slab (3.1 x 3.1 x 8 mm voxels) of contralateral sensorimotor cortex (white box) was extracted; images A-E were obtained by five different data analytic strategies. Voxel intensity is displayed using a linear grayscale. The 5 x 5 correlation matrix (lower left: Pearson product-moment correlation; upper right: concordance correlation coefficient for reproducibility) provides quantitative measures of spatial activation pattern similarity.

Results. Figure 1 indicates that all models appear to reproduce coarse features of activated foci, yet images B to E become progressively dissimilar from A (see correlation matrix). A permutation test of A vs. B indicates that these two spatial activation patterns are not significantly different ($p > 0.05$) and that, under this model pair, the spatial activation pattern is reproducible. However, differences in model complexity appear to alter the local structure of activated foci. Multivariate and nonlinear models (C-E) may capture spatio-temporal effects in fMRI time series that are averaged out or otherwise obscured by simpler models (A, B).

Conclusion. Superficial similarity between fMRI summary images can belie complex spatio-temporal behaviors of underlying signal and noise. Modern, locally-adaptive statistical techniques attempt to capture these behaviors and reflect their important effects. The concordance correlation coefficient for reproducibility has advantages over traditional similarity measures and can be used to compare spatial activation patterns in functional neuroimages derived from different data analytic strategies. The concordance measure may also be useful in other contexts, such as in comparisons of replications across time for single subjects, between multiple subjects, groups of subjects and functional neuroimaging modalities.

Acknowledgment. Funded in part by Human Brain Project grant R01 DA09246.

References

1. Lin LIK. Biometrics 1989, 45: 255-268.
2. Press WH et al. Numerical Recipes. Cambridge University Press, 1989.
3. Strother SC et al. J. Cereb. Blood Flow Metab. 1995, 15: 738-775.
4. Mardia KV, Kent JT, Bibby JM. Multivariate Analysis. London: Academic Press, 1979.
5. Svarer C, Hansen LK, Larsen J. Proc IEEE Int Conf Neural Networks 1993, 46-51.
6. Lange N, Zeger SL. Submitted, J. Roy. Stat. Soc. 1996.

Appendix J

NNSP '96 contribution

This appendix contains the paper "Design and Evaluation of Neural Classifiers," presented at the IEEE 1996 workshop on Neural Networks for Signal Processing. The focus of this paper is feedforward networks applied to classification by transforming the network outputs into probabilities by the "SoftMax" transformation and training by minimizing the entropic cost function augmented by a regularization term. A Gauss-Newton like approximation to the Hessian is derived and used for second-order training as well as pruning by the OBD scheme; generalization ability is estimated using an analytical test error estimate. The resulting algorithmic framework is applied to the contiguity problem as well as to the "glass" classification problem from the PROBEN1 benchmark problem collection.

Reference for the paper: [HMPHL96].

DESIGN AND EVALUATION OF NEURAL CLASSIFIERS

Mads Hintz-Madsen, Morten With Pedersen,
Lars Kai Hansen, and Jan Larsen
CONNECT, Dept. of Mathematical Modeling, B. 349
Technical University of Denmark
DK-2800 Lyngby, Denmark

Phone: (+45) 4525 3885, Fax: (+45) 4588 0117
Email: hintz, with, lkhansen, jlarsen@ei.dtu.dk

Abstract - In this paper we propose a method for design of feed-forward neural classifiers based on regularization and adaptive architectures. Using a penalized maximum likelihood scheme we derive a modified form of the entropic error measure and an algebraic estimate of the test error. In conjunction with Optimal Brain Damage pruning the test error estimate is used to optimize the network architecture. The scheme is evaluated on an artificial and a real world problem.

INTRODUCTION

Pattern recognition is an important aspect of most scientific fields and indeed the objective of most neural network applications. Some of the by now classic applications of neural networks like Sejnowski and Rosenbergs "NetTalk" concern classification of patterns into a finite number of categories. In modern approaches to pattern recognition the objective is to produce class probabilities for a given pattern. Using Bayes decision theory, the "hard" classifier selects the class with the highest class probability, hence minimizing the probability of error. The conventional approach to pattern recognition is statistical and concerns the modeling of class probability distributions for patterns produced by a stationary stochastic source by a certain set of basis functions, e.g., Parzen windows or Gaussian mixtures.

In this paper we define and analyze a system for design and evaluation of feed-forward neural classifiers based on regularization and adaptive architectures. The proposed scheme is a generalization of the approach we have suggested for time series processing [1, 2] and for binary classification in the context of a medical application [3]. The key concept of the new methodology for optimization of neural classifiers is an asymptotic estimate of the test error of the classifier providing an algebraic expression in terms of the training error and a complexity estimate. Our approach is a penalized maximum likelihood scheme. The likelihood is formulated using a simple stationary noisy channel model of the pattern source. For any fixed input pattern there can be defined a fixed probability distribution over a fixed finite set of classes. The training set involves simple labelled data, i.e., for each input vector we are provided with a single class label. The task of the network is to estimate the relative frequencies of class labels for a given pattern. In conjunction with SoftMax normalization of the outputs of a standard, computationally universal, feed-forward network we recover a slightly modified form of the so-called entropic error measure [4]. For a fixed architecture the neural network weights are estimated using a Gauss-Newton method [5], while the model architecture is optimized using Optimal Brain Damage [6]. The problem of proper selection of regularization parameters is also briefly discussed, see also [7].

The salient features of the approach are: Efficient Newton optimization, pruning by Optimal Brain Damage and evaluation of network architectures by an algebraic test error estimate.

NEURAL CLASSIFIERS

Let us assume that we have a training set, D, consisting of q input-output pairs

$$D = \{(\mathbf{x}^\mu, y^\mu) | \mu = 1, ..., q\} \tag{1}$$

where $\mathbf{x}$ is an input vector consisting of n_I elements and y is the corresponding class label. In this presentation we will assume that the class label is of the definite form $y = 1, ..., n_O$, with n_O being the number of classes. An alternative soft target assignment might be relevant in some practical contexts where the target could be, e.g., an estimate of class probabilities for the given input.

We aim to model the posterior probability distribution

$$p(y = i|\mathbf{x}), \qquad i = 1, \ldots, n_O. \tag{2}$$

In some applications it might be desirable to use a rejection threshold when classifying, that is if all of the posterior probabilities fall below this threshold then no classification decision is made, see e.g., [3].

To represent these distributions we choose the following network architecture:

$$h_j(\mathbf{x}^\mu) \quad = \quad \tanh\left(\sum_{k=1}^{n_I} w_{jk} x_k^\mu + w_{j0}\right) \tag{3}$$

$$\phi_i(\mathbf{x}^\mu) \quad = \quad \sum_{j=1}^{n_H} W_{ij} h_j(\mathbf{x}^\mu) + W_{i0} \tag{4}$$

with n_I input units, n_H hidden units, n_O output units, and parameters $\mathbf{u} = (\mathbf{w}, \mathbf{W})$, where w_{j0} and W_{i0} are thresholds. To ensure that the outputs, $\phi_i(\mathbf{x}^\mu)$, can be interpreted as probabilities, we use the normalized exponential transformation known as SoftMax [4]:

$$\hat{p}(y^\mu = i|\mathbf{x}^\mu) \equiv \frac{\exp(\phi_i(\mathbf{x}^\mu))}{\sum_{i'=1}^{n_O} \exp(\phi_{i'}(\mathbf{x}^\mu))} \tag{5}$$

where $\hat{p}(y^\mu = i|\mathbf{x}^\mu)$ is the estimated probability, that $\mathbf{x}^\mu$ belongs to class i.

Assuming that the training data are drawn independently, the likelihood of the model can be expressed as

$$P(D|\mathbf{u}) = \prod_{\mu=1}^{q} \prod_{i=1}^{n_O} \hat{p}(y^\mu = i|\mathbf{x}^\mu)^{\delta_{i,y^\mu}} \tag{6}$$

where $\delta_{i,y^\mu} = 1$ if $i = y^\mu$, otherwise $\delta_{i,y^\mu} = 0$.

Training is based on the minimization of the negative log-likelihood

$$E(\mathbf{u}) = -\frac{1}{q} \log P(D|\mathbf{u}) = \frac{1}{q} \sum_{\mu=1}^{q} \epsilon(\mathbf{x}^\mu, y^\mu, \mathbf{u}) \tag{7}$$

where

$$\epsilon(\mathbf{x}^\mu, y^\mu, \mathbf{u}) = -\sum_{i=1}^{n_O} \delta_{i,y^\mu} \left[\phi_i(\mathbf{x}^\mu) - \log \left(\sum_{i'=1}^{n_O} \exp(\phi_{i'}(\mathbf{x}^\mu)) \right) \right]. \tag{8}$$

In order to eliminate overfitting and ensure numerical stability, we augment the cost function by a regularization term, e.g., a simple weight decay,

$$C(\mathbf{u}) = E(\mathbf{u}) + \frac{1}{2} \mathbf{u}^{\mathrm{T}} \mathbf{R} \mathbf{u} \tag{9}$$

where $\mathbf{R}$ is a positive definite matrix. In this paper we consider a diagonal matrix with elements $2\alpha_j/q$.

The gradient of (7) is

$$\frac{\partial E(\mathbf{u})}{\partial u_j} = -\frac{1}{q} \sum_{\mu=1}^{q} \sum_{i=1}^{n_O} \left[\delta_{i,y^\mu} - \hat{p}(y^\mu = i|\mathbf{x}^\mu) \right] \frac{\partial \phi_i(\mathbf{x}^\mu)}{\partial u_j}. \tag{10}$$

The matrix of second derivatives (the Hessian) can be expressed as

$$H_{jk} \equiv \frac{\partial^2 E(\mathbf{u})}{\partial u_j \partial u_k} = \frac{1}{q} \sum_{\mu=1}^{q} \sum_{i=1}^{n_O} \sum_{i'=1}^{n_O} \hat{p}(y^\mu = i|\mathbf{x}^\mu) \left[\delta_{i,i'} - \hat{p}(y^\mu = i'|\mathbf{x}^\mu) \right] \frac{\partial \phi_{i'}(\mathbf{x}^\mu)}{\partial u_k} \frac{\partial \phi_i(\mathbf{x}^\mu)}{\partial u_j} \tag{11}$$

where we have used a Gauss-Newton like approximation.

Using matrix/vector notation the Gauss-Newton paradigm of updating the weights can now be computed as [5]

$$\mathbf{u}^{\text{new}} = \mathbf{u} - \eta \left(\mathbf{H} + \mathbf{R}\right)^{-1} \left[\frac{\partial E}{\partial \mathbf{u}} + \mathbf{R}\mathbf{u}\right] \tag{12}$$

where $\mathbf{Ru}$ and $\mathbf{R}$ are the first and second derivatives of the regularization term, respectively, and η is a parameter, that may be used to ensure a decrease in the cost function, e.g., by line search.

A natural approach for determining the regularization parameters is by minimizing the test error with respect to the regularization parameters. Here one may use an estimate of the test error as derived in the next section. Let us now consider the case with only two different weight decays: α_w for the input-to-hidden weights and α_W for the hidden-to-output weights. By sampling the space spanned by α_w and α_W with e.g., a 3x3 grid and computing the estimated test error, it is possible to fit e.g., a paraboloid[1] in a least-square sense to the sample points, locate the minimum[2] of the paraboloid and use the weight decays found for the design of the network.

A different approach using a validation set for determining the regularization parameters is described in [7].

Test Error Estimate

One of the main objectives in our approach is to estimate a network model with a high generalization ability. In order to obtain this we need an estimate of the generalization ability of a model. The generalization or test error for a given network $\mathbf{u}$ may be defined as

$$E_{\text{test}}(\mathbf{u}) = \int d\mathbf{x}dy P(\mathbf{x}, y)\epsilon(\mathbf{x}, y, \mathbf{u}) \tag{13}$$

where $P(\mathbf{x}, y)$ is the true underlying distribution of examples and $\epsilon(\mathbf{x}, y, \mathbf{u})$ is the error on example $(\mathbf{x}, y)$. Since the test error involves an average over all possible examples, it is in general not accessible, but it can be estimated by using additional statistical assumptions, see e.g., [3] and [8], thus giving us the following estimate for the average test error of a network $\mathbf{u}$ estimated on a training set D [8],

$$\langle \widehat{E_{\text{test}}} \rangle = E_{\text{train}}(\mathbf{u}(D)) + \frac{N_{\text{eff}}}{q} \tag{14}$$

[1]Paraboloid: $(z - z_0) = (x - x_0)^2/a^2 + (y - y_0)^2/b^2$.

[2]In case the minimum is located outside the sample-grid, one should relocate the grid and find a new minimum.

where $E_{\mathrm{train}}(\mathbf{u}(D))$ is the training error of the model. The effective number of parameters is given by $N_{\mathrm{eff}} = \mathrm{Tr}[\mathbf{H}(\mathbf{H}+\mathbf{R})^{-1}]$, where $\mathbf{R}$ is the second derivative of the regularization term. This estimate of the test error averaged over all possible training sets may be used to select the optimal network e.g., among a nested family of pruned networks; hence, be used as a pruning stop criterion similarly to our procedure for evaluation of function approximation networks [1, 2].

Pruning with Optimal Brain Damage

In order to reduce and optimize a networks architecture, we recommend to apply a pruning scheme such as *Optimal Brain Damage* (OBD) [6]. The aim of OBD is to estimate the importance of the weights for the training error and rank the weights according to their importance. If the importance is estimated using a second order expansion of the training error around its minimum, the *saliency* for a weight u_i is [1]

$$s_i = \left(R_{ii} + \frac{1}{2}H_{ii}\right)u_i^2 \tag{15}$$

where the Hessian H_{ii} is given by (11) and R_{ii} is i'th diagonal element of $\mathbf{R}$.

By repeatedly removing weights with the smallest saliencies and retraining the resulting network, a nested family of networks is obtained. Here we may use the previously derived test error estimate to select the "optimal" network.

EXPERIMENTS

The proposed methodology for designing neural classifiers has been evaluated on the artificial *contiguity problem* and the real world *glass classification problem.* The latter is a part of the Proben1 neural network benchmark collection [9].

The Contiguity Problem

The contiguity problem has in several cases been used for evaluating optimization schemes, see e.g., [10]. The boolean input vector (± 1) is interpreted as a one-dimensional image and connected clumps of $+1$'s are counted. Two classes are defined: those with two and three clumps. We consider the case, where $n_I = 10$. In this case there are 792 legal input patterns consisting of 432 patterns with three clumps and 360 with two clumps. We use a randomly selected training set with 150 patterns and a test set with 510 patterns both containing an even split of the two classes.

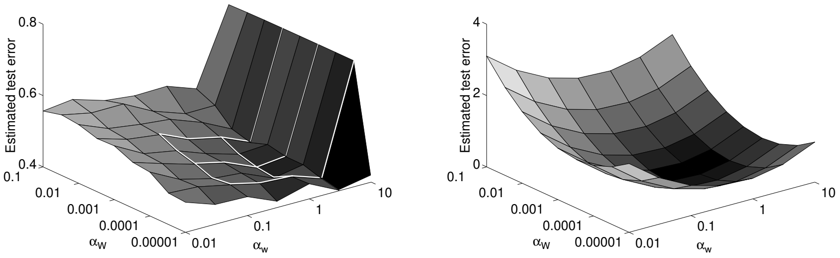


Figure 1: Left: The estimated test error for the *contiguity problem* as function of the weight decay parameters. Right: Paraboloid fitted to the 3x3 grid shown in the left panel. Minimum located at $(\alpha_w, \alpha_W) = (0.68, 2.8 \cdot 10^{-4})$.

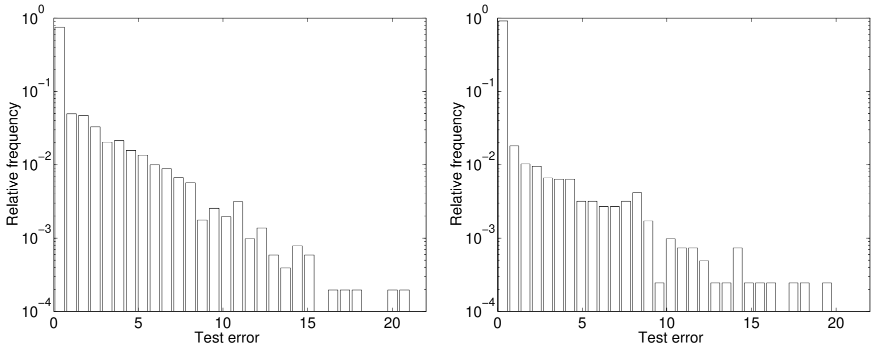


Figure 2: Left: The distribution of the test error for 10 fully connected *contiguity* networks combined. Notice the "long tail" of the distribution resulting in a high mean error (0.94) and a small median error (0.022) i.e., the mean is predominantly driven by a few examples with high error. Right: The distribution of the test error for 10 pruned networks selected by the minimum of the estimated test error combined. The mean error is 0.37 and the median error is 0.0014 showing a significant performance improvement as result of pruning.

Initially a network architecture consisting of 10 input units, 8 hidden units and 2 output units was chosen. The weight decay parameters were estimated by using the previously described sample-grid technique. This is shown in figure 1. Next ten fully connected networks were trained[3] using the estimated weight decay parameters, subsequently pruned using the OBD saliency ranking, removing one weight per iteration. In figure 2 the distribution of the test error is shown. The error distribution shows that the mean error is predominantly driven by a few examples with a high error, thus suggesting that one should monitor the median error as well in order to get a good indication of a network's performance. Seven of the ten pruned networks had a classification[4] error on the test set between 0% and 3.3%, while three networks had

[3]Training was stopped when the 2-norm of the gradient vector was below 10^{-5}.

[4]Following Bayes decision theory, the network output with the highest probability de-

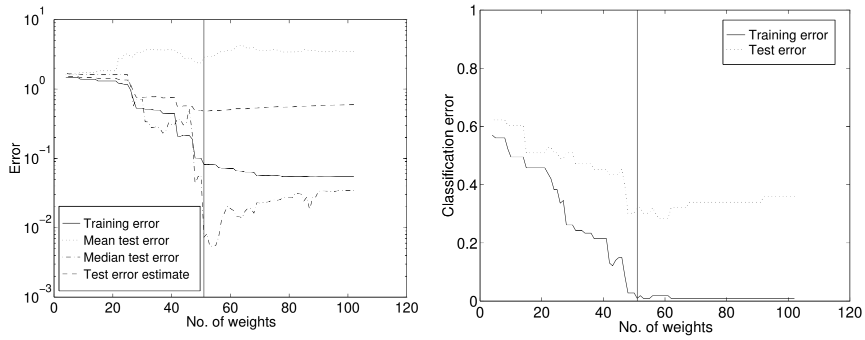


Figure 3: Pruning of a *glass classification* network using the small training set. The vertical line indicates the "optimal" network selected by the minimum of the estimated test error. Notice the similarity in the development of the median error and the classification error on the test set.

an error of 16-19%. In [10] seven of ten networks had an error of 8-38% using the same size of training set, while three networks had errors around 0%. Compared with these results, our classifier design scheme has a significantly higher yield.

The Glass Classification Problem

The task in the glass classification problem is to classify glass splinters into six classes. The glass splinters have been chemical analyzed and nine different measures have been extracted from the analysis, see [9] for details. The original dataset (*glass1*) consists of 214 examples divided into a training set (107), a validation set (54) and a test set (53). Since our approach doesn't require a validation set, we have used two different training scenarios: one using the original training set and one using a new training set consisting of the original training and validation set.

The initial network architecture chosen consisted of 9 input units, 6 hidden units and 6 output units. We estimated the regularization parameters using the sample-grid technique and the small training set. The parameters were found to be $\alpha_w = 2.2 \cdot 10^{-2}$ and $\alpha_W = 4.7 \cdot 10^{-4}$.

In figure 3 and 4 we show the pruning results of networks trained with the small and large training set, respectively, using the estimated regularization parameters. The "optimal" network found with the small training set had a classification error of 32% on the test set, while the "optimal" network found with the large training set had an error of 28%. In [9] Prechelt reports a test error of 32% for a fixed network architecture using the small training set. The validation set is used to stop training, thus he effectively uses both the training and validation set for training [11]. Our approach using the

termines the class label.

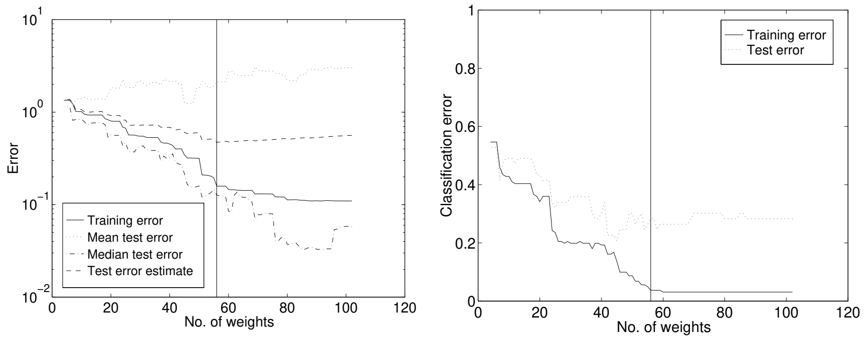


Figure 4: Pruning of a *glass classification* network using the large training set. The vertical line indicates the "optimal" network selected by the estimated test error. Notice the overall lower classification error on the test set compared to figure 3.

estimated test error for model selection eliminates the need for a validation set, thus allowing us to use more data for the actual training resulting in a better generalization performance. In a forthcoming paper the problem of comparing the performance of neural network models is addressed [12].

For comparison a standard *k-Nearest-Neighbor*[5] (k-N-N) classification was performed using the large training set. The training error may be computed from the training set by including each training pattern in the majority vote. A *leave-one-out* "validation" error on the training set may be computed by excluding each training pattern from the vote. Finally, the test patterns may be classified by voting among the k nearest neighbors found among the training patterns. Using the *leave-one-out* validation error we found that $k = 2$ was optimal for this data set. The 2-N-N scheme had a classification error of 34% on the test set. Thus the performance of the optimized k-N-N scheme cannot match Prechelt's or our networks.

CONCLUSION

We have developed a methodology for design and evaluation of neural classifiers. The approach was applied to the *contiguity problem* and the *glass classification problem*. It was shown that the test error estimator for classifiers could be used to select optimal networks among families of pruned networks, thus increasing the generalization ability compared to non-pruned networks. Currently, the aim is to establish more empirical data for the validation of the neural classifier design approach.

[5]Within k-N-N a pattern is classified according to a majority vote among its k nearest neighbors using the simple Euclidean metric.

ACKNOWLEDGMENT

This research is supported by the Danish Research Councils for the Natural and Technical Sciences through the Danish Computational Neural Network Center. Jan Larsen thanks the Radio-Parts Foundation for financial support.

REFERENCES

[1] C. Svarer, L.K. Hansen, and J. Larsen. "On Design and Evaluation of Tapped-Delay Neural Network Architectures". **Proceedings of the 1993 IEEE International Conference on Neural Networks**, pages 46–51, 1993.

[2] C. Svarer, L.K. Hansen, J. Larsen, and C.E. Rasmussen. "Designer Networks for Time Series Processing". **Proceedings of the 1993 IEEE Workshop on Neural Networks for Signal Processing III**, pages 78–97, 1993.

[3] M. Hintz-Madsen, L.K. Hansen, J.Larsen, E. Olesen, and K.T. Drzewiecki. "Design and Evaluation of Neural Classifiers - Application to Skin Lesion Classification". **Proceedings of the 1995 IEEE Workshop on Neural Networks for Signal Processing V**, pages 484–493, 1995.

[4] J.S. Bridle. "Probabilistic Interpretation of Feedforward Classification Network Outputs with Relationships to Statistical Pattern Recognition". **Neurocomputing - Algorithms, Architectures and Applications**, 6:227–236, 1990.

[5] G.A.F. Seber and C.J. Wild. **Nonlinear Regression**. John Wiley & Sons, New York, New York, 1995.

[6] Y. Le Cun, J. Denker, and S. Solla. "Optimal Brain Damage". **Advances in Neural Information Processing Systems**, 2:598–605, 1990.

[7] J. Larsen, L.K. Hansen, C. Svarer, and M. Ohlsson. "Design and Regularization of Neural Networks: The Optimal Use of A Validation Set". **Proceedings of the 1996 IEEE Workshop on Neural Networks for Signal Processing VI**, 1996.

[8] S. Amari and N. Murata. "Statistical Theory of Learning Curves under Entropic Loss Criterion". **Neural Computation**, 5:140–153, 1993.

[9] Lutz Prechelt. "PROBEN1 — A set of benchmarks and benchmarking rules for neural network training algorithms". **Technical Report**

21/94, Fakultät für Informatik, Universität Karlsruhe, Germany, 1994. *Available via anonymous ftp* `ftp.ira.uka.de/pub/papers/techreports/1994/1994-21.ps.Z`.

[10] J. Gorodkin, L.K. Hansen, A. Krogh, C. Svarer, and O. Winther. "A Quantitative Study of Pruning by Optimal Brain Damage". **International Journal of Neural Systems**, 4:159–169, 1993.

[11] J. Sjöberg. "Non-Linear System Identification with Neural Networks". **Ph.D. Thesis no. 381, Department of Electrical Engineering, Linköping University, Sweden**, 1995.

[12] J. Larsen et al. "Empirical Comparison of Neural Network Models". **In preparation**, 1996.

Appendix K

Asilomar '96 contribution

This appendix contains the paper "Pruning Boltzmann Networks and Hidden Markov Models," presented at the Asilomar 1996 Conference on Signals, Systems, and Computers. In this paper a Gauss-Newton like approximation to the Hessian is derived for the relative entropy cost function applied to general Boltzmann networks and the application of second-order training as well as the OBD and OBS pruning schemes are suggested for this model type. It is discussed how Hidden Markov Models may be pruned by converting them into equivalent Boltzmann chains, and the viability of pruning Boltzmann networks is illustrated for a Boltzmann zipper applied to an artificial problem.

Reference for the paper: [PS96].

PRUNING BOLTZMANN NETWORKS AND HIDDEN MARKOV MODELS

Morten With Pedersen[1] *David G. Stork*[2]

[1] Section for Digital Signal Processing, Department of Mathematical Modelling
Technical University of Denmark B321, DK-2800 Lyngby, DENMARK
mwp@imm.dtu.dk

[2]Machine Learning and Perception Group, Ricoh California Research Center
2882 Sand Hill Road Suite 115 , Menlo Park, CA 94025-7022 USA
stork@crc.ricoh.com

ABSTRACT

We present sensitivity-based pruning algorithms for general Boltzmann networks. Central to our methods is the efficient calculation of a second-order approximation to the true weight saliencies in a cross-entropy error. Building upon recent work which shows a formal correspondence between linear *Boltzmann chains* and Hidden Markov Models (HMMs), we argue that our method can be applied to HMMs as well. We illustrate pruning on *Boltzmann zippers*, which are equivalent to two HMMs with cross-connection links. We verify that our second-order approximation preserves the rank ordering of weight saliencies and thus the proper weight is pruned at each pruning step. In all our experiments in small problems, pruning reduces the generalization error; in most cases the pruned networks facilitate interpretation as well.

1. INTRODUCTION

There is an enormous body of simulation work demonstrating the value of architecture optimization for networks for pattern classification, and this has properly led to great interest in both theoretical foundations and in new algorithms. There are two basic viewpoints toward this issue: regularization (or penalty based) and sensitivity based. According to the viewpoint of regularization, one seeks to impose some desired property in the final solution, for instance smoothness. Thus in weight decay one penalizes large weights and therefore favors smoother decision boundaries. According to the viewpoint of sensitivity, one seeks to eliminate those parameters (e.g., weights) that have the smallest effect on the training error, thereby restricting the model without severely penalizing the training error. For instance, Optimal Brain Damage (OBD) [1] and Optimal Brain Surgeon (OBS) [2] eliminate weights that are predicted to have the least effect on the training error.

In fact both views stem from a deeper notion concerning the incorporation of model priors or structural risk minimization [3]. Despite their fundamental unity, in computational practice it is convenient to adopt one or the other of these views. For instance, regularization by weight decay is easy to incorporate *during* learning; pruning by sensitivity-based methods is traditionally performed *after* training with examples. (We note too that in practice one can use *both* methods when creating networks.) In this paper we mention briefly the application of weight decay but our primary contribution is a new sensitivity-based pruning algorithm for Boltzmann networks.

Most pruning methods have been developed for networks of nonlinear units — feed-forward or recurrent — typically trained by backpropagation or second-order methods. However pruning in another class of network, Boltzmann networks, has not received adequate attention. Though typically slower and a bit more difficult to train than feedforward neural networks, Boltzmann networks nevertheless have some desirable properties: natural handling of missing data (during training or recall), pattern completion, and superior avoidance of local energy minima during training. Based on the close relationship between certain Boltzmann networks and Hidden Markov Models [4,5], we now know that such Boltzmann networks can posess benefits of HMMs too, most particularly dynamic time adjustment.

Our paper is organized as follows: In Section 2. we provide a short review of Boltzmann networks. In Section 3., we show a second-order expansion of the cross-entropy cost function and present our sensitivity based pruning algorithm. In Section 4. we apply our method to a Boltzmann architecture that is of particular interest for integration of two channels having different time scales. We conclude in Section 5. with some thoughts on future directions.

2. BOLTZMANN NETWORKS

Boltzmann networks are stochastic networks with both visible and hidden units (cf., [6] for an introduction and the notation we use here). We let the subscript α denote the states of the visible units and β the states of the hidden units. The superscript $+$ denotes iterating the network with the visible units clamped to a desired pattern, and $-$ denotes the visible units running freely, or unclamped. The energy function for the Boltzmann network is usually defined as

$$E = -\frac{1}{2}\sum_{ij} w_{ij} s_i s_j, \qquad (1)$$

where w_{ij} is the (bi-directional) weight connecting units i and j, and s_i is the (binary) state of unit i. At *thermal equilibrium* the probability of finding the units in a given state configuration $\alpha\beta$ when the visible units are unclamped is given by the Boltzmann distribution

$$P_{\alpha\beta}^{-} = \frac{1}{Z} e^{-E_{\alpha\beta}}, \tag{2}$$

where Z is the normalizing partition function and $E_{\alpha\beta}$ is the energy (dependent on the weights) when the visible units are in states α and the hidden units are in states β; for clarity in the following, we have incorporated the temperature into the energy term. Thus, the probability P_α^- of finding the visible units in joint states α is found by summing over the possible hidden unit configurations β.

When training Boltzmann networks we want the probabilities of the freely running network P_α^- to match those of the environment/training examples P_α^+. As a measure of the difference between the two probability distributions we use the *Kullback-Leibler* measure, or relative entropy, as our cost function:

$$\begin{aligned} H(\mathbf{w}) &= \sum_\alpha P_\alpha^+ \ln \frac{P_\alpha^+}{P_\alpha^-} \\ &= -\sum_\alpha P_\alpha^+ \ln P_\alpha^- + \mathit{const}, \end{aligned} \tag{3}$$

where *const* is a constant determined solely by the environment, and is hence independent of the weights $\mathbf{w}$. When training using gradient descent we need the derivatives of $H(\mathbf{w})$ with respect to the bi-directional weights w_{ij} connecting units i and j:

$$\begin{aligned} \frac{\partial H(\mathbf{w})}{\partial w_{ij}} &= -\sum_\alpha P_\alpha^+ \frac{\partial \ln P_\alpha^-}{\partial w_{ij}} \\ &= -\sum_\alpha P_\alpha^+ \left\langle -\frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right\rangle_\alpha^+ + \left\langle -\frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right\rangle^- \\ &= < s_i s_j >^- - < s_i s_j >^+, \end{aligned} \tag{4}$$

where $< \cdots >_\alpha^+$ is the mean value given that the visible units are clamped in states α, and $< \cdots >^-$ is the mean when all units are free running.

3. PRUNING

The first derivatives lead to the traditional first-order training methods [6]. However, for training using second-order methods and for our pruning algorithm we need the second derivatives of the entropic cost. These second derivatives are calculated as:

$$\begin{aligned} &\frac{\partial^2 H(\mathbf{w})}{\partial w_{ij}\partial w_{pq}} = \\ &- \left[\sum_\alpha \frac{P_\alpha^+}{P_\alpha^-} \cdot \frac{\partial^2 P_\alpha^-}{\partial w_{ij}\partial w_{pq}} - \sum_\alpha \frac{\partial P_\alpha^-}{\partial w_{ij}} \cdot \frac{P_\alpha^+}{(P_\alpha^-)^2} \cdot \frac{\partial P_\alpha^-}{\partial w_{pq}} \right] \\ &= \sum_\alpha \frac{\partial \ln P_\alpha^-}{\partial w_{ij}} \cdot P_\alpha^+ \cdot \frac{\partial \ln P_\alpha^-}{\partial w_{pq}} - \sum_\alpha \frac{P_\alpha^+}{P_\alpha^-} \cdot \frac{\partial^2 P_\alpha^-}{\partial w_{ij}\partial w_{pq}} \\ &\approx \sum_\alpha \frac{\partial \ln P_\alpha^-}{\partial w_{ij}} \cdot P_\alpha^+ \cdot \frac{\partial \ln P_\alpha^-}{\partial w_{pq}}. \end{aligned} \tag{5}$$

The approximation in Eq. 5 is justified if we assume that the problem at hand is realizable, i.e., there exists a set of *optimal* weights $\mathbf{w}^*$, for which $P_\alpha^- = P_\alpha^+, \forall\alpha$ [7]. For these weights, the term in Eq. 5 involving second derivatives of the unclamped probabilities P_α^- reads

$$\begin{aligned} \sum_\alpha \frac{P_\alpha^+}{P_\alpha^-} \cdot \frac{\partial^2 P_\alpha^-}{\partial w_{ij}\partial w_{pq}} &= \sum_\alpha \frac{\partial^2 P_\alpha^-}{\partial w_{ij}\partial w_{pq}} \\ &= \frac{\partial^2}{\partial w_{ij}\partial w_{pq}} \left(\sum_\alpha P_\alpha^- \right) \\ &= 0, \end{aligned} \tag{6}$$

where in the first step we used the fact that $P_\alpha^+ = P_\alpha^-$ at $\mathbf{w} = \mathbf{w}^*$, and in the last step that $\sum_\alpha P_\alpha^- = 1$. Thus, close to the optimal weights $\mathbf{w}^*$ the term in Eq. 5 involving second derivatives vanish. The remaining term involves terms of a form calculated from Eq. 4:

$$\begin{aligned} \frac{\partial \ln P_\alpha^-}{\partial w_{ij}} &= \left\langle -\frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right\rangle_\alpha^+ - \left\langle -\frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right\rangle^- \\ &= < s_i s_j >_\alpha^+ - < s_i s_j >^- . \end{aligned} \tag{7}$$

Thus we obtain the second derivatives:

$$\begin{aligned} \frac{\partial^2 H(\mathbf{w})}{\partial w_{ij}\partial w_{pq}} &\approx \sum_\alpha \frac{\partial \ln P_\alpha^-}{\partial w_{ij}} \cdot P_\alpha^+ \cdot \frac{\partial \ln P_\alpha^-}{\partial w_{pq}} \\ &= \sum_\alpha P_\alpha^+ < s_i s_j >_\alpha^+ < s_p s_q >_\alpha^+ \\ &\quad - < s_p s_q >^- \sum_\alpha P_\alpha^+ < s_i s_j >_\alpha^+ \\ &\quad - < s_i s_j >^- \sum_\alpha P_\alpha^+ < s_p s_q >_\alpha^+ \\ &\quad + < s_i s_j >^- < s_p s_q >^- \\ &= < s_i s_j s_p s_q >^+ - < s_p s_q >^- < s_i s_j >^+ \\ &\quad - < s_i s_j >^- < s_p s_q >^+ \\ &\quad + < s_i s_j >^- < s_p s_q >^- . \end{aligned} \tag{8}$$

We note that the second derivatives involves only terms already computed when calculating the gradient, thus implementation is straightforward and yields little computational burden beyond that needed for gradient descent learning.

3.1. Pruning using OBD and OBS

Two well-known methods for pruning traditional feedforward and recurrent networks are Optimal Brain Damage (OBD) [1] and Optimal Brain Surgeon (OBS)

[2]. Both methods use second-order expansions of the error to estimate the importance of the parameters. OBD uses a diagonal approximation to the Hessian to calculate the saliency of a weight by estimating the change in error when the weight is set to zero. OBS uses the full Hessian, and the change in training error is estimated including the effect of reestimating the remaining parameters in the model to a new minimum within the quadratic approximation. The rationale behind both methods is that if we remove the least salient weights according to training error, we gracefully relieve the danger of overfitting, and thereby improve generalization.

The method we present here is firmly rooted in the logic of OBD/OBS; the key novelty is the equation for second derivatives, Eq. 8. We note that for traditional Boltzmann networks, removing a weight from the model is equal to setting it to zero, since the weight no longer provides contribution to the energy (Eq. 1). Thus, the logic of OBD and OBS can be applied directly for these models as well, though using the second derivatives derived above. Even though it is usually intractable to compute the numerical value of the entropic cost (Eq. 3), this does not affect the pruning methods since they only measure the *change* in cost, working from approximations using information already provided by the learning algorithms.

3.2. Pruning Hidden Markov Models

It has been shown that for certain tree-like connectivity of Boltzmann networks it is possible and computationally tractable to compute the expressions in Section 2. and 3. *exactly* [8]; this, therefore, provides greater accuracy for the saliency estimates provided by our method. Such computations have been used to a specific topology of Boltzmann networks called Boltzmann *chains* [4]. It was shown that any first-order HMM can be represented by an equivalent Boltzmann chain [4]. It was furthermore shown that under the condition that all state sequences have a mandatory end state, Boltzmann chains can be represented by first-order HMMs as well [5].

In traditional research on HMMs, the topology of HMMs for a given task has been chosen by hand or found by 'exhaustive' search. However we suggest that the topology can be optimized by *converting the HMM into a corresponding Boltzmann chain and performing pruning on this model.* The resulting chain is then converted back into an HMM for reestimation of the remaining parameters or, alternatively, the reestimation is done for the Boltzmann chain, only converting back the *optimal* model.

Pruning a weight in a Boltzmann chain representing an HMM should be equivalent to setting the corresponding transition probability in the HMM to zero, thus preventing hidden state sequences including the transition in question from contributing to the probability of a given observation sequence. This means that the weight should be *set equal to* $-\infty$ *if pruned in the Boltzmann chain*, yielding zero contribution to the partition function for state sequences including the transition in question. This is consistent with the expressions in [4] for converting the parameters in an HMM into the weights of a Boltzmann chain, which is accomplished by applying the natual logarithm to the transition probabilites.

When pruning Boltzmann chains representing HMMs we must modify our algorithms somewhat however, since these methods estimate the effect on the cost when a parameter in the chain is set to zero. Instead we are forced to set the weights to $-\infty$ one by one and compute the resulting change in error. This is possible since we are able to perform exact calculations of the entropic cost function for this special topology of Boltzmann networks. Also, we should be careful if/when pruning weights representing observation probabilities, since this means that the observation is no longer possible when in a particular hidden state.

4. EXPERIMENT

It has been shown how to model correlated discrete time series on disparate timescales using cross-connected parallel Boltzmann chains [4], which we call Boltzmann *zippers* (Fig. 1). These models can be interpreted as interconnected HMMs. Such models are of particular interest where one must integrate information from two time series having different inherent time scales, as for instance speechreading, where the fast acoustic information must be integrated with the slow visual information [9]. Here we use pruning to investigate the utilization of the cross-connection links.

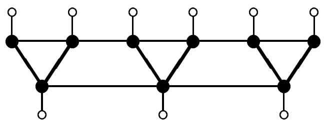


Figure 1. Topology of a Boltzmann zipper. White circles represent groups of visible units/states, dark circles represent groups of hidden units/states.

We generated synthetic patterns by two left-right HMMs. In order to generate observations on two different time scales, a *fast* model was iterated twice as often as a *slow* model. The fast model had three hidden states and three observation symbols, the slow had two hidden states and two observation symbols. The last states in the models were connected, increasing the probability of making transitions to the last hidden state in one model if the other model was already in its last state. Thus, the fast model was able to make direct transitions to its last state *if* the slow model was already in its last state.

Three hundred pairwise sequences of lengths 14 and 7 (respectively) were generated, and a fully connected

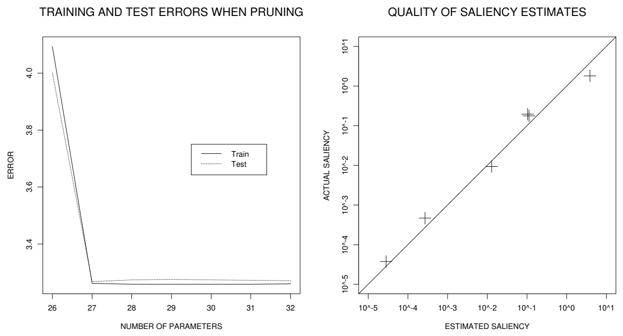


Figure 2. Left panel: Cross-entropy error on training and independent test sets (150 patterns each) versus the number of bidirectional weights between units. Pruning proceeds from right to left, and at the extreme left (26 weights), all cross connections have been removed. Right panel: The true saliences in a full network versus the saliencies estimated to second order. Note the excellent agreement over six orders of magnitude.

Boltzmann zipper having the same number of hidden and visible states as the underlying HMM-models was trained, using 150 examples for training and 150 examples as a separate test set. The model, initially having 32 parameters, was trained using gradient descent followed by a second-order damped Gauss-Newton method. In order to ensure numerical stability and facilitate training, the entropic cost function was augmented by a small quadratic weight decay term.

In order to investigate the utilization of the cross-connections in the zipper, we used our method (in the diagonal Hessian approximation) on these six weights. Since it is not clear how these weights relate to the transition probabilities in the HMMs we chose to limit the degrees of freedom by setting the weights to zero when pruned. The remaining weights were retrained by the second-order method after each weight elimination. In the left panel of Fig. 2 we show the results of pruning. We note that the errors are left almost unchanged until we prune the final cross-connection, after which the errors increase significantly. This is consistent with the model from which the data was generated, and indicates that the zipper has captured the underlying structure well. In fact, the lowest test error was obtained using only one cross connection, and the error rises dramatically if this last connection is also pruned.

In the right panel of Fig. 2 we illustrate the quality of the saliency estimates. For the fully connected zipper, we plot the estimated saliency for the cross connections versus the actual saliency computed by setting the weight to zero and calculating the resulting change in training error. We note that the estimates are approximately equal to the actual saliencies. Note that the rank ordering of the estimated saliency is the same as the actual saliencies; and thus the correct weight is deleted.

In 12 pruning experiments such as just described (in which the generating model had a single cross-connection link), we found that all 12 resulting networks displayed better generalization than the unpruned network. Of these, we found that 9 displayed best generalization using a single link, as expected.

5. CONCLUSION AND FUTURE WORK

We have derived the second derivatives of the entropic cost function and shown how to use these for sensitivity-based pruning of general Boltzmann networks. We have described how to extend this approach to Hidden Markov Models by transforming these into equivalent Boltzmann chains. Finally, we illustrated the viability of pruning on the cross-connections of Boltzmann zippers.

Clearly our method must be further demonstrated on large and realistic problems, such as speechreading. It will be interesting to see if our method, when applied to Boltzmann chains, is computationally more efficient than the exhaustive search method used throughout speech research for highly trained models.

Acknowledgements

This work was completed during a visit to the Ricoh California Research Center, supported by the Danish Natural Science and Technical Research Councils through the Computational Neural Network Center (CONNECT). The authors would like to thank Lars Kai Hansen and Greg Wolff for support, and Lawrence Saul for valuable discussions and for making available his Boltzmann zipper code.

References

[1] Y. Le Cun, J. S. Denker & S. A. Solla (1990) "Optimal Brain Damage," in D. S. Touretzky (ed.), *Advances in Neural Information Processing Systems – 2*, pp. 598-605. San Mateo, CA: Morgan-Kaufmann.

[2] B. Hassibi & D. G. Stork (1993) "Second order derivatives for network pruning: Optimal Brain Surgeon," in S. J. Hanson, J. D. Cowan & C. L. Giles (eds.) *Advances in Neural Information Processing Systems – 5*, pp. 164-171. San Mateo, CA: Morgan-Kaufmann.

[3] V. Vapnik (1982) **Estimation of Dependences Based on Empirical Data** New York: Springer-Verlag.

[4] L. K. Saul & M. I. Jordan (1995) "Boltzmann Chains and Hidden Markov Models," in G. Tesauro, D. Touretzky & T. Leen (eds.), *Advances in Neural Information Processing Systems – 7*, pp. 435-442. Cambridge, MA: MIT Press.

[5] D. J. MacKay (1996) "Equivalence of Linear Boltzmann Chains and Hidden Markov Models," *Neural Computation* **8**(1) 178-181.

[6] S. Haykin (1994) **Neural Networks: A Comprehensive Foundation** New York: Macmillan.

[7] L. K. Hansen (1996) unpublished manuscript.

[8] L. K. Saul & M. I. Jordan (1994) "Learning in Boltzmann trees," *Neural Computation* **6**(6) 1174-1184.

[9] D. G. Stork & M. E. Hennecke (eds.) (1996) **Speechreading by Humans and Machines** New York: Springer-Verlag.

Appendix L

NNSP '97 contribution (a)

This appendix contains the paper "Training Recurrent Networks," accepted for presentation at the IEEE 1997 workshop on Neural Networks for Signal Processing. This paper directs the attention towards the numerical aspects of recurrent network training. It is described how ill-conditioning may arise during training of recurrent networks, and how such numerical problems are reduced when augmenting the quadratic cost by a simple weight decay term. Examples of ill-conditioning and the effects of weight decay are given for a small recurrent network trained on the Santa Fe laser series. Furthermore a comparison between training methods is provided for a larger recurrent network, illustrating the superiority of the damped Gauss-Newton method over gradient descent once the need for regularization has been recognized.

Reference for the paper: [Ped97].

TRAINING RECURRENT NETWORKS

Morten With Pedersen
CONNECT, Department of Mathematical Modelling, Building 321
Technical University of Denmark, DK-2800 Lyngby, Denmark
Phone: + 45 45253920 Fax: + 45 45872599
email: mwp@imm.dtu.dk

Abstract - Training recurrent networks is generally believed to be a difficult task. Excessive training times and lack of convergence to an acceptable solution are frequently reported. In this paper we seek to explain the reason for this from a numerical point of view and show how to avoid problems when training. In particular we investigate ill-conditioning, the need for and effect of regularization and illustrate the superiority of second-order methods for training.

INTRODUCTION

Recurrent neural networks are an interesting class of models for signal processing as they are able to build up internal memory suited for the task at hand and thus often lead to compact model representations. However, it is generally believed to be a difficult task to train this type of networks. Several authors have addressed the learning problem for recurrent networks, e.g., in the context of sequence classification when required to store information for an arbitrary period of time [1, 5] but to the best of the authors knowledge no one have treated the problem from a general *numerical* point of view.

Feedforward networks were treated extensively from a numerical point of view in [7] where it was illustrated how training forms an extremely ill-conditioned optimization problem. In this contribution we extend this analysis to include recurrent networks. In particular we identify redundant connections and illustrate how ill-conditioning may otherwise arise, which motivates the use of regularization.

Having acknowledged the need for regularization makes way for the highly effective second-order methods for training. In this contribution we particulary focus on the *damped* Gauss-Newton method and illustrate how this method by far outperforms gradient descent on a time series prediction problem, namely the Santa Fe laser data. The focus in this contribution is on time series prediction, but the results generalize to other applications as well.

ARCHITECTURE

The general architecture of the networks considered here are fully connected feedback networks with one hidden layer of nonlinear units and a single linear output unit. The output $y(t)$ of the network is linear in order to allow for arbitrary dynamical range, and is given by

$$y(t) = \sum_{i=1}^{N_h} w_{oi} s_i(t) + w_{ob} \tag{1}$$

where N_h is the number of hidden units, w_{oi} is the weight to the output unit from hidden unit j and w_{ob} is a bias weight. The output $s_i(t)$ from hidden unit i at time t is computed as

$$s_i(t) = f\left(\sum_{j=1}^{N_h} w_{ij} s_j(t-1) + w_{io} y(t-1) + \sum_{k=1}^{N_I} w_{ik} x_k(t) + w_{ib} \right) \tag{2}$$

where w_{ij} is the weight *to* hidden unit i *from* hidden unit j, w_{io} is the weight *to* hidden unit i *from* the output unit and w_{ib} is the bias weight for hidden unit i. $x_k(t)$ is the k'th element in the external input vector $\mathbf{x}(t)$ at time t and N_I is the total number of external inputs. $f(\cdot)$ is the nonlinear activation function, in this work we use $f(x) = \tanh(x)$.

Note that the update of the recurrent network presented above is *layered*, as the outputs $s_i(t)$ from the hidden units are computed immediately *before* the computation of the output unit output. This is opposed to the update presented in e.g. [10] where all the units are updated simultaneously. In [6] it was shown that when using fully recurrent networks for forecasting, layered update is preferable since synchronous update of the units effectively results in a two-step ahead predictor. Note also that the linear output unit does not have feedback of its own previous value. This is in order to avoid stability problems that are otherwise likely to occur.

Training

In this work we focus on time series prediction in which case the input vector contains delayed elements of the time series, $\mathbf{x}(t) = [x(t), \ldots, x(t - N_I + 1)]$, and the network output is a prediction of the next value in the series, $\widehat{x}(t+1) = y(t)$. Training the network means adjusting the weights so as to minimize a cost function. Most applications are based on the sum of squared errors,

$$E(\mathbf{w}) = \frac{1}{2} \sum_{t=1}^{T} [e(t)]^2 \quad , \quad e(t) = x(t+1) - y(t) \tag{3}$$

where T denotes the number of training examples and $\mathbf{w}$ is the concatenated set of parameters. The adjustment of the parameters is done *off line* by an iterative sheme, $\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \Delta \mathbf{w}_k$, where $\Delta \mathbf{w}_k$ indicates the direction of change and η is the (adaptive) size of the step. When training recurrent neural networks the most commonly used scheme is gradient descent, where the direction $\Delta \mathbf{w}_k$ is equal to the gradient $\mathbf{g}$, $g_i = \partial E(\mathbf{w}_k)/\partial w_i$. Unfortunately this method suffers from extremely slow convergence, and the quality of resulting solutions is often not satisfactory.

Experiments have shown that much more efficient training can be obtained by using second-order methods [6]. Here we focus on the *damped* Gauss-Newton method [3], in which the search direction $\Delta \mathbf{w}_k$ is determined by

$$\Delta \mathbf{w}_k = \mathbf{H}^{-1}\mathbf{g} \tag{4}$$

where $\mathbf{H}$ is the positive semidefinite approximation to the Hessian,

$$H_{ij} = \sum_{t=1}^{T} \frac{\partial y(t)}{\partial w_i}\frac{\partial y(t)}{\partial w_j} \approx \frac{\partial^2 E(\mathbf{w}_k)}{\partial w_i \partial w_j} = \sum_{t=1}^{T}\left[\frac{\partial y(t)}{\partial w_i}\frac{\partial y(t)}{\partial w_j} - e(t)\frac{\partial^2 y(t)}{\partial w_i \partial w_j}\right] \tag{5}$$

In each iteration k the step size η is determined by line search which makes the method globally convergent [3]; here we recommend a simple approach where η is halved until a decrease in the cost is obtained [3]. The iterations are continued until convergence, determined by a sufficiently small length of the gradient, $\|\mathbf{g}\|_2 < \varepsilon$. The Gauss-Newton method involves finding the solution to a linear system of equations $\mathbf{H}\Delta\mathbf{w}_k = \mathbf{g}$ in each iteration, but the increased computational burden is justified by a dramatic increase in convergence and thus reduction of overall training time, even for large networks as we shall see. However, the success of the damped Gauss-Newton method relies heavily on the conditioning of the training problem, as is the case for gradient descent.

ILL-CONDITIONING

When training using either gradient descent or the Gauss-Newton method, a measure of great importance for the convergence is the condition number of the Hessian $\mathbf{H}$. For a symmetric positive definite matrix $\mathbf{H}$, the condition number is defined as $\kappa(\mathbf{H}) = \lambda_{max}/\lambda_{min}$, the ratio between the largest and smallest eigenvalue of $\mathbf{H}$. If the condition number is *large*, the Hessian becomes *ill-conditioned*. The convergence rate will suffer and the solution to the linear system of equations (4) in the Gauss-Newton method becomes unreliable. As a rule of thumb the solution may not be trustworthy if $\kappa(\mathbf{H}) > \epsilon^{-1/2}$, where ϵ denotes the machine precision [3]. For the IEEE 64-bit floating point representation this is equivalent to $\kappa(\mathbf{H}) > 6.7 \cdot 10^7$. This may seem as a large number, but this order of magnitude is not uncommon in the framework of either feedforward networks [7] or recurrent networks as we shall see.

In [2] it was shown that an eigenvalue of the order of the number of input variables could be avoided if the mean was subtracted from each of the input variables $x_k(t)$ and if a symmetric activation function is used. However, these simple countermeasures are not adequate for avoiding ill-conditioning in recurrent networks, as the analysis in the following will show.

The Hessian (5) can also be written as

$$\mathbf{H} = \mathbf{J}^{\mathrm{T}}\mathbf{J}\ ,\ J_{ti} = \frac{\partial y(t)}{\partial w_i} \tag{6}$$

where $\mathbf{J}$ is the Jacobian matrix, whose columns are the partial derivatives of the network output at each timestep in the training series. If $\mathbf{J}$ is rank-deficient some of the columns are linearly dependent, which is indicated by singular values with the value zero in an SVD analysis. This again leads to a singular Hessian and thus an infinite condition number. In practice it is rare to find columns in **J** that are *exactly* dependent and thus singular values that are exactly zero [7]. However, it is often the case that columns are *nearly* linearly dependent, which leads to very small singular values of **J** and thus large condition numbers for the Hessian **H**. In the following sections we describe situations leading to ill-conditioning of **J** for recurrent networks, arising from both exact and approximate linearly dependencies between columns in **J**.

Exact dependency

For the type of recurrent networks defined by (1) and (2) there is built-in rank deficiency in the Jacobian since it is easy to show that some of the columns in **J** will *always* be linear combinations of each other. This is illustrated by an example for a small network, but the result apply for networks with an arbitrary number of hidden units. The network considered here involves only one external input and one hidden unit, and the output is thus defined as

$$
\begin{aligned}
y(t) &= w_{o1}s_1(t) + w_{ob} &(7)\\
s_1(t) &= f\left(w_{11}s_1(t-1) + w_{1o}y(t-1) + w_{1x}x(t) + w_{1b}\right) &(8)\\
&= f\left((w_{11} + w_{1o}w_{o1})s_1(t-1) + w_{1x}x(t) + (w_{1b} + w_{1o}w_{ob})\right) &(9)
\end{aligned}
$$

where (9) is obtained by insertion of (7) in (8). We see that the network output will remain unchanged as long as the total weighting k_1 of $s_1(t-1)$, $k_1 = w_{11} + w_{1o}w_{o1}$, and the total bias k_2 on the hidden unit, $k_2 = w_{1b} + w_{1o}w_{ob}$, remains constant. w_{o1} and w_{ob} can not be changed without directly affecting the network output (7) and are therefore kept fixed which we denote by $*$. However, changes in w_{11}, w_{1o} and w_{1b} that satisfies *both* expressions

$$
\begin{array}{ccccccc}
w_{11} & + & w_{o1}^* \cdot w_{1o} & + & 0 & = & k_1\\
0 & + & w_{ob}^* \cdot w_{1o} & + & w_{1b} & = & k_2
\end{array} \qquad (10)
$$

will leave the network output unchanged. The expressions (10) form hyperplanes in parameter space spanned by w_{11}, w_{1o} and w_{1b} and their line of intersection is computed as $(w_{11}, w_{1o}, w_{1b}) = (k_1, 0, k_2) + t(-w_{o1}^*, 1, -w_{ob}^*)$, parametrized by t. The line defines a direction in parameter space in which the network output is constant. The constant network output means that derivatives are zero in this direction. Thus, columns in the Jacobian corresponding to (w_{11}, w_{1o}, w_{1b}) are linearly dependent.

When investigating Jacobians for the dependency problem outlined above it is however uncommon to encounter singular values *exactly* equal to zero; but according to the derivations this clearly ought to be the case. The reason for this is the initialization of previous state values when starting up the network. If the recurrent network starts iteration at time $t = 1$ it is common practice [10] to set the previous states of the hidden units as well as their derivatives to zero, $s_i(0) = 0 \;\;,\;\; \partial s_i(0)/\partial \mathbf{w} = 0$. This startup procedure clearly marks an initial discontinuity in the recursive equations (7) and (8) governing the feedback network. Thus initially the partial derivatives wrt. the involved weights in the Jacobian will generally *not* be linearly dependent. But after a few iterations indicating a transient, the dependency arises with increasing accuracy. The linear dependency is eliminated if we omit the feedback weights w_{io} *from* the output *to* the hidden units i, as the degeneracy can then no longer occur. This elimination has no influence on the modeling capabilities of the network since the remaining weights can be adjusted so that the network output remains unaffected.

Approximate dependency

Even though removal of the feedback weights w_{io} leading from the linear output to the hidden units removes the problem of almost exact rank-deficiency in the Jacobian for recurrent networks it does not eliminate ill-conditioning as experiments show. In [7] the problem of ill-conditioning was analyzed for *feedforward* networks by careful examination of the components entering the partial derivatives $\partial y(t)/\partial w_i$ of the network output and it was found that ill-conditioning in the Jacobian can arise from at least these three reasons (assuming that the external inputs are not proportional):

1. The output from a hidden unit is saturated and constant ($\equiv \pm 1$).
2. The outputs from two hidden units are approximately proportional.
3. The derivatives of two hidden unit outputs wrt. their activations are approximately proportional.

Theoretical and empirical examinations of the components entering the partial derivatives for *recurrent* networks reveal that ill-conditioning may arise here from the same reasons; such analysis is however not included here.

Situation 2 where the outputs of two hidden units are proportional and thus highly correlated often occurs in practice; e.g., in [9] high correlation between hidden unit outputs was found and studied for feedforward networks.

The effects of situation 2 are similar to the effects of exact dependencies described above, as we can determine directions in parameter space in which the cost function is approximately constant. For recurrent networks this situation is much more severe than for feedforward networks since the degeneracy will not only affect weights leading to the output, but also many weights connecting the hidden units as the experiments will show.

The scenarios listed above lead to nearly linearly dependencies between the columns of $\mathbf{J}$ and thus to small eigenvalues in $\mathbf{H}$. However, the condition number of a matrix is determined by the *ratio* between the largest and smallest eigenvalues, thus problems do not only arise from small singular values but also from large values. As mentioned, the situations described above will lead to directions in parameter space where the cost is approximately constant, thus when training using the Gauss-Newton method the search direction will be dominated by these directions leading to an unrestrained growth in the magnitude of the affected weights. This again leads to a significant growth in the magnitude of several of the columns in the Jacobian since many derivatives are dominated by terms of the form [10]

$$\frac{\partial s_k(t)}{\partial w_{pq}} \propto \sum_{j=1}^{N_h} w_{kj} \frac{\partial s_j(t-1)}{\partial w_{pq}} \tag{11}$$

which becomes large if the weights w_{kj} become large. The large elements in the Jacobian lead to an overall upward scale of the elements in $\mathbf{J}^{\mathrm{T}}\mathbf{J}$ and thus to an upward shift of the eigenvalues.

REGULARIZATION

A traditional method for handling ill-conditioning is by *regularizing* the cost function [3, 4]. A simple yet highly effective regularization can be obtained by augmenting the cost function by a simple quadratic weight decay [4],

$$C(\mathbf{w}) = E(\mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \tag{12}$$

Simple weight decay is often primarily considered as a means for avoiding overfitting as it puts constraints on the parameters and thus reduces the degrees of freedom. Weight decay should however also be considered from its regularizing effects. The immediate effect is that α gets added to the diagonal of the Hessian which puts a lower bound on the smallest eigenvalues, since it is easy to show that $\lambda(\mathbf{H} + \alpha\mathbf{1}) = \lambda(\mathbf{H}) + \alpha$. Another effect is the limit imposed on the growth of the weights which prevents near singular directions in parameter space from dominating the search directions obtained by the Gauss-Newton method, thus greatly improving the efficiency of the optimization. The constraints put on the weights by the regularization has a smoothing effect on the cost function which was clearly illustrated in [6]. Here it was also demonstrated that the significance of the second order term ignored in (5) diminishes when using simple weight decay as regularization.

EXPERIMENTS

In the first experiment we illustrate how ill-conditioning results from some of the situations described herein and how regularization improves training. For this experiment we used a simple recurrent network to predict the laser data from the Santa Fe time series prediction competition [8]. The data were scaled so that the first 1000 points used for training had zero mean and unit variance and the following 100 values were used as a test set. The network used had one external input and three hidden units; there were *no feedback from the linear output unit* to the hidden units as found appropriate above. Training was performed initially using five iterations of gradient descent followed by the damped Gauss-Newton method. In the left panel of Figure 1 is shown the evolution of the *mean* squared errors normalized by the variance of the sets (NMSE, [8]) when training without regularization. It seems that training is converging to a solution, but this is *not* the case as the evolution of the weights

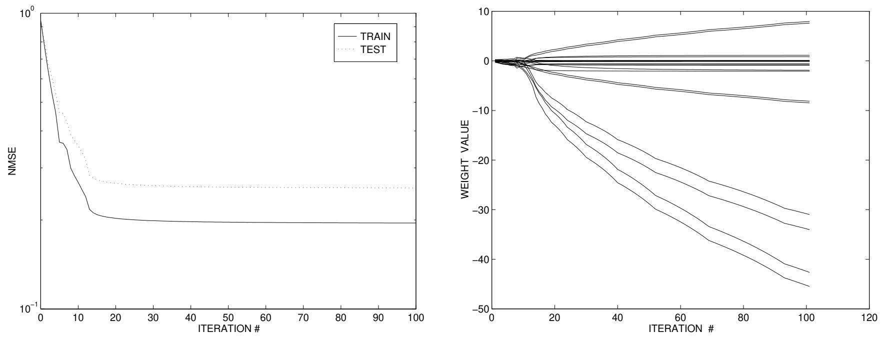


Figure 1: Training without regularization. Left panel: Evolution of training and test errors. Right panel: Evolution of the weight values.

in the right panel of Figure 1 shows. What happens is that the outputs of two hidden units become almost proportional; this is revealed by the cosine to the angle θ between vectors containing their outputs on the training set which at iteration 100 is $\cos\theta = 0.9998$. This corresponds to situation 2 listed above. The weights that grow in magnitude are the pairs of weights leading *from* these two units *to* every unit in the network including the output. Note that the error and thus the network output is unaffected since the effects of the changes in the growing weights cancel out due to the dependency between the hidden units.

The condition number during training is shown in the left panel of Figure 2 and is seen to grow enormously. The rapid increase occurs shortly after the initiation of the second-order method which quickly 'discovers' the dependency between the hidden unit outputs. The near singular Hessian $\mathbf{H}$ leads to very large weight changes in some directions when solving (4). The large steps are however handled by the line search which returns very small step sizes, indicated by the smooth increase in the weight magnitudes. In the right panel of Figure 2 is shown the eigenvalues of the Hessian after iterations 7, 20 and 100. At each of the iterations it is seen that the condition number results from both very small as well as very large eigenvalues and we note that as

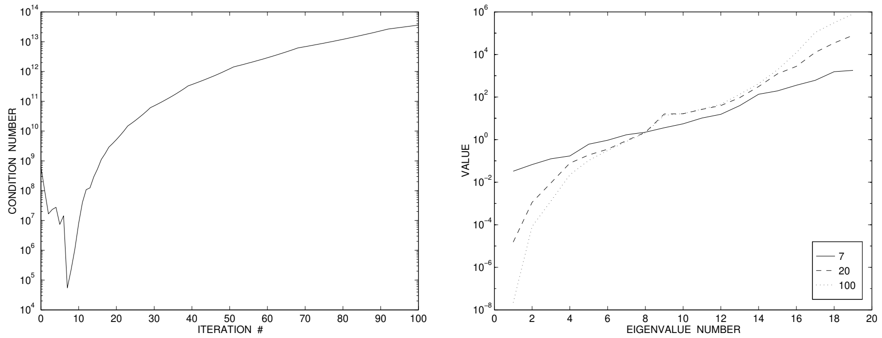


Figure 2: Training without regularization. Left panel: Evolution of the condition number for $\mathbf{H}$. Right panel: Eigenvalues after iterations 7, 20 and 100.

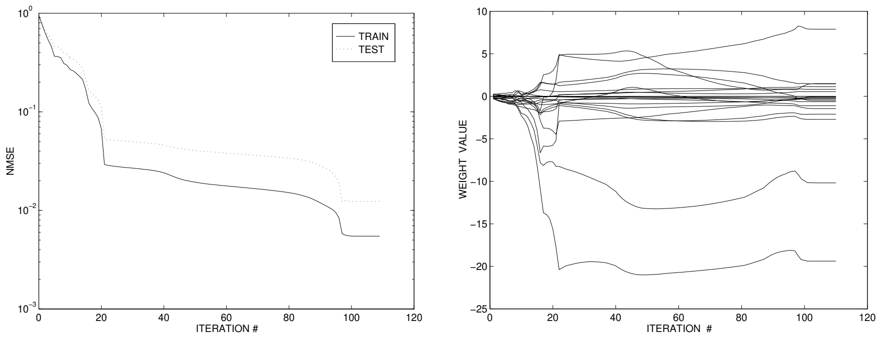


Figure 3: Training with regularization, $\alpha = 10^{-3}$. Left panel: Evolution of training and test errors. Right panel: Evolution of the weight values.

training progresses the eigenvalues extend both upward and downward.

The training was then repeated using the exact same initial weights and the same training approach, but now with a regularization term added to the cost function, using $\alpha = 10^{-3}$. In the left panel of Figure 3 is shown the resulting evolution of the errors. The positive effect of the regularization is evident, as the final errors are several orders of magnitude below the levels shown in Figure 1 obtained without regularization. Furthermore the stopping criterion $\|\mathbf{g}\|_2 < 10^{-4}$ was satisfied; in the previous experiment using no regularization the gradient norm grew proportional to the condition number.

In the right panel of Figure 3 we see that the regularization term limits the growth of the weights compared to Figure 1. Some however still grow large as does the condition number shown in the left panel of Figure 4. Even though the condition number grows to 10^8 the damped Gauss-Newton method still manages to find a minimum. Experience shows that for this method successful training to a (local) minimum can be obtained for condition numbers up to about 10^8 in magnitude. This may depend on the decomposition algorithm used when solving (4), here we use the fast and stable Cholesky factorization [3]. From the right panel of Figure 4 we learn that the reduction in condition number is obtained *only* from an increase in the smallest eigenvalues resulting

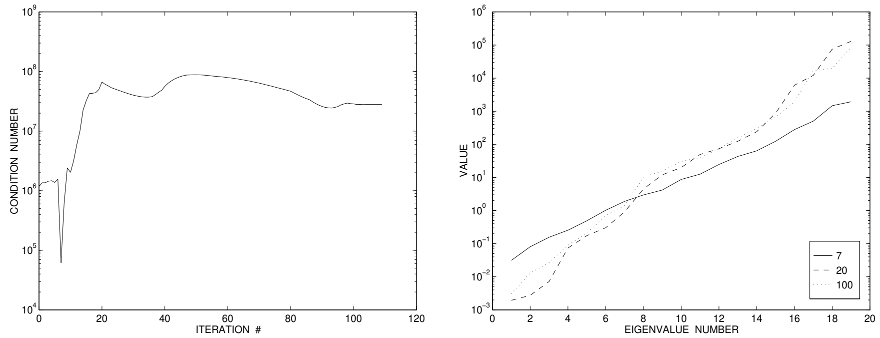


Figure 4: Training with regularization, $\alpha = 10^{-3}$. Left panel: Evolution of the condition number for $\mathbf{H}$. Right panel: Eigenvalues after iterations 7, 20 and 100.

from the regularization. The largest eigenvalues are of the same order of magnitude as when training without weight decay, see Figure 2. This is due to the still fairly large weight magnitudes. If the regularization term α is further increased the larger eigenvalues will also be affected; but so will the modeling capabilities of the network, leading to increased errors.

In the final experiment we compare the performance of damped Gauss-Newton with a gradient descent algorithm also using the step-size halving line search. The problem is still prediction of the laser series but using larger networks with a single input and nine hidden units, 109 weights in total (no feedback from the output to the hidden units). Thus, each iteration using damped Gauss-Newton involved solution of a 109 by 109 linear system of equations. Six initial networks were generated by initializing their weights with values drawn from a uniform distribution over the interval $[-0.3; 0.3]$. The training algorithms were then compared when starting from the same six initial networks, both using regularization $\alpha = 0.02$. The resulting evolution of errors is shown in Figure 5; in the left panel we see the resulting errors using the damped Gauss-Newton method, in the right panel using gradient descent. Using both methods the stopping criteria was set to $\|\mathbf{g}\|_2 < 10^{-4}$ or maximum 10000 iterations.

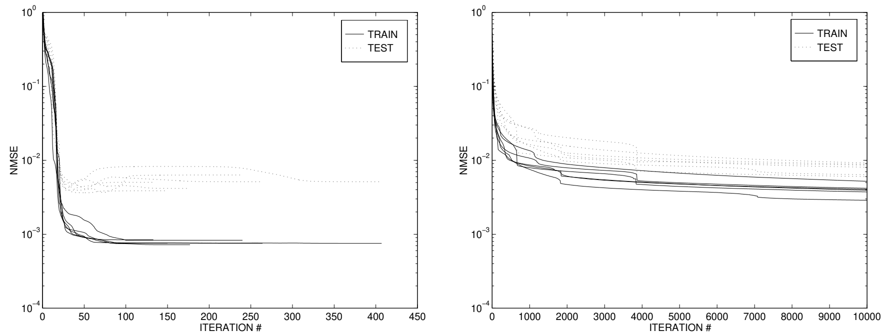


Figure 5: Evolution of errors using different optimization methods. Left panel: Damped Gauss-Newton method. Right panel: Gradient descent with line search.

For the damped Gauss-Newton method the stopping criterion was met in all six runs. The average training error (Normalized Mean Squared Error) was $7.7 \cdot 10^{-4}$, the average test error was $4.9 \cdot 10^{-3}$. The *average* time for a complete training run was 200 seconds. For gradient descent the stopping criterion was never met, the termination of the algorithm in each run was due to maximum number of iterations reached. The average training error obtained after the maximum allowed 10000 iterations was $4.0 \cdot 10^{-3}$, the average test error was $7.8 \cdot 10^{-3}$. The average time used for obtaining these error levels was 8140 seconds. Note that the levels of both training and test errors obtained using gradient descent are much higher than the levels obtained using the damped Gauss-Newton method even though gradient descent used a factor of 50 times more iterations and a factor of 40 times more computer time. Thus, even though an iteration of the damped Gauss-Newton method is computationally more costly than an iteration of gradient descent, the additional cost is highly justified by the vastly increased convergence rate. Similar justification has been observed for networks with up to 300 parameters.

CONCLUSION

In this paper we have focused on sources of ill-conditioning and thus the need for regularization when training recurrent networks especially using second-order methods. Once this need is recognized dramatic improvement in convergence rate and quality of solution is obtained, even for large size problems.

ACKNOWLEDGMENTS

The author would like to thank Lars Kai Hansen and Jan Larsen for support. This research is supported by the Danish Natural and Technical Research Coucils through the Computational Neural Network Center (CONNECT).

REFERENCES

[1] Y. Bengio, P. Simard and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," **IEEE Transactions on Neural Networks**, vol. 5, no. 2, pp. 157–166, 1994.

[2] Y. L. Cun, I. Kanter and S. A. Solla, "Eigenvalues of covariance matrices: Application to neural-network learning," **Physical Review Letters**, vol. 66, no. 18, pp. 2396–2399, 1991.

[3] J. E. Dennis and R. B. Schnabel, **Numerical Methods for Unconstrained Optimization and Nonlinear Equations**, Englewood Cliffs, NJ: Prentice-Hall, 1983.

[4] S. Haykin, **Neural Networks, A Comprehensive Foundation**, New York, NY: Macmillan, 1994.

[5] S. Hochreiter and J. Schmidhuber, "Long short term memory," Tech. Rep. FKI-207-95, Fakultat fur Informatik, Technische Universitat Munchen, Munchen, 1995.

[6] M. W. Pedersen and L. K. Hansen, "Recurrent networks: Second order properties and pruning," in G. Tesauro, D. Touretzky and T. Leen, eds., **Advances in Neural Information Processing Systems**, The MIT Press, 1995, vol. 7, pp. 673–680.

[7] S. Saarinen, R. Bramley and G. Cybenko, "Ill-conditioning in neural network training problems," **SIAM Journal on Scientific Computing**, vol. 14, pp. 693–714, 1993.

[8] A. S. Weigend and N. A. Gershenfeld, eds., **Time Series Prediction: Forecasting the Future and Understanding the Past**, Reading, MA: Addison-Wesley, 1993.

[9] A. S. Weigend and D. E. Rumelhart, "The effective dimension of the space of hidden units," in E. Keramides, ed., **Proceedings of INTERFACE'91: Computing Science and Statistics**, Springer Verlag, 1992.

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

Appendix M

NNSP '97 contribution (b)

This appendix contains the paper "Interpretation of Recurrent Neural Networks," accepted for presentation at the IEEE 1997 workshop on Neural Networks for Signal Processing. In this paper it is suggested to measure the *memory* of a recurrent network applied to time series prediction on a *test* set. Expressions are provided for both an "average" memory measure as well as a "short term" memory measure. As the network memory is defined in terms of generalization error, attention is directed towards the generation of a *learning curve*. Learning curves are generated for recurrent networks trained on the Santa Fe laser series as well as the Mackey-Glass series, and average memory as well as short term memory is measured for selected networks.

Reference for the paper: [PL97].

INTERPRETATION OF RECURRENT NEURAL NETWORKS

Morten With Pedersen and Jan Larsen
CONNECT, Department of Mathematical Modelling, Building 321
Technical University of Denmark, DK-2800 Lyngby, Denmark
Phones: + 45 4525 + ext. 3920,3923 Fax: + 45 45872599
emails: mwp@imm.dtu.dk, jl@imm.dtu.dk

Abstract - This paper addresses techniques for interpretation and characterization of trained recurrent nets for time series problems. In particular, we focus on assessment of effective memory and suggest an operational definition of memory. Further we discuss the evaluation of learning curves. Various numerical experiments on time series prediction problems are used to illustrate the potential of the suggested methods.

INTRODUCTION

It is widely recognized that recurrent neural networks (RNNs) are flexible tools for time series processing, system identification and control problems, see e.g., [3]. Feed-forward networks can accommodate dynamics by having a lag space of past input and target values; however, a fully recurrent network with internal feedbacks allows for even more sophisticated dynamics. While fully RNN architectures are the ultimate tool for modeling dynamic relations, the comprehension of the networks is a challenging subject of ongoing research. Theoretical investigations of modeling capabilities of RNNs have been reported, see e.g., [2], [4], [7]. However, to the authors knowledge, there is no general theory of the dynamic behavior of a general RNN except for very special models like the Hopfield network, see e.g., [3]. This indeed indicates that theoretical analysis of RNNs is extremely complicated. On the other hand, one might pursue a more computational approach. The general computational tools from non-linear dynamic systems analysis like phase portraits, stability analysis, measurement of fractal dimensions or Lyapunov exponents (see e.g., [1], [3]) may be applied to the analysis of RNNs.

The motivation for this paper is evaluation and interpretation of *trained* recurrent networks, and to suggest and discuss simple operational techniques. In particular, we focus on the *learning curve* and present a new method to determine the effective *memory* of a recurrent network which conveys the relevant time scale of the dynamics.

NETWORK ARCHITECTURE

The objective is to model a non-linear dynamic relation among a discrete-time input signal $x(t)$ and a discrete time target signal, $d(t)$. The general architecture of the RNN considered in this presentation is based on [5] and consists of a single hidden layer of fully connected nonlinear units and one output unit. In particular, we focus on a network with only one external input, viz. the most recent value, $x(t)$. That is, the only information available about previous inputs stems from the memory build up internally in the net. The advantage using these networks is that the tedious problem of determining the optimal lag space of previous inputs is converted into determining the optimal network architecture in terms of connections and number of hidden neurons.

The network has a linear output in order to allow for arbitrary dynamic range, and at time t the prediction of the target $d(t)$ is given by,

$$y(t) = \sum_{i=1}^{N_h} w_{oi} \cdot s_i(t) + w_{ob} \tag{1}$$

where N_h is the number of hidden units, w_{oi} is the weight to the output unit from hidden unit i and w_{ob} is the output bias weight. The ith *state*, $s_i(t)$, is the output of a hidden unit computed as

$$s_i(t) = f\left(\sum_{j=1}^{N_h} w_{ij} \cdot s_j(t-1) + w_{ix} \cdot x(t) + w_{ib}\right) \tag{2}$$

where w_{ij} is the weight to hidden unit i from hidden unit j, w_{ix} is the weight from the external input $x(t)$, and w_{ib} is the bias weight. $f(\cdot)$ is the nonlinear activation function $\tanh(x)$. Note that the update of the units is *layered* [5]: at each time step the hidden units are updated before the output unit.

TRAINING AND GENERALIZATION

Suppose we have a training set of related values of inputs and targets $\mathcal{T} = \{x(t), d(t)\}_{t=1}^{T}$ where T is the number of training samples. Training is done by adjusting the weights so as to minimize a cost function. Here we employ the sum of squared errors augmented by a simple weight decay regularization term

$$C(\mathbf{w}) = \frac{1}{2}\sum_{t=1}^{T} e^2(t) + \frac{\alpha}{2}|\mathbf{w}|^2 \ , \ \ e(t) = d(t) - y(t) \tag{3}$$

where $\mathbf{w}$ is the concatenated set of weights and α is a small regularization parameter. Training aims at minimizing the cost function $C(\mathbf{w})$ and is thoroughly treated for RNNs in [6].

Suppose that training provides the estimated weight vector $\widehat{\mathbf{w}}$. Let $\boldsymbol{\pi}$ be an initial state vector[1] of the "true" data generating system leading to the training set $\mathcal{T}$ and define an associated probability distribution[2] $p(\boldsymbol{\pi})$. Further, define $\mathbf{x}(t) = [x(t), x(t-1), \cdots x(T+1)]^\top$ and let $p\,(d(t), \mathbf{x}(t)\,|\,\mathcal{T}, \boldsymbol{\pi})\,,\ t > T$, be the true joint probability density function of $[d(t), \mathbf{x}(t)]$ conditioned on the initial state $\boldsymbol{\pi}$ and the training set $\mathcal{T}$. The true joint p.d.f. is assumed to be time-independent (i.e., stationary). The generalization error of the trained net is defined as the expected squared prediction error on future data *immediately* succeeding the training data, i.e., for $t > T$,

$$G(\widehat{\mathbf{w}}) = \int [d(t) - y(t; \widehat{\mathbf{w}})]^2 \cdot p\,(d(t), \mathbf{x}(t)\,|\,\mathcal{T}, \boldsymbol{\pi}) \cdot p(\boldsymbol{\pi})\; dd(t) d\mathbf{x}(t) d\boldsymbol{\pi} \qquad (4)$$

Thus the generalization error is the ensemble average of the squared error over 1) possible realizations of $[d(t), \mathbf{x}(t)]$ due to inherent stochastic processes in the data generating system, and 2) over possible initial states leading to the particular training set.

We estimate the generalization error by,

$$\widehat{G}(\widehat{\mathbf{w}}) = \frac{1}{V} \sum_{t=T+1}^{T+V} e^2(t; \widehat{\mathbf{w}}) \qquad (5)$$

where V is the number of test samples.

LEARNING CURVE

The learning curve expresses the average generalization error over all possible training sets of a particular size T as a function of T and is an important tool for verifying whether enough data is available for proper training of the network. Moreover, the shape of the curve provides insight into the nature of the problem as demonstrated in the experimental section.

Practical considerations may lead to more restricted definitions. Here we compute the learning curve as the estimated generalization error when gradually expanding the training set. That is, there is no average over different sets of a particular size.

NETWORK MEMORY

A characteristic of recurrent neural networks is their ability to build up an internal memory representing the "history" of previous inputs on which the predictions of future values is based. The significance of this internal memory is especially clear when using RNNs having only one external input. Without the ability to create internal memory this class of networks would be useless.

Once a recurrent network is trained, the basic idea here is to define an integer variable M which expresses the effective memory of past values of

[1] The initial state captures the all information about the time series for $t \leq 0$.

[2] E.g., that all initial states are equally likely.

the input signal $x(t)$. The memory thus provides a partial insight into the functionality and dynamics of the network. The experimental section gives examples of interpreting the dynamics using this simple concept. Recurrent networks with only one external input can not give individual contribution to each previous input $x(t-m)$ but must store their own representation. Consequently, the RNN has a certain *memory profile*. We are currently pursuing the idea of determining the memory profile.

A feed-forward network does not possess any internal memory, i.e., the memory is explicitly determined by the memory contained in the preprocessing of the input signal. The standard approach is to feed the signals from a tapped delay line $[x(t), x(t-1), \cdots x(t-M)]$ into the network and the memory thus equals M.

The capacity of the internal memory of a recurrent network increases when the number of hidden units (i.e., the dimension of the state vector) increases as the state vector contains all information about previous inputs. However, to our knowledge, there is no reports on quantizing the notion of memory in recurrent networks. In the following we attempt to provide a definition of the memory of a specific trained recurrent network.

The output from the RNN defined in (1), (2) is based on the current and – in principle – infinitely many previous inputs[3], as shown by,

$$y(t) = y\left(t|\widehat{\mathbf{w}}, x(t), x(t-1), \ldots, x(-\infty)\right). \tag{6}$$

In order to determine the effective *average* memory of the recurrent network we suggest to evaluate an estimate of the generalization error, i.e., prediction error on a test set, using predictions based on only a *limited* number of previous inputs. This generalization error is then compared to the error obtained using all – in principle infinitely many – previous inputs.

In particular, when evaluating the generalization error using only the m most recent inputs, we compute,

$$\widehat{G}_m(\widehat{\mathbf{w}}) = \frac{1}{V} \sum_{t=T+1}^{T+V} \left[d(t) - y\left(t|\widehat{\mathbf{w}}, x(t), x(t-1), \ldots, x(t-m)\right)\right]^2, \ m \geq 0 \tag{7}$$

where V is the size of the test set. $y\left(t|\widehat{\mathbf{w}}, x(t), x(t-1), \ldots, x(t-m)\right)$ is computed for each $t \in [T+1; T+V]$ by *resetting*[4] the states $s_i(t-m-1)$, $i = 1, 2, \cdots, N_h$, to zero and then iterate the network from time $t-m$ until time t, using the output $y(t)$ at this time as the prediction of $d(t)$. In the first iteration, calculating $y\left(t-m|\widehat{\mathbf{w}}, x(t-m)\right)$, the network thus functions as a feed-forward network since the previous states of the hidden units – and thereby all previous external inputs – have no influence on the network output. Then, the network gradually builds up a representation of the past in

[3]This is also true for a RNN in which previous values of the output is fed back to the input.

[4]Setting the hidden unit states $s_i(t-m-1)$ to zero is equivalent to erasing the memory of the network regarding inputs before time $t-m$.

the hidden units during the next $m+1$ iterations before it makes its prediction at time t.

The resulting errors $\widehat{G}_0(\widehat{\mathbf{w}}), \widehat{G}_1(\widehat{\mathbf{w}}), \ldots$ are then compared to $\widehat{G}_\infty(\widehat{\mathbf{w}})$ denoting the error obtained when using all available previous inputs, i.e., no resetting of the hidden unit states at any time. The memory M is now defined as,

$$M = \inf \left\{ m \,\middle|\, \forall\, m' \geq m,\ \frac{|\widehat{G}_{m'}(\widehat{\mathbf{w}}) - \widehat{G}_\infty(\widehat{\mathbf{w}})|}{\widehat{G}_\infty(\widehat{\mathbf{w}})} < \epsilon \right\} \tag{8}$$

where ϵ is a *small* number. Thus, the memory, M, denotes the minimal number of previous inputs beyond which additional inputs are insignificant.

The memory measure outlined above determines the number of previous inputs that the network needs knowledge about in order to obtain good predictions on *all* samples in the test set. Thus the measure can be interpreted as the *average* memory of the network. A recurrent network, however, is a *dynamic* system whose internal characteristics can be highly influenced by the nature of the input series. Especially, if the input series exhibits regions of non-stationary behavior, the network dynamics including memory must clearly be affected. Such changes in dynamics are not captured by the average memory measure and we may define a *local memory*, in accordance with (8), using a local generalization error estimate[5]

$$\widehat{G}_m(t; \widehat{\mathbf{w}}) = \frac{1}{K} \sum_{t'=t-K+1}^{t} [d(t') - y\,(t'|\widehat{\mathbf{w}}, x(t'), x(t'-1), \ldots, x(t'-m))]^2, \tag{9}$$

where $m \geq 0$, $t > T$, and $1 \leq K \leq V$ is the size of a smaller test set. Choosing K too small gives rise to a very noisy measure of the generalization error; however, in principle a good resolution of changes in memory requirement. On the other hand, increasing K improves generalization accuracy but reduces the resolution of changes in memory.

EXPERIMENTS

The proposed methods for estimating the learning curves and memory are evaluated on two chaotic time series prediction problems, viz. the laser series from the Santa Fe time series competition [9] and the artificially generated Mackey-Glass series [8].

The laser series is illustrated in the left panel of Figure 1. Let $z(t)$ denotes the series, then identification is done by training the network to perform a one step ahead prediction, i.e., we use $x(t) = z(t)$ and $d(t) = z(t+1)$. All available 10093 samples are used and scaled to zero mean and unit variance. From these data we construct a learning curve. The training series are obtained by

[5]Notice, by defining this measure for all $t > T$ some of the first values are based partly on training examples.

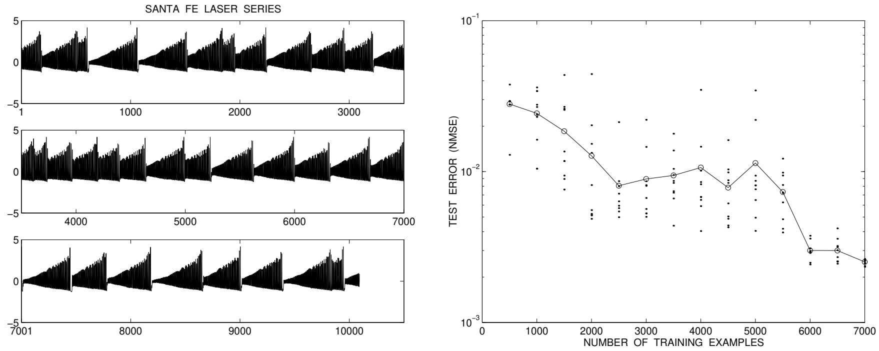


Figure 1: Left panel: The Santa Fe laser series. Right panel: Learning curve for the laser data. Dots denote error for individual nets, the connected circles indicate the average.

extending backwards in time from point 7000 and the last 3093 points in the series are used as test series. For instance, a training set of size 1000 involves training using $z(6000)$ through $z(7000)$. The employed nets have one external input and ten hidden units. For each number of increasing training set sizes, we train ten networks using different random initial weights and compute the resulting normalized mean squared error (NMSE) on the test set. NMSE is defined by

$$\text{NMSE} = \frac{|\mathcal{S}|^{-1} \sum_{t \in \mathcal{S}} e^2(t; \widehat{\mathbf{w}})}{\widehat{\text{var}}(d(t))} \tag{10}$$

where t runs over the set $\mathcal{S}$ in question (i.e., either training or test set), $|\mathcal{S}|$ is the size of the set, and $\widehat{\text{var}}(\cdot)$ denotes the empirical variance.

The learning curve is shown in the right panel of Figure 1. Initially the test error drops as the size of the training set is increased, but from training set size 2500 to 5500 the average test error is fairly constant. This can be explained by visual inspection of the laser series as the "shape" of many collapses between the corresponding points 1500–4500 seems atypical for the test series. We see a significant drop in test error when increasing the training set size from 5500 to 6000 points which might be explained by the fact that the training set now incorporates an additional collapse very similar in shape to the ones in the test series. These observations suggest that for the laser series, the concept of an example should be conceived on several time scales: there are the pointwise examples corresponding to each single input presented to the network; but more important, there obviously exists "super examples" consisting of a whole section of the time series. If additional super examples or sections are not similar to the sections encountered in the test series, generalization will not improve as seen in the right panel of Figure 1.

We now examine the memory of selected networks. The left panel of Figure 2 depicts the normalized version of Eq. (7) for increasing values of lag space m when evaluating one of the networks with low test error trained on 7000 examples. The horizontal dotted line indicates the normalized level

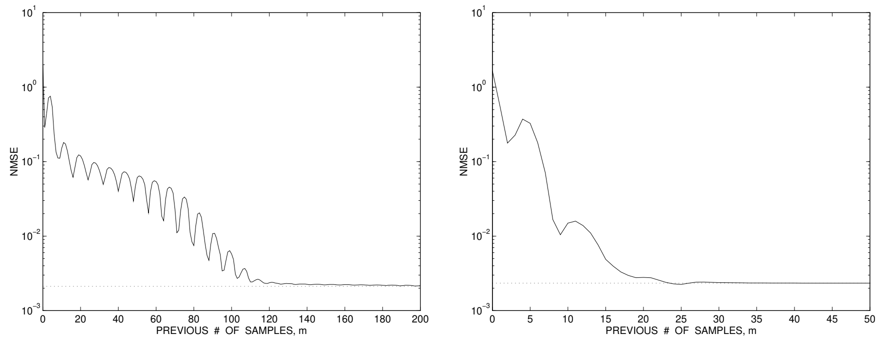


Figure 2: Left panel: Measuring *average* memory for one of the networks with low generalization error trained on 7000 examples from the laser series. Right panel: Measuring *average* memory for another of the networks trained on 7000 examples.

$\widehat{G}_\infty(\widehat{\mathbf{w}})$ using all available previous inputs. It seems that the network has a memory somewhere between 120 and 200. The precision ϵ in (8) denotes a level below which we consider the two errors as equivalent. The value of the memory thus naturally depends on the choice of ϵ as shown in Table 1. In the right panel of Figure 2 the normalized test error for increasing lag

ϵ	0.05	0.025	0.01
M	150	183	198

Table 1: The value of the memory dependence on ϵ for curve in the left panel of Figure 2.

space m for another of the nets trained on 7000 points is shown. We note that for this network the memory M is less sensitive to ϵ, as it is between 23–25 for $\epsilon \leq 0.18$. We also note that the memory is much shorter than for the previous network even though the test errors are almost identical. Note, since the network complexity[6] is restricted, a network with short memory is able to allow for more individual contribution of each of the previous inputs $x(t-n)$ than a network with long memory. The memory profile of a short term memory net is thus more fine grained than that of a long term memory net (with the same complexity). One might claim that a compact memory model is better tuned to the problem.

In the left panel of Figure 3 we illustrate the average memory of the network with lowest test error when training on only 500 examples. We notice that by limiting the memory the error can actually become lower than $\widehat{G}_\infty$. This effect often occurs for *overtrained* networks which is also the case here. The memory of the network is highly specialized on the training set; limiting the memory acts as regularization and actually improves the performance on the test set.

We now illustrate that the memory of a recurrent network indeed is a

[6] E.g., measured by the number of hidden neurons.

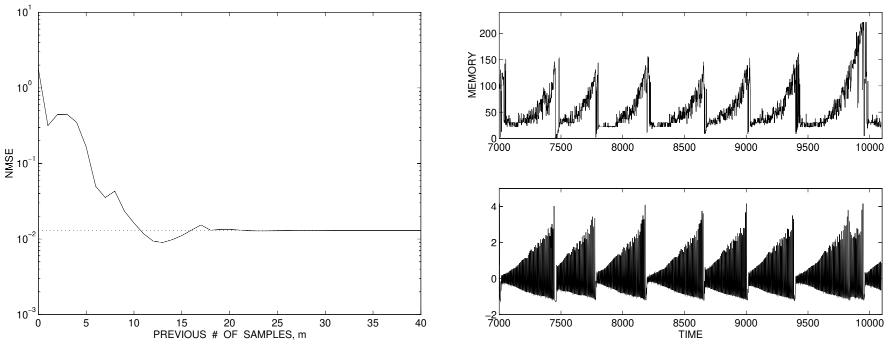


Figure 3: Left panel: Measuring *average* memory for best network trained on 500 examples from the laser series. Right panel: Measuring *local* memory with threshold $\epsilon = 0.01$ using five point average, $K = 5$.

dynamic quantity by examining the *local* memory defined by Eq. (8) and (9) for the network whose *average* memory is shown in the left panel of Figure 2. The right panel of Figure 3 and the left panel of Figure 4 illustrate the dynamic memory measure using precision $\epsilon = 0.01$ and averaging over $K = 5$ and $K = 50$ examples, respectively. The memory is seen to be very dependent upon where in the laser series it is measured; the closer to a collapse, the larger. The memory required around the last collapse is significantly larger than around the previous collapses. This may be explained by the observation that the characteristics of the laser series just before the last collapse is highly atypical from the rest of the test series. The memory in the right panel of Figure 3 averaging over only $K = 5$ previous errors is seen to be a very noisy quantity. As K is increased the error measure becomes smoother. Recall from Table 1 that the average memory for $\epsilon = 0.01$ is $M = 198$; however, the illustrations of the local memory shows that by omitting the last collapse the average memory would be measured to 150, approximately.

The Mackey-Glass series is a standard problem of nonlinear dynamics and results from the integration of a differential equation, see e.g., [8]. Standard

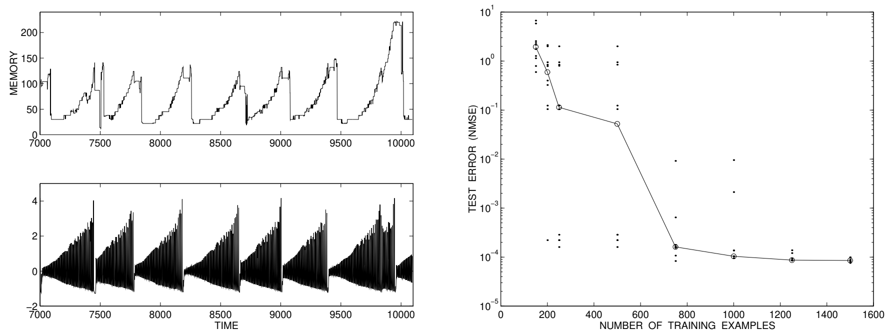


Figure 4: Left panel: Measuring *local* memory with threshold $\epsilon = 0.01$ using fifty point average, $K = 50$. Right panel: Learning curve for the Mackey-Glass series.

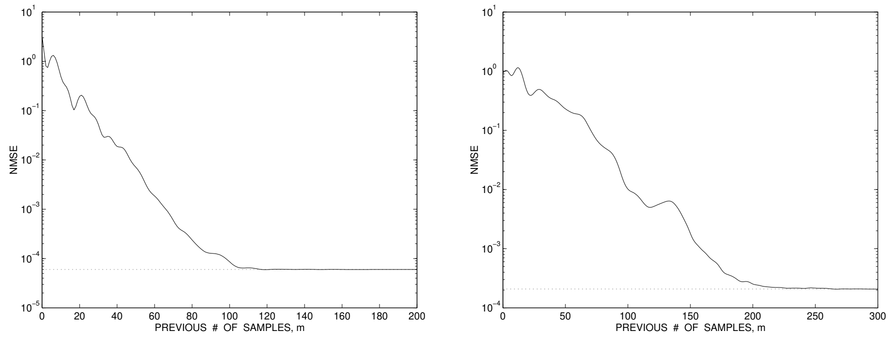


Figure 5: Measuring *average* memory for networks trained on 1500 examples from the Mackey-Glass series. Left Panel: Network having short memory. Right panel: Network having long memory.

practice is to implement a six step ahead predictor, i.e., modeling $z(t)$ from a lag space vector $\mathbf{x}(t) = [z(t-6), z(t-12), \ldots, z(t-6n_I)]$ using feed-forward networks. Here we implement the six step ahead predictor with target value $d(t) = z(t)$ using a recurrent network with only one external input, $x(t) = z(t-6)$, and ten hidden units. In the right panel of Figure 4 is shown a learning curve for the Mackey-Glass series when training on up to 1500 samples and testing on the following 7000 samples. For each training set size ten networks were trained. The learning curve indicates that more than 1000 examples are needed in order to obtain consistently good results on the test set. We then determined the average memory defined by Eq. (7) for the properly trained networks with the lowest errors on the test set. Using the threshold $\epsilon = 0.01$ we found that the networks implemented a memory in the range of 118–263, as seen from Figure 5.

The memories implemented by the recurrent networks are surprisingly long. In order to obtain comparable performance using feed-forward networks six external inputs are needed, thus spanning a total of only 31 previous samples. This is the minimal memory neccessary for good performance provided weighting of individual lags is possible, however, a RNN's memory profile is more coarse grained reducing the possibilty of individual weighting. Furthermore, maintaining information about all previous input values seems to bias recurrent networks towards the implementation of a long *effective* memory.

The long memory implemented by the recurrent networks seems to be of prime importance for the *robustness* of these models. Preliminary experiments indicate that recurrent networks are far more resilient to noise pertubations of the input data than comparable feed-forward networks. Examination of the robustness of recurrent networks is a topic of ongoing research.

CONCLUSION

In this paper we have focused on determining the *effective* memory of recurrent neural networks when used for time series processing, equivalent to the span of the externally provided lag space for feed-forward networks. In particular, we have suggested an operational definition which measures the memory of a fully trained RNN on a test set. The viability of the method is illustrated on two chaotic time series problems.

ACKNOWLEDGMENTS

This research was supported by the Danish Natural Science and Technical Research Councils through the Computational Neural Network Center (CONNECT). JL furthermore acknowledge the Radio Parts Foundation for financial support. Lars Kai Hansen is acknowledged for stimulating discussions.

REFERENCES

[1] H.D.I. Abarbanel: **Analysis of Observed Chaotic Data**, New York, NY: Springer-Verlag, 1996.

[2] M. Casey: "The Dynamics of Discrete-Time Computation, with Application to Recurrent Neural Networks and Finite State Machine Extraction," **Neural Computation**, vol. 8, pp. 1135–1178, 1996.

[3] S. Haykin: **Neural Networks: A Comprehensive Foundation**, New York, New York: Macmillan College Publishing Company, 1994.

[4] T. Lin, B.G. Horne, P. Tino & C.L. Giles: "Learning Long-term Dependencies with NARX Recurrent Neural Networks," **IEEE Transactions on Neural Networks**, vol. 7, no. 6, p. 1329, 1996.

[5] M.W. Pedersen & L.K. Hansen: "Recurrent Networks: Second Order Properties and Pruning," in G. Tesauro, D. Touretzky & T. Leen (eds.) **Advances in Neural Information Processing Systems 7**, Cambridge, MA: The MIT Press, 1995, pp. 673–680.

[6] M.W. Pedersen: "Training Recurrent Networks," in **Proceedings of the IEEE Workshop on Neural Networks for Signal Processing VII**, Piscataway, New Jersey: IEEE, 1997.

[7] H.T. Siegelmann, B.G. Horne & C.L. Giles: "Computational Capabilities of Recurrent NARX Neural Networks," **Technical Report UMIACS-TR-95-12, IEEE Transactions on Systems, Man and Cybernetics**, 1997 (in press).

[8] C. Svarer, L. K. Hansen, J. Larsen & C. E. Rasmussen: "Designer Networks for Time Series Processing," in C. A. Kamm, G. M. Kuhn, B. Yoon, R. Chellappa & S. Y. Kung (eds.), **Proceedings of the IEEE Workshop on Neural Networks for Signal Processing 3**, Piscataway, New Jersey: IEEE, pp. 78–87, 1993.

[9] A.S. Weigend, & N.A. Gershenfeld (eds.): **Time Series Prediction: Forecasting the Future and Understanding the Past**, Santa Fe Institute Studies in the Sciences of Complexity, Reading, MA: Addison-Wesley, 1993.

Appendix N

NIPS*97 submission

This appendix contains the paper "Second-Order Methods in Boltzmann Learning: An Application to Speechreading," submitted to the 1997 Neural Information Processing Systems conference. The focus of this paper is once more the application of second-order methods for training and pruning of general Boltzmann networks. An approximation to the Hessian is derived for the relative entropy cost function applied to general Boltzmann networks. The improvement in training when employing second-order methods is illustrated for Boltzmann chains applied to the identification of an HMM, and attention is directed towards the importance of regularization. Pruning is illustrated for Boltzmann zippers applied to a real-world speechreading problem. It is illustrated how pruning by OBD reduces the model complexity and improves the generalization ability. Furthermore the accuracy of the saliency estimates is illustrated, indicating the quality of the Hessian approximation.

Reference for the paper: [PS98].

Second-Order Methods in Boltzmann Learning: An Application to Speechreading

Morten With Pedersen*
Section for Digital Signal Processing
Department of Mathematical Modelling
Technical University of Denmark B321
DK-2800 Lyngby, DENMARK
mwp@imm.dtu.dk

David G. Stork
The MLP Group
Ricoh Silicon Valley
2882 Sand Hill Road Suite 115
Menlo Park, CA 94025-7022 USA
stork@crc.ricoh.com

Abstract

We introduce second-order methods for training and pruning of general Boltzmann networks trained with cross-entropy error. In particular, we derive the second derivatives for the entropic cost function. We illustrate pruning on *Boltzmann zippers*, applied to real-world data — a speechreading (lipreading) problem.

1 INTRODUCTION

Second-order methods for training feed-forward neural networks, such as Gauss-Newton, Levenberg-Marquardt and quasi-Newton algorithms [2], can greatly reduce learning time compared to first-order methods such as backpropagation. Boltzmann networks are among the slowest networks to train, and thus it is natural that we apply second-order methods to them as well. While there are several differences between traditional feed-forward networks and Boltzmann networks, at a sufficiently abstract level these networks are similar enough that the approaches developed for traditional networks can be taken over to Boltzmann networks, as we shall see.

Second-order methods have another important use in neural networks: pruning. There is an enormous body of simulation work demonstrating the value of architecture optimization for networks for pattern classification, and this has properly led to great interest in both theoretical foundations and in new algorithms. There are two basic viewpoints toward this issue: regularization (or penalty based) and sensitivity based. According to the viewpoint of regularization, one seeks to impose

*Corresponding author. Category: Algorithms and Architectures. Presentation: Oral.

some desired property in the final solution, for instance smoothness. Thus in weight decay one penalizes large weights and therefore favors smoother decision boundaries. According to the viewpoint of sensitivity, one seeks to eliminate those parameters (e.g., weights) that have the smallest effect on the training error, thereby restricting the model without severely penalizing the training error. Recent methods (special cases of the Wald statistic) eliminate weights that are predicted to have the least effect on the training error [1, 3]. In this work we introduce these methods (OBD and OBS) in the context of Boltzmann networks.

2 BOLTZMANN NETWORKS

Though typically slower and a bit more difficult to train than feedforward neural networks, Boltzmann networks nevertheless have some desirable properties: natural handling of missing data (during both training and application), pattern completion, and superior avoidance of local energy minima during training. Boltzmann networks are stochastic networks with both visible and hidden units (cf., [4] for an introduction and the notation we use here). We let the subscript α denote the states of the visible units and β the states of the hidden units. The superscript $+$ denotes iterating the network with the visible units clamped to a desired pattern, and $-$ denotes the visible units running freely, or unclamped. The energy function for the Boltzmann network is usually defined as

$$E = -\frac{1}{2}\sum_{ij} w_{ij} s_i s_j, \tag{1}$$

where w_{ij} is the (bi-directional) weight connecting units i and j, and s_i is the (binary) state of unit i. At *thermal equilibrium* the probability of finding the units in a given state configuration $\alpha\beta$ when the visible units are unclamped is given by the Boltzmann distribution

$$P^-_{\alpha\beta} = \frac{1}{Z} e^{-E_{\alpha\beta}}, \tag{2}$$

where Z is the normalizing partition function and $E_{\alpha\beta}$ is the energy (dependent on the weights) when the visible units are in states α and the hidden units are in states β; for clarity in the following, we have incorporated the *temperature* into the energy term. Thus, the probability P^-_α of finding the visible units in joint states α is found by summing over the possible hidden unit configurations β.

When training Boltzmann networks we want the probabilities of the freely running network P^-_α to match those of the environment/training examples P^+_α. As a measure of the difference between the two probability distributions we use the *Kullback-Leibler* measure, or relative entropy, as our cost function:

$$H(\mathbf{w}) = \sum_\alpha P^+_\alpha \ln \frac{P^+_\alpha}{P^-_\alpha} = -\sum_\alpha P^+_\alpha \ln P^-_\alpha + \mathit{const}, \tag{3}$$

where *const* is a constant determined solely by the environment, and is hence independent of the weights $\mathbf{w}$. When training using gradient descent we need the derivatives of $H(\mathbf{w})$ with respect to the bi-directional weights w_{ij} connecting units i and j:

$$\begin{aligned}\frac{\partial H(\mathbf{w})}{\partial w_{ij}} &= -\sum_\alpha P^+_\alpha \frac{\partial \ln P^-_\alpha}{\partial w_{ij}} = -\sum_\alpha P^+_\alpha \left\langle -\frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right\rangle^+_\alpha + \left\langle -\frac{\partial E_{\alpha\beta}}{\partial w_{ij}} \right\rangle^- \\ &= < s_i s_j >^- - < s_i s_j >^+, \end{aligned} \tag{4}$$

where $< \cdots >^+_\alpha$ is the mean value given that the visible units are clamped in states α, and $< \cdots >^-$ is the mean when all units are free running.

3 SECOND-ORDER METHODS

The first derivatives lead to the traditional first-order training methods [4]. However, for faster training methods and for pruning algorithms we need the second derivatives of the entropic cost. These second derivatives are calculated as:

$$\begin{aligned}\frac{\partial^2 H(\mathbf{w})}{\partial w_{ij}\partial w_{pq}} &= -\left[\sum_\alpha \frac{P_\alpha^+}{P_\alpha^-}\cdot\frac{\partial^2 P_\alpha^-}{\partial w_{ij}\partial w_{pq}} - \sum_\alpha \frac{\partial P_\alpha^-}{\partial w_{ij}}\cdot\frac{P_\alpha^+}{\left(P_\alpha^-\right)^2}\cdot\frac{\partial P_\alpha^-}{\partial w_{pq}}\right] \\ &\approx \sum_\alpha \frac{\partial \ln P_\alpha^-}{\partial w_{ij}}\cdot P_\alpha^+\cdot\frac{\partial \ln P_\alpha^-}{\partial w_{pq}}. \end{aligned} \tag{5}$$

The approximation in Eq. 5 is equivalent to that made in *Fisher's method of scoring* [7] and also corresponds to the Gauss-Newton approximation to the Hessian of a quadratic cost in which the term with second derivatives is ignored. As is the case for the quadratic cost function, the approximation becomes exact in the limit of infinitely many examples, provided the network is not underparametrized. In that limit, the parameters that minimize the entropic cost function will converge towards a set of *optimal* weights $\mathbf{w}^*$ for which $P_\alpha^- = P_\alpha^+$, $\forall\alpha$. For these weights the term in Eq. 5 involving second derivatives of the unclamped probabilities P_α^- reads

$$\sum_\alpha \frac{P_\alpha^+}{P_\alpha^-}\cdot\frac{\partial^2 P_\alpha^-}{\partial w_{ij}\partial w_{pq}} = \sum_\alpha \frac{\partial^2 P_\alpha^-}{\partial w_{ij}\partial w_{pq}} = \frac{\partial^2}{\partial w_{ij}\partial w_{pq}}\left(\sum_\alpha P_\alpha^-\right) = 0, \tag{6}$$

where in the first step we used the fact that $P_\alpha^+ = P_\alpha^-$ at $\mathbf{w} = \mathbf{w}^*$, and in the last step that $\sum_\alpha P_\alpha^- = 1$. Thus for weights sufficiently "close" to the optimal, $\mathbf{w}^*$, the term in Eq. 5 involving second derivatives will be "small" and the rest involves terms of a form calculated from Eq. 4:

$$\frac{\partial \ln P_\alpha^-}{\partial w_{ij}} = \left\langle -\frac{\partial E_{\alpha\beta}}{\partial w_{ij}}\right\rangle_\alpha^+ - \left\langle -\frac{\partial E_{\alpha\beta}}{\partial w_{ij}}\right\rangle^- = < s_i s_j >_\alpha^+ - < s_i s_j >^- . \tag{7}$$

Thus we obtain the simplified form of the second derivatives:

$$\begin{aligned}\frac{\partial^2 H(\mathbf{w})}{\partial w_{ij}\partial w_{pq}} &\approx \sum_\alpha \frac{\partial \ln P_\alpha^-}{\partial w_{ij}}\cdot P_\alpha^+\cdot\frac{\partial \ln P_\alpha^-}{\partial w_{pq}} \\ &= \sum_\alpha P_\alpha^+ < s_i s_j >_\alpha^+ < s_p s_q >_\alpha^+ - < s_p s_q >^- \sum_\alpha P_\alpha^+ < s_i s_j >_\alpha^+ \\ &\quad - < s_i s_j >^- \sum_\alpha P_\alpha^+ < s_p s_q >_\alpha^+ + < s_i s_j >^- < s_p s_q >^- . \end{aligned} \tag{8}$$

We note that the approximation to the second derivatives is positive definite which is an advantage when using the second derivatives for second-order training. The approximation involves only terms already computed when calculating the gradient, and thus implementation is straightforward, requiring little computation beyond that needed for computing the gradient.

3.1 Learning: The damped Gauss-Newton method

Learning in Boltzmann networks can be sped up if a more efficient optimization technique than gradient descent is used. Here we suggest the damped Gauss-Newton method,[2] in which the direction of weight changes in iteration k is computed as

$$\Delta\mathbf{w}_k = -\eta[H''(\mathbf{w}_k)]^{-1}H'(\mathbf{w}_k). \tag{9}$$

This corresponds to the direction towards the minimum of a second-order expansion of the entropic cost function around the current iteration point. The term "damped" refers to the use of a line search such as simple bisection in order to determine the step size η. The line search makes the method globally convergent and is simple if the Boltzmann network has a structure that allows for exact computation of likelihoods.

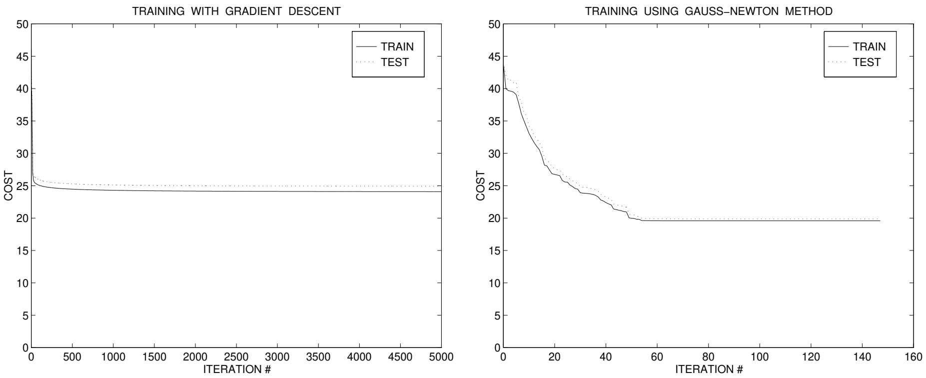


Figure 1: Learning in a Boltzmann chain (see text). Left panel: Training using gradient descent. Right panel: Training using the Gauss-Newton method.

Figure 1 compares the performance between gradient descent and the damped Gauss-Newton method when training a Boltzmann chain [6] from the same initial weights. The problem was identification of a left-to-right HMM having three observation states and three hidden states from fixed-length observation sequences generated by the HMM. In this example we observed a 30% increase in computation time per iteration when using the Gauss-Newton method; the increase is however highly justified by a much more rapid convergence. In ten runs, only once did gradient descent reach the level of error obtained using the Gauss-Newton method within 5000 iterations; the Gauss-Newton method only needed 100–200 iterations.

Regularization is crucial for the application of second-order training methods to especially Boltzmann chains as the Hessian for the entropic cost function is inherently rank-deficient for this model type. This is so since addition of an arbitrary constant to all the weights in these networks will not influence the Boltzmann distribution Eq. 2 and therefore leaves the entropic cost unchanged. Augmentation by a simple quadratic weight decay term, $\lambda/2\,\mathbf{w}^{\mathrm{T}}\mathbf{w}$, solves this problem.

3.2 Pruning: Saliency based methods

Saliency based pruning algorithms developed for feed-forward nets include Optimal Brain Damage (OBD) [1] and Optimal Brain Surgeon (OBS) [3]. Both use second-order expansions of the error to estimate the importance, or saliency, of the parameters if these are reset to zero. The rationale behind both methods is that if we remove the least salient weights according to training error, we gracefully relieve the danger of overfitting, and thereby simplify the network and (often) improve generalization.

Having derived the second derivatives for the entropic cost function allows for the application of OBD and OBS to Boltzmann networks. For simplicity, we here focus on the analog of OBD, for which the saliency for parameter j is then computed as

$$\delta H_j^{\mathrm{OBD}} = \left(\lambda + \frac{1}{2} \frac{\partial^2 H(\mathbf{w})}{\partial w_j^2} \right) w_j^2 \tag{10}$$

when working from a cost function augmented by a quadratic weight decay term. We note that the pruning methods only requires information already provided by the learning algorithms. Both pruning as well as the weight decay will bias the parameters towards the value zero and thus bias the Boltzmann distribution towards a smooth uniform distribution.

4 SPEECHREADING PROBLEM

Speechreading — audio-visual speech recognition, or more colloquially "lipreading" — refers to the use of *both* visual and audio signals for speech recognition [8]. There is ample evidence, from a number of independent research groups, that the incorporation of visual information can improve recognition accuracy, especially in noisy environments.[8]

4.1 Sensory integration

The central novel problem in speechreading is sensory integration — how to best integrate (and learn) information from the acoustic and visual channels. There are three principal methods: early, intermediate and late integration. In early integration, the (preprocessed) audio and video features are effectively concatenated to provide an expanded feature vector, which is then recognized. In late integration, there are, effectively, two separate classifiers: one for the video information, one for the audio information. The probability estimates of the categories, provided by the two channels, are then pooled for overall classification. Intermediate integration is a somewhat vague term which depends upon the fundamental recognition engine in question (HMMs, neural nets, Dynamic Time Warping, ...), but has come to mean any integration scheme "between" early and late integration.

Any speech recognition architecture must be insensitive to the variable rate of natural speech, and for this reason Hidden Markov Models have enjoyed great popularity in the speech community. However, both early and late integration using HMMs seem to be suboptimal for the automatic speechreading problem. In early integration, the acoustic and visual features are concatenated and input to a single HMM. The problem is that, for the vast majority of words, the acoustic signal requires more hidden states than the visual, and thus any *single* HMM represents a poor compromise. In late integration, one has two separate HMMs, with the appropriate, different number of hidden states, and pool their probability estimates for the final classification. However, this precludes the learning of low-level bi-modal features, for instance based on both the instantaneous visual shape of the mouth and the sound.

In short, it appears that an HMM-based architecture supporting some form of intermediate integration is needed — one where the number of hidden states can differ between the two channels, while also allowing cross-modal features to be learned. Recent work has shown how two linked Boltzmann chains [6] (which have been dubbed "Boltzmann zippers" because of their architecture) have this ability. It is for this reason we explored their use on a real-world speechreading problem.

4.2 Experiment

We constructed an isolated-word speechreading system using Boltzmann zippers for classifying the three nonsense utterances `/asklee/`, `/asklaa/` and `/askluu/` (Fig. 2). These were chosen because they illustrate well a vexing problem in speech recognition: co-articulation. The sound of the `/s/` phoneme differs dramatically in

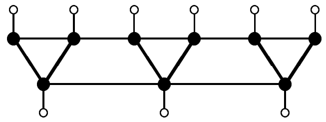


Figure 2: Topology of a Boltzmann zipper. The white circles represent groups of visible units/states; at the top for the "fast" (in speechreading, the acoustic channel), at the bottom the "slow" (visual) channel. Dark circles represent groups of hidden units/states. This architecture is formally equivalent to two HMMs, fully cross-connected by trainable weights.

these utterances, due to the influence of lip rounding associated with the different vowels coming "later". We hypothesized that, for the linked-HMM models for these utterances, interactions between the "later" visual states would have significant interactions with the "early" acoustic ones. Indeed, an analysis of the trained zipper weights confirmed this hypothesis.

Repetitions of each word were recorded (including video information) from a single talker in a low-noise environment. Each of the three sets of utterances was divided into two sets: 15 utterances for training and five for testing. The audio part of each utterance consisted of a sequence of 48 feature vectors, the video information for each utterance consisted of a sequence of 24 feature vectors. Thus, the video information was obtained at half the speed of the audio information. Each video feature vector was a collection of visual parameters representing height and width of the mouth opening, thickness of the upper lip etc. The feature vectors were converted into discrete observations using the LBG algorithm.

A Boltzmann zipper having five 'fast' hidden unit states and three 'slow' hidden unit states was trained for each of the utterances using second-order methods combined with OBD pruning. In the left panel of Fig. 3 is shown the evolution of the entropic cost for the model trained on the `/asklee/` utterances as parameters were pruned away. Initially both the training and test error decreased as parameters were removed. This was caused by the Gauss-Newton algorithm being able to locate lower-lying local minima during retraining of the reduced model. As the zipper was increasingly restricted by pruning the training error began to increase again, as is common in pruning experiments. Around 60 parameters left in the model the test error suddenly dropped to the lowest level obtained. This is explained by the

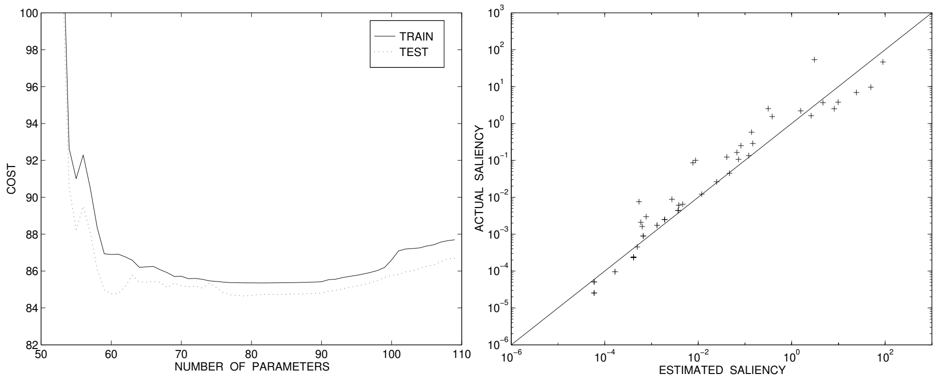


Figure 3: Left: Evolution of the relative entropy cost function for the model trained on the `/asklee/` data as the parameters of a Boltzmann zipper are pruned to the zero. Right: Estimated versus actual saliencies for the fully connected Boltzmann zipper.

zipper being sufficiently gracefully restricted and thereby reducing the possibilities of overfitting the training sequences.

In the right panel of Fig. 3 we illustrate the quality of the saliency estimates. For the fully connected zipper, we plot the estimated saliency versus the actual saliency computed by setting the weight to zero and calculating the resulting change in training error. We note that especially for low saliency weights the estimates are approximately equal to the actual saliencies. Note that rank ordering of parameters according to estimated saliency is consistent with rank ordering according to actual saliency; and thus the correct weight is pruned at each step.

Similar results were obtained for the Boltzmann zippers optimized for the `/asklaa/` and `/askluu/` utterances. The three optimal zippers were then combined into a speechreading system, which was able to correctly classify the 15 test utterances.

5 CONCLUSION AND FUTURE WORK

We have derived the second derivatives of the entropic cost function for Boltzmann networks and have applied these for training using a damped Gauss-Newton method as well as well as pruning by Optimal Brain Damage. We demonstrated the methods on Boltzmann zippers (linked HMMs) on a limited, isolated-word speechreading task. The improved speed and ease of implementation should remove some of the criticisms of these methods, and lead to greater use of this powerful architecture. Our results suggest that second-order methods, and Boltzmann zippers themselves, can be employed on larger, more complex pattern recognition problems as well.

Acknowledgments

This work was completed partly during a visit to the Ricoh California Research Center, supported by the Danish Natural Science and Technical Research Councils through the Computational Neural Network Center (CONNECT). The authors would like to thank Lars Kai Hansen and Greg Wolff for support, and Lawrence Saul for valuable discussions and for making available his Boltzmann zipper code.

References

[1] Y. L. Cun, J. S. Denker and S. A. Solla, "Optimal Brain Damage," in D. Touretzky, ed., **Advances in Neural Information Processing Systems**, Morgan Kaufmann, San Mateo, CA, 1990, vol. 2, pp. 598–605.

[2] P. E. Gill, W. Murray and M. H. Wright, **Practical Optimization**, London: Academic Press, 1981.

[3] B. Hassibi and D. G. Stork, "Second Order Derivatives for Network Pruning: Optimal Brain Surgeon," in S. J. Hanson, J. D. Cowan and C. L. Giles, eds., **Advances in Neural Information Processing Systems**, Morgan Kaufmann, San Mateo, CA, 1993, vol. 5, pp. 164–171.

[4] S. Haykin, **Neural Networks. A Comprehensive Foundation**, New York: Macmillan College Publishing, 1994.

[5] D. J. C. MacKay, "Equivalence of Linear Boltzmann Chains and Hidden Markov Models," **Neural Computation**, vol. 8, no. 1, pp. 178–181, 1996.

[6] L. K. Saul and M. I. Jordan, "Boltzmann Chains and Hidden Markov Models," in G. Tesauro, D. Touretzky and T. Leen, eds., **Advances in Neural Information Processing Systems**, The MIT Press, 1995, vol. 7, pp. 435–442.

[7] G. A. F. Seber and C. J. Wild, **Nonlinear Regression**, New York: John Wiley & Sons, 1989.

[8] D. G. Stork and M. E. Hennecke, eds., **Speechreading by Humans and Machines**, New York: Springer-Verlag, 1996.

Bibliography

[Aba96] H. D. I. Abarbanel. *Analysis of Observed Chaotic Data.* Springer-Verlag, New York, NY, 1996.

[AHS85] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A Learning Algorithm for Boltzmann Machines. *Cognitive Science*, (9):147–169, 1985.

[AK89] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines.* John Wiley, Chichester, 1989.

[Aka69] H. Akaike. Fitting Autoregressive Models for Prediction. *Annals of the Institute of Statistical Mathematics*, 21:243–247, 1969.

[BGM94] M. Bianchini, M. Gori, and M. Maggini. On the Problem of Local Minima in Recurrent Neural Networks. *IEEE Transactions on Neural Networks*, 5(2):167–177, March 1994.

[Bis95] C. M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, 1995.

[Bjö96] Å. Björck. *Numerical Methods for Least Squares Problems.* SIAM, Philadelphia, PA, 1996.

[BSF94] Y. Bengio, P. Simard, and P. Frasconi. Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[BT91] A. D. Back and A. C. Tsoi. FIR and IIR Synapses, a New Neural Network Architecture for Time Series Modeling. *Neural Computation*, 3(3):375–385, 1991.

[Cas92] M. Casdagli. A Dynamical Systems Approach to Modeling Input-Output Systems. In M. Casdagli and S. Eubank, editors, *Nonlinear Modeling and Forecasting*, volume XII of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 265–281. Addison-Wesley, Redwood City, 1992.

[CBD+90] Y. Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L.D. Jackel. Handwritten Digit Recognition with a Backpropagation Network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 396–404. Morgan Kaufmann, San Mateo, CA, 1990.

[CDS90] Y. Le Cun, J. S. Denker, and S. A. Solla. Optimal Brain Damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 598–605. Morgan Kaufmann, San Mateo, CA, 1990.

[CEFG91] M. Casdagli, S. Eubank, J. D. Farmer, and J. Gibson. State Space Reconstruction in the Presence of Noise. *Physica D*, 51:52–98, 1991.

[CFP94] G. Castellano, A. M. Fanelli, and M. Pelillo. Pruning in Recurrent Neural Networks. In M. Marinaro and P. G. Morasso, editors, *International Conference on Artificial Neural Networks ICANN'94 Sorrento*, pages 451–454. Springer, 1994.

[CKS91] Y. Le Cun, I. Kanter, and S. A. Solla. Eigenvalues of Covariance Matrices: Application to Neural-Network Learning. *Physical Review Letters*, 66(18):2396–2399, 1991.

[Cyb89] G. Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control Signals, and Systems*, 2:303–314, 1989.

[DS83] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[dVP92] B. de Vries and J. Principe. The Gamma Model - A New Neural Network for Temporal Processing. *Neural Networks*, 5(4):565–576, 1992.

[Elm90] J. L. Elman. Finding Structure in Time. *Cognitive Science*, 14:179–211, 1990.

[FGS92] P. Frasconi, M. Gori., and G. Soda. Local Feedback Multilayered Networks. *Neural Computation*, 4(1):120–130, 1992.

[Fun89] K. Funahashi. On the Approximate Realization of Continuous Mappings by Neural Networks. *Neural Networks*, 2:183–192, 1989.

[GBD92] S. Geman, E. Bienenstock, and R. Doursat. Neural Networks and the Bias/Variance Dilemma. *Neural Computation*, 4(1):1–58, 1992.

[GHK+93] J. Gorodkin, L. K. Hansen, A. Krogh, C. Svarer, and O. Winther. A Quantitative Study of Pruning by Optimal Brain Damage. *International Journal of Neural Systems*, 4:159–171, 1993.

[GL96] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[GLH97] C. L. Giles, T. Lin, and B. G. Horne. Remembering the Past: The Role of Embedded Memory in Recurrent Neural Network Architectures. In *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing VII*, Piscataway, New Jersey, 1997. IEEE. To Appear.

[GMW81] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, London, 1981.

[GO94] C. L. Giles and C. W. Omlin. Pruning Recurrent Neural Networks for Improved Generalization Performance. *IEEE Transactions on Neural Networks*, 5(5):848–851, September 1994.

[Gol91] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[Gou97] C. Goutte. Extracting the Relevant Delays in Time Series Modelling. In *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing VII*, Piscataway, New Jersey, 1997. IEEE. To Appear.

[GP83] P. Grassberger and I. Procaccia. Measuring the Strangeness of Strange Attractors. *Physica D*, 9:189–208, 1983.

[Gra84] R. M. Gray. Vector Quantization. *IEEE ASSP Magazine*, pages 4–29, April 1984.

[Hay88] J. P. Hayes. *Computer Architecture and Organization.* Computer Science Series. McGraw-Hill, Singapore, second edition, 1988.

[Hay94] S. Haykin. *Neural Networks. A Comprehensive Foundation.* Macmillan College Publishing, New York, 1994.

[Hen96] M. E. Hennecke. *Audio-Visual Speech Recognition: Preprocessing, Learning and Sensory Integration.* PhD thesis, Stanford University, 1996.

[HKP91] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, Redwood City, CA, 1991.

[HMPHL96] M. Hintz-Madsen, M. W. Pedersen, L. K. Hansen, and J. Larsen. Design and Evaluation of Neural Classifiers. In *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing VI*, pages 223–232, Piscataway, New Jersey, 1996. IEEE.

[Hop82] J. J. Hopfield. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *Proceedings of the National Academy of Sciences, USA*, 79:2554–2558, 1982.

[HP94] L. K. Hansen and M. W Pedersen. Controlled Growth of Cascade Correlation Nets. In M. Marinaro and P. G. Morasso, editors, *International Conference on Artificial Neural Networks ICANN'94 Sorrento*, pages 797–801. Springer, 1994.

[HR94] L. K. Hansen and C. E. Rasmussen. Pruning from Adaptive Regularization. *Neural Computation*, 6(6):1223–1232, 1994.

[HRSL94] L. K. Hansen, C. E. Rasmussen, C. Svarer, and J. Larsen. Adaptive Regularization. In *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing IV*, pages 78–87, Piscataway, New Jersey, 1994. IEEE.

[HS93] B. Hassibi and D. G. Stork. Second Order Derivatives for Network Pruning: Optimal Brain Surgeon. In Stephen José Hanson, Jack D. Cowan, and C. Lee Giles, editors, *Advances in Neural Information Processing Systems*, volume 5, pages 164–171. Morgan Kaufmann, San Mateo, CA, 1993.

[HS95] S. Hochreiter and J. Schmidhuber. Long Short Term Memory. Technical Report FKI-207-95, Fakultät für Informatik, Technische Universität München, München, 1995.

[HSP96] M. E. Hennecke, D. G. Stork, and K. V. Prasad. Visionary Speech: Looking Ahead to Practical Speechreading Systems. In D. G. Stork and M. E. Hennecke, editors, *Speechreading by Humans and Machines*, volume 150 of *NATO ASI Series, Series F: Computer and Systems Science*, pages 331–350. Springer Verlag, Berlin, 1996.

[HSW89] K. Hornik, M. Stinchcombe, and H. White. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*, 2:359–366, 1989.

[Hüb93] U. Hübner. Lorenz-like Chaos in NH_3-FIR Lasers. In A. S. Weigend and N. A. Gershenfeld, editors, *Time Series Prediction: Forecasting the Future and Understanding the Past*, pages 73–104. Addison–Wesley, Reading, MA, 1993.

[Jor88] M. I. Jordan. Supervised Learning and Systems with Excess Degrees of Freedom. In *Proceedings of the 1988 Connectionist Models Summer School*, pages 62–75, San Mateo, CA, 1988. Morgan Kaufmann.

[JR91] B. Juang and L. R. Rabiner. Hidden Markov Models for Speech Recognition. *Technometrics*, 33(3):251–272, August 1991.

[KBM$^+$95] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden Markov Models in Computational Biology: Applications to Protein Modeling. *Journal of Molecular Biology*, (235):1501–1531, 1995.

[Koh78] Z. Kohavi. *Switching and Finite State Automata.* Tata McGraw-Hill Publishing Company, New Delhi, second edition, 1978.

[KZM94] G. Kechriotis, E. Zervas, and E. S. Manolakos. Using Recurrent Neural Networks for Adaptive Communication Channel Equalization. *IEEE Transactions on Neural Networks*, 5(2):267–278, March 1994.

[Lar93] J. Larsen. *Design of Neural Network Filters.* PhD thesis, Technical University of Denmark, Electronics Institute, March 1993.

[LB85] I. J. Leontaritis and S. A. Billings. Input-Output Parametric Models for Non-linear Systems. *International Journal of Control*, 41:303–344, 1985.

[LGHK97] T. Lin, C. L. Giles, B. G. Horne, and S. Y. Kung. A Delay Damage Model Selection Algorithm for NARX Neural Networks. *IEEE Transactions on Signal Processing*, 1997. Accepted. Special Issue on Neural Network Applications to Signal Processing.

[LH94] J. Larsen and L. K. Hansen. Generalization Performance of Regularized Neural Network Models. In J. Vlontzos, J.N. Whang, and E. Wilson, editors, *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing IV*, pages 42–51, Piscataway, New Jersey, 1994. IEEE.

[LHG96] T. Lin, B. G. Horne, and C. L. Giles. How Memory Orders Affect the Performance of NARX Networks. Technical Report UMIACS-TR-96-76 and CS-TR-3706, Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland, 1996.

[LHP$^+$96] N. Lange, L. K. Hansen, M. W. Pedersen, R. L. Savoy, and S. C. Strother. A Concordance Correlation Coefficient for Reproducibility of Spatial Activation Patterns. In A. W. Toga, R. S. J. Frackowiak, and J. C. Mazziotta, editors, *Second International Conference on Functional Mapping of the Human Brain*, number 3 in NeuroImage, page S75. Academic Press, Orlando, FL, 1996.

[LHSO96] J. Larsen, L. K. Hansen, C. Svarer, and M. Ohlsson. Design and Regularization of Neural Networks: The Optimal Use of a Validation Set. In S. Usui, Y. Tohkura, S. Katagiri, and E. Wilson, editors, *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing VI*, pages 62–71, Piscataway, New Jersey, 1996. IEEE.

[LHTG96] T. Lin, B. G. Horne, P. Tiño, and C. L. Giles. Learning Long-Term Dependencies is not as Difficult with NARX Networks. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 577–583. The MIT Press, 1996.

[Lju87] L. Ljung. *System Identification. Theory for the User.* Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[LSAH97] J. Larsen, C. Svarer, L. N. Andersen, and L. K. Hansen. Adaptive Regularization in Neural Network Modeling. In G. B. Orr, K. Müller, and R. Caruana, editors, *Book of Tricks.* Springer-Verlag, 1997. In press.

[LT89] L. S. Liebovitch and T. Toth. A Fast Algorithm to Determine Fractal Dimension by Box Counting. *Physics Letters A*, 141:386–390, 1989.

[Mac96] D. J. C. MacKay. Equivalence of Linear Boltzmann Chains and Hidden Markov Models. *Neural Computation*, 8(1):178–181, 1996.

[MD89] J. Moody and C. J. Darken. Fast Learning in Networks of Locally-Tuned Processing Units. *Neural Computation*, 1(2):281–294, 1989.

[MG77] M. C. Mackey and L. Glass. Oscillation and Chaos in Physiological Control Systems. *Science*, 197:287, 1977.

[Møl93] M. Møller. *Efficient Training of Feed-Forward Neural Networks.* PhD thesis, Aarhus University, Computer Science Department, Denmark, 1993.

[Moo91] J. Moody. Note on Generalization, Regularization and Architecture Selection. In *Proceedings of the First IEEE Workshop on Neural Networks for Signal Processing*, pages 1–10, Piscataway, New Jersey, 1991. IEEE.

[Moo92] J. E. Moody. The Effective Number of Parameters: An Analysis of Generalization and Regularization in Nonlinear Learning Systems. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems*, volume 4, pages 847–854. Morgan Kaufmann Publishers, Inc., 1992.

[Moz93] M. C. Mozer. Neural Net Architectures for Temporal Sequence Processing. In A. S. Weigend and N. A. Gershenfeld, editors, *Time Series Prediction:*

Forecasting the Future and Understanding the Past, pages 243–264. Addison–Wesley, Reading, MA, 1993.

[MP43] W. S. McCulloch and W. Pitts. A Logical Calculus of Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[Nør96] P. M. Nørgaard. *System Identification and Control with Neural Networks.* PhD thesis, Technical University of Denmark, Department of Automation, June 1996.

[NRRU+94] O. Nerrand, P. Roussel-Ragot, D. Urbani, L. Personnaz, and G. Dreyfus. Training Recurrent Neural Networks: Why and How? An Illustration in Dynamical Process Modeling. *IEEE Transactions on Neural Networks*, 5(2):178–184, March 1994.

[OS89] A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[PA87] C. Peterson and J. R. Anderson. A Mean Field Theory Learning Algorithm for Neural Networks. *Complex Systems*, (1):995–1019, 1987.

[Pap84] A. Papoulis. *Probability, Random Variables, and Stochastic Processes.* McGraw Hill, New York, 1984.

[Par92] J. E. Parkum. *Recursive Identification of Time-Varying Systems.* PhD thesis, Technical University of Denmark, Department of Mathematical Modelling, 1992.

[PC89] T. S. Parker and L. O. Chua. *Practical Numerical Algorithms for Chaotic Systems.* Springer-Verlag, New York, 1989.

[Ped94] M. W. Pedersen. Tidsserieprædiktion med Rekursive Neurale Netværk. Master's thesis, Technical University of Denmark, Section for Digital Signal Processing, Department of Mathematical Modelling, August 1994. In Danish.

[Ped97] M. W. Pedersen. Training Recurrent Networks. In *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing VII*, Piscataway, New Jersey, 1997. IEEE. To Appear.

[PF94] G. V. Puskorius and L. A. Feldkamp. Neurocontrol of Nonlinear Dynamical Systems with Kalman Filter Trained Recurrent Networks. *IEEE Transactions on Neural Networks*, 5(2):279–297, March 1994.

[PFTV92] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, second edition, 1992.

[PH95] M. W. Pedersen and L. K. Hansen. Recurrent Networks: Second Order Properties and Pruning. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 673–680. The MIT Press, 1995.

[PHL96] M. W. Pedersen, L. K. Hansen, and J. Larsen. Pruning with Generalization Based Weight Saliencies: γOBD, γOBS. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 521–527. The MIT Press, 1996.

[PL97] M. W. Pedersen and J. Larsen. Interpretation of Recurrent Neural Networks. In *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing VII*, Piscataway, New Jersey, 1997. IEEE. To Appear.

[PS89] C. Peterson and C. Söderberg. A New Method for Mapping Optimization Problems onto Neural Networks. *International Journal of Neural Systems*, 1(1):3–22, 1989.

[PS93] F. J. Pineda and J. C. Sommerer. Estimating Generalized Dimensions and Choosing Time Delays: A Fast Algorithm. In A. S. Weigend and N. A. Gershenfeld, editors, *Time Series Prediction: Forecasting the Future and Understanding the Past*, pages 367–385. Addison–Wesley, Reading, MA, 1993.

[PS96] M. W. Pedersen and D. G. Stork. Pruning Boltzmann Networks and Hidden Markov Models. In *Proceedings of the 30th Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 258–261, Pacific Grove, CA, November 1996. IEEE Press.

[PS98] M. W. Pedersen and D. G. Stork. Second-Order Methods in Boltzmann Learning: An Application to Speechreading. In *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998. Submitted.

[Rab89] L. R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[Ras93] C. E. Rasmussen. Generalization in Neural Networks. Master's thesis, Technical University of Denmark, Electronics Institute, August 1993.

[RF91] T. Robinson and F. Fallside. A Recurrent Error Propagation Network Speech Recognition System. *Computer Speech and Language*, 5:259–274, 1991.

[Rip96] B. D. Ripley. *Pattern Recognition and Neural Networks.* Cambridge University Press, Cambridge, UK, 1996.

[RK97] S. K. Riis and A. Krogh. Hidden Neural Networks: A Framework for HMM/NN Hybrids. In *Proceedings of the 1997 IEEE International Conference on Acoustics, Speech & Signal Processing*, pages 3233–3236, Munich, Germany, 1997. IEEE.

[RM86] D. E. Rumelhart and J. L. McClelland, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1. Foundations.* The MIT Press, Cambridge, MA, 1986.

[SBC93] S. Saarinen, R. Bramley, and G. Cybenko. Ill-Conditioning in Neural Network Training Problems. *SIAM Journal on Scientific Computing*, 14:693–714, 1993.

[Sen73] E. Seneta. *Non-Negative Matrices.* Springer-Verlag, New York, 1973.

[SHG97] H. T. Siegelmann, B. G. Horne, and C. L. Giles. Computational Capabilities of Recurrent NARX Neural Networks. *IEEE Trans. on Systems, Man and Cybernetics*, 27(2):209, 1997.

[SHL93] C. Svarer, L. K. Hansen, and J. Larsen. On Design and Evaluation of Tapped Delay Line Networks. In *Proceedings of the 1993 IEEE International Conference on Neural Networks*, pages 46–51, San Francisco, 1993.

[SHLR93] C. Svarer, L. K. Hansen, J. Larsen, and C. E. Rasmussen. Designer Networks for Time Series Processing. In *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing III*, pages 78–87, Piscataway, New Jersey, 1993. IEEE.

[SJ94] L. K. Saul and M. I. Jordan. Learning in Boltzmann Trees. *Neural Computation*, 6(6):1174–1184, 1994.

[SJ95] L. K. Saul and M. I. Jordan. Boltzmann Chains and Hidden Markov Models. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 435–442. The MIT Press, 1995.

[Sjö95] J. Sjöberg. *Non-Linear System Identification with Neural Networks.* PhD thesis, Linköping University, Division of Automatic Control, Sweden, 1995.

[SL91] D. R. Seidl and R. D. Lorenz. A Structure by which a Recurrent Neural Network Can Approximate a Nonlinear Dynamic System. In *Proceedings of the International Joint Conference on Neural Networks 1991*, volume II, pages 709–714, July 1991.

[SL96] D. G. Stork and H. Lu. Speechreading by Boltzmann Zippers. In *Machines that Learn*, Snowbird, UT, April 1996.

[Sør94] O. Sørensen. *Neural Networks in Control Applications.* PhD thesis, Aalborg University, Department of Control Engineering, 1994.

[SSSD90] D. B. Schwartz, S. A. Solla, V. K. Samalam, and J. S. Denker. Exhaustive Learning. *Neural Computation*, 2(3):374–385, 1990.

[Sto97] D. G. Stork, editor. *HAL's Legacy: 2001's Computer as Dream and Reality.* MIT Press, New York, 1997.

[SUH90] K. Stokbro, D. K. Umberger, and J. A. Hertz. Exploiting Neurons with Localized Receptive Fields to Learn Chaos. *Complex Systems*, 4:603–622, 1990.

[Sva94] C. Svarer. *Neural Networks for Signal Processing.* PhD thesis, Technical University of Denmark, Electronics Institute, December 1994.

[SW89] G. A. F. Seber and C. J. Wild. *Nonlinear Regression.* John Wiley & Sons, New York, 1989.

[SYC91] T. Sauer, J. A Yorke, and M. Casdagli. Embedology. *Journal of Statistical Physics*, 65(3/4):579–618, 1991.

[Tak81] F. Takens. Detecting Strange Attractors in Turbulence. In *Dynamical Systems and Turbulence, Warwick 1980*, volume 898 of *Lecture Notes in Mathematics*, pages 366–381. Springer-Verlag, 1981.

[TB94] A. C. Tsoi and A. D. Back. Locally Recurrent Globally Feedforward Networks: A Critical Review of Architectures. *IEEE Transactions on Neural Networks*, 5(2):229–239, March 1994.

[Tho91] H. H. Thodberg. Improving Generalization of Neural Networks Through Pruning. *International Journal of Neural Systems*, 1(4):317–326, 1991.

[Wan90] E. A. Wan. Temporal Backpropagation for FIR Neural Networks. In *Proceedings of International Joint Conference on Neural Networks, San Diego*, pages 575–580, Piscataway, NY, 1990. IEEE Service Center.

[Wan93] E. A. Wan. Time Series Prediction by Using a Connectionist Network with Internal Delay Lines. In A. S. Weigend and N. A. Gershenfeld, editors, *Time Series Prediction: Forecasting the Future and Understanding the Past*, pages 195–217. Addison–Wesley, Reading, MA, 1993.

[WG93] A. S. Weigend and N. A. Gershenfeld, editors. *Time Series Prediction: Forecasting the Future and Understanding the Past.* Addison-Wesley, Reading, MA, 1993.

[Whi89] H. White. Learning in Artificial Neural Networks: A Statistical Perspective. *Neural Computation*, 1(4):425–464, 1989.

[WHR90] A. S. Weigend, B. A. Huberman, and D. E. Rumelhart. Predicting the Future: A Connectionist Approach. *International Journal of Neural Systems*, 1:193–209, 1990.

[WHR92] A. S. Weigend, B. A. Huberman, and D. E. Rumelhart. Predicting Sunspots and Exchange Rates with Connectionist Networks. In M. Casdagli and S. Eubank, editors, *Nonlinear Modeling and Forecasting*, pages 395–432, Redwood City, CA, 1992. Addison-Wesley.

[Wil63] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1963.

[WM96a] L. Wu and J. Moody. A Smoothing Regularizer for Feedforward and Recurrent Neural Networks. *Neural Computation*, 8(3):461–489, 1996.

[WM96b] L. Wu and J. Moody. A Smoothing Regularizer for Recurrent Neural Networks. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 458–464. The MIT Press, 1996.

[WP90] R. J. Williams and J. Peng. An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories. *Neural Computation*, 2(4):490–501, 1990.

[WR91] A. S. Weigend and D. E. Rumelhart. The Effective Dimension of the Space of Hidden Units. In *Proceedings of International Joint Conference on Neural Networks, Singapore*, pages 2069–2074, Piscataway, NY, 1991. IEEE Service Center.

[WS85] B. Widrow and S. D. Stearns. *Adaptive Signal Processing.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[Wu83] F. Y. Wu. The Potts Model. *Rev. Mod. Phys.*, 54:235, 1983.

[Wul92] N. H. Wulff. Learning Dynamics with Recurrent Networks. Master's thesis, Niels Bohr Institute, Blegdamsvej 17, 2100 Copenhagen Ø, Denmark, April 1992.

[WZ89] R. J. Williams and D. Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1:270–280, 1989.

[Yul27] G. Yule. On a Method of Investigating Periodicity in Disturbed Series with Special Reference to Wolfer's Sunspot Numbers. *Phil. Trans. Roy. Soc. London*, A 226:267–298, 1927.