

Lecture Notes in Computer Science

7700

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

Grégoire Montavon
Geneviève B. Orr
Klaus-Robert Müller (Eds.)

Neural Networks: Tricks of the Trade

Second Edition



Volume Editors

Grégoire Montavon
Technische Universität Berlin
Department of Computer Science
Franklinstr. 28/29, 10587 Berlin, Germany
E-mail: gregoire.montavon@tu-berlin.de

Geneviève B. Orr
Willamette University
Department of Computer Science
900 State Street, Salem, OR 97301, USA
E-mail: gorr@willamette.edu

Klaus-Robert Müller
Technische Universität Berlin
Department of Computer Science
Franklinstr. 28/29, 10587 Berlin, Germany
and
Korea University
Department of Brain and Cognitive Engineering
Anam-dong, Seongbuk-gu, Seoul 136-713, Korea
E-mail: klaus-robert.mueller@tu-berlin.de

ISSN 0302-9743

e-ISSN 1611-3349

ISBN 978-3-642-35288-1

e-ISBN 978-3-642-35289-8

DOI 10.1007/978-3-642-35289-8

Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012952591

CR Subject Classification (1998): F.1, I.2.6, I.5.1, C.1.3, F.2, J.3

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

© Springer-Verlag Berlin Heidelberg 1998, 2012

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface to the Second Edition

There have been substantial changes in the field of neural networks since the first edition of this book in 1998. Some of them have been driven by external factors such as the increase of available data and computing power. The Internet made public massive amounts of labeled and unlabeled data. The ever-increasing raw mass of user-generated and sensed data is made easily accessible by databases and Web crawlers. Nowadays, anyone having an Internet connection can parse the 4,000,000+ articles available on Wikipedia and construct a dataset out of them. Anyone can capture a Web TV stream and obtain days of video content to test their learning algorithm.

Another development is the amount of available computing power that has continued to rise at steady rate owing to progress in hardware design and engineering. While the number of cycles per second of processors has thresholded due to physics limitations, the slow-down has been offset by the emergence of processing parallelism, best exemplified by the massively parallel graphics processing units (GPU). Nowadays, everybody can buy a GPU board (usually already available in consumer-grade laptops), install free GPU software, and run computation-intensive simulations at low cost.

These developments have raised the following question: Can we make use of this large computing power to make sense of these increasingly complex datasets? Neural networks are a promising approach, as they have the intrinsic modeling capacity and flexibility to represent the solution. Their intrinsically distributed nature allows one to leverage the massively parallel computing resources.

During the last two decades, the focus of neural network research and the practice of training neural networks underwent important changes. Learning in deep (or “deep learning”) has to a certain degree displaced the once more prevalent regularization issues, or more precisely, changed the practice of regularizing neural networks. Use of unlabeled data via unsupervised layer-wise pretraining or deep unsupervised embeddings is now often preferred over traditional regularization schemes such as weight decay or restricted connectivity. This new paradigm has started to spread over a large number of applications such as image recognition, speech recognition, natural language processing, complex systems, neuroscience, and computational physics.

The second edition of the book *reloads* the first edition with more tricks. These tricks arose from 14 years of theory and experimentation (from 1998 to 2012) by some of the world’s most prominent neural networks researchers. These tricks can make a substantial difference (in terms of speed, ease of implementation, and accuracy) when it comes to putting algorithms to work on real problems. Tricks may not necessarily have solid theoretical foundations or formal validation. As Yoshua Bengio states in Chap. 19, “the wisdom distilled here should be taken as a guideline, to be tried and challenged, not as a practice set in stone” [1].

The second part of the new edition starts with tricks to faster optimize neural networks and make more efficient use of the potentially infinite stream of data presented to them. Chapter 18 [2] shows that a simple stochastic gradient descent (learning one example at a time) is suited for training most neural networks. Chapter 19 [1] introduces a large number of tricks and recommendations for training feed-forward neural networks and choosing the multiple hyperparameters.

When the representation built by the neural network is highly sensitive to small parameter changes, for example, in recurrent neural networks, second-order methods based on mini-batches such as those presented in Chap. 20 [9] can be a better choice. The seemingly simple optimization procedures presented in these chapters require their fair share of tricks in order to work optimally. The software *Torch7* presented in Chap. 21 [5] provides a fast and modular implementation of these neural networks.

The novel second part of this volume continues with tricks to incorporate invariance into the model. In the context of image recognition, Chap. 22 [4] shows that translation invariance can be achieved by learning a k-means representation of image patches and spatially pooling the k-means activations. Chapter 23 [3] shows that invariance can be injected directly in the input space in the form of elastic distortions. Unlabeled data are ubiquitous and using them to capture regularities in data is an important component of many learning algorithms. For example, we can learn an unsupervised model of data as a first step, as discussed in Chaps. 24 [7] and 25 [10], and feed the unsupervised representation to a supervised classifier. Chapter 26 [12] shows that similar improvements can be obtained by learning an unsupervised embedding in the deep layers of a neural network, with added flexibility.

The book concludes with the application of neural networks to modeling time series and optimal control systems. Modeling time series can be done using a very simple technique discussed in Chap. 27 [8] that consists of fitting a linear model on top of a “reservoir” that implements a rich set of time series primitives. Chapter 28 [13] offers an alternative to the previous method by directly identifying the underlying dynamical system that generates the time series data. Chapter 29 [6] presents how these system identification techniques can be used to identify a Markov decision process from the observation of a control system (a sequence of states and actions in the reinforcement learning terminology). Chapter 30 [11] concludes by showing how the control system can be dynamically improved by fitting a neural network as the control system explores the space of states and actions.

The book intends to provide a timely snapshot of tricks, theory, and algorithms that are of use. Our hope is that some of the chapters of the new second edition will become our companions when doing experimental work—eventually becoming classics, as some of the papers of the first edition have become. Eventually in some years, there may be an urge to reload again...

Acknowledgments. This work was supported by the World Class University Program through the National Research Foundation of Korea funded by the Ministry of Education, Science, and Technology, under Grant R31-10008. The editors also acknowledge partial support by DFG (MU 987/17-1).

References

- [1] Bengio, Y.: Practical Recommendations for Gradient-based Training of Deep Architectures. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 437–478. Springer, Heidelberg (2012)
- [2] Bottou, L.: Stochastic Gradient Descent Tricks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 421–436. Springer, Heidelberg (2012)
- [3] Ciresan, D.C., Meier, U., Gambardella, L.M., Schmidhuber, J.: Deep Big Multilayer Perceptrons for Digit Recognition. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 581–598. Springer, Heidelberg (2012)
- [4] Coates, A., Ng, A.Y.: Learning Feature Representations with k-means. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 561–580. Springer, Heidelberg (2012)
- [5] Collobert, R., Kavukcuoglu, K., Farabet, C.: Implementing Neural Networks Efficiently. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 537–557. Springer, Heidelberg (2012)
- [6] Duell, S., Udluft, S., Sterzing, V.: Solving Partially Observable Reinforcement Learning Problems with Recurrent Neural Networks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 687–707. Springer, Heidelberg (2012)
- [7] Hinton, G.E.: A Practical Guide to Training Restricted Boltzmann Machines. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 621–637. Springer, Heidelberg (2012)
- [8] Lukoševičius, M.: A Practical Guide to Applying Echo State Networks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 659–686. Springer, Heidelberg (2012)
- [9] Martens, J., Sutskever, I.: Training Deep and Recurrent Networks with Hessian-free Optimization. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 479–535. Springer, Heidelberg (2012)
- [10] Montavon, G., Müller, K.-R.: Deep Boltzmann Machines and the Centering Trick. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 621–637. Springer, Heidelberg (2012)
- [11] Riedmiller, M.: 10 Steps and Some Tricks to Set Up Neural Reinforcement Controllers. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 735–757. Springer, Heidelberg (2012)
- [12] Weston, J., Ratle, F., Collobert, R.: Deep Learning Via Semi-supervised Embedding. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 639–655. Springer, Heidelberg (2012)
- [13] Zimmermann, H.-G., Tietz, C., Grothmann, R.: Forecasting with Recurrent Neural Networks: 12 Tricks. In: NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 687–707. Springer, Heidelberg (2012)

Table of Contents

Introduction	1
Speeding Learning	
Preface	7
1. Efficient BackProp	9
<i>Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus-Robert Müller</i>	
Regularization Techniques to Improve Generalization	
Preface	49
2. Early Stopping — But When?	53
<i>Lutz Prechelt</i>	
3. A Simple Trick for Estimating the Weight Decay Parameter	69
<i>Thorsteinn S. Rögnvaldsson</i>	
4. Controlling the Hyperparameter Search in MacKay’s Bayesian Neural Network Framework	91
<i>Tony Plate</i>	
5. Adaptive Regularization in Neural Network Modeling	111
<i>Jan Larsen, Claus Svarer, Lars Nonboe Andersen, and Lars Kai Hansen</i>	
6. Large Ensemble Averaging	131
<i>David Horn, Ury Naftaly, and Nathan Intrator</i>	
Improving Network Models and Algorithmic Tricks	
Preface	139
7. Square Unit Augmented, Radially Extended, Multilayer Perceptrons . .	143
<i>Gary William Flake</i>	
8. A Dozen Tricks with Multitask Learning	163
<i>Rich Caruana</i>	
9. Solving the Ill-Conditioning in Neural Network Learning	191
<i>Patrick van der Smagt and Gerd Hirzinger</i>	
10. Centering Neural Network Gradient Factors	205
<i>Nicol N. Schraudolph</i>	
11. Avoiding Roundoff Error in Backpropagating Derivatives	225
<i>Tony Plate</i>	

Representing and Incorporating Prior Knowledge in Neural Network Training

Preface	231
12. Transformation Invariance in Pattern Recognition – Tangent Distance and Tangent Propagation	235
<i>Patrice Y. Simard, Yann A. LeCun, John S. Denker, and Bernard Victorri</i>	
13. Combining Neural Networks and Context-Driven Search for On-line, Printed Handwriting Recognition in the Newton	271
<i>Larry S. Yaeger, Brandyn J. Webb, and Richard F. Lyon</i>	
14. Neural Network Classification and Prior Class Probabilities	295
<i>Steve Lawrence, Ian Burns, Andrew Back, Ah Chung Tsoi, and C. Lee Giles</i>	
15. Applying Divide and Conquer to Large Scale Pattern Recognition Tasks	311
<i>Jürgen Fritsch and Michael Finke</i>	

Tricks for Time Series

Preface	339
16. Forecasting the Economy with Neural Nets: A Survey of Challenges and Solutions	343
<i>John Moody</i>	
17. How to Train Neural Networks	369
<i>Ralph Neuneier and Hans Georg Zimmermann</i>	

Big Learning in Deep Neural Networks

Preface	419
18. Stochastic Gradient Descent Tricks	421
<i>Léon Bottou</i>	
19. Practical Recommendations for Gradient-Based Training of Deep Architectures	437
<i>Yoshua Bengio</i>	
20. Training Deep and Recurrent Networks with Hessian-Free Optimization	479
<i>James Martens and Ilya Sutskever</i>	
21. Implementing Neural Networks Efficiently	537
<i>Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet</i>	

Better Representations: Invariant, Disentangled and Reusable

Preface	559
22. Learning Feature Representations with K-Means	561
<i>Adam Coates and Andrew Y. Ng</i>	
23. Deep Big Multilayer Perceptrons for Digit Recognition	581
<i>Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber</i>	
24. A Practical Guide to Training Restricted Boltzmann Machines	599
<i>Geoffrey E. Hinton</i>	
25. Deep Boltzmann Machines and the Centering Trick	621
<i>Grégoire Montavon and Klaus-Robert Müller</i>	
26. Deep Learning via Semi-supervised Embedding	639
<i>Jason Weston, Frédéric Ratle, and Ronan Collobert</i>	

Identifying Dynamical Systems for Forecasting and Control

Preface	657
27. A Practical Guide to Applying Echo State Networks	659
<i>Mantas Lukoševičius</i>	
28. Forecasting with Recurrent Neural Networks: 12 Tricks	687
<i>Hans-Georg Zimmermann, Christoph Tietz, and Ralph Grothmann</i>	
29. Solving Partially Observable Reinforcement Learning Problems with Recurrent Neural Networks	709
<i>Siegmund Duell, Steffen Udluft, and Volkmar Sterzing</i>	
30. 10 Steps and Some Tricks to Set up Neural Reinforcement Controllers	735
<i>Martin Riedmiller</i>	
Author Index	759
Subject Index	761

Introduction*

It is our belief that researchers and practitioners acquire, through experience and word-of-mouth, techniques and heuristics that help them successfully apply neural networks to difficult real world problems. Often these “tricks” are theoretically well motivated. Sometimes they are the result of trial and error. However, their most common link is that they are usually hidden in people’s heads or in the back pages of space-constrained conference papers. As a result newcomers to the field waste much time wondering why their networks train so slowly and perform so poorly.

This book is an outgrowth of a 1996 NIPS workshop called *Tricks of the Trade* whose goal was to begin the process of gathering and documenting these tricks. The interest that the workshop generated, motivated us to expand our collection and compile it into this book. Although we have no doubt that there are many tricks we have missed, we hope that what we have included will prove to be useful, particularly to those who are relatively new to the field. Each chapter contains one or more tricks presented by a given author (or authors). We have attempted to group related chapters into sections, though we recognize that the different sections are far from disjoint. Some of the chapters (e.g. 1,13,17) contain entire systems of tricks that are far more general than the category they have been placed in.

Before each section we provide the reader with a summary of the tricks contained within, to serve as a quick overview and reference. However, we do not recommend applying tricks before having read the accompanying chapter. Each trick may only work in a particular context that is not fully explained in the summary. This is particularly true for the chapters that present systems where combinations of tricks must be applied together for them to be effective.

Below we give a coarse roadmap of the contents of the individual chapters.

Speeding Learning

The book opens with a chapter based on Leon Bottou and Yann LeCun’s popular workshop on efficient backpropagation where they present a system of tricks for speeding the minimization process. Included are tricks that are very simple to implement as well as more complex ones, e.g. based on second-order methods. Though many of the readers may recognize some of these tricks, we believe that this chapter provides both: a thorough explanation of their theoretical basis as well as an understanding of the subtle interactions among them.

This chapter provides an ideal introduction for the reader. It starts with discussing fundamental tricks addressing input representation, initialization, target

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

values, choice of learning rates, choice of the nonlinearity, and so on. Subsequently, the authors introduce in great detail tricks for estimation and approximation of the Hessian in neural networks. This provides the basis for a discussion of second-order algorithms, fast training methods like the stochastic Levenberg-Marquardt algorithm, and tricks for learning rate adaptation.

Regularization Techniques to Improve Generalization

Fast minimization is important but only if we can also insure good generalization. We therefore next include a collection of chapters containing a range of approaches for improving generalization. As one might expect, there are no tricks that work well in all situations. However, many examples and discussions are included to help the reader to decide which will work best for their own problem.

Chapter 2 addresses what is one of the most commonly used techniques: early stopping. Here Lutz Prechelt discusses the pitfalls of this seemingly simple technique. He quantifies the tradeoff between generalization and training time for various stopping criteria, which leads to a trick for picking an appropriate criterion.

Using a weight decay penalty term in the cost function is another common method for improving generalization. The difficulty, however, is in finding a good estimate of the weight decay parameter. In chapter 3, Thorsteinn Rögnvaldsson presents a fast technique for finding a good estimate, surprisingly, by using information measured at the early stopping point. Experimental evidence for its usefulness is given in several applications.

Tony Plate in chapter 4 treats the penalty terms along the lines of MacKay, i.e. as hyperparameters to be found through iterative search. He presents and compares tricks for making the hyperparameter search in classification networks work in practice by speeding it up and simplifying it. Key to his success is a control of the frequency of the hyperparameter updates and a better strategy in cases where the Hessian becomes out-of-bounds.

In chapter 5, Jan Larsen et al. present a trick for adapting regularization parameters by using simple gradient descent (with respect to the regularization parameters) on the validation error. The trick is tested on both classification and regression problems.

Averaging over multiple predictors is a well known method for improving generalization. Two questions that arise are how many predictors are “enough” and how does the number of predictors affect the stopping criteria for early stopping. In the final chapter of this section, David Horn et al. present solutions to these questions by providing a method for estimating the error of an infinite number of predictors. They then demonstrate this trick for a prediction task.

Improving Network Models and Algorithmic Tricks

In this section we examine tricks that help improve the network model. Even though standard multilayer perceptrons (MLPs) are, in theory, universal ap-

proximators, other architectures may provide a more natural fit to a problem. A better fit means that training is faster and that there is a greater likelihood of finding a good and stable solution. For example, radial basis functions (RBFs) are preferred for problems that exhibit local features in a finite region. Of course, which architecture to choose is not always obvious.

In chapter 7, Gary Flake presents a trick that gives MLPs the power of both an MLP and an RBF so that one does not need to choose between them . This trick is simply to add extra inputs whose values are the square of the regular inputs. Both a theoretical and intuitive explanation are presented along with a number of simulation examples.

Rich Caruana in chapter 8 shows that performance can be improved on a main task by adding extra outputs to a network that predict related tasks. This technique, known as multi-task learning (MTL), trains these extra outputs in parallel with the main task. This chapter presents multiple examples of what one might use as these extra outputs as well as techniques for implementing MTL effectively. Empirical examples include mortality rankings for pneumonia and road-following in a network learning to steer a vehicle.

Patrick van der Smagt and Gerd Hirzinger consider in chapter 9 the ill-conditioning of the Hessian in neural network training and propose using what they call a linearly augmented feed-forward network, employing input/output short-cut connections that share the input/hidden weights. This gives rise to better conditioning of the learning problem and, thus, to faster learning, as shown in a simulation example with data from a robot arm.

In chapter 10, Nicol Schraudolph takes the idea of scaling and centering the inputs even further than chapter 1 by proposing to center all factors in the neural network gradient: inputs, activities, error signals and hidden unit slopes. He gives experimental evidence for the usefulness of this trick.

In chapter 11, Tony Plate's short note reports a numerical trick for computing derivatives more accurately with only a small memory overhead.

Representation and Incorporating Prior Knowledge in Neural Network Training

Previous chapters (e.g. Chapter 1) present very general tricks for transforming inputs to improve learning: prior knowledge of the problem is not taken into account explicitly (of course regularization, as discussed in Chapters 2-5, implicitly assumes a prior but on the weight distribution). For complex, difficult problems, however, it is not enough to take a black box approach, no matter how good that black box might be. This section examines how prior knowledge about a problem can be used to greatly improve learning. The questions asked include how to best represent the data, how to make use of this representation for training, and how to take advantage of the invariances that are present. Such issues are key for proper neural network training. They are also at the heart of the tricks pointed out by Patrice Simard, et al. in the first chapter of this section. Here, the authors present a particularly interesting perspective on how

to incorporate prior knowledge into data. They also give the first review of the tangent distance classification method and related techniques evolving from it such as tangent prop. These methods are applied to the difficult task of optical character recognition (OCR).

In chapter 13, Larry Yaeger, et al. give an overview of the tricks and techniques for on-line handwritten character recognition that were eventually used in the Apple Computer's Newton MessagePad® and eMate®. Anyone who has used these systems knows that their handwriting recognition capability works exceedingly well. Although many of the issues that are discussed in this chapter overlap with those in OCR, including representation and prior knowledge, the solutions are complementary. This chapter also gives a very nice overview of what design choices proved to be efficient as well as how different tricks such as choice of learning rate, over-representation of more difficult patterns, negative training, error emphasis and so on work together.

Whether it be handwritten character recognition, speech recognition or medical applications, a particularly difficult problem encountered is the unbalanced class prior probabilities that occur, for example, when certain writing styles and subphoneme classes are uncommon or certain illnesses occur less frequently. Chapter 13 briefly discusses this problem in the context of handwriting recognition and presents a heuristic which controls the frequency with which samples are picked for training.

In chapter 14, Steve Lawrence, et al. discuss the issue of unbalanced class prior probabilities in greater depth. They present and compare several different heuristics (prior scaling, probabilistic sampling, post scaling and class membership equalization) one of which is similar to the one in chapter 13. They demonstrate their tricks solving an ECG classification problem and provide some theoretical explanations.

Many training techniques work well for small to moderate size nets. However when problems consist of thousands of classes and millions of examples, not uncommon in applications such as speech recognition, many of these techniques break down. This chapter by Jürgen Fritsch and Michael Finke is devoted to the issue of large scale classification problems and representation design in general. Here the problem of unbalanced class prior probabilities is also tackled.

Although Fritsch and Finke specifically exemplify their design approach for the problem of building a large vocabulary speech recognizer, it becomes clear that these techniques are also applicable to the general construction of an appropriate hierarchical decision tree. A particularly interesting result in this paper is that the structural design to incorporate prior knowledge about speech done by a human speech expert was outperformed by their machine learning technique using an agglomerative clustering algorithm to choose the structure of the decision tree.

Tricks for Time Series

We close the book with two papers on the subject of time series and economic forecasting. In the first of these chapters, John Moody presents an excellent

survey of both the challenges of macroeconomic forecasting as well a number of neural network solutions. The survey is followed by a more detailed description of smoothing regularizers, model selection methods (e.g. AIC, effective number of parameters, nonlinear cross-validation), and input selection via sensitivity-based input pruning. Model interpretation and visualization are also discussed.

In the final chapter, Ralph Neuneier and Hans Georg Zimmermann present an impressive integrated system for neural network training of time series and economic forecasting. Every aspect of the system is discussed including input preprocessing, cost functions, handling of outliers, architecture, regularization techniques, as well as solutions for dealing with the problem of bottom-heavy networks, i.e. the input dimension is large while the output dimension is very small. There is also a thought-provoking discussion of the Observer-Observer dilemma: we want both to create a model based on observed data while, at the same time, use this model to judge the correctness of new incoming data. Even those people not interested specifically in economic forecasting are encouraged to read this very useful example of how to incorporate prior (system) knowledge into training.

Final Remark

As a final remark, we note that some of the views taken in the chapters are contradictory, e.g. some authors favor one regularization method over another, while other authors make exactly the opposite statement. On the one hand, one can explain these discrepancies by stating that the field is still very active and therefore opposing viewpoints will inevitably exist until more is understood. On the other hand, it may be that both (contradicting) views are correct but on different data sets and in different applications, e.g. an approach that considers noisy time-series needs algorithms with a completely different robustness than in, say, an OCR setting. In this sense, the present book mirrors an active field and a variety of applications with its diversity of views.

August 1998

Jenny & Klaus

Acknowledgements. We would like to thank all authors for their collaboration. Special thanks to Steven Lemm for considerable help with the typesetting. K.-R.M. acknowledges partial financial support from DFG (grant JA 379/51 and JA 379/7) and EU ESPRIT (grant 25387-STORM).

Speeding Learning^{*}

Preface

There are those who argue that developing fast algorithms is no longer necessary because computers have become so fast. However, we believe that the complexity of our algorithms and the size of our problems will always expand to consume all cycles available, regardless of the speed of our machines. Thus, there will never come a time when computational efficiency can or should be ignored. Besides, in the quest to find solutions faster, we also often find better and more stable solutions as well. This section is devoted to techniques for making the learning process in backpropagation (BP) faster and more efficient. It contains a single chapter based on a workshop by Leon Bottou and Yann LeCun. While many alternative learning systems have emerged since the time BP was first introduced, BP is still the most widely used learning algorithm. The reason for this is its simplicity, efficiency, and its general effectiveness on a wide range of problems. Even so, there are many pitfalls in applying it, which is where all these tricks enter.

Chapter 1 begins gently by introducing us to a few practical tricks that are very simple to implement. Included are easy to understand qualitative explanations of each. There is a discussion of **stochastic (on-line) vs batch mode learning** where the advantages and disadvantages of both are presented while making it clear that stochastic learning is most often preferred (p. 13). There is a trick that aims at maximizing the per iteration information presented to the network simply by knowing **how best to shuffle the examples** (p. 15). This is followed by an entire set of tricks that must be coordinated together for maximum effectiveness. These include:

- how to **normalize, decorrelate, and scale the inputs** (p. 16)
- how to **choose the sigmoid** (p. 17)
- how to **set target values** (classification) (p. 19)
- how to **initialize the weights** (p. 20)
- how to **pick the learning rates** (p. 20).

Additional issues discussed include the effectiveness of momentum and the choice between radial basis units and sigmoid units (p. 21).

Chapter 1 then introduces us to a little of the theory, providing deeper understanding of some of the preceding tricks. Included are discussions of the effect of learning rates on the speed of learning and of the relationship between the Hessian matrix, the error surface, and the learning rates. Simple examples of linear and multilayer nets are provided to illustrate the theoretical results.

The chapter next enters more difficult territory by giving an overview of second order methods (p. 31). Quickly summarized here, they are

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

Newton method: generally impractical to use since it requires inverting the full Hessian and works only in batch mode.

conjugate gradient: an $O(N)$ algorithm that doesn't use the Hessian, but requires a line search and so works only in batch mode.

Quasi-Newton, Broyden-Fletcher-Goldfarb-Shanno (BFGS) method: an $O(N^2)$ algorithm that computes an estimate of the inverse Hessian. It requires line search and also only works in batch mode.

Gauss-Newton method: an $O(N^3)$ algorithm that uses the square Jacobi approximation of the Hessian. Mainly used for batch and works only for mean squared error loss functions.

Levenberg Marquardt method: extends the Gauss-Newton method to include a regularization parameter for stability.

Second order methods can greatly speed learning at each iteration but often at an excessive computational cost. However, by replacing the exact Hessian with an approximation of either the full or partial Hessian, the benefits of second order information can still be reaped without incurring as great a computational cost.

The first and most direct method for **estimating the full Hessian** is finite differences which simply requires little more than two backpropagations to compute each row of the Hessian (p. 35). Another is to use the square Jacobian approximation which guarantees a positive semi-definite matrix which may be beneficial for improving stability. If even more simplification is desired, one can just compute the diagonal elements of the Hessian. All of the methods mentioned here are easily implemented using BP.

Unfortunately, for very large networks, many of the classical second order methods do not work well because storing the Hessian is far too expensive and because batch mode, required by most of the methods, is too slow. *On-line* second order methods are needed instead. One such technique presented here is a **stochastic diagonal Levenberg Marquardt method** (p. 40).

If all that is needed is the **product of the Hessian with an arbitrary vector** rather than the Hessian itself, then much time can be saved using a method that computes this entire product directly using only a single backpropagation step (p. 37). Such a technique can be used to compute the largest eigenvalue and associated eigenvector of the Hessian. The inverse of the largest eigenvalue can then be used to obtain a good estimate of the learning rate.

Finally, three useful tricks are presented for **computing the principal eigenvalue** and vector without having to compute the Hessian: the power method, Taylor expansion, and an on-line method (p. 42).

Jenny & Klaus

1

Efficient BackProp[★]

Yann A. LeCun¹, Léon Bottou¹, Genevieve B. Orr², and Klaus-Robert Müller³

¹ Image Processing Research Department AT& T Labs - Research, 100 Schulz Drive,
Red Bank, NJ 07701-7033, USA

² Willamette University, 900 State Street, Salem, OR 97301, USA

³ GMD FIRST, Rudower Chaussee 5, 12489 Berlin, Germany

{yann,leonb}@research.att.com, gorr@willamette.edu, klaus@first.gmd.de

Abstract. The convergence of back-propagation learning is analyzed so as to explain common phenomenon observed by practitioners. Many undesirable behaviors of backprop can be avoided with tricks that are rarely exposed in serious technical publications. This paper gives some of those tricks, and offers explanations of why they work.

Many authors have suggested that second-order optimization methods are advantageous for neural net training. It is shown that most “classical” second-order methods are impractical for large neural networks. A few methods are proposed that do not have these limitations.

1.1 Introduction

Backpropagation is a very popular neural network learning algorithm because it is conceptually simple, computationally efficient, and because it often works. However, getting it to work well, and sometimes to work at all, can seem more of an art than a science. Designing and training a network using backprop requires making many seemingly arbitrary choices such as the number and types of nodes, layers, learning rates, training and test sets, and so forth. These choices can be critical, yet there is no foolproof recipe for deciding them because they are largely problem and data dependent. However, there are heuristics and some underlying theory that can help guide a practitioner to make better choices.

In the first section below we introduce standard backpropagation and discuss a number of simple heuristics or tricks for improving its performance. We next discuss issues of convergence. We then describe a few “classical” second-order non-linear optimization techniques and show that their application to neural network training is very limited, despite many claims to the contrary in the literature. Finally, we present a few second-order methods that do accelerate learning in certain cases.

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

1.2 Learning and Generalization

There are several approaches to automatic machine learning, but much of the successful approaches can be categorized as *gradient-based learning methods*. The learning machine, as represented in Figure 1.1, computes a function $M(Z^p, W)$ where Z^p is the p -th input pattern, and W represents the collection of adjustable parameters in the system. A cost function $E^p = C(D^p, M(Z^p, W))$, measures the discrepancy between D^p , the “correct” or desired output for pattern Z^p , and the output produced by the system. The average cost function $E_{train}(W)$ is the average of the errors E^p over a set of input/output pairs called the training set $\{(Z^1, D^1), \dots, (Z^P, D^P)\}$. In the simplest setting, the learning problem consists in finding the value of W that minimizes $E_{train}(W)$. In practice, the performance of the system on a training set is of little interest. The more relevant measure is the error rate of the system in the field, where it would be used in practice. This performance is estimated by measuring the accuracy on a set of samples disjoint from the training set, called the test set. The most commonly used cost function is the Mean Squared Error:

$$E^p = \frac{1}{2}(D^p - M(Z^p, W))^2, \quad E_{train} = \frac{1}{P} \sum_{p=1}^P E^p$$

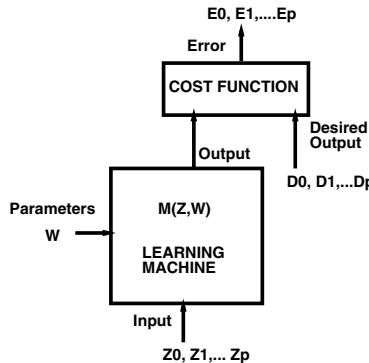


Fig. 1.1. Gradient-based learning machine

This chapter is focused on strategies for improving the process of minimizing the cost function. However, these strategies must be used in conjunction with methods for maximizing the network’s ability to *generalize*, that is, to predict the correct targets for patterns the learning system has not previously seen (e.g. see chapters 2, 3, 4, 5 for more detail).

To understand generalization, let us consider how backpropagation works. We start with a set of samples each of which is an input/output pair of the function to be learned. Since the measurement process is often noisy, there may be errors in the samples. We can imagine that if we collected multiple *sets* of samples then each set would look a little different because of the noise and because of the different points sampled. Each of these data sets would also result in networks with minima that are slightly different from each other and from the

true function. In this chapter, we concentrate on improving the process of finding the minimum for the particular set of examples that we are given. Generalization techniques try to correct for the errors introduced into the network as a result of our choice of dataset. Both are important.

Several theoretical efforts have analyzed the process of learning by minimizing the error on a training set (a process sometimes called Empirical Risk Minimization) [40, 41].

Some of those theoretical analyses are based on decomposing the generalization error into two terms: bias and variance (see e.g. [12]). The bias is a measure of how much the network output, averaged over all possible data sets differs from the desired function. The variance is a measure of how much the network output varies between datasets. Early in training, the bias is large because the network output is far from the desired function. The variance is very small because the data has had little influence yet. Late in training, the bias is small because the network has learned the underlying function. However, if trained too long, the network will also have learned the noise specific to that dataset. This is referred to as overtraining. In such a case, the variance will be large because the noise varies between datasets. It can be shown that the minimum total error will occur when the sum of bias and variance are minimal.

There are a number of techniques (e.g. early stopping, regularization) for maximizing the generalization ability of a network when using backprop. Many of these techniques are described in later chapters 2, 3, 5, 4.

The idea of this chapter, therefore, is to present minimization strategies (given a cost function) and the tricks associated with increasing the speed and quality of the minimization. It is however clear that the choice of the model (model selection), the architecture and the cost function is crucial for obtaining a network that generalizes well. So keep in mind that if the wrong model class is used and no proper model selection is done, then even a superb minimization will clearly not help very much. In fact, the existence of overtraining has led several authors to suggest that inaccurate minimization algorithms can be better than good ones.

1.3 Standard Backpropagation

Although the tricks and analyses in this paper are primarily presented in the context of “classical” multi-layer feed-forward neural networks, many of them also apply to most other gradient-based learning methods.

The simplest form of multilayer learning machine trained with gradient-based learning is simply a stack of modules, each of which implements a function $X_n = F_n(W_n, X_{n-1})$, where X_n is a vector representing the output of the module, W_n is the vector of tunable parameters in the module (a subset of W), and X_{n-1} is the module’s input vector (as well as the previous module’s output vector). The input X_0 to the first module is the input pattern Z^p . If the partial derivative of

E^p with respect to X_n is known, then the partial derivatives of E^p with respect to W_n and X_{n-1} can be computed using the backward recurrence

$$\begin{aligned}\frac{\partial E^p}{\partial W_n} &= \frac{\partial F}{\partial W}(W_n, X_{n-1}) \frac{\partial E^p}{\partial X_n} \\ \frac{\partial E^p}{\partial X_{n-1}} &= \frac{\partial F}{\partial X}(W_n, X_{n-1}) \frac{\partial E^p}{\partial X_n}\end{aligned}\quad (1.1)$$

where $\frac{\partial F}{\partial W}(W_n, X_{n-1})$ is the Jacobian of F with respect to W evaluated at the point (W_n, X_{n-1}) , and $\frac{\partial F}{\partial X}(W_n, X_{n-1})$ is the Jacobian of F with respect to X . The Jacobian of a vector function is a matrix containing the partial derivatives of all the outputs with respect to all the inputs. When the above equations are applied to the modules in reverse order, from layer N to layer 1, all the partial derivatives of the cost function with respect to all the parameters can be computed. The way of computing gradients is known as back-propagation.

Traditional multi-layer neural networks are a special case of the above system where the modules are alternated layers of matrix multiplications (the weights) and component-wise sigmoid functions (the units):

$$Y_n = W_n X_{n-1} \quad (1.2)$$

$$X_n = F(Y_n) \quad (1.3)$$

where W_n is a matrix whose number of columns is the dimension of X_{n-1} , and number of rows is the dimension of X_n . F is a vector function that applies a sigmoid function to each component of its input. Y_n is the vector of weighted sums, or *total inputs*, to layer n .

Applying the chain rule to the equation above, the classical backpropagation equations are obtained:

$$\frac{\partial E^p}{\partial y_n^i} = f'(y_n^i) \frac{\partial E^p}{\partial x_n^i} \quad (1.4)$$

$$\frac{\partial E^p}{\partial w_n^{ij}} = x_{n-1}^j \frac{\partial E^p}{\partial y_n^i} \quad (1.5)$$

$$\frac{\partial E^p}{\partial x_{n-1}^k} = \sum_i w_n^{ik} \frac{\partial E^p}{\partial y_n^i}. \quad (1.6)$$

The above equations can also be written in matrix form:

$$\frac{\partial E^p}{\partial Y_n} = F'(Y_n) \frac{\partial E^p}{\partial X_n} \quad (1.7)$$

$$\frac{\partial E^p}{\partial W_n} = X_{n-1} \frac{\partial E^p}{\partial Y_n} \quad (1.8)$$

$$\frac{\partial E^p}{\partial X_{n-1}} = W_n^T \frac{\partial E^p}{\partial Y_n}. \quad (1.9)$$

The simplest learning (minimization) procedure in such a setting is the gradient descent algorithm where W is iteratively adjusted as follows:

$$W(t) = W(t-1) - \eta \frac{\partial E}{\partial W}. \quad (1.10)$$

In the simplest case, η is a scalar constant. More sophisticated procedures use variable η . In other methods η takes the form of a diagonal matrix, or is an estimate of the inverse Hessian matrix of the cost function (second derivative matrix) such as in the Newton and Quasi-Newton methods described later in the chapter. A proper choice of η is important and will be discussed at length later.

1.4 A Few Practical Tricks

Backpropagation can be very slow particularly for multilayered networks where the cost surface is typically non-quadratic, non-convex, and high dimensional with many local minima and/or flat regions. There is no formula to guarantee that (1) the network will converge to a good solution, (2) convergence is swift, or (3) convergence even occurs at all. However, in this section we discuss a number of tricks that can greatly improve the chances of finding a good solution while also decreasing the convergence time often by orders of magnitude. More detailed theoretical justifications will be given in later sections.

1.4.1 Stochastic versus Batch Learning

At each iteration, equation (1.10) requires a complete pass through the entire dataset in order to compute the *average* or true gradient. This is referred to as batch learning since an entire “batch” of data must be considered before weights are updated. Alternatively, one can use stochastic (online) learning where a *single* example $\{Z^t, D^t\}$ is chosen (e.g. randomly) from the training set at each iteration t . An *estimate* of the true gradient is then computed based on the error E^t of that example, and then the weights are updated:

$$W(t+1) = W(t) - \eta \frac{\partial E^t}{\partial W}. \quad (1.11)$$

Because this estimate of the gradient is noisy, the weights may not move precisely down the gradient at each iteration. As we shall see, this “noise” at each iteration can be advantageous. Stochastic learning is generally the preferred method for basic backpropagation for the following three reasons:

Advantages of Stochastic Learning

1. Stochastic learning is usually *much* faster than batch learning.
2. Stochastic learning also often results in better solutions.
3. Stochastic learning can be used for tracking changes.

Stochastic learning is most often *much* faster than batch learning particularly on large redundant datasets. The reason for this is simple to show. Consider the simple case where a training set of size 1000 is inadvertently composed of 10 identical copies of a set with 100 samples. Averaging the gradient over all 1000 patterns gives the exact same result as computing the gradient based on just the first 100. Thus, batch gradient descent is wasteful because it recomputes

the same quantity 10 times before one parameter update. On the other hand, stochastic gradient will see a full epoch as 10 iterations through a 100-long training set. In practice, examples rarely appear more than once in a dataset, but there are usually clusters of patterns that are very similar. For example in phoneme classification, all of the patterns for the phoneme /æ/ will (hopefully) contain much of the same information. It is this redundancy that can make batch learning much slower than on-line.

Stochastic learning also often results in better solutions because of the noise in the updates. Nonlinear networks usually have multiple local minima of differing depths. The goal of training is to locate one of these minima. Batch learning will discover the minimum of whatever basin the weights are initially placed. In stochastic learning, the noise present in the updates can result in the weights jumping into the basin of another, possibly deeper, local minimum. This has been demonstrated in certain simplified cases [15, 30].

Stochastic learning is also useful when the function being modeled is changing over time, a quite common scenario in industrial applications where the data distribution changes gradually over time (e.g. due to wear and tear of the machines). If the learning machine does not detect and follow the change it is impossible to learn the data properly and large generalization errors will result. With batch learning, changes go undetected and we obtain rather bad results since we are likely to average over several rules, whereas on-line learning – if operated properly (see below in section 1.4.7) – will track the changes and yield good approximation results.

Despite the advantages of stochastic learning, there are still reasons why one might consider using batch learning:

Advantages of Batch Learning

1. Conditions of convergence are well understood.
2. Many acceleration techniques (e.g. conjugate gradient) only operate in batch learning.
3. Theoretical analysis of the weight dynamics and convergence rates are simpler.

These advantages stem from the same noise that make stochastic learning advantageous. This noise, which is so critical for finding better local minima also prevents full convergence to the minimum. Instead of converging to the exact minimum, the convergence stalls out due to the weight fluctuations. The size of the fluctuations depend on the degree of noise of the stochastic updates. The variance of the fluctuations around the local minimum is proportional to the learning rate η [28, 27, 6]. So in order to reduce the fluctuations we can either decrease (anneal) the learning rate or have an adaptive batch size. In theory [13, 30, 36, 35] it is shown that the optimal annealing schedule of the learning rate is of the form

$$\eta \sim \frac{c}{t}, \quad (1.12)$$

where t is the number of patterns presented and c is a constant. In practice, this may be too fast (see chapter 13).

Another method to remove noise is to use “mini-batches”, that is, start with a small batch size and increase the size as training proceeds. Møller discusses one method for doing this [25] and Orr [31] discusses this for linear problems. However, deciding the rate at which to increase the batch size and which inputs to include in the small batches is as difficult as determining the proper learning rate. Effectively the size of the learning rate in stochastic learning corresponds to the respective size of the mini batch.

Note also that the problem of removing the noise in the data may be less critical than one thinks because of generalization. Overtraining may occur long before the noise regime is even reached.

Another advantage of batch training is that one is able to use second order methods to speed the learning process. Second order methods speed learning by estimating not just the gradient but also the curvature of the cost surface. Given the curvature, one can estimate the approximate location of the actual minimum.

Despite the advantages of batch updates, stochastic learning is still often the preferred method particularly when dealing with very large data sets because it is simply much faster.

1.4.2 Shuffling the Examples

Networks learn the fastest from the most unexpected sample. Therefore, it is advisable to choose a sample at each iteration that is the most unfamiliar to the system. Note, this applies only to stochastic learning since the order of input presentation is irrelevant for batch¹. Of course, there is no simple way to know which inputs are information rich, however, a very simple trick that crudely implements this idea is to simply choose successive examples that are from *different* classes since training examples belonging to the same class will most likely contain similar information.

Another heuristic for judging how much new information a training example contains is to examine the error between the network output and the target value when this input is presented. A large error indicates that this input has not been learned by the network and so contains a lot of new information. Therefore, it makes sense to present this input more frequently. Of course, by “large” we mean relative to all of the other training examples. As the network trains, these relative errors will change and so should the frequency of presentation for a particular input pattern. A method that modifies the probability of appearance of each pattern is called an *emphasizing scheme*.

Choose Examples with Maximum Information Content

1. Shuffle the training set so that successive training examples never (rarely) belong to the same class.
2. Present input examples that produce a large error more frequently than examples that produce a small error.

¹ The order in which gradients are summed in batch may be affected by roundoff error if there is a significant range of gradient values.

However, one must be careful when perturbing the normal frequencies of input examples because this changes the relative importance that the network places on different examples. This may or may not be desirable. For example, *this technique applied to data containing outliers can be disastrous* because outliers can produce large errors yet should not be presented frequently. On the other hand, this technique can be particularly beneficial for boosting the performance for infrequently occurring inputs, e.g. /z/ in phoneme recognition (see chapter 13, 14).

1.4.3 Normalizing the Inputs

Convergence is usually faster if the average of each input variable over the training set is close to zero. To see this, consider the extreme case where all the inputs are positive. Weights to a particular node in the first weight layer are updated by an amount proportional to δx where δ is the (scalar) error at that node and x is the input vector (see equations (1.5) and (1.10)). When all of the components of an input vector are positive, all of the updates of weights that feed into a node will be the same sign (i.e. $\text{sign}(\delta)$). As a result, these weights can only all decrease or all increase *together* for a given input pattern. Thus, if a weight vector must change direction it can only do so by zigzagging which is inefficient and thus very slow.

In the above example, the inputs were all positive. However, in general, any shift of the average input away from zero will bias the updates in a particular direction and thus slow down learning. Therefore, it is good to shift the inputs so that the average over the training set is close to zero. This heuristic should be applied at all layers which means that we want the average of the *outputs* of a node to be close to zero because these outputs are the inputs to the next layer [19], chapter 10. This problem can be addressed by coordinating how the inputs are transformed with the choice of sigmoidal activation function. Here we discuss the input transformation. The discussion of the sigmoid follows.

Convergence is faster not only if the inputs are shifted as described above but also if they are scaled so that all have about the same covariance, C_i , where

$$C_i = \frac{1}{P} \sum_{p=1}^P (z_i^p)^2. \quad (1.13)$$

Here, P is the number of training examples, C_i is the covariance of the i^{th} input variable and z_i^p is the i^{th} component of the p^{th} training example. Scaling speeds learning because it helps to balance out the rate at which the weights connected to the input nodes learn. The value of the covariance should be matched with that of the sigmoid used. For the sigmoid given below, a covariance of 1 is a good choice.

The exception to scaling all covariances to the same value occurs when it is known that some inputs are of less significance than others. In such a case, it can be beneficial to scale the less significant inputs down so that they are “less visible” to the learning process.

Transforming the Inputs

1. The average of each input variable over the training set should be close to zero.
2. Scale input variables so that their covariances are about the same.
3. Input variables should be uncorrelated if possible.

The above two tricks of shifting and scaling the inputs are quite simple to implement. Another trick that is quite effective but more difficult to implement is to decorrelate the inputs. Consider the simple network in Figure 1.2. If inputs are uncorrelated then it is possible to solve for the value of w_1 that minimizes the error without any concern for w_2 , and vice versa. In other words, the two variables are independent (the system of equations is diagonal). With correlated inputs, one must solve for both simultaneously which is a much harder problem. Principal component analysis (also known as the Karhunen-Loeve expansion) can be used to remove *linear* correlations in inputs [10].

Inputs that are linearly dependent (the extreme case of correlation) may also produce degeneracies which may slow learning. Consider the case where one input is always twice the other input ($z_2 = 2z_1$). The network output is constant along lines $W_2 = v - (1/2)W_1$, where v is a constant. Thus, the gradient is zero along these directions (see Figure 1.2). Moving along these lines has absolutely no effect on learning. We are trying to solve in 2-D what is effectively only a 1-D problem. Ideally we want to remove one of the inputs which will decrease the size of the network.

Figure 1.3 shows the entire process of transforming inputs. The steps are (1) shift inputs so the mean is zero, (2) decorrelate inputs, and (3) equalize covariances.

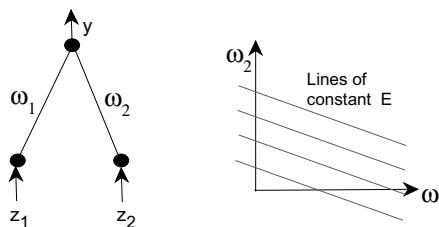
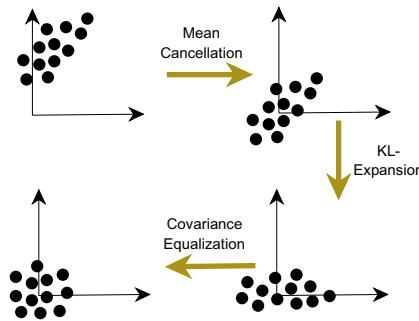


Fig. 1.2. Linearly dependent inputs

1.4.4 The Sigmoid

Nonlinear activation functions are what give neural networks their nonlinear capabilities. One of the most common forms of activation function is the sigmoid which is a monotonically increasing function that asymptotes at some finite value as $\pm\infty$ is approached. The most common examples are the standard logistic function $f(x) = 1/(1 + e^{-x})$ and hyperbolic tangent $f(x) = \tanh(x)$ shown in Figure 1.4. Sigmoids that are symmetric about the origin (e.g. see Figure 1.4b) are preferred for the same reason that inputs should be normalized, namely,

**Fig. 1.3.** Transformation of inputs

because they are more likely to produce outputs (which are *inputs* to the next layer) that are on average close to zero. This is in contrast, say, to the logistic function whose outputs are always positive and so must have a mean that is positive.

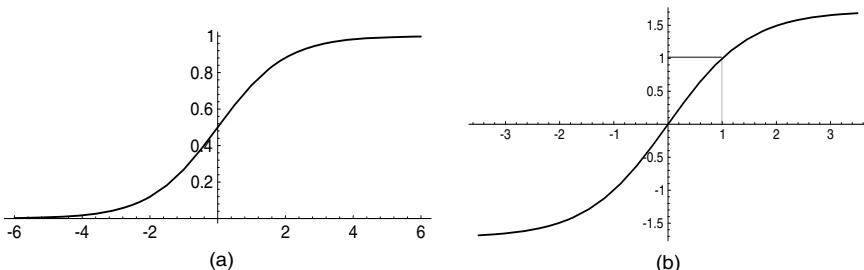


Fig. 1.4. (a) Not recommended: the standard logistic function, $f(x) = 1/(1 + e^{-x})$.
 (b) Recommended: $f(x) = 1.7159 \tanh(\frac{2}{3}x)$.

Sigmoids

1. Symmetric sigmoids such as hyperbolic tangent often converge faster than the standard logistic function.
2. A recommended sigmoid [19] is: $f(x) = 1.7159 \tanh(\frac{2}{3}x)$. Since the \tanh function is sometimes computationally expensive, an approximation of it by a ratio of polynomials can be used instead.
3. Sometimes it is helpful to add a small linear term, e.g. $f(x) = \tanh(x) + ax$ so as to avoid flat spots.

The constants in the recommended sigmoid given above have been chosen so that, *when used with transformed inputs* (see previous discussion), the variance of the outputs will also be close to 1 because the effective gain of the sigmoid is roughly 1 over its useful range. In particular, this sigmoid has the properties

(a) $f(\pm 1) = \pm 1$, (b) the second derivative is a maximum at $x = 1$, and (c) the effective gain is close to 1.

One of the potential problems with using symmetric sigmoids is that the error surface can be *very* flat near the origin. For this reason it is good to avoid initializing with very small weights. Because of the saturation of the sigmoids, the error surface is also flat far from the origin. Adding a small linear term to the sigmoid can sometimes help avoid the flat regions (see chapter 9).

1.4.5 Choosing Target Values

In classification problems, target values are typically binary (e.g. $\{-1,+1\}$). Common wisdom might seem to suggest that the target values be set at the value of the sigmoid's asymptotes. However, this has several drawbacks.

First, instabilities can result. The training process will try to drive the output as close as possible to the target values, which can only be achieved asymptotically. As a result, the weights (output and even hidden) are driven to larger and larger values where the sigmoid derivative is close to zero. The very large weights increase the gradients, however, these gradients are then multiplied by an exponentially small sigmoid derivative (except when a twisting term² is added to the sigmoid) producing a weight update close to zero. As a result, the weights may become stuck.

Second, when the outputs saturate, the network gives no indication of confidence level. When an input pattern falls near a decision boundary the output class is uncertain. Ideally this should be reflected in the network by an output value that is in between the two possible target values, i.e. not near either asymptote. However, large weights tend to force all outputs to the tails of the sigmoid regardless of the uncertainty. Thus, the network may predict a wrong class without giving any indication of its low confidence in the result. Large weights that saturate the nodes make it impossible to differentiate between typical and nontypical examples.

A solution to these problems is to set the target values to be within the range of the sigmoid, rather than at the asymptotic values. Care must be taken, however, to insure that the node is not restricted to only the linear part of the sigmoid. Setting the target values to the point of the maximum second derivative on the sigmoid is the best way to take advantage of the nonlinearity without saturating the sigmoid. This is another reason the sigmoid in Figure 1.4b is a good choice. It has maximum second derivative at ± 1 which correspond to the binary target values typical in classification problems.

Targets

Choose target values at the point of the maximum second derivative on the sigmoid so as to avoid saturating the output units.

² A twisting term is a small linear term added to the node output, e.g. $f(x) = \tanh(x) + ax$.

1.4.6 Initializing the Weights

The starting values of the weights can have a significant effect on the training process. Weights should be chosen randomly but in such a way that the sigmoid is primarily activated in its linear region. If weights are all very large then the sigmoid will saturate resulting in small gradients that make learning slow. If weights are very small then gradients will also be very small. Intermediate weights that range over the sigmoid's linear region have the advantage that (1) the gradients are large enough that learning can proceed and (2) the network will learn the linear part of the mapping before the more difficult nonlinear part.

Achieving this requires coordination between the training set normalization, the choice of sigmoid, and the choice of weight initialization. We start by requiring that the distribution of the outputs of each node have a standard deviation (σ) of approximately 1. This is achieved at the input layer by normalizing the training set as described earlier. To obtain a standard deviation close to 1 at the output of the first hidden layer we just need to use the above recommended sigmoid together with the requirement that the input to the sigmoid also have a standard deviation $\sigma_y = 1$. Assuming the inputs, y_i , to a unit are uncorrelated with variance 1, the standard deviation of the units weighted sum will be

$$\sigma_{y_i} = \left(\sum_j w_{ij}^2 \right)^{1/2}. \quad (1.14)$$

Thus, to insure that the σ_{y_i} are approximately 1 the weights should be randomly drawn from a distribution with mean zero and a standard deviation given by

$$\sigma_w = m^{-1/2} \quad (1.15)$$

where m is the number of inputs to the unit.

Initializing Weights

Assuming that:

1. the training set has been normalized, and
2. the sigmoid from Figure 1.4b has been used

then weights should be randomly drawn from a distribution (e.g. uniform) with mean zero and standard deviation

$$\sigma_w = m^{-1/2} \quad (1.16)$$

where m is the fan-in (the number of connections feeding *into* the node).

1.4.7 Choosing Learning Rates

There is at least one well-principled method (described in section 1.9.2) for estimating the ideal learning rate η . Many other schemes (most of them rather empirical) have been proposed in the literature to automatically adjust the learning

rate. Most of those schemes decrease the learning rate when the weight vector “oscillates”, and increase it when the weight vector follows a relatively steady direction. The main problem with these methods is that they are not appropriate for stochastic gradient or on-line learning because the weight vector fluctuates all the time.

Beyond choosing a single global learning rate, it is clear that picking a different learning rate η_i for each weight can improve the convergence. A well-principled way of doing this, based on computing second derivatives, is described in section 1.9.1. The main philosophy is to make sure that all the weights in the network converge roughly at the same speed.

Depending upon the curvature of the error surface, some weights may require a small learning rate in order to avoid divergence, while others may require a large learning rate in order to converge at a reasonable speed. Because of this, learning rates in the lower layers should generally be larger than in the higher layers (see Figure 1.21). This corrects for the fact that in most neural net architectures, the second derivative of the cost function with respect to weights in the lower layers is generally smaller than that of the higher layers. The rationale for the above heuristics will be discussed in more detail in later sections along with suggestions for how to choose the actual value of the learning rate for the different weights (see section 1.9.1).

If shared weights are used such as in time-delay neural networks (TDNN) [42] or convolutional networks [20], the learning rate should be proportional to the square root of the number of connections sharing that weight, because we know that the gradients are a sum of more-or-less independent terms.

Equalize the Learning Speeds

- give each weight its own learning rate
- learning rates should be proportional to the square root of the number of inputs to the unit
- weights in lower layers should typically be larger than in the higher layers

Other tricks for improving the convergence include:

Momentum. Momentum

$$\Delta w(t+1) = \eta \frac{\partial E_{t+1}}{\partial w} + \mu \Delta w(t),$$

can increase speed when the cost surface is highly nonspherical because it damps the size of the steps along directions of high curvature thus yielding a larger effective learning rate along the directions of low curvature [43] (μ denotes the strength of the momentum term). It has been claimed that momentum generally helps more in batch mode than in stochastic mode, but no systematic study of this are known to the authors.

Adaptive Learning Rates. Many authors, including Sompolinsky et al. [37], Darken & Moody [9], Sutton [38], Murata et al. [28] have proposed rules for automatically adapting the learning rates (see also [16]). These rules control the speed of convergence by increasing or decreasing the learning rate based on the error.

We assume the following facts for a learning rate adaptation algorithm: (1) the smallest eigenvalue of the Hessian (see Eq.(1.27)) is sufficiently smaller than the second smallest eigenvalue and (2) therefore after a large number of iterations, the parameter vector $w(t)$ will approach the minimum from the direction of the minimum eigenvector of the Hessian (see Eq.(1.27), Figure 1.5). Under these conditions the evolution of the estimated parameter can be thought of as a one-dimensional process and the minimum eigenvector \mathbf{v} can be approximated (for a large number of iterations: see Figure 1.5) by

$$\mathbf{v} = \langle \frac{\partial E}{\partial w} \rangle / \| \langle \frac{\partial E}{\partial w} \rangle \|,$$

where $\| \cdot \|$ denotes the L^2 norm. Hence we can adopt a projection

$$\xi = \langle \mathbf{v}^T \frac{\partial E}{\partial w} \rangle = \| \langle \frac{\partial E}{\partial w} \rangle \|$$

to the approximated minimum Eigenvector \mathbf{v} as a one dimensional measure of the distance to the minimum. This distance can be used to control the learning rate (for details see [28])

$$w(t+1) = w(t) - \eta_t \frac{\partial E_t}{\partial w}, \quad (1.17)$$

$$\mathbf{r}(t+1) = (1 - \delta)\mathbf{r}(t) + \delta \frac{\partial E_t}{\partial w}, \quad (0 < \delta < 1) \quad (1.18)$$

$$\eta(t+1) = \eta(t) + \alpha \eta(t) (\beta \|\mathbf{r}(t+1)\| - \eta(t)), \quad (1.19)$$

where δ controls the leak size of the average, α, β are constants and \mathbf{r} is used as auxiliary variable to calculate the leaky average of the gradient $\frac{\partial E}{\partial w}$.

Note that this set of rules is easy to compute and straightforward to implement. We simply have to keep track of an additional vector in Eq.(1.18): the averaged gradient \mathbf{r} . The norm of this vector then controls the size of the learning rate (see Eq.(1.19)). The algorithm follows the simple intuition: far away from the minimum (large distance ξ) it proceeds in big steps and close to the minimum it anneals the learning rate (for theoretical details see [28]).

1.4.8 Radial Basis Functions vs Sigmoid Units

Although most systems use nodes based on dot products and sigmoids, many other types of units (or layers) can be used. A common alternative is the radial basis function (RBF) network (see [7, 26, 5, 32]) In RBF networks, the dot product of the weight and input vector is replaced with a Euclidean distance

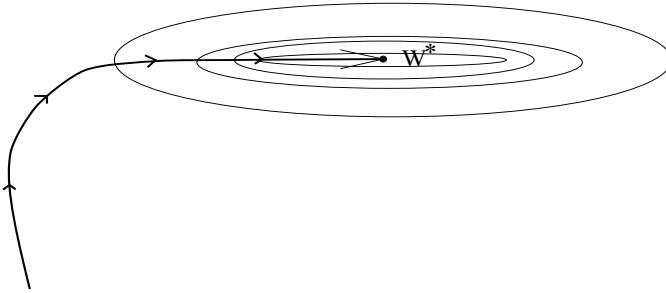


Fig. 1.5. Convergence of the flow. During the final stage of learning the average flow is approximately one dimensional towards the minimum \mathbf{w}^* and it is a good approximation of the minimum eigenvalue direction of the Hessian.

between the input and weight and the sigmoid is replaced by an exponential. The output activity is computed, e.g. for one output, as

$$g(x) = \sum_{i=1}^N w_i \exp\left(-\frac{1}{2\sigma_i^2} \|x - \nu_i\|^2\right),$$

where ν_i (σ_i) is the mean (standard deviation) of the i -th Gaussian. These units can replace or coexist with the standard units and they are usually trained by combination of gradient descent (for output units) and unsupervised clustering for determining the means and widths of the RBF units.

Unlike sigmoidal units which can cover the entire space, a single RBF unit covers only a small local region of the input space. This can be an advantage because learning can be faster. RBF units may also form a better set of basis functions to model the input space than sigmoid units, although this is highly problem dependent (see chapter 7). On the negative side, the locality property of RBFs may be a disadvantage particularly in high dimensional spaces because many units are needed to cover the spaces. RBFs are more appropriate in (low dimensional) upper layers and sigmoids in (high dimensional) lower layers.

1.5 Convergence of Gradient Descent

1.5.1 A Little Theory

In this section we examine some of the theory behind the tricks presented earlier. We begin in one dimension where the update equation for gradient descent can be written as

$$W(t+1) = W(t) - \eta \frac{dE(W)}{dW}. \quad (1.20)$$

We would like to know how the value of η affects convergence and the learning speed. Figure 1.6 illustrates the learning behavior for several different sizes of η

when the weight W starts out in the vicinity of a local minimum. In one dimension, it is easy to define the optimal learning rate, η_{opt} , as being the learning rate that will move the weight to the minimum, W_{min} , in precisely one step (see Figure 1.6(i)b). If η is smaller than η_{opt} then the stepsize will be smaller and convergence will take multiple timesteps. If η is between η_{opt} and $2\eta_{opt}$ then the weight will oscillate around W_{min} but will eventually converge (Figure 1.6(i)c). If η is more than twice the size of η_{opt} (Figure 1.6(i)d) then the stepsize is so large that the weight ends up farther from W_{min} than before. Divergence results.

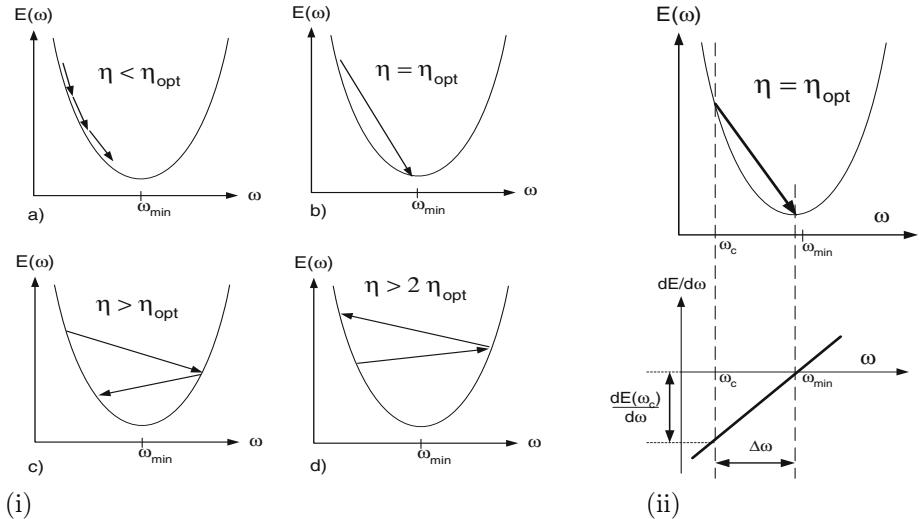


Fig. 1.6. Gradient descent for different learning rates

What is the optimal value of the learning rate η_{opt} ? Let us first consider the case in 1-dimension. Assuming that E can be approximated by a quadratic function, η_{opt} can be derived by first expanding E in a Taylor series about the current weight, W_c :

$$E(W) = E(W_c) + (W - W_c) \frac{dE(W_c)}{dW} + \frac{1}{2}(W - W_c)^2 \frac{d^2E(W_c)}{dW^2} + \dots, \quad (1.21)$$

where we use the shorthand $\frac{dE(W_c)}{dW} \equiv \frac{dE}{dW}|_{W=W_c}$. If E is quadratic the second order derivative is constant and the higher order terms vanish. Differentiating both sides with respect to W then gives

$$\frac{dE(W)}{dW} = \frac{dE(W_c)}{dW} + (W - W_c) \frac{d^2E(W_c)}{dW^2}. \quad (1.22)$$

Setting $W = W_{min}$ and noting that $dE(W_{min})/dW = 0$, we are left after rearranging with

$$W_{min} = W_c - \left(\frac{d^2E(W_c)}{dW^2} \right)^{-1} \frac{dE(W_c)}{dW}. \quad (1.23)$$

Comparing this with the update equation (1.20), we find that we can reach a minimum in one step if

$$\eta_{opt} = \left(\frac{d^2 E(W_c)}{dW^2} \right)^{-1}. \quad (1.24)$$

Perhaps an easier way to obtain this same result is illustrated in Figure 1.6(ii). The bottom graph plots the gradient of E as a function of W . Since E is quadratic, the gradient is simply a straight line with value zero at the minimum and $\frac{\partial E(W_c)}{\partial W}$ at the current weight W_c . $\partial^2 E / \partial^2 W$ is simply the slope of this line and is computed using the standard slope formula

$$\partial^2 E / \partial^2 W = \frac{\partial E(W_c) / \partial W - 0}{W_c - W_{min}}. \quad (1.25)$$

Solving for W_{min} then gives equation (1.23).

While the learning rate that gives fastest convergence is η_{opt} , the largest learning rate that can be used without causing divergence is (also see Figure 1.6(i)d)

$$\eta_{max} = 2\eta_{opt}. \quad (1.26)$$

If E is not exactly quadratic then the higher order terms in equation (1.21) are not precisely zero and (1.23) is only an approximation. In such a case, it may take multiple iterations to locate the minimum even when using η_{opt} , however, convergence can still be quite fast.

In multiple dimensions, determining η_{opt} is a bit more difficult because the right side of (1.24) is a matrix H^{-1} where H is called the Hessian whose components are given by

$$H_{ij} \equiv \frac{\partial^2 E}{\partial W_i \partial W_j} \quad (1.27)$$

with $1 \leq i, j \leq N$, and N equal to the total number of weights.

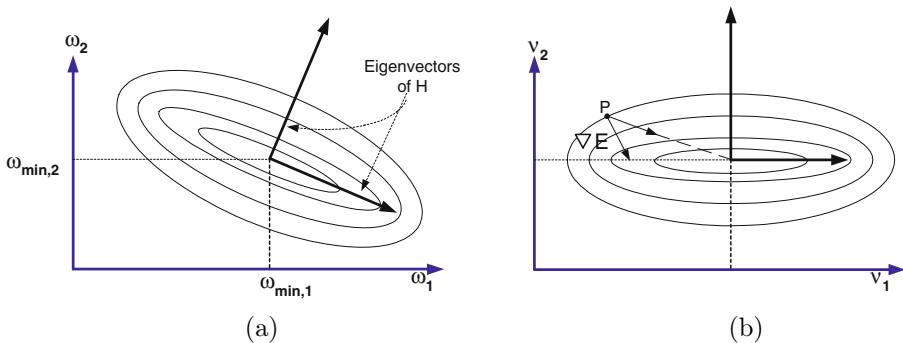
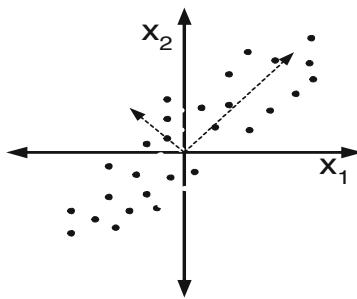
H is a measure of the curvature of E . In two dimensions, the lines of constant E for a quadratic cost are oval in shape as shown in Figure 1.7. The eigenvectors of H point in the directions of the major and minor axes. The eigenvalues measure the steepness of E along the corresponding eigendirection.

Example. In the least mean square (LMS) algorithm, we have a single layer linear network with error function

$$E(W) = \frac{1}{2P} \sum_{p=1}^P |d^p - \sum_i w_i x_i^p|^2 \quad (1.28)$$

where P is the number of training vectors. The Hessian in this case turns out to be the same as the covariance matrix of the inputs,

$$H = \frac{1}{P} \sum_p x^p x^{pT}. \quad (1.29)$$

**Fig. 1.7.** Lines of constant E **Fig. 1.8.** For the LMS algorithm, the eigenvectors and eigenvalues of H measure the spread of the inputs in input space

Thus, each eigenvalue of H is also a measure of the covariance or spread of the inputs along the corresponding eigendirection as shown in Figure 1.8.

Using a scalar learning rate is problematic in multiple dimensions. We want η to be large so that convergence is fast along the shallow directions of E (small eigenvalues of H), however, if η is too large the weights will diverge along the steep directions (large eigenvalues of H). To see this more specifically, let us again expand E , but this time about a minimum

$$E(W) \approx E(W_{min}) + \frac{1}{2}(W - W_{min})^T H_{(W_{min})}(W - W_{min}). \quad (1.30)$$

Differentiating (1.30) and using the result in the update equation (1.20) gives

$$W(t+1) = W(t) - \eta \frac{\partial E(t)}{\partial W} \quad (1.31)$$

$$= W(t) - \eta H_{(W_{min})}(W(t) - W_{min}). \quad (1.32)$$

Subtracting W_{min} from both sides gives

$$(W(t+1) - W_{min}) = (I - \eta H(W_{min}))(W(t) - W_{min}). \quad (1.33)$$

If the prefactor $(I - \eta H(W_{min}))$ is a matrix transformation that always shrinks a vector (i.e. its eigenvalues all have magnitude less than 1) then the update equation will converge.

How does this help with choosing the learning rates? Ideally we want different learning rates along the different eigendirections. This is simple if the eigendirections are lined up with the coordinate axes of the weights. In such a case, the weights are uncoupled and we can assign each weight its own learning rate based on the corresponding eigenvalue. However, if the weights are coupled then we must first rotate H such that H is diagonal, i.e. the coordinate axes line up with the eigendirections (see Figure 1.7b). This is the purpose of diagonalizing the Hessian discussed earlier.

Let Θ be the rotation matrix such that

$$A = \Theta H \Theta^T \quad (1.34)$$

where A is diagonal and $\Theta^T \Theta = I$. The cost function then can be written as

$$E(W) \approx E(W_{min}) + \frac{1}{2} [(W - W_{min})^T \Theta^T] [\Theta H(W_{min}) \Theta^T] [\Theta(W - W_{min})]. \quad (1.35)$$

Making a change of coordinates to $\nu = \Theta(W - W_{min})$ simplifies the above equation to

$$E(\nu) \approx E(0) + \frac{1}{2} \nu^T A \nu \quad (1.36)$$

and the transformed update equation becomes

$$\nu(t+1) = (I - \eta A) \nu(t). \quad (1.37)$$

Note that $I - \eta A$ is diagonal with diagonal components $1 - \eta \lambda_i$. This equation will converge if $|1 - \eta \lambda_i| < 1$, i.e. $\eta < \frac{2}{\lambda_i}$ for all i . If constrained to have a *single* scalar learning rate for all weights then we must require

$$\eta < \frac{2}{\lambda_{max}} \quad (1.38)$$

in order to avoid divergence, where λ_{max} is the largest eigenvalue of H . For fastest convergence we have

$$\eta_{opt} = \frac{1}{\lambda_{max}}. \quad (1.39)$$

If λ_{min} is a lot smaller than λ_{max} then convergence will be very slow along the λ_{min} direction. In fact, convergence time is proportional to the condition number $\kappa \equiv \lambda_{max}/\lambda_{min}$ so that it is desirable to have as small an eigenvalue spread as possible.

However, since we have rotated H to be aligned with the coordinate axes, (1.37) consists actually of N independent 1-dimensional equations. Therefore, we can choose a learning rate for each weight independent of the others. We see that the optimal rate for the i^{th} weight ν_i is $\eta_{opt,i} = \frac{1}{\lambda_i}$.

1.5.2 Examples

Linear Network. Figure 1.10 displays a set of 100 examples drawn from two Gaussian distributed classes centered at $(-0.4, -0.8)$ and $(0.4, 0.8)$. The eigenvalues of the covariance matrix are 0.84 and 0.036. We train a single layer linear network with 2 inputs, 1 output, 2 weights, and 1 bias (see Figure (1.9)) using the LMS algorithm in batch mode. Figure 1.11 displays the weight trajectory and error during learning when using a learning rates of $\eta = 1.5$ and 2.5. Note that the learning rate (see Eq. 1.38) $\eta_{max} = 2/\lambda_{max} = 2/.84 = 2.38$ will cause divergences as is evident for $\eta = 2.5$.

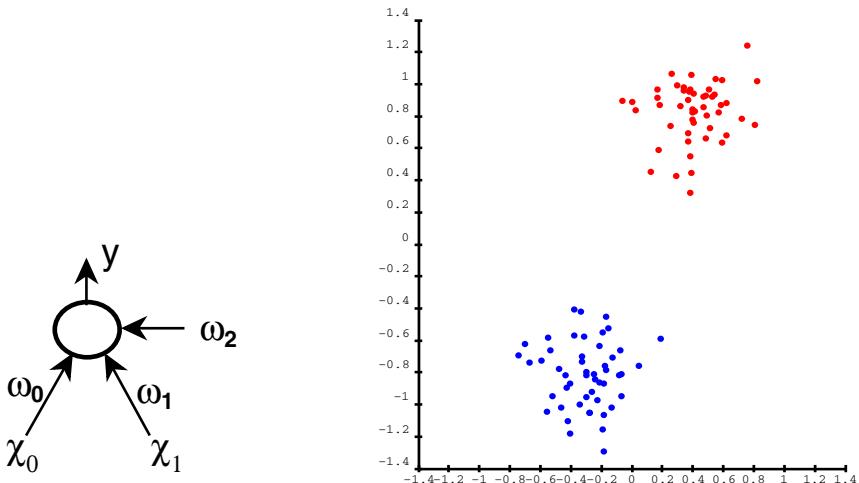


Fig. 1.9. Simple linear network

Fig. 1.10. Two classes drawn from gaussian distributions centered at $(-0.4, -0.8)$ and $(0.4, 0.8)$

Figure 1.12 shows the same example using stochastic instead of batch mode learning. Here, a learning rate of $\eta = 0.2$ is used. One can see that the trajectory is much noisier than in batch mode since only an estimate of the gradient is used at each iteration. The cost is plotted as a function of epoch. An epoch here is simply defined as 100 input presentations which, for stochastic learning, corresponds to 100 weight updates. In batch, an epoch corresponds to one weight update.

Multilayer Network. Figure 1.14 shows the architecture for a very simple multilayer network. It has 1 input, 1 hidden, and 1 output node. There are 2 weights and 2 biases. The activation function is $f(x) = 1.71 \tanh((2/3)x)$. The training set contains 10 examples from each of 2 classes. Both classes are Gaussian distributed with standard deviation 0.4. Class 1 has a mean of -1 and class 2 has a mean of +1. Target values are -1 for class 1 and +1 for class 2. Figure 1.13 shows the stochastic trajectory for the example.

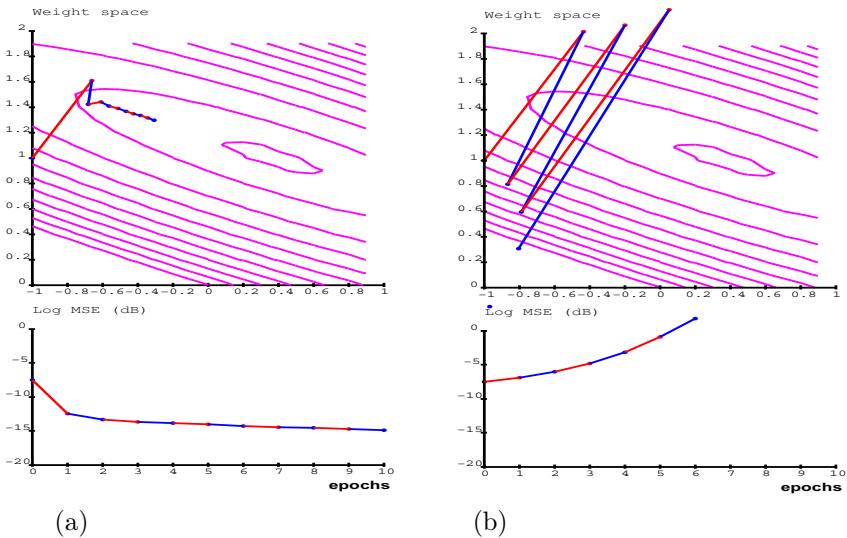


Fig. 1.11. Weight trajectory and error curve during learning for (a) $\eta = 1.5$ and (b) $\eta = 2.5$

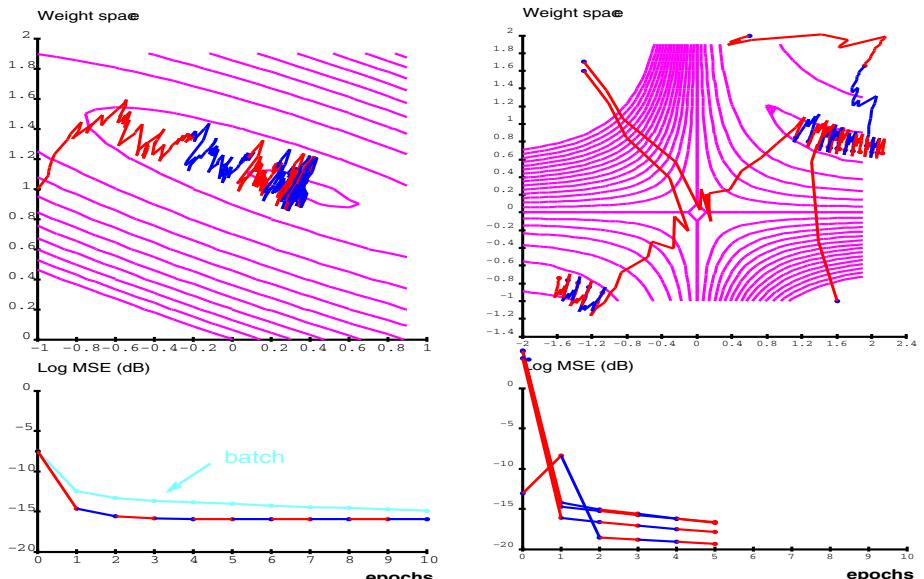


Fig. 1.12. Weight trajectory and error curve during stochastic learning for $\eta = 0.2$

Fig. 1.13. Weight trajectories and errors for 1-1 network trained using stochastic learning

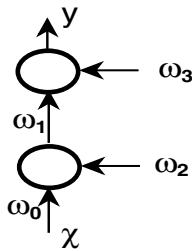


Fig. 1.14. The minimal multilayer network

1.5.3 Input Transformations and Error Surface Transformations Revisited

We can use the results of the previous section to justify several of the tricks discussed earlier.

Subtract the means from the input variables

The reason for the above trick is that a nonzero mean in the input variables creates a *very large* eigenvalue. This means the condition number will be large, i.e. the cost surface will be steep in some directions and shallow in others so that convergence will be very slow. The solution is to simply preprocess the inputs by subtracting their means.

For a single linear neuron, the eigenvectors of the Hessian (with means subtracted) point along the principal axes of the cloud of training vectors (recall Figure 1.8). Inputs that have a large variation in spread along different directions of the input space will have a large condition number and slow learning. And so we recommend:

Normalize the variances of the input variables.

If the input variables are correlated, this will not make the error surface spherical, but it will possibly reduce its eccentricity.

Correlated input variables usually cause the eigenvectors of H to be rotated away from the coordinate axes (Figure 1.7a versus 1.7b) thus weight updates are not decoupled. Decoupled weights make the “one learning rate per weight” method optimal, thus, we have the following trick:

Decorrelate the input variables.

Now suppose that the input variables of a neuron have been decorrelated, the Hessian for this neuron is then diagonal and its eigenvalues point along the coordinate axes. In such a case the gradient is not the best descent direction as can be seen in Fig 1.7b. At the point P, an arrow shows that gradient does not point towards the minimum. However, if we instead assign each weight its own learning rate (equal the inverse of the corresponding eigenvalue) then the

descent direction will be in the direction of the other arrow that points directly towards the minimum:

Use a separate learning rate for each weight.

1.6 Classical Second Order Optimization Methods

In the following we will briefly introduce the Newton, conjugate gradient, Gauss-Newton, Levenberg Marquardt and the Quasi-Newton (BFGS) method (see also [11, 34, 3, 5]).

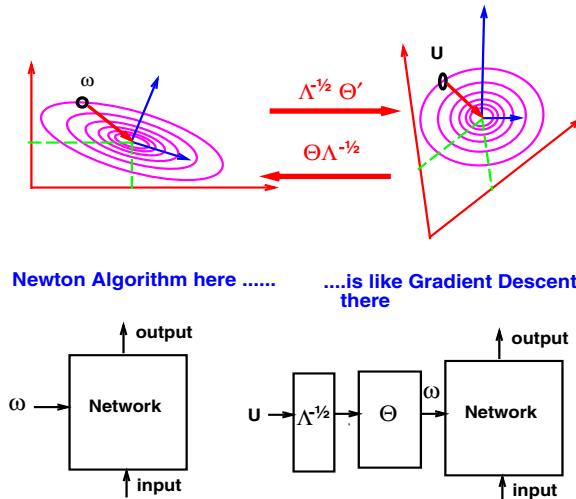


Fig. 1.15. Sketch of the whitening properties of the Newton algorithm

1.6.1 Newton Algorithm

To get an understanding of the Newton method let us recapitulate the results from section 1.5.1. Assuming a quadratic loss function E (see Eq.(1.21)) as depicted in Figure 1.6(ii), we can compute the weight update along the lines of Eq.(1.21)-(1.23)

$$\Delta w = \eta \left(\frac{\partial^2 E}{\partial w^2} \right)^{-1} \frac{\partial E}{\partial w} = \eta H(w)^{-1} \frac{\partial E}{\partial w}, \quad (1.40)$$

where η must be chosen in the range $0 < \eta < 1$ since E is in practice not perfectly quadratic. In this equation information about the Hessian H is taken into account. If the error function was quadratic one step would be sufficient to converge.

Usually the energy surface around the minimum is rather ellipsoid, or in the extreme like a taco shell, depending on the conditioning of the Hessian. A whitening transform, well known from signal processing literature [29] can change this ellipsoid shape to a spherical shape through $u = \Theta \Lambda^{1/2} w$ (see Figure 1.15 and

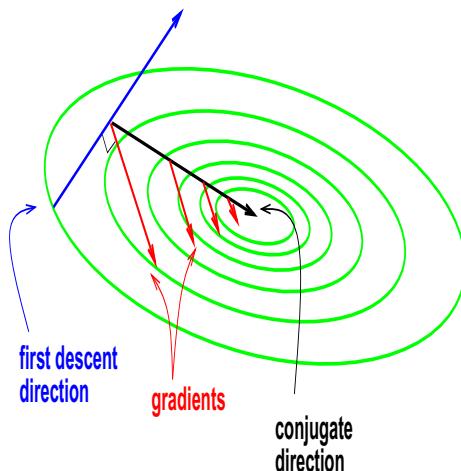


Fig. 1.16. Sketch of conjugate gradient directions in a 2D error surface

Eq.(1.34)). So the inverse Hessian in Eq.(1.40) basically spheres out the error surface locally. The following two approaches can be shown to be equivalent: (a) use the Newton algorithm in an untransformed weight space and (b) do usual gradient descent in a whitened coordinate system (see Figure 1.15) [19].

Summarizing, the Newton algorithm converges in one step if the error function is quadratic and (unlike gradient descent) it is invariant with respect to linear transformations of the input vectors. This means that the convergence time is not affected by shifts, scaling and rotation of input vectors. However one of the main drawbacks is that an $N \times N$ Hessian matrix must be stored and inverted, which takes $O(N^3)$ per iterations and is therefore impractical for more than a few variables. Since the error function is in general non-quadratic, there is no guarantee of convergence. If the Hessian is not positive definite (if it has some zero or even negative Eigenvalues where the error surface is flat or some directions are curved downward), then the Newton algorithm will diverge, so the Hessian *must be* positive definite. Of course the Hessian matrix of multilayer networks is in general not positive definite everywhere. For these reasons the Newton algorithm in its original form is not usable for general neural network learning. However it gives good insights for developing more sophisticated algorithms, as discussed in the following.

1.6.2 Conjugate Gradient

There are several important properties in conjugate gradient optimization: (1) it is a $O(N)$ method, (2) it doesn't use the Hessian explicitly, (3) it attempts to find descent directions that try to minimally spoil the result achieved in the previous iterations, (4) it uses a line search, and most importantly, (5) it works only for batch learning.

The third property is shown in Figure 1.16. Assume we pick a descent direction, e.g. the gradient, then we minimize along a line in this direction (line search). Subsequently we should try to find a direction along which the gradient does not change its direction, but merely its length (conjugate direction), because moving along this direction will not spoil the result of the previous iteration. The evolution of the descent directions ρ_k at iteration k is given as

$$\rho_k = -\nabla E(w_k) + \beta_k \rho_{k-1}, \quad (1.41)$$

where the choice of β_k can be done either according to Fletcher and Reeves [34]

$$\beta_k = \frac{\nabla E(w_k)^T \nabla E(w_k)}{\nabla E(w_{k-1})^T \nabla E(w_{k-1})} \quad (1.42)$$

or Polak and Ribiere

$$\beta_k = \frac{(\nabla E(w_k) - \nabla E(w_{k-1}))^T \nabla E(w_k)}{\nabla E(w_{k-1})^T \nabla E(w_{k-1})}. \quad (1.43)$$

Two directions ρ_k and ρ_{k-1} are defined as conjugate if

$$\rho_k^T H \rho_{k-1} = 0,$$

i.e. conjugate directions are orthogonal directions in the space of an identity Hessian matrix (see Figure 1.17). Very important for convergence in both choices is a good line search procedure. For a perfectly quadratic function with N variables a convergence within N steps can be proved. For non-quadratic functions Polak and Ribiere's choice seems more robust. Conjugate gradient (1.41) can also be viewed as a smart choice for choosing the momentum term known in neural network training. It has been applied with large success in multi-layer network training on problems that are moderate sized with rather low redundancy in the data. Typical applications range from function approximation, robotic control [39], time-series prediction and other real valued problems where high accuracy is wanted. Clearly on large and redundant (classification) problems stochastic backpropagation is faster. Although attempts have been made to define mini-batches [25], the main disadvantage of conjugate gradient methods remains that it is a batch method (partly due to the precision requirements in line search procedure).

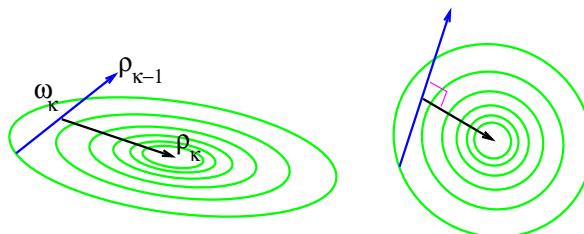


Fig. 1.17. Sketch of conjugate gradient directions in a 2D error surface

1.6.3 Quasi-Newton (BFGS)

The Quasi-Newton (BFGS) method (1) iteratively computes an estimate of the inverse Hessian, (2) is an $O(N^2)$ algorithm, (3) requires line search and (4) it works only for batch learning.

The positive definite estimate of the inverse Hessian is done directly without requiring matrix inversion and by only using gradient information. Algorithmically this can be described as follows: (1) first a positive definite matrix M is chosen, e.g. $M = I$, (2) then the search direction is set to

$$\rho(t) = M(t)\nabla E(w(t)),$$

(3) a line search is performed along ρ , which gives the update for the parameters at time t

$$w(t) = w(t-1) - \eta(t)\rho(t).$$

Finally (4) the estimate of the inverse Hessian is updated. Compared to the Newton algorithm the Quasi-Newton approach only needs gradient information. The most successful Quasi-Newton algorithm is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method. The update rule for the estimate of the inverse Hessian is

$$M(t) = M(t-1) \left(1 + \frac{\phi^T M \phi}{\delta^T \phi} \right) \frac{\delta \delta^T}{\delta^T \phi} - \left(\frac{\delta \phi^T M + M \phi \delta^T}{\delta^T \phi} \right), \quad (1.44)$$

where some abbreviations have been used for the following $N \times 1$ vectors

$$\begin{aligned} \phi &= \nabla E(w(t)) - \nabla E(w(t-1)) \\ \delta &= w(t) - w(t-1). \end{aligned} \quad (1.45)$$

Although, as mentioned above, the complexity is only $O(N^2)$, we are still required to store a $N \times N$ matrix, so the algorithm is only practical for small networks with non-redundant training sets. Recently some variants exist that aim to reduce storage requirements (see e.g. [3]).

1.6.4 Gauss-Newton and Levenberg Marquardt

Gauss-Newton and Levenberg Marquardt algorithm (1) use the square Jacobi approximation, (2) are mainly designed for batch learning, (3) have a complexity of $O(N^3)$ and (4) most important, they work only for mean squared error loss functions. The Gauss-Newton algorithm is like the Newton algorithm, however the Hessian is approximated by the square of the Jacobian (see also section 1.7.2 for a further discussion)

$$\Delta w = \left(\sum_p \frac{\partial f(w, x_p)^T}{\partial w} \frac{\partial f(w, x_p)}{\partial w} \right)^{-1} \nabla E(w). \quad (1.46)$$

The Levenberg Marquardt method is like the Gauss-Newton above, but it has a regularization parameter μ that prevents it from blowing up, if some eigenvalues are small

$$\Delta w = \left(\sum_p \frac{\partial f(w, x_p)}{\partial w}^T \frac{\partial f(w, x_p)}{\partial w} + \mu I \right)^{-1} \nabla E(w), \quad (1.47)$$

where I denotes the unity matrix. The Gauss Newton method is valid for quadratic cost functions however a similar procedure also works with Kullback-Leibler cost and is called Natural Gradient (see e.g. [1, 44, 2]).

1.7 Tricks to Compute the Hessian Information in Multilayer Networks

We will now discuss several techniques aimed at computing full or partial Hessian information by (a) finite difference method, (b) square Jacobian approximation (for Gauss-Newton and Levenberg-Marquardt algorithm), (c) computation of the diagonal of the Hessian and (d) by obtaining a product of the Hessian and a vector without computing the Hessian. Other semi-analytical techniques that allow the computation of the full Hessian are omitted because they are rather complicated and also require many forward/backward propagation steps [5, 8].

1.7.1 Finite Difference

We can write the k -th line of the Hessian

$$H^{(k)} = \frac{\partial(\nabla E(w))}{\partial w_k} \sim \frac{\nabla E(w + \delta \phi_k) - \nabla E(w)}{\delta},$$

where $\phi_k = (0, 0, 0, \dots, 1, \dots, 0)$ is a vector of zeros and only one 1 at the k -th position. This can be implemented with a simple recipe: (1) compute the total gradient by multiple forward and backward propagation steps. (2) Add δ to the k -th parameter and compute again the gradient, and finally (3) subtract both results and divide by δ . Due to numerical errors in this computation scheme the resulting Hessian might not be perfectly symmetric. In this case it should be *symmetrized* as described below.

1.7.2 Square Jacobian Approximation for the Gauss-Newton and Levenberg-Marquardt Algorithms

Assuming a mean squared cost function

$$E(w) = \frac{1}{2} \sum_p (d_p - f(w, x_p))^T (d_p - f(w, x_p)) \quad (1.48)$$

then the gradient is

$$\frac{\partial E(w)}{\partial w} = - \sum_p (d_p - f(w, x_p))^T \frac{\partial f(w, x_p)}{\partial w} \quad (1.49)$$

and the Hessian follows as

$$H(w) = \sum_p \frac{\partial f(w, x_p)}{\partial w}^T \frac{\partial f(w, x_p)}{\partial w} + \sum_p (d_p - f(w, x_p))^T \frac{\partial^2 f(w, x_p)}{\partial w \partial w}. \quad (1.50)$$

A simplifying approximation of the Hessian is the square of the Jacobian which is a positive semi-definite matrix of dimension: $N \times O$

$$H(w) \sim \sum_p \frac{\partial f(w, x_p)}{\partial w}^T \frac{\partial f(w, x_p)}{\partial w}, \quad (1.51)$$

where the second term from Eq.(1.50) was dropped. This is equivalent to assuming that the network is a linear function of the parameters w . Again this is readily implemented for the k -th column of the Jacobian: for all training patterns, (1) we forward propagate, then (2) set the activity of the output units to 0 and only the k -th output to 1, (3) a backpropagation step is taken and the gradient is accumulated.

1.7.3 Backpropagating Second Derivatives

Let us consider a multi-layer system with some functional blocks with N_i inputs, N_o outputs and N parameters of the form $O = F(W, X)$. Now assume we knew $\partial^2 E / \partial O^2$, which is a $N_o \times N_o$ matrix. Then it is straight forward to compute this matrix

$$\frac{\partial^2 E}{\partial W^2} = \frac{\partial O}{\partial W}^T \frac{\partial^2 E}{\partial O^2} \frac{\partial O}{\partial W} + \frac{\partial E}{\partial O} \frac{\partial^2 O}{\partial W^2}. \quad (1.52)$$

We can drop the second term in Eq.(1.52) and the resulting estimate of the Hessian is positive semi-definite. A further reduction is achieved, if we ignore all but the diagonal terms of $\frac{\partial^2 E}{\partial O^2}$:

$$\frac{\partial^2 E}{\partial w_i^2} = \sum_k \frac{\partial^2 E}{\partial o_k^2} \left(\frac{\partial o_k}{\partial w_i} \right)^2. \quad (1.53)$$

A similar derivation can be done to obtain the N_i times N_i matrix $\partial^2 E / \partial x^2$.

1.7.4 Backpropagating the Diagonal Hessian in Neural Nets

Backpropagation procedures for computing the diagonal Hessian are well known [18, 4, 19]. It is assumed that each layer in the network has the functional form $o_i = f(y_i) = f(\sum_j w_{ij} x_j)$ (see Figure 1.18 for the sigmoidal network). Using the Gauss-Newton approximation (dropping the term that contain $f''(y)$) we obtain:

$$\frac{\partial^2 E}{\partial y_k^2} = \frac{\partial^2 E}{\partial o_k^2} (f'(y_k))^2, \quad (1.54)$$

$$\frac{\partial^2 E}{\partial w_{ki}^2} = \frac{\partial^2 E}{\partial y_k^2} x_i^2 \quad (1.55)$$

and

$$\frac{\partial^2 E}{\partial x_i^2} \sum_k \frac{\partial^2 E}{\partial y_k^2} w_{ki}^2. \quad (1.56)$$

With f being a Gaussian nonlinearity as shown in Figure 1.18 for the RBF networks we obtain

$$\frac{\partial^2 E}{\partial w_{ki}^2} = \frac{\partial^2 E}{\partial y_k^2} (x_i - w_{ki})^2 \quad (1.57)$$

and

$$\frac{\partial^2 E}{\partial x_i^2} = \sum_k \frac{\partial^2 E}{\partial y_k^2} (x_i - w_{ki})^2. \quad (1.58)$$

The cost of computing the diagonal second derivatives by running these equations from the last layer to the first one is essentially the same as the regular backpropagation pass used for the gradient, except that the square of the weights are used in the weighted sums. This technique is applied in the “optimal brain damage” pruning procedure (see [21]).

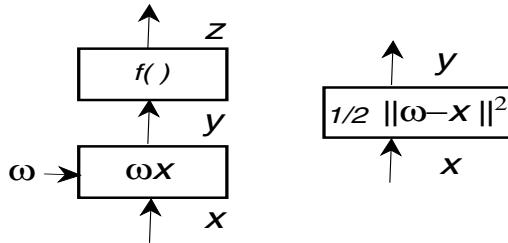


Fig. 1.18. Backpropagating the diagonal Hessian: sigmoids (left) and RBFs (right)

1.7.5 Computing the Product of the Hessian and a Vector

In many methods that make use of the Hessian, the Hessian is used exclusively in products with a vector. Interestingly, there is a way of computing such products *without* going through the trouble of computing the Hessian itself. The finite difference method can fulfill this task for an arbitrary vector Ψ

$$H\Psi \sim \frac{1}{\alpha} \left(\frac{\partial E}{\partial w}(w + \alpha\Psi) - \frac{\partial E}{\partial w}(w) \right), \quad (1.59)$$

using only two gradient computations (at point w and $w + \alpha\Psi$ respectively), which can be readily computed with backprop (α is a small constant).

This method can be applied to compute the principal eigenvector and eigenvalue of H by the power method. By iterating and setting

$$\Psi(t+1) = \frac{H\Psi(t)}{\|\Psi(t)\|}, \quad (1.60)$$

the vector $\Psi(t)$ will converge to the largest eigenvector of H and $\|\Psi(t)\|$ to the corresponding eigenvalue [23, 14, 10]. See also [33] for an even more accurate method that (1) does not use finite differences and (2) has similar complexity.

1.8 Analysis of the Hessian in Multi-layer Networks

It is interesting to understand how some of the tricks shown previously influence on the Hessian, i.e. how does the Hessian change with architecture and details of the implementation. Typically, the eigenvalue distribution of the Hessian looks like the one sketched in Figure 1.20: a few small eigenvalues, many medium ones and few very large ones. We will now argue that the *large eigenvalues* will cause the trouble in the training process because [23, 22]

- non-zero mean inputs or neuron states [22] (see also chapter 10)
- wide variations of the second derivatives from layer to layer
- correlation between state variables.

To exemplify this, we show the eigenvalue distribution of a network trained on OCR data in Figure 1.20. Clearly, there is a wide spread of eigenvalues (see Figure 1.19) and we observe that the ratio between e.g. the first and the eleventh eigenvalue is about 8. The long tail of the eigenvalue distribution (see Figure 1.20) is rather painful because the ratio between the largest and smallest eigenvalue gives the conditioning of the learning problem. A large ratio corresponds to a big difference in the axis of the ellipsoidal shaped error function: the larger the ratio, the more we find a taco-shell shaped minima, which are extremely steep towards the small axis and very flat along the long axis.

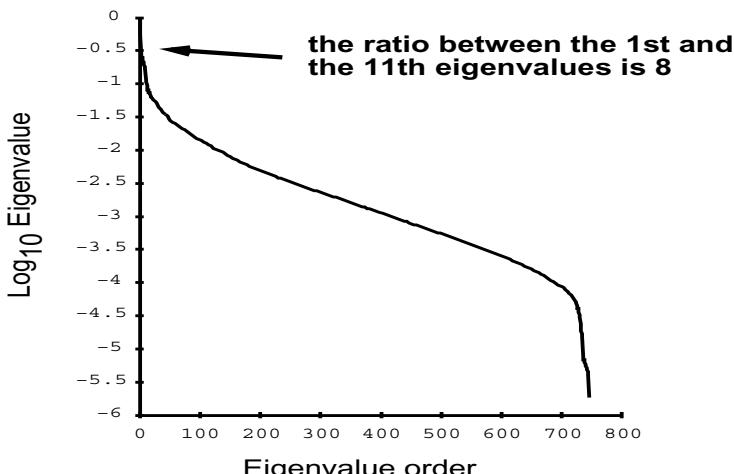


Fig. 1.19. Eigenvalue spectrum in a 4 layer shared weights network ($256 \times 128 \times 64 \times 10$) trained on 320 handwritten digits

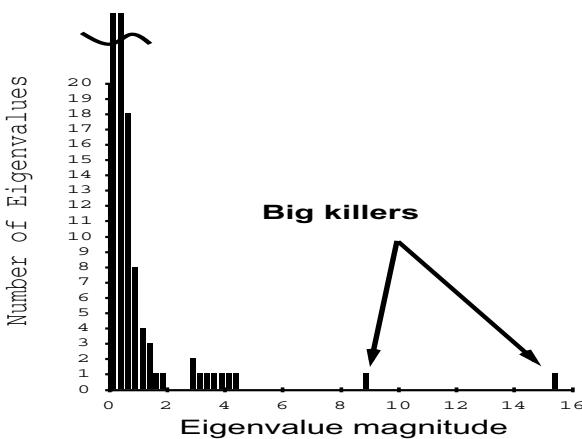


Fig. 1.20. Eigenvalue spectrum in a 4 layer shared weights network ($256 \times 128 \times 64 \times 10$) trained on 320 handwritten digits

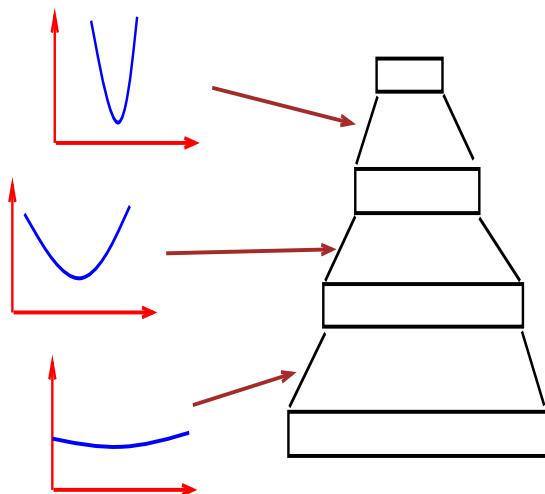


Fig. 1.21. Multilayered architecture: the second derivative is often smaller in lower layers

Another general characteristic of the Hessian in multi-layer networks is the spread between layers. In Figure 1.21 we roughly sketch how the shape of the Hessian varies from being rather flat in the first layer to being quite steep in the last layer. This affects the learning speed and can provide an ingredient to explain the slow learning in lower layers and the fast (sometime oscillating) learning in the last layer. A trick to compensate this different scale of learning is to use the inverse diagonal Hessian to control the learning rate (see also section 1.6, chapter 17).

1.9 Applying Second Order Methods to Multilayer Networks

Before we concentrate in this section on how to tailor second order techniques for training large networks, let us first repeat some rather pessimistic facts about applying classical second order methods. Techniques using full Hessian information (Gauss -Newton, Levenberg-Marquardt and BFGS) can only apply to very small networks trained in batch mode, however those small networks are not the ones that need speeding up the most. Most second order methods (conjugate gradient, BFGS, ...) require a line-search and can therefore not be used in the stochastic mode. Many of the tricks discussed previously apply only to batch learning. From our experience we know that a carefully tuned stochastic gradient descent is hard to beat on large classification problems. For smaller problems that require accurate real-valued outputs like in function approximation or control problems, we see that conjugate gradient (with Polak-Ribiere Eq.(1.43)) offers the best combination of speed, reliability and simplicity. Several attempts using “mini batches” in applying conjugate gradient to large and redundant problems have been made recently [17, 25, 31]. A variant of conjugate gradient optimization (called scaled CG) seems interesting: here the line search procedure is replaced by a 1D Levenberg Marquardt type algorithm [24].

1.9.1 A Stochastic Diagonal Levenberg Marquardt Method

To obtain a stochastic version of the Levenberg Marquardt algorithm the idea is to compute the diagonal Hessian through a running estimate of the second derivative with respect to each parameter. The instantaneous second derivative can be obtained via backpropagation as shown in the formulas of section 1.7. As soon as we have those running estimates we can use them to compute individual learning rates for each parameter

$$\eta_{ki} = \frac{\epsilon}{\langle \frac{\partial^2 E}{\partial w_{ki}^2} \rangle + \mu}, \quad (1.61)$$

where ϵ denotes the global learning rate, and $\langle \frac{\partial^2 E}{\partial w_{ki}^2} \rangle$ is a running estimate of the diagonal second derivative with respect to w_{ki} . μ is a parameter to prevent η_{ki}

from blowing up in case the second derivative is small, i.e. when the optimization moves in flat parts of the error function. The running estimate is computed as

$$\langle \frac{\partial^2 E}{\partial w_{ki}^2} \rangle_{new} = (1 - \gamma) \langle \frac{\partial^2 E}{\partial w_{ki}^2} \rangle_{old} + \gamma \frac{\partial^2 E^p}{\partial w_{ki}^2}, \quad (1.62)$$

where γ is a small constant that determines the amount of memory that is being used. The second derivatives can be computed prior to training over e.g. a subset of the training set. Since they change only very slowly they only need to be reestimated every few epochs. Note that the additional cost over regular backpropagation is negligible and convergence is – as a rule of thumb – about three times faster than a carefully tuned stochastic gradient algorithm.

In Figure 1.22 and 1.23 we see the convergence of the stochastic diagonal Levenberg Marquardt method (1.61) for a toy example with two different sets of learning rates. Obviously the experiment shown Figure 1.22 contains fewer fluctuations than in Figure 1.23 due to smaller learning rates.

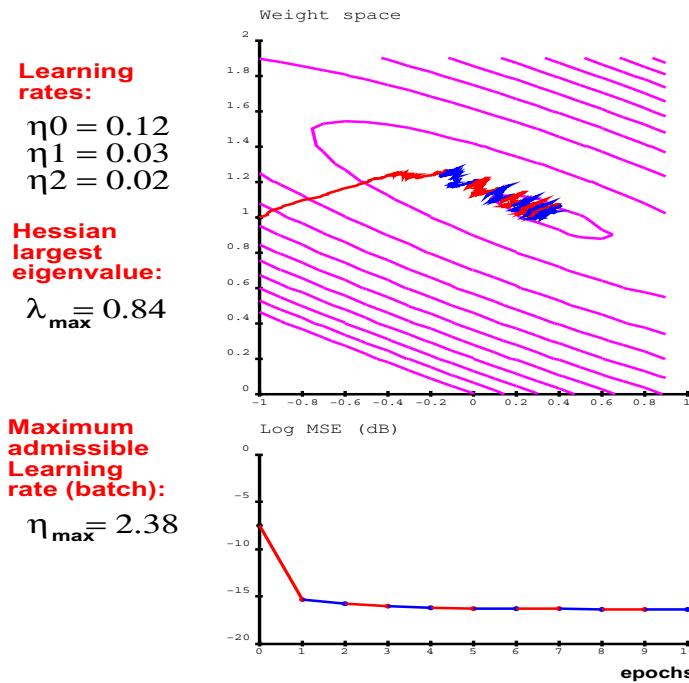


Fig. 1.22. Stochastic diagonal Levenberg-Marquardt algorithm. Data set from 2 Gaussians with 100 examples. The network has one linear unit, 2 inputs and 1 output, i.e. three parameters (2 weights, 1 bias).

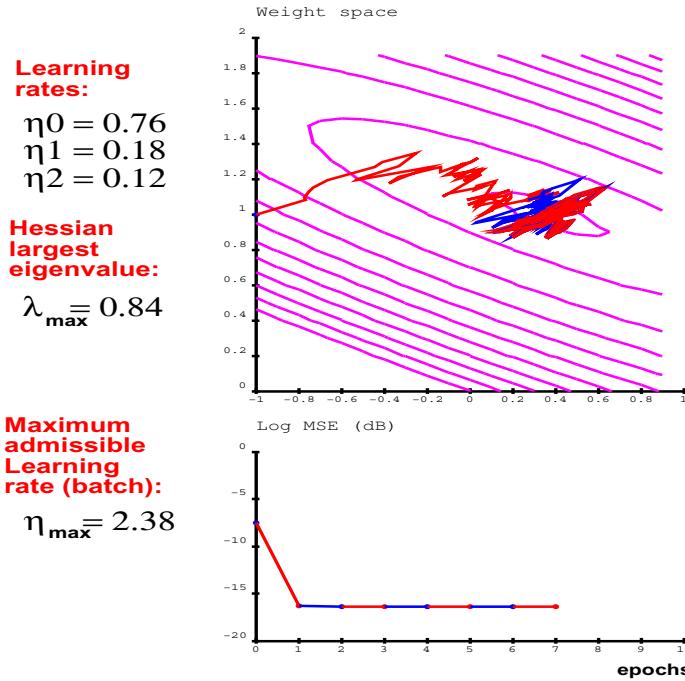


Fig. 1.23. Stochastic diagonal Levenberg-Marquardt algorithm. Data set from 2 Gaussians with 100 examples. The network has one linear unit, 2 inputs and 1 output, i.e. three parameters (2 weights, 1 bias).

1.9.2 Computing the Principal Eigenvector/Vector of the Hessian

In the following we give three tricks for computing the principal eigenvalue/Vector of the Hessian without having to compute the Hessian itself. Remember that in section 1.4.7 we also introduced a method to approximate the smallest eigenvector of the Hessian (without having to compute the Hessian) through averaging (see also [28]).

Power Method. We repeat the result of our discussion in section 1.7.5: starting from a random initial vector Ψ , the iteration

$$\Psi_{new} = H \frac{\Psi_{old}}{\|\Psi_{old}\|},$$

will eventually converge to the principal eigenvector (or a vector in the principal eigenspace) and $\|\Psi_{old}\|$ will converge to the corresponding eigenvalue [14, 10].

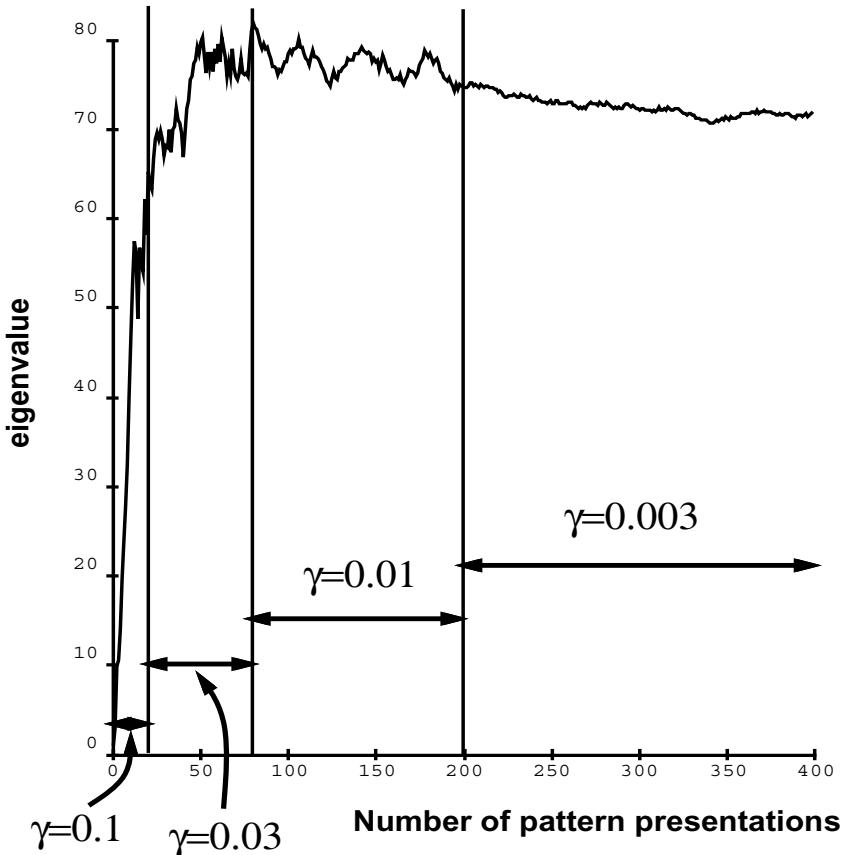


Fig. 1.24. Evolution of the eigenvalue as a function of the number of pattern presentations for a shared weight network with 5 layers, 64638 connections and 1278 free parameters. The training set consists of 1000 handwritten digits.

Taylor Expansion. Another method makes use of the fact that small perturbations of the gradient also lead to the principal eigenvector of H

$$\Psi_{new} = \frac{1}{\alpha} \left(\frac{\partial E}{\partial w}(w + \alpha \frac{\Psi_{old}}{\|\Psi_{old}\|}) - \frac{\partial E}{\partial w}(w) \right), \quad (1.63)$$

where α is a small constant. One iteration of this procedure requires two forward and two backward propagation steps for each pattern in the training set.

Online Computation of Ψ . The following rule makes use of the running average to obtain the largest eigenvalue of the average Hessian very fast

$$\Psi_{new} = (1 - \gamma)\Psi + \frac{1}{\alpha} \left(\frac{\partial E^p}{\partial w}(w + \alpha \frac{\Psi_{old}}{\|\Psi_{old}\|}) - \frac{\partial E}{\partial w}(w) \right). \quad (1.64)$$

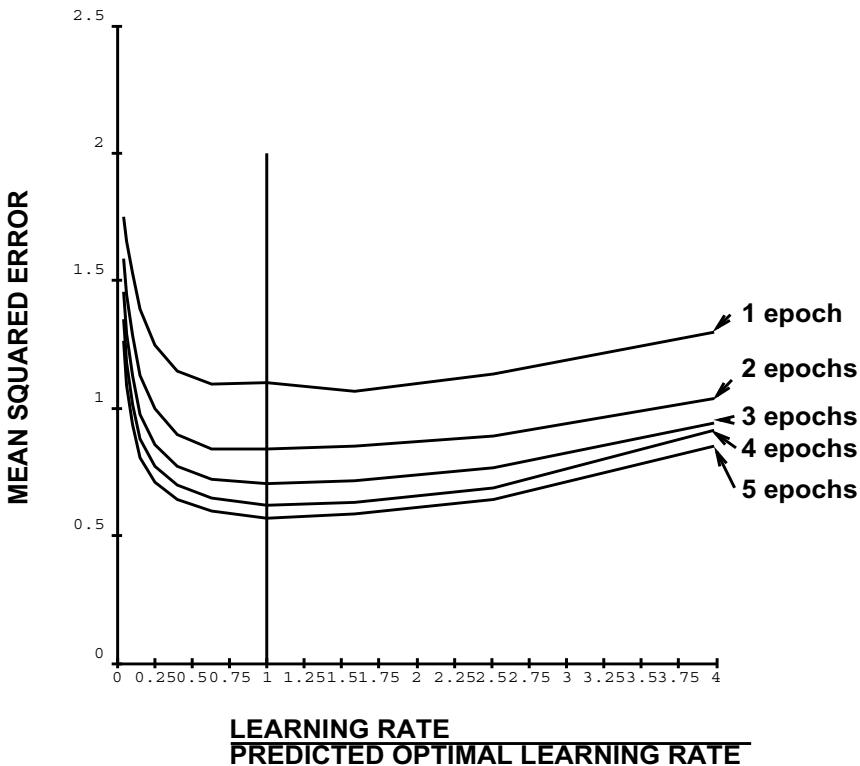


Fig. 1.25. Mean squared error as a function of the ratio between learning rate and predicted optimal learning rate for a fully connected network ($784 \times 30 \times 10$). The training set consists of 300 handwritten digits.

To summarize, the eigenvalue/vector computations:

1. a random vector is chosen for initialization of Ψ ,
2. an input pattern is presented with desired output, a forward and backward propagation, step is performed and the gradients $G(w)$ are stored,
3. $\alpha \frac{\Psi_{old}}{\|\Psi_{old}\|}$ is added to the current weight vector w ,
4. a forward and backward propagation step is performed with the perturbed weight vector and the gradients $G(w')$ are stored,
5. the difference $1/\alpha(G(w') - G(w))$ is computed and the running average of the eigenvector is updated,
6. we loop from (2)-(6) until a reasonably stable result is obtained for Ψ ,
7. the optimal learning rate is then given as

$$\eta_{opt} = \frac{1}{\|\Psi\|}.$$

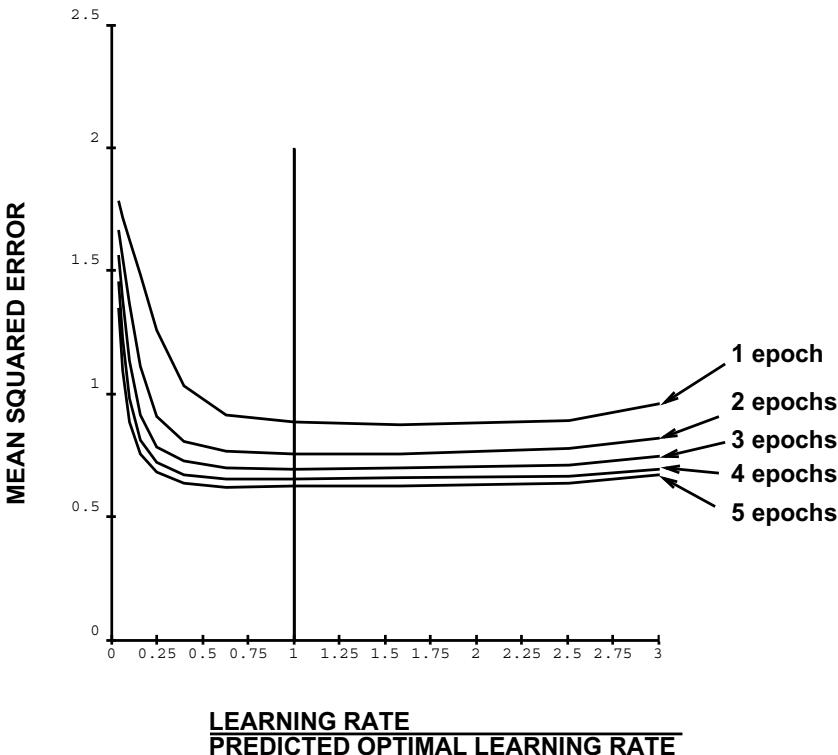


Fig. 1.26. Mean squared error as a function of the ratio between learning rate and predicted optimal learning rate for a shared weight network with 5 layers ($1024 \times 1568 \times 392 \times 400 \times 100 \times 10$), 64638 (local) connections and 1278 free parameters (shared weights). The training set consists of 1000 handwritten digits.

In Figure 1.24 we see the evolution of the eigenvalue as a function of the number of pattern presentations for a neural network in a handwritten character recognition task. In practice we adapt the leak size of the running average in order to get fewer fluctuations (as also indicated on the figure). In the figure we see that after fewer than 100 pattern presentations the correct order of magnitude for the eigenvalue, i.e. the learning rate is reached. From the experiments we also observe that the fluctuations of the average Hessian over training are small.

In Figure 1.25 and 1.26 we start with the same initial conditions, and perform a fixed number of epochs with learning rates computed by multiplying the predicted learning rate by a predefined constant. Choosing constant 1 (i.e. using the predicted optimal rate) always gives residual errors which are very close to the error achieved by the best choice of the constant. In other words, the “predicted optimal rate” is optimal enough.

1.10 Discussion and Conclusion

According to the recommendations mentioned above, a practitioner facing a multi-layer neural net training problem would go through the following steps:

- shuffle the examples
- center the input variables by subtracting the mean
- normalize the input variable to a standard deviation of 1
- if possible, decorrelate the input variables.
- pick a network with the sigmoid function shown in figure 1.4
- set the target values within the range of the sigmoid, typically +1 and -1.
- initialize the weights to random values as prescribed by 1.16.

The preferred method for training the network should be picked as follows:

- if the training set is large (more than a few hundred samples) and redundant, and if the task is classification, use stochastic gradient with careful tuning, or use the stochastic diagonal Levenberg Marquardt method.
- if the training set is not too large, or if the task is regression, use conjugate gradient.

Classical second-order methods are impractical in almost all useful cases.

The non-linear dynamics of stochastic gradient descent in multi-layer neural networks, particularly as it pertains to generalization, is still far from being well understood. More theoretical work and systematic experimental work is needed.

Acknowledgement. Y.L. & L.B. & K.-R. M. gratefully acknowledge mutual exchange grants from DAAD and NSF.

References

- [1] Amari, S.: Neural learning in structured parameter spaces — natural riemannian gradient. In: Mozer, M.C., Jordan, M.I., Petsche, T. (eds.) *Advances in Neural Information Processing Systems*, vol. 9, p. 127. MIT Press (1997)
- [2] Amari, S.: Natural gradient works efficiently in learning. *Neural Computation* 10(2), 251–276 (1998)
- [3] Battiti, R.: First- and second-order methods for learning: Between steepest descent and newton's method. *Neural Computation* 4, 141–166 (1992)
- [4] Becker, S., LeCun, Y.: Improving the convergence of backpropagation learning with second oder metho ds. In: Touretzky, D., Hinton, G., Sejnowski, T. (eds.) *Proceedings of the 1988 Connectionist Models Summer School*, pp. 29–37. Lawrence Erlbaum Associates (1989)
- [5] Bishop, C.M.: *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford (1995)
- [6] Bottou, L.: Online algorithms and stochastic approximations. In: Saad, D. (ed.) *Online Learning in Neural Networks* (1997 Workshop at the Newton Institute). The Newton Institute Series. Cambridge University Press, Cambridge (1998)
- [7] Broomhead, D.S., Lowe, D.: Multivariable function interpolation and adaptive networks. *Complex Systems* 2, 321–355 (1988)

- [8] Buntine, W.L., Weigend, A.S.: Computing second order derivatives in Feed-Forward networks: A review. *IEEE Transactions on Neural Networks* (1993) (to appear)
- [9] Darken, C., Moody, J.E.: Note on learning rate schedules for stochastic optimization. In: Lippmann, R.P., Moody, J.E., Touretzky, D.S. (eds.) *Advances in Neural Information Processing Systems*, vol. 3, pp. 832–838. Morgan Kaufmann, San Mateo (1991)
- [10] Diamantaras, K.I., Kung, S.Y.: *Principal Component Neural Networks*. Wiley, New York (1996)
- [11] Fletcher, R.: *Practical Methods of Optimization*, ch. 8.7: Polynomial time algorithms, 2nd edn., pp. 183–188. John Wiley & Sons, New York (1987)
- [12] Geman, S., Bienenstock, E., Doursat, R.: Neural networks and the bias/variance dilemma. *Neural Computation* 4(1), 1–58 (1992)
- [13] Goldstein, L.: Mean square optimality in the continuous time Robbins Monro procedure. Technical Report DRB-306, Dept. of Mathematics, University of Southern California, LA (1987)
- [14] Golub, G.H., Van Loan, C.F.: *Matrix Computations*, 2nd edn. Johns Hopkins University Press, Baltimore (1989)
- [15] Heskes, T.M., Kappen, B.: On-line learning processes in artificial neural networks. In: Tayler, J.G. (ed.) *Mathematical Approaches to Neural Networks*, vol. 51, pp. 199–233. Elsevier, Amsterdam (1993)
- [16] Jacobs, R.A.: Increased rates of convergence through learning rate adaptation. *Neural Networks* 1, 295–307 (1988)
- [17] Kramer, A.H., Sangiovanni-Vincentelli, A.: Efficient parallel learning algorithms for neural networks. In: Touretzky, D.S. (ed.) *Proceedings of the 1988 Conference on Advances in Neural Information Processing Systems*, pp. 40–48. Morgan Kaufmann, San Mateo (1989)
- [18] LeCun, Y.: Modèles connexionnistes de l'apprentissage (connectionist learning models). PhD thesis, Université P. et M. Curie, Paris VI (1987)
- [19] LeCun, Y.: Generalization and network design strategies. In: Pfeifer, R., Schreter, Z., Fogelman, F., Steels, L. (eds.) *Proceedings of the International Conference Connectionism in Perspective*, University of Zürich, October 10-13. Elsevier, Amsterdam (1988)
- [20] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Handwritten digit recognition with a backpropagation network. In: Touretzky, D.S. (ed.) *Advances in Neural Information Processing Systems*, vol. 2. Morgan Kaufmann, San Mateo (1990)
- [21] LeCun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In: Touretzky, D.S. (ed.) *Advances in Neural Information Processing Systems*, vol. 2, pp. 598–605 (1990)
- [22] LeCun, Y., Kanter, I., Solla, S.A.: Second order properties of error surfaces. In: *Advances in Neural Information Processing Systems*, vol. 3. Morgan Kaufmann, San Mateo (1991)
- [23] LeCun, Y., Simard, P.Y., Pearlmutter, B.: Automatic learning rate maximization by on-line estimation of the hessian's eigenvectors. In: Giles, Hanson, Cowan (eds.) *Advances in Neural Information Processing Systems*, vol. 5. Morgan Kaufmann, San Mateo (1993)
- [24] Møller, M.: A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks* 6, 525–533 (1993)
- [25] Møller, M.: Supervised learning on large redundant training sets. *International Journal of Neural Systems* 4(1), 15–25 (1993)

- [26] Moody, J.E., Darken, C.J.: Fast learning in networks of locally-tuned processing units. *Neural Computation* 1, 281–294 (1989)
- [27] Murata, N.: PhD thesis, University of Tokyo (1992) (in Japanese)
- [28] Murata, N., Müller, K.-R., Ziehe, A., Amari, S.: Adaptive on-line learning in changing environments. In: Mozer, M.C., Jordan, M.I., Petsche, T. (eds.) *Advances in Neural Information Processing Systems*, vol. 9, p. 599. MIT Press (1997)
- [29] Oppenheim, A.V., Schafer, R.W.: *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs (1975)
- [30] Orr, G.B.: Dynamics and Algorithms for Stochastic learning. PhD thesis, Oregon Graduate Institute (1995)
- [31] Orr, G.B.: Removing noise in on-line search using adaptive batch sizes. In: Mozer, M.C., Jordan, M.I., Petsche, T. (eds.) *Advances in Neural Information Processing Systems*, vol. 9, p. 232. MIT Press (1997)
- [32] Orr, M.J.L.: Regularization in the selection of radial basis function centers. *Neural Computation* 7(3), 606–623 (1995)
- [33] Pearlmutter, B.A.: Fast exact multiplication by the hessian. *Neural Computation* 6, 147–160 (1994)
- [34] Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: *Numerical Recipes in C: The art of Scientific Programming*. Cambridge University Press, Cambridge (1988)
- [35] Saad, D. (ed.): *Online Learning in Neural Networks* (1997 Workshop at the Newton Institute). The Newton Institute Series. Cambridge University Press, Cambridge (1998)
- [36] Saad, D., Solla, S.A.: Exact solution for on-line learning in multilayer neural networks. *Physical Review Letters* 74, 4337–4340 (1995)
- [37] Sompolinsky, H., Barkai, N., Seung, H.S.: On-line learning of dichotomies: algorithms and learning curves. In: Oh, J.-H., Kwon, C., Cho, S. (eds.) *Neural Networks: The Statistical Mechanics Perspective*, pp. 105–130. World Scientific, Singapore (1995)
- [38] Sutton, R.S.: Adapting bias by gradient descent: An incremental version of delta-bar-delta. In: Swartout, W. (ed.) *Proceedings of the 10th National Conference on Artificial Intelligence*, pp. 171–176. MIT Press, San Jose (July 1992)
- [39] van der Smagt, P.: Minimisation methods for training feed-forward networks. *Neural Networks* 7(1), 1–11 (1994)
- [40] Vapnik, V.: *The Nature of Statistical Learning Theory*. Springer, New York (1995)
- [41] Vapnik, V.: *Statistical Learning Theory*. Wiley, New York (1998)
- [42] Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., Lang, K.J.: Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing ASSP-37*, 328–339 (1989)
- [43] Wiegerinck, W., Komoda, A., Heskes, T.: Stochastic dynamics of learning with momentum in neural networks. *Journal of Physics A* 27, 4425–4437 (1994)
- [44] Yang, H.H., Amari, S.: The efficiency and the robustness of natural gradient descent learning rule. In: Jordan, M.I., Kearns, M.J., Solla, S.A. (eds.) *Advances in Neural Information Processing Systems*, vol. 10. MIT Press (1998)

Regularization Techniques to Improve Generalization*

Preface

Good tricks for regularization are extremely important for improving the generalization ability of neural networks. The first and most commonly used trick is **early stopping**, which was originally described in [11]. In its simplest version, the trick is as follows:

Take an independent validation set, e.g. take out a part of the training set, and monitor the error on this set during training. The error on the training set will decrease, whereas the error on the validation set will first decrease and then increase. The early stopping point occurs where the error on the validation set is the lowest. It is here that the network weights provide the best generalization.

As Lutz Prechelt points out in chapter 2, the above picture is highly idealized. In practice, the shape of the error curve on the validation set is more likely very ragged with multiple minima. Choosing the “best” early stopping point then involves a trade-off between (1) improvement of generalization and (2) speed of learning. If speed is not an issue then, clearly, the safest strategy is to train all the way until the minimum error on the training set is found, while monitoring the location of the lowest error rate on the validation set. Of course, this can take a prohibitive amount of computing time. This chapter presents less costly strategies employing a number of different stopping criteria, e.g. when the ratio between the *generalization loss* and the *progress* exceeds a given threshold (see p. 57). A large simulation study using various benchmark problems is used in the discussion and analysis of the differences (with respect to e.g. robustness, effectiveness, training time, ...) between these proposed **stopping criteria** (see p. 60ff.). So far theoretical studies [12, 1, 6] have not studied this trade-off.

Weight decay is also a commonly used technique for controlling capacity in neural networks. Early stopping is considered to be fast, but it is not well defined (keep in mind the pitfalls mentioned in chapter 2). On the other hand, weight decay regularizers [5, 2] are well understood, but finding a suitable parameter λ to control the strength of the weight decay term can be tediously time consuming. Thorsteinn Rögnvaldsson proposes a simple trick for **estimating** λ by making use of the *best of both worlds* (see p. 75): simply compute the gradient at the early stopping solution \mathbf{W}_{es} and divide it by the norm of \mathbf{W}_{es} ,

$$\hat{\lambda} = \|\nabla E(\mathbf{W}_{es})\| / \|\mathbf{W}_{es}\|.$$

Other penalties are also possible. The trick is speedy, since we neither have to do a complete training nor a scan of the whole λ parameter space, and the accuracy of the determined $\hat{\lambda}$ is good, as seen from some interesting simulations.

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

Tony Plate in chapter 4 treats the penalty factors for the weights (hyperparameters) along the **Bayesian framework** of MacKay [8] and Neal [9]. There are two levels in searching for the best network. The inner loop is a minimization of the training error keeping the hyperparameters fixed, whereas the outer loop searches the hyperparameter space with the goal of maximizing the evidence of having generated the data. This whole procedure is rather slow and computationally expensive, since, in theory, the inner search needs to converge (to a local minimum) at each outer loop search step. When applied to classification networks using the cross-entropy error function the outer-loop search can be unstable with the hyperparameter values oscillating wildly or going to inappropriate extremes. To make this Bayesian framework work better in practice, Tony Plate proposes a number of tricks that speed and simplify the **hyperparameter search** strategies (see p. 96). In particular, his search strategies center around the questions: (1) how often (when) should the hyperparameters be updated (see p. 96) and (2) what should be done if the Hessian is out-of-bounds (see p. 97ff.). To discuss the effects of the choices made in (1) and (2), Tony Plate uses simulations based on artificial examples and concludes with a concise set of rules for making the hyperparameter framework work better.

In chapter 5, Jan Larsen et al. formulate an iterative gradient descent scheme for **adapting** their **regularization parameters** (note, different regularizers can be used for input/hidden and hidden/output weights). The trick is simple: perform gradient descent on the validation set errors with respect to the regularization parameters, and iteratively use the results for updating the estimate of the regularization parameters (see p. 116). This method holds for a variety of penalty terms (e.g. weight decay). The computational overhead is negligible for computing the gradients, however, an inverse Hessian has to be estimated. If second order methods are used for training, then the inverse Hessian may already be available, so there is little additional effort. Otherwise obtaining full Hessian information is rather tedious and limits the approach to smaller applications (see discussion in chapter 1). Nevertheless approximations of the Hessian (e.g. diagonal) could also be used to limit the computation time. Jan Larsen, et al., demonstrate the applicability of their trick on classification (vowel data) and regression (time-series prediction) problems.

Averaging over multiple predictors is a well known method for improving generalization (see e.g. [10, 3, 7, 13]). David Horn et al. raises two questions in ensemble training: (1) how many predictors are “enough” and (2) how does the number of predictors affect the stopping criteria for early stopping (see p. 134). They present solutions for answering these questions by providing a method for estimating the error of an infinite number of predictors and they demonstrate the usefulness of their trick for the sunspot prediction task. Additional theoretical reasoning is given to explain their success in terms of variance minimization within the ensemble.

Jenny & Klaus

References

- [1] Amari, S., Murata, N., Müller, K.-R., Finke, M., Yang, H.H.: Asymptotic statistical theory of overtraining and cross-validation. *IEEE Transactions on Neural Networks* 8(5), 985–996 (1997)
- [2] Bishop, C.M.: *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford (1995)
- [3] Breiman, L.: Bagging predictors. *Machine Learning* 26(2), 123–140 (1996)
- [4] Cowan, J.D., Tesauro, G., Alspector, J. (eds.): *Advances in Neural Information Processing Systems 6*, San Mateo, CA. Morgan Kaufman Publishers Inc. (1994)
- [5] Girosi, F., Jones, M., Poggio, T.: Regularization theory and neural networks architectures. *Neural Computation* 7(2), 219–269 (1995)
- [6] Kearns, M.: A bound on the error of cross validation using the approximation and estimation rates, with consequences for the training-test split. *Neural Computation* 9(5), 1143–1161 (1997)
- [7] Lincoln, W.P., Skrzypek, J.: Synergy of clustering multiple back propagation networks. In: Touretzky, D.S. (ed.) *Advances in Neural Information Processing Systems 2*, San Mateo, CA, pp. 650–657. Morgan Kaufmann (1990)
- [8] McKay, D.J.C.: A practical Bayesian framework for backpropagation networks. *Neural Computation* 4, 448–472 (1992)
- [9] Neal, R.M.: *Bayesian Learning for Neural Networks*. Lecture Notes in Statistics, vol. 118. Springer, New York (1996)
- [10] Perrone, M.P.: *Improving Regression Estimation: Averaging Methods for Variance Reduction with Extensions to General Convex Measure Optimization*. PhD thesis, Brown University (May 1993)
- [11] Plaut, D.C., Nowlan, S.J., Hinton, G.E.: Experiments on learning by back-propagation. Technical Report Computer Science Dept. Tech. Report, Pittsburgh, PA (1986)
- [12] Wang, C., Venkatesh, S.S., Judd, J.S.: Optimal stopping and effective machine complexity in learning. In: [4] (1994)
- [13] Wolpert, D.H.: Stacked generalization. *Neural Networks* 5(2), 241–259 (1992)

Early Stopping — But When?*

Lutz Prechelt

Fakultät für Informatik, Universität Karlsruhe
 D-76128 Karlsruhe, Germany
 prechelt@ira.uka.de
<http://www.ipd.ira.uka.de/~prechelt/>

Abstract. Validation can be used to detect when overfitting starts during supervised training of a neural network; training is then stopped before convergence to avoid the overfitting (“early stopping”). The exact criterion used for validation-based early stopping, however, is usually chosen in an ad-hoc fashion or training is stopped interactively. This trick describes how to select a stopping criterion in a systematic fashion; it is a trick for either speeding learning procedures or improving generalization, whichever is more important in the particular situation. An empirical investigation on multi-layer perceptrons shows that there exists a tradeoff between training time and generalization: From the given mix of 1296 training runs using different 12 problems and 24 different network architectures I conclude slower stopping criteria allow for small improvements in generalization (here: about 4% on average), but cost *much* more training time (here: about factor 4 longer on average).

2.1 Early Stopping Is Not Quite as Simple

2.1.1 Why Early Stopping?

When training a neural network, one is usually interested in obtaining a network with optimal generalization performance. However, all standard neural network architectures such as the fully connected multi-layer perceptron are prone to overfitting [10]: While the network *seems* to get better and better, i.e., the error on the training set decreases, at some point during training it actually begins to get worse again, i.e., the error on unseen examples increases. The idealized expectation is that during training the generalization error of the network evolves as shown in Figure 2.1. Typically the generalization error is estimated by a validation error, i.e., the average error on a *validation set*, a fixed set of examples not from the training set.

There are basically two ways to fight overfitting: reducing the number of dimensions of the parameter space or reducing the effective size of each dimension.

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

Techniques for reducing the number of parameters are greedy constructive learning [7], pruning [5, 12, 14], or weight sharing [18]. Techniques for reducing the size of each parameter dimension are regularization, such as weight decay [13] and others [25], or early stopping [17]. See also [8, 20] for an overview and [9] for an experimental comparison.

Early stopping is widely used because it is simple to understand and implement and has been reported to be superior to regularization methods in many cases, e.g. in [9].

2.1.2 The Basic Early Stopping Technique

In most introductory papers on supervised neural network training one can find a diagram similar to the one shown in Figure 2.1. It is claimed to show the evolution over time of the per-example error on the training set and on a validation set not used for training (the *training error curve* and the *validation error curve*). Given this behavior, it is clear how to do early stopping using validation:

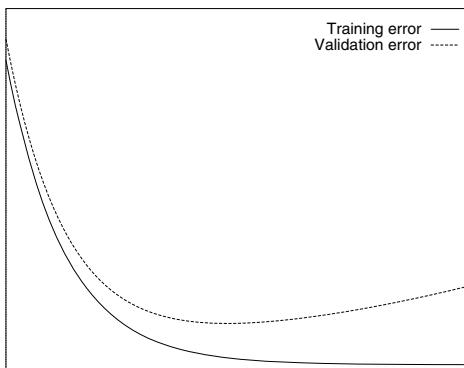


Fig. 2.1. Idealized training and validation error curves. Vertical: errors; horizontal: time.

1. Split the training data into a training set and a validation set, e.g. in a 2-to-1 proportion.
2. Train only on the training set and evaluate the per-example error on the validation set once in a while, e.g. after every fifth epoch.
3. Stop training as soon as the error on the validation set is higher than it was the last time it was checked.
4. Use the weights the network had in that previous step as the result of the training run.

This approach uses the validation set to anticipate the behavior in real use (or on a test set), assuming that the error on both will be similar: The validation error is used as an estimate of the generalization error.

2.1.3 The Ugliness of Reality

However, for real neural network training the validation set error does not evolve as smoothly as shown in Figure 2.1, but looks more like in Figure 2.2. See Section 2.4 for a rough explanation of this behavior. As we see, the validation error can still go further down after it has begun to increase — plus in a realistic setting we do never know the exact generalization error but estimate it by the validation set error instead. There is no obvious rule for deciding when the minimum of the generalization error is obtained. Real validation error curves almost always have more than one local minimum. The above curve exhibits as many as 16 local minima before severe overfitting begins at about epoch 400. Of these local minima, 4 are the global minimum up to where they occur. The optimal stopping point in this example would be epoch 205. Note that stopping in epoch 400 compared to stopping shortly after the first “deep” local minimum at epoch 45 trades an about sevenfold increase of learning time for an improvement of validation set error by 1.1% (by finding the minimum at epoch 205). If representative data is used, the validation error is an unbiased estimate of the actual network performance; so we expect a 1.1% decrease of the generalization error in this case. Nevertheless, overfitting might sometimes go undetected because the validation set is finite and thus not perfectly representative of the problem.

Unfortunately, the above or any other validation error curve is not *typical* in the sense that all curves share the same qualitative behavior. Other curves might never reach a better minimum than the first, or than, say, the third; the mountains and valleys in the curve can be of very different width, height, and shape. The only thing all curves seem to have in common is that the differences between the first and the following local minima are not huge.

As we see, choosing a stopping criterion predominantly involves a tradeoff between training time and generalization error. However, some stopping criteria may typically find better tradeoffs than others. This leads to the question of

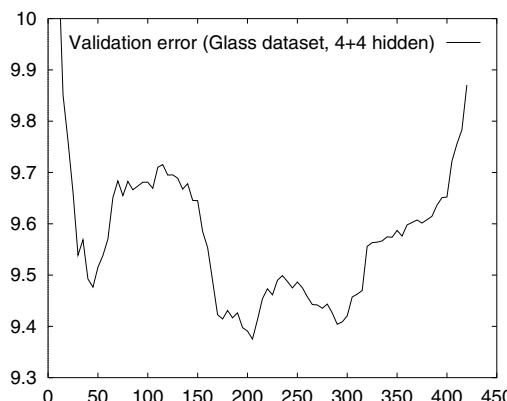


Fig. 2.2. A real validation error curve. Vertical: validation set error; horizontal: time (in training epochs).

which criterion to use with cross validation to decide when to stop training. This is why we need the present trick: To tell us how to *really* do early stopping.

2.2 How to Do Early Stopping Best

What we need is a predicate that tells us when to stop training. We call such a predicate a *stopping criterion*. Among all possible stopping criteria we are searching for those which yield the lowest generalization error and also for those with the best “price-performance ratio”, i.e., that require the least training for a given generalization error or that (on average) result in the lowest generalization error for a certain training time.

2.2.1 Some Classes of Stopping Criteria

There are a number of plausible stopping criteria and this work considers three classes of them. To formally describe the criteria, we need some definitions first. Let E be the objective function (error function) of the training algorithm, for example the squared error. Then $E_{tr}(t)$, the training set error (for short: training error), is the average error per example over the training set, measured after epoch t . $E_{va}(t)$, the validation error, is the corresponding error on the validation set and is used by the stopping criterion. $E_{te}(t)$, the test error, is the corresponding error on the test set; it is not known to the training algorithm but estimates the generalization error and thus benchmarks the quality of the network resulting from training. In real life, the generalization error is usually unknown and only the validation error can be used to estimate it.

The value $E_{opt}(t)$ is defined to be the lowest validation set error obtained in epochs up to t :

$$E_{opt}(t) := \min_{t' \leq t} E_{va}(t')$$

Now we define the *generalization* at epoch t to be the relative increase of the validation error over the minimum-so-far (in percent):

$$GL(t) = 100 \cdot \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right)$$

High generalization loss is one obvious candidate reason to stop training, because it directly indicates overfitting. This leads us to the **first class of stopping criteria: stop as soon as the generalization loss exceeds a certain threshold**. We define the class GL_α as

$$GL_\alpha : \text{stop after first epoch } t \text{ with } GL(t) > \alpha$$

However, we might want to suppress stopping if the training is still progressing very rapidly. The reasoning behind this approach is that when the training error still decreases quickly, generalization losses have higher chance to be “repaired”; we assume that often overfitting does not begin until the error decreases only

slowly. To formalize this notion we define a *training strip of length k* to be a sequence of k epochs numbered $n+1 \dots n+k$ where n is divisible by k . The training *progress* (in per thousand) measured after such a training strip is then

$$P_k(t) := 1000 \cdot \left(\frac{\sum_{t'=t-k+1}^t E_{tr}(t')}{k \cdot \min_{t'=t-k+1}^t E_{tr}(t')} - 1 \right)$$

that is, “how much was the average training error during the strip larger than the minimum training error during the strip?” Note that this progress measure is high for unstable phases of training, where the training set error goes up instead of down. This is intended, because many training algorithms sometimes produce such “jitter” by taking inappropriately large steps in weight space. The progress measure is, however, guaranteed to approach zero in the long run unless the training is globally unstable (e.g. oscillating).

Now we can define the **second class of stopping criteria: use the quotient of generalization loss and progress.**

$$PQ_\alpha : \text{stop after first end-of-strip epoch } t \text{ with } \frac{GL(t)}{P_k(t)} > \alpha$$

In the following we will always assume strips of length 5 and measure the validation error only at the end of each strip.

A completely different kind of stopping criterion relies only on the sign of the changes in the generalization error. We define the **third class of stopping criteria: stop when the generalization error increased in s successive strips.**

UP_s : stop after epoch t iff UP_{s-1} stops after epoch $t-k$ and

$$E_{va}(t) > E_{va}(t-k)$$

UP_1 : stop after first end-of-strip epoch t with $E_{va}(t) > E_{va}(t-k)$

The idea behind this definition is that when the validation error has increased not only once but during s consecutive strips, we assume that such increases indicate the beginning of final overfitting, independent of how large the increases actually are. The UP criteria have the advantage of measuring change locally so that they can be used in the context of pruning algorithms, where errors must be allowed to remain much higher than previous minima over long training periods.

None of these criteria alone can guarantee termination. We thus complement them by the rule that training is stopped when the progress drops below 0.1 or after at most 3000 epochs.

All stopping criteria are used in the same way: They decide to stop at some time t during training and the result of the training is then the set of weights that exhibited the lowest validation error $E_{opt}(t)$. Note that in order to implement this scheme, only one duplicate weight set is needed.

2.2.2 The Trick: Criterion Selection Rules

These three classes of stopping criteria GL , UP , and PQ were evaluated on a variety of learning problems as described in Section 2.3 below. The results

indicate that “slower” criteria, which stop later than others, on the average lead to improved generalization compared to “faster” ones. However, the training time that has to be expended for such improvements is rather large on average and also varies dramatically when slow criteria are used. The systematic differences between the criteria *classes* are only small.

For training setups similar to the one used in this work, the following rules can be used for selecting a stopping criterion:

1. Use fast stopping criteria unless small improvements of network performance (e.g. 4%) are worth large increases of training time (e.g. factor 4).
2. To maximize the probability of finding a “good” solution (as opposed to maximizing the average quality of solutions), use a *GL* criterion.
3. To maximize the average quality of solutions, use a *PQ* criterion if the network overfits only very little or an *UP* criterion otherwise.

2.3 Where and How Well Does This Trick Work?

As no mathematical analysis of the properties of stopping criteria is possible today (see Section 2.4 for the state of the art), we resort to an experimental evaluation.

We want to find out which criteria will achieve how much generalization using how much training time on which kinds of problems. To achieve broad coverage, we use 12 different network topologies, 12 different learning tasks, and 14 different stopping criteria. To keep the experiment feasible, only one training algorithm is used.

2.3.1 Concrete Questions

To derive and evaluate the stopping criteria selection rules presented above we need to answer the following questions:

1. *Training time*: How long will training take with each criterion, i.e., how *fast* or *slow* are they?
2. *Efficiency*: How much of this training time will be redundant, i.e., will occur after the to-be-chosen validation error minimum has been seen?
3. *Effectiveness*: How good will the resulting network performance be?
4. *Robustness*: How sensitive are the above qualities of a criterion to changes of the learning problem, network topology, or initial conditions?
5. *Tradeoffs*: Which criteria provide the best time-performance tradeoff?
6. *Quantification*: How can the tradeoff be quantified?

The answers will directly lead to the rules already presented above in Section 2.2.2. To find the answers to the questions we record for a large number of runs when each criterion would stop and what the associated network performance would be.

2.3.2 Experimental Setup

Approach. To measure network performance, we partition each dataset into two disjoint parts: *Training data* and *test data*. The training data is further subdivided into a *training set* of examples used to adjust the network weights and a *validation set* of examples used to estimate network performance during training as required by the stopping criteria. The validation set is never used for weight adjustment. This decision was made in order to obtain pure stopping criteria results. In contrast, in a real application after a reasonable stopping time has been computed, one would include the validation set examples in the training set and retrain from scratch.

Stopping Criteria. The stopping criteria examined were $GL_1, GL_2, GL_3, GL_5, PQ_{0.5}, PQ_{0.75}, PQ_1, PQ_2, PQ_3, UP_2, UP_3, UP_4, UP_6$, and UP_8 . All criteria where evaluated simultaneously, i.e., each single training run returned one result for each of the criteria. This approach reduces the variance of the estimation.

Learning Tasks. Twelve different problems were used, all from the PROBEN1 NN benchmark set [19]. All problems are real datasets from realistic application domains; they form a sample of a broad class of domains, but none of them exhibits extreme nonlinearity. The problems have between 8 and 120 inputs, between 1 and 19 outputs, and between 214 and 7200 examples. All inputs and outputs are normalized to range 0...1. Nine of the problems are classification tasks using 1-of-n output encoding (*cancer*, *card*, *diabetes*, *gene*, *glass*, *heart*, *horse*, *soybean*, and *thyroid*), three are approximation tasks (*building*, *flare*, and *hearta*).

Datasets and Network Architectures. The examples of each problem were partitioned into training (50%), validation (25%), and test set (25% of examples) in three different random ways, resulting in 36 datasets. Each of these datasets was trained with 12 different feedforward network topologies: one hidden layer networks with 2, 4, 8, 16, 24, or 32 hidden nodes and two hidden layer networks with 2+2, 4+2, 4+4, 8+4, 8+8, or 16+8 hidden nodes in the first+second hidden layer, respectively; all these networks were fully connected including all possible shortcut connections. For each of the network topologies and each dataset, two runs were made with linear output units and one with sigmoidal output units using the activation function $f(x) = x/(1 + |x|)$.

Training Algorithm. All runs were done using the RPROP training algorithm [21] using the squared error function and the parameters $\eta^+ = 1.1$, $\eta^- = 0.5$, $\Delta_0 \in 0.05 \dots 0.2$ randomly per weight, $\Delta_{max} = 50$, $\Delta_{min} = 0$, initial weights $-0.5 \dots 0.5$ randomly. RPROP is a fast backpropagation variant that is about as fast as quickprop [6] but more stable without adjustment of the parameters. RPROP requires epoch learning, i.e., the weights are updated only once per epoch. Therefore, the algorithm is fast without parameter tuning for small training sets but not recommendable for large training sets. Lack of parameter tuning helps to avoid the common methodological error of tuning parameters using the test error.

2.3.3 Experiment Results

Altogether, 1296 training runs were made for the comparison, giving 18144 stopping criteria performance records for the 14 criteria. 270 of these records (or 1.5%) from 125 different runs reached the 3000 epoch limit instead of using the stopping criterion itself.

The results for each stopping criterion averaged over all 1296 runs are shown in Table 2.1. Figure 2.3 describes the variance embedded in the means given in the table. I will now explain and then interpret the entries in both, table and figure. Note that the discussion is biased by the particular collection of criteria chosen for the study.

Definitions. For each run, we define $E_{va}(C)$ as the minimum validation error found until criterion C indicates to stop; it is the error after epoch number $t_m(C)$ (read: “time of minimum”). $E_{te}(C)$ is the corresponding test error and characterizes network performance. Stopping occurs after epoch $t_s(C)$ (read: “time of stop”). A *best* criterion \hat{C} of a particular run is one with minimum t_s of all those (among the examined) with minimum E_{va} , i.e., a criterion that found the best validation error fastest. There may be several best, because multiple criteria may stop at the same epoch. Note that there is no single criterion \hat{C} because \hat{C} changes from run to run. C is called *good* in a particular run if $E_{va}(C) = E_{va}(\hat{C})$, i.e., if it is among those that found the lowest validation set error, no matter how fast or slow.

2.3.4 Discussion: Answers to the Questions

We now discuss the questions raised in Section 2.3.1.

Table 2.1. Behavior of stopping criteria. S_{GL_2} is normalized training time, B_{GL_2} is normalized test error (both relative to GL_2). r is the training time redundancy, P_g is the probability of finding a good solution. For further description please refer to the text.

C	training time		efficiency and effectiveness			
	$S_{\hat{c}}(C)$	$S_{GL_2}(C)$	$r(C)$	$B_{\hat{c}}(C)$	$B_{GL_2}(C)$	$P_g(C)$
UP_2	0.792	0.766	0.277	1.055	1.024	0.587
GL_1	0.956	0.823	0.308	1.044	1.010	*0.680
UP_3	1.010	1.264	0.419	*1.026	1.003	0.631
GL_2	1.237	1.000	0.514	1.034	1.000	*0.723
UP_4	1.243	1.566	0.599	*1.020	0.997	0.666
$PQ_{0.5}$	1.253	1.334	0.663	1.027	1.002	0.658
$PQ_{0.75}$	1.466	1.614	0.863	1.021	0.998	0.682
GL_3	1.550	1.450	*0.712	1.025	0.994	*0.748
PQ_1	1.635	1.796	1.038	1.018	0.994	0.704
UP_6	1.786	2.381	1.125	*1.012	0.990	0.737
GL_5	2.014	2.013	1.162	1.021	0.991	*0.772
PQ_2	2.184	2.510	1.636	1.012	0.990	0.768
UP_8	2.485	3.259	1.823	*1.010	0.988	0.759
PQ_3	2.614	3.095	2.140	1.009	0.988	0.800

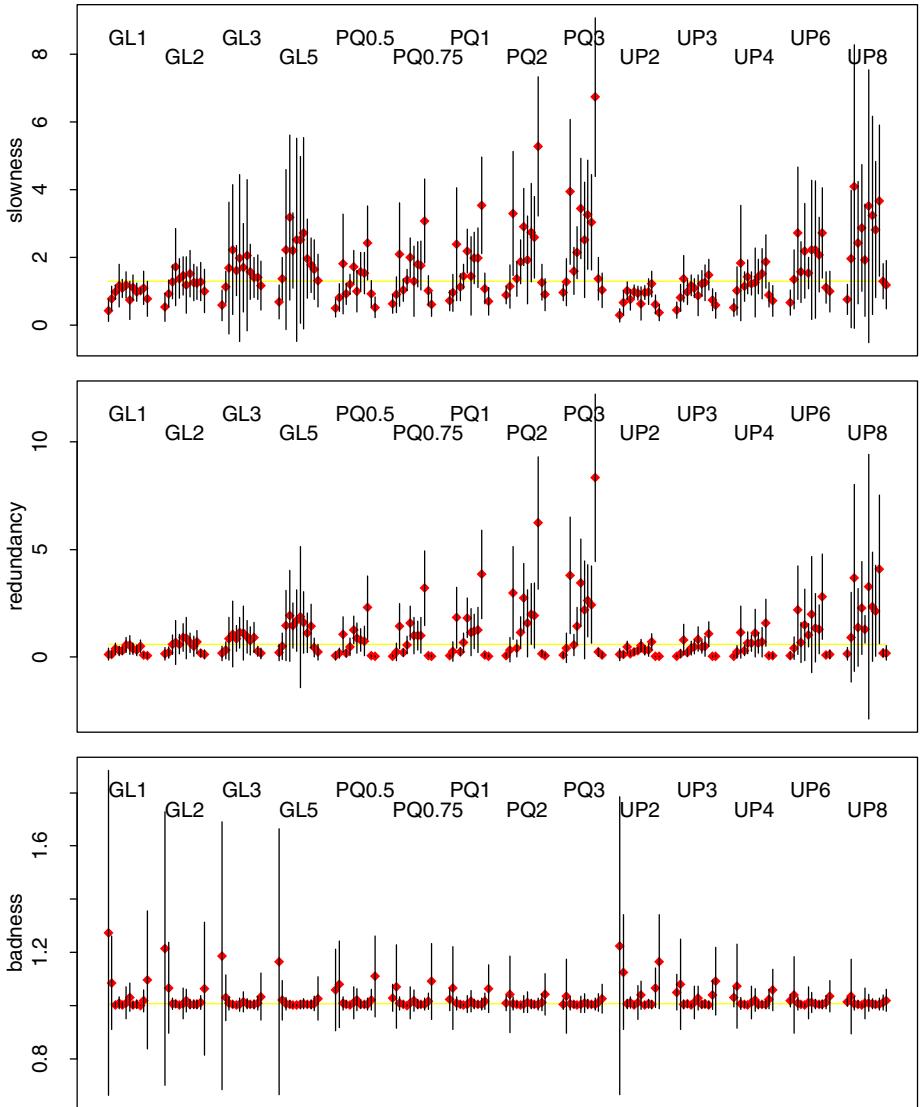


Fig. 2.3. Variance of slowness $S_{\hat{C}}(C)$ (top), redundancy $r(C)$ (middle), and badness $B_{\hat{C}}(C)$ (bottom) for each pair of learning problem and stopping criterion. In each of the 168 columns, the dot represents the mean computed from 108 runs; learning problem and stopping criterion are fixed, while three other parameters are varied (12 topologies \times 3 runs \times 3 dataset variants). The length of the line is twice the standard deviation within these 108 values. Within each block of dot-line plots, the plots represent (in order) the problems building, cancer, card, diabetes, flare, gene, glass, heart, hearta, horse, soybean, thyroid. The horizontal line marks the median of the means. Note: When comparing the criteria groups, remember that overall the PQ criteria chosen are slower than the others. It is unfair to compare, for example, $PQ_{0.5}$ to GL_1 and UP_2 .

1. *Training time:* The *slowness* of a criterion C in a run, relative to another criterion x is $S_x(C) := t_s(C)/t_s(x)$, i.e., the relative total training time. As we see, the times relative to a fixed criterion as shown in column $S_{GL_2}(C)$ vary by more than factor 4. Therefore, the decision for a particular stopping criterion influences training times dramatically, even if one considers only the range of criteria used here. In contrast, even the slowest criteria train only about 2.5 times as long as the fastest criterion of each run that finds the same result, as indicated in column $S_{\hat{C}}(C)$. This shows that the training times are not completely unreasonable even for the slower criteria, but do indeed pay off to some degree.

2. *Efficiency:* The *redundancy* of a criterion can be defined as $r(C) := (t_s(C)/t_m(C)) - 1$. It characterizes how long the training continues after the final solution has been seen. $r(C) = 0$ would be perfect, $r(C) = 1$ means that the criterion trains twice as long as necessary. Low values indicate efficient criteria. As we see, the slower a criterion is, the less efficient it tends to get. Even the fastest criteria “waste” about one fifth of their overall training time. The slower criteria train twice as long as necessary to find the same solution.

3. *Effectiveness:* We define the *badness* of a criterion C in a run relative to another criterion x as $B_x(C) := E_{te}(C)/E_{te}(x)$, i.e., its relative error on the test set. $P_g(C)$ is the fraction of the 1296 runs in which C was a good criterion. This is an estimate of the probability that C is good in a run. As we see from the P_g column, even the fastest criteria are fairly effective. They reach a result as good as the best (of the same run) in about 60% of the cases. On the other hand, even the slowest criteria are not at all infallible; they achieve about 80%. However, P_g says nothing about how *far* from the optimum the non-good runs are. Columns $B_{\hat{C}}(C)$ and $B_{GL_2}(C)$ indicate that these differences are usually rather small: column $B_{GL_2}(C)$ shows that even the criteria with the lowest error achieve only about 1% lower error on the average than the relatively fast criterion GL_2 . In column $B_{\hat{C}}(C)$ we see that several only modestly slow criteria have just about 2% higher error on the average than the best criteria of the same run. For obtaining the lowest possible generalization error, independent of training time, it appears that one has to use an extreme criterion such as GL_{50} or even use a conjunction of all three criteria classes with high parameter values.

4. *Robustness:* We call a criterion *robust* to the degree that its performance is independent of the learning problem and the learning environment (network topology, initial conditions etc.). Optimal robustness would mean that in Figure 2.3 all dots within a block are at the same height (problem independence) and all lines have length zero (environment independence). Note that slowness and badness are measured relative to the best criterion of the same program run. We observe the following:

- With respect to slowness and redundancy, slower criteria are much less robust than faster ones. In particular the PQ criteria are quite sensitive to the learning problem, with the card and horse problems being worst in this experimental setting.
- With respect to badness, the picture is completely different: slower criteria tend to be slightly *more* robust than faster ones. PQ criteria are a little

more robust than the others while GL criteria are significantly less robust. All criteria are more or less unstable for the building, cancer, and thyroid problems. In particular, all GL criteria have huge problems with the building problem, whose dataset 1 is the only one that is partitioned non-randomly; it uses chronological order of examples, see [19]. The slower variants of the other criteria types are nicely robust in this case.

- Similar statements apply when one analyzes the influence of only large or only small network topologies separately (not shown in any figure or table). One notable exception was the fact that for networks with very few hidden nodes the PQ criteria are more cost-effective than both the GL and the UP criteria for minimizing $B_{\hat{C}}(C)$. The explanation may be that such small networks do not overfit severely; in this case it is advantageous to take training progress into account as an additional factor to determine when to stop training.

Overall, fast criteria improve the predictability of the training time, while slow ones improve the predictability of the solution quality.

5. *Best tradeoffs*: Despite the common overall trend, some criteria may be more cost-effective than others, i.e., provide better tradeoffs between training time and resulting network performance. Column $B_{\hat{C}}$ of the table suggests that the best tradeoffs between test error and training time are (in order of increasing willingness to spend lots of training time) UP_3 , UP_4 , and UP_6 , if one wants to minimize the expected network performance from a single run. These criteria are also robust. If on the other hand one wants to make several runs and pick the network that seems to be best (based on its validation error), P_g is the relevant metric and the GL criteria are preferable. The best tradeoffs are marked with a star in the table. Figure 2.4 illustrates these results. The upper curve corresponds to column $B_{\hat{C}}$ of the table (plotted against column $S_{\hat{C}}$); local minima indicate

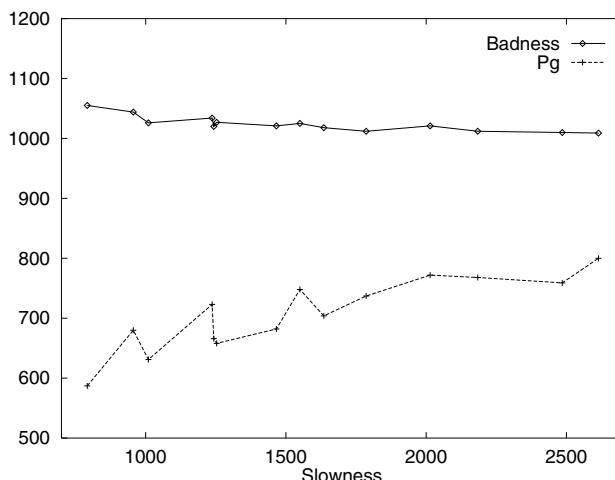


Fig. 2.4. Badness $B_{\hat{C}}(C)$ and P_g against slowness $S_{\hat{C}}(C)$ of criteria

criteria with the best tradeoffs. The lower curve corresponds to column P_g ; local maxima indicate the criteria with the best tradeoffs. All measurements are scaled by 1000.

6. *Quantification:* From columns $S_{GL_2}(C)$ and $B_{GL_2}(C)$ we can quantify the tradeoff involved in the selection of a stopping criterion as follows: In the range of criteria examined we can roughly trade a 4% decrease in test error (from 1.024 to 0.988) for an about fourfold increase in training time (from 0.766 to 3.095). Within this range, some criteria are somewhat better than others, but there is no panacea.

2.3.5 Generalization of These Results

It is difficult to say whether or how these results apply to different contexts than those of the above evaluation. Speculating though, I would expect that the behavior of the stopping criteria

- is similar for other learning rules, unless they frequently make rather extreme steps in parameter space,
- is similar for other error functions, unless they are discontinuous,
- is similar for other learning tasks, as long as they are in the same ballpark with respect to their nonlinearity, number of inputs and outputs, and amount of available training data.

Note however, that at least with respect to the learning task deviations do occur (see Figure 2.3). More research is needed in order to describe which properties of the learning tasks lead to which differences in stopping criteria behavior — or more generally: in order to understand how which features of tasks influence learning methods.

2.4 Why This Works

Detailed theoretical analyses of the error curves cannot yet be done for the most interesting cases such as sigmoidal multi-layer perceptrons trained on a modest number of examples; today they are possible for restricted scenarios only [1, 2, 3, 24] and do usually not aim at finding the optimal stopping criterion in a way comparable to the present work. However, a simplification of the analysis performed by Wang et al. [24] or the alternative view induced by the bias/variance decomposition of the error as described by Geman et al. [10] can give some insights why early stopping behaves as it does.

At the beginning of training (phase I), the error is dominated by what Wang et al. call the *approximation error* — the network has hardly learned anything and is still very biased. During training this part of the error is further and further reduced. At the same time, however, another component of the error increases: the *complexity error* that is induced by the increasing variance of the network model as the possible magnitude and diversity of the weights grows. If we train long enough, the error will be dominated by the complexity error

(phase III). Therefore, there is a phase during training, when the approximation and complexity (or: bias and variance) components of the error compete but none of them dominates (phase II). See Amari et al. [1, 2] for yet another view of the training process, using a geometrical interpretation. The task of early stopping as described in the present work is to detect when phase II ends and the dominance of the variance part begins.

Published theoretical results on early stopping appear to provide some nice techniques for practical application: Wang et al. [24] offer a method for computing the stopping point based on complexity considerations — without using a separate validation set at all. This could save precious training examples. Amari et al. [1, 2] compute the optimal split proportion of training data into training and validation set.

On the other hand, unfortunately, the practical applicability of these theoretical analyses is severely restricted. Wang et al.’s analysis applies to networks where only output weights are being trained; no hidden layer training is captured. It is unclear to what degree the results apply to the multi-layer networks considered here. Amari et al.’s analysis applies to the asymptotic case of very many training examples. The analysis does not give advice on stopping criteria; it shows that early stopping is not useful when very many examples are available but does not cover the much more frequent case when training examples are scarce.

There are several other theoretical works on early stopping, but none of them answers our practical questions. Thus, given these theoretic results, one is still left with making a good stopping decision for practical cases of multilayer networks with only few training examples and faced with a complicated evolution of the validation set error as shown in Figure 2.2. This is why the present empirical investigation was necessary.

The jagged form of the validation error curve during phase II arises because neither bias nor variance change monotonically, let alone smoothly. The bias error component may change abruptly because training algorithms never perform gradient descent, but take finite steps in parameter space that sometimes have severe results. The observed variance error component may change abruptly because, first, the validation set error is only an estimate of the actual generalization error and, second, the effect of a parameter change may be very different in different parts of parameter space.

Quantitatively, the different error minima that occur during phase II are quite close together in terms of size, but may be rather far apart in terms of training epoch. The exact validation error behavior seems rather unpredictable when only a short left section of the error curve is given. The behavior is also very different for different training situations.

For these reasons no class of stopping criteria has any big advantage over another (on average, for the mix of situations considered here), but scaling the same criterion to be slower always tends to gain a little generalization.

References

- [1] Amari, S., Murata, N., Müller, K.-R., Finke, M., Yang, H.: Statistical theory of overtraining - is cross-validation effective. In: [23], pp. 176–182 (1996)
- [2] Amari, S., Murata, N., Müller, K.-R., Finke, M., Yang, H.: Asymptotic statistical theory of overtraining and cross-validation. *IEEE Trans. on Neural Networks* 8(5), 985–996 (1997)
- [3] Baldi, P., Chauvin, Y.: Temporal evolution of generalization during learning in linear networks. *Neural Computation* 3, 589–603 (1991)
- [4] Cowan, J.D., Tesauro, G., Alspector, J. (eds.): *Advances in Neural Information Processing Systems 6*. Morgan Kaufman Publishers Inc., San Mateo (1994)
- [5] Le Cun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In: [22], pp. 598–605 (1990)
- [6] Fahlman, S.E.: An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (September 1988)
- [7] Fahlman, S.E., Lebiere, C.: The Cascade-Correlation learning architecture. In: [22], pp. 524–532 (1990)
- [8] Fiesler, E.: Comparative bibliography of ontogenetic neural networks (1994) (submitted for publication)
- [9] Finnoff, W., Hergert, F., Zimmermann, H.G.: Improving model selection by non-convergent methods. *Neural Networks* 6, 771–783 (1993)
- [10] Geman, S., Bienenstock, E., Doursat, R.: Neural networks and the bias/variance dilemma. *Neural Computation* 4, 1–58 (1992)
- [11] Hanson, S.J., Cowan, J.D., Giles, C.L. (eds.): *Advances in Neural Information Processing Systems 5*. Morgan Kaufman Publishers Inc., San Mateo (1993)
- [12] Hassibi, B., Stork, D.G.: Second order derivatives for network pruning: Optimal brain surgeon. In: [11], pp. 164–171 (1993)
- [13] Krogh, A., Hertz, J.A.: A simple weight decay can improve generalization. In: [16], pp. 950–957 (1992)
- [14] Levin, A.U., Leen, T.K., Moody, J.E.: Fast pruning using principal components. In: [4] (1994)
- [15] Lippmann, R.P., Moody, J.E., Touretzky, D.S. (eds.): *Advances in Neural Information Processing Systems 3*. Morgan Kaufman Publishers Inc., San Mateo (1991)
- [16] Moody, J.E., Hanson, S.J., Lippmann, R.P. (eds.): *Advances in Neural Information Processing Systems 4*. Morgan Kaufman Publishers Inc., San Mateo (1992)
- [17] Morgan, N., Bourlard, H.: Generalization and parameter estimation in feedforward nets: Some experiments. In: [22], pp. 630–637 (1990)
- [18] Nowlan, S.J., Hinton, G.E.: Simplifying neural networks by soft weight-sharing. *Neural Computation* 4(4), 473–493 (1992)
- [19] Prechelt, L.: PROBEN1 — A set of benchmarks and benchmarking rules for neural network training algorithms. Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, Germany, Anonymous,
`ftp://pub/papers/techreports/1994/1994-21.ps.gz` on,
`ftp.ira.uka.de` (September 1994)
- [20] Reed, R.: Pruning algorithms — a survey. *IEEE Transactions on Neural Networks* 4(5), 740–746 (1993)
- [21] Riedmiller, M., Braun, H.: A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In: Proc. of the IEEE Intl. Conf. on Neural Networks, San Francisco, CA, pp. 586–591 (April 1993)

- [22] Touretzky, D.S. (ed.): Advances in Neural Information Processing Systems 2. Morgan Kaufman Publishers Inc., San Mateo (1990)
- [23] Touretzky, D.S., Mozer, M.C., Hasselmo, M.E. (eds.): Advances in Neural Information Processing Systems 8. MIT Press, Cambridge (1996)
- [24] Wang, C., Venkatesh, S.S., Judd, J.S.: Optimal stopping and effective machine complexity in learning. In: [4] (1994)
- [25] Weigend, A.S., Rumelhart, D.E., Huberman, B.A.: Generalization by weight-elimination with application to forecasting. In: [15], pp. 875–882 (1991)

A Simple Trick for Estimating the Weight Decay Parameter^{*}

Thorsteinn S. Rögnvaldsson

Centre for Computer Architecture (CCA), Halmstad University, P.O. Box 823,
 S-301 18 Halmstad, Sweden
 denni@cca.hh.se
<http://www.hh.se/staff/denni/>

Abstract. We present a simple trick to get an approximate estimate of the weight decay parameter λ . The method combines early stopping and weight decay, into the estimate

$$\hat{\lambda} = \|\nabla E(\mathbf{W}_{es})\| / \|2\mathbf{W}_{es}\|,$$

where \mathbf{W}_{es} is the set of weights at the early stopping point, and $E(\mathbf{W})$ is the training data fit error.

The estimate is demonstrated and compared to the standard cross-validation procedure for λ selection on one synthetic and four real life data sets. The result is that $\hat{\lambda}$ is as good an estimator for the optimal weight decay parameter value as the standard search estimate, but orders of magnitude quicker to compute.

The results also show that weight decay can produce solutions that are significantly superior to committees of networks trained with early stopping.

3.1 Introduction

A regression problem which does not put constraints on the model used is ill-posed [21], because there are infinitely many functions that can fit a finite set of training data perfectly. Furthermore, real life data sets tend to have noisy inputs and/or outputs, which is why models that fit the data perfectly tend to be poor in terms of out-of-sample performance. Since the modeler's task is to find a model for the underlying function while not overfitting to the noise, models have to be based on criteria which include other qualities besides their fit to the training data.

In the neural network community the two most common methods to avoid overfitting are *early stopping* and *weight decay* [17]. Early stopping has the advantage of being quick, since it shortens the training time, but the disadvantage

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

of being poorly defined and not making full use of the available data. Weight decay, on the other hand, has the advantage of being well defined, but the disadvantage of being quite time consuming. This is because much time is spent with selecting a suitable value for the weight decay parameter (λ), by searching over several values of λ and estimating the out-of-sample performance using e.g. cross validation [25].

In this paper, we present a very simple method for estimating the weight decay parameter, for the standard weight decay case. This method combines early stopping with weight decay, thus merging the quickness of early stopping with the more well defined weight decay method, providing a weight decay parameter which is essentially as good as the standard search method estimate when tested empirically.

We also demonstrate in this paper that the arduous process of selecting λ can be rewarding compared to simpler methods, like e.g. combining networks into committees [16].

The paper is organized as follows: In section 2 we present the background of how and why weight decay or early stopping should be used. In section 3 we review the standard method for selecting λ and also introduce our new estimate. In section 4 we give empirical evidence on how well the method works, and in section 5 we summarize our conclusions.

3.2 Ill-Posed Problems, Regularization, and Such Things...

3.2.1 Ill-Posed Problems

In what follows, we denote the input data by $\mathbf{x}(n)$, the target data by $y(n)$, and the model (neural network) output by $f(\mathbf{W}, \mathbf{x}(n))$, where \mathbf{W} denotes the parameters (weights) for the model. We assume a target data generating process of the form

$$y(n) = \phi[\mathbf{x}(n)] + \varepsilon(n) \quad (3.1)$$

where ϕ is the *underlying function* and $\varepsilon(n)$ are sampled from a stationary uncorrelated (IID) zero mean noise process with variance σ^2 . We select models f from a model family F , e.g. multilayer perceptrons, to learn an approximation to the underlying function ϕ , based on the training data. That is, we are searching for

$$f^* \equiv f(\mathbf{W}^*) \in F \text{ such that } E(f^*, \phi) \leq E(f, \phi) \quad \forall f \in F, \quad (3.2)$$

where $E(f, \phi)$ is a measure of the “distance” between the model f and the true model ϕ . Since we only have access to the target values y , and not the underlying function ϕ , $E(f, \phi)$ is often taken to be the mean square error

$$E(f, \phi) \rightarrow E(f, y) = E(\mathbf{W}) = \frac{1}{2N} \sum_{n=1}^N [y(n) - f(\mathbf{W}, \mathbf{x}(n))]^2. \quad (3.3)$$

Unfortunately, minimizing (3.3) is, more often than not, an ill-posed problem. That is, it does not meet the following three requirements [21]:

- The model (e.g. neural network) can learn the function training data, i.e. there *exists* a solution $f^* \in F$.
- The solution is *unique*.
- The solution is *stable* under small variations in the training data set. For instance, training with two slightly different training data sets sampled from the same process must result in similar solutions (similar when evaluated on e.g. test data).

The first and second of these requirements are often not considered serious problems. It is always possible to find a multilayer perceptron that learns the training data perfectly by using many internal units, since any continuous function can be constructed with a single hidden layer network with sigmoid units (see e.g. [6]), and we may be happy with any solution and ignore questions on uniqueness. However, a network that has learned the training data perfectly will be very sensitive to changes in the training data. Fulfilling the first requirement is thus usually in conflict with fulfilling the third requirement, which is a really important requirement. A solution which changes significantly with slightly different training sets will have very poor generalization properties.

3.2.2 Regularization

It is common to introduce so-called regularizers¹ in order to make the learning task well posed (or at least less ill-posed). That is, instead of only minimizing an error of fit measure like (3.3) we augment it with a regularization term $\lambda R(\mathbf{W})$ which expresses e.g. our prior beliefs about the solution.

The error functional then takes the form

$$E(\mathbf{W}) = \frac{1}{2N} \sum_{n=1}^N [y(n) - f(\mathbf{W}, \mathbf{x}(n))]^2 + \lambda R(\mathbf{W}) = E_0(\mathbf{W}) + \lambda R(\mathbf{W}), \quad (3.4)$$

where λ is the *regularization parameter* which weighs the importance of $R(\mathbf{W})$ relative to the error of fit $E_0(\mathbf{W})$.

The effect of the regularization term is to shrink the model family F , or make some models more likely than others. As a consequence, solutions become more stable to small perturbations in the training data.

The term “regularization” encompasses all techniques which make use of penalty terms added to the error measure to avoid overfitting. This includes e.g. weight decay [17], weight elimination [26], soft weight sharing [15], Laplacian weight decay [12] [27], and smoothness regularization [2] [9] [14]. Certain forms of “hints” [1] can also be called regularization.

3.2.3 Bias and Variance

The benefit of regularization is often described in the context of *model bias* and *model variance*. This originates from the separation of the expected generalization error $\langle E_{gen} \rangle$ into three terms [8]

¹ Called “stabilizers” by Tikhonov [21].

$$\begin{aligned}
\langle E_{gen} \rangle &= \left\langle \int [y(\mathbf{x}) - f(\mathbf{x})]^2 p(\mathbf{x}) d\mathbf{x} \right\rangle \\
&= \int [\phi(\mathbf{x}) - \langle f(\mathbf{x}) \rangle]^2 p(\mathbf{x}) d\mathbf{x} + \left\langle \int [f(\mathbf{x}) - \langle f(\mathbf{x}) \rangle]^2 p(\mathbf{x}) d\mathbf{x} \right\rangle + \\
&\quad \left\langle \int [y(\mathbf{x}) - \phi(\mathbf{x})]^2 p(\mathbf{x}) d\mathbf{x} \right\rangle \\
&= \text{Bias}^2 + \text{Variance} + \sigma^2,
\end{aligned} \tag{3.5}$$

where $\langle \rangle$ denotes taking the expectation over an ensemble of training sets. Here $p(\mathbf{x})$ denotes the input data probability density.

A high sensitivity to training data noise corresponds to a large model variance. A large bias term means either that $\phi \notin F$, or that ϕ is downweighted in favour of other models in F . We thus have a trade-off between model bias and model variance, which corresponds to the trade-off between the first and third requirements on well-posed problems.

Model bias is weighed versus model variance by selecting both a parametric form for $R(\mathbf{W})$ and an optimal² value for the regularization parameter λ .

Many neural network practitioners ignore the first part and choose weight decay by default, which corresponds to a Gaussian parametric form for the prior on \mathbf{W} . Weight decay is, however, not always the best choice (in fact, it is most certainly not the best choice for all problems). Weight decay does not for instance consider the function the network is producing, it only puts a constraint on the parameters. Another, perhaps more correct, choice would be to constrain the higher order derivatives of the network function (which is commonplace in statistics) like in e.g. [14].

3.2.4 Bayesian Framework

From a Bayesian and maximum likelihood perspective, *prior* information about the model (f) is weighed against the *likelihood* of the training data (D) through Bayes theorem (see [4] for a discussion on this). Denote the probability for observing data set D by $p(D)$, the prior distribution of models f by $p(f)$, and the likelihood for observing the data D , if f is the correct model, by $p(D|f)$. We then have for the posterior probability $p(f|D)$ for the model f given the observed data D

$$\begin{aligned}
p(f|D) &= \frac{p(D|f)p(f)}{p(D)} \Rightarrow \\
-\ln p(f|D) &= -\log p(D|f) - \ln p(f) + \ln p(D) \Rightarrow \\
-\ln p(f|D) &= \sum_{n=1}^N [y(n) - f(\mathbf{W}, \mathbf{x}(n))]^2 - \ln p(f) + \text{constant},
\end{aligned} \tag{3.6}$$

where Gaussian noise ε is assumed in the last step. If we identify $2N\lambda R(\mathbf{W})$ with the negative logarithm of the model prior, $-\ln p(f)$, then maximizing $p(f|D)$ is equivalent to minimizing expression (3.4).

² Optimality is usually measured via cross-validation or some similar method.

From this perspective, choosing $R(\mathbf{W})$ is equivalent to choosing a parameterized form for the model prior $p(f)$, and selecting a value for λ corresponds to estimating the parameters for the prior.

3.2.5 Weight Decay

Weight decay [17] is the neural network equivalent to the Ridge Regression [11] method. In this case $R(\mathbf{W}) = \|\mathbf{W}\|^2 = \sum_k w_k^2$ and the error functional is

$$E(\mathbf{W}) = E_0(\mathbf{W}) + \lambda R(\mathbf{W}) = \frac{1}{2N} \sum_{n=1}^N [y(n) - f(\mathbf{W}, \mathbf{x}(n))]^2 + \lambda \|\mathbf{W}\|^2, \quad (3.7)$$

and λ is usually referred to as the *weight decay parameter*. In the Bayesian framework, weight decay means implicitly imposing the model prior

$$p[f(\mathbf{W})] = \sqrt{\frac{\lambda}{2\pi\sigma^2}} \exp\left(-\frac{\lambda\|\mathbf{W}\|^2}{2\sigma^2}\right) \quad (3.8)$$

where σ^2 is the variance of the noise in the data.

Weight decay often improves the generalization properties of neural network models, for reasons outlined above.

3.2.6 Early Stopping

Undoubtedly, the simplest and most widely used method to avoid overfitting is to stop training before the training set has been learned perfectly. This is done by setting aside a fraction of the training data for estimating the out-of-sample performance. This data set is called the validation data set. Training is then stopped when the error on the validation set starts to increase. Early stopping often shortens the training time significantly, but suffers from being ill-defined since there really is no well defined stopping point, and wasteful with data, since a part of the data is set aside.

There is a connection between early stopping and weight decay, if learning starts from small weights, since weight decay applies a potential which forces all weights towards zero. For instance, Sjöberg and Ljung [20] show that, if a constant learning rate η is used, the number of iterations n at which training is stopped is related to the weight decay parameter λ roughly as

$$\lambda \sim \frac{1}{2\eta n}. \quad (3.9)$$

This does not, however, mean that using early stopping is equivalent to using weight decay in practice. Expression (3.9) is based on a constant learning rate, a local expansion around the optimal stopping point, ignoring local minima, and assumes small input noise levels, which may not reflect the situation when overfitting is a serious problem. The choice of learning algorithm can also affect

the early stopping point, and one cannot expect (3.9) to hold exactly in the practical case.

Inspired by this connection between early stopping and weight decay, we use early stopping in the following section to estimate the weight decay parameter λ .

3.3 Estimating λ

From a pure Bayesian point of view, the prior is something we know/assume in advance and do not use the training data to select (see e.g. [5]). There is consequently no such thing as “ λ selection” in the pure Bayesian model selection scheme. This is of course perfectly fine if the prior is correct. However, if we suspect that our choice of prior is less than perfect, then we are better off if we take an “empirical Bayes” approach and use the data to tune the prior, through λ .

Several options for selecting λ have been proposed. Weigend et al. [26] present, for a slightly different weight cost term, a set of heuristic rules for changing λ during the training. Although Weigend et al. demonstrate the use of these heuristics on a couple of time series problems, we cannot get these rules to work consistently to our satisfaction. A more principled approach is to try several values of λ and estimate the out-of-sample error, either by correcting the training error, with some factor or term, or by using cross-validation. The former is done in e.g. [10], [23], and [24] (see also references therein). The latter is done by e.g. [25].

The method of using validation data for estimating the out-of-sample error is robust but slow since it requires training several models. We use cross-validation here because of its reliability.

3.3.1 Search Estimates

Finding the optimal λ requires the use of a search algorithm, which must be robust because the validation error can be very noisy. A simple and straightforward way is to start at some large λ where the validation error is large, due to the large model bias, and step towards lower values until the out-of-sample error becomes large again, due to the large model variance. In our experience, it often makes sense to do the search in $\log \lambda$ (i.e. with equally spaced increments in $\log \lambda$).

The result of such a search is a set of K values $\{\lambda_k\}$ with corresponding average n -fold cross validation errors $\{\log E_{nCV,k}\}$ and standard deviations $\{\sigma_{nCV,k}\}$ for the validation errors. These are defined as

$$\log E_{nCV,k} = \frac{1}{n} \sum_{j=1}^n \log E_{j,k} \quad (3.10)$$

$$\sigma_{nCV,k}^2 = \frac{1}{n-1} \sum_{j=1}^n (\log E_{j,k} - \log E_{nCV,k})^2 \quad (3.11)$$

when $\lambda = \lambda_k$. The number of validation data sets is n and $E_{j,k}$ denotes the validation error when $\lambda = \lambda_k$ and we use validation set j . Taking logarithms is motivated by our observation that the validation error distribution looks approximately log-normal and we use this in our selection of the optimal λ value below.

Once the search is finished, the optimal λ is selected. This is not necessarily trivial since a large range of values may look equally good, or one value may have a small average cross-validation error with a large variation in this error, and another value may have a slightly higher average cross-validation error with a small variation in this error. The simplest approach is to look at a plot of the validation errors versus λ and make a judgement on where the optimal λ is, but this adds an undesired subjectiveness to the choice. Another is to take a weighted average over the different λ values, which is what we use here (see Ripley [19] for a discussion on variants of λ selection methods).

Our estimate for the optimal λ is the value

$$\hat{\lambda}_{opt} = \frac{\sum_{k=1}^K n_k \lambda_k}{\sum_{k=1}^K n_k} \quad (3.12)$$

where n_k is the number of times λ_k corresponds to the minimum validation error when we sample validation errors from K log-normal distributions with means $\log E_{nCV,k}$ and standard deviations $\sigma_{nCV,k}$, assuming that the validation errors are independent. This is illustrated on a hypothetical example in Figure 3.1. The choice (3.12) was done after confirming that it often agrees well with our subjective choice for λ . We refer to this below as a ‘‘Monte Carlo estimate’’ of λ .

3.3.2 Two Early Stopping Estimates

If \mathbf{W}^* is the set of weights when $E(\mathbf{W})$ in eq. (3.4) is minimized, then

$$\nabla E(\mathbf{W}^*) = \nabla E_0(\mathbf{W}^*) + \lambda \nabla R(\mathbf{W}^*) = 0, \quad (3.13)$$

which implies

$$\lambda = \frac{\|\nabla E_0(\mathbf{W}^*)\|}{\|\nabla R(\mathbf{W}^*)\|} \quad (3.14)$$

for the regularization parameter λ . Thus, if we have a reasonable estimate of \mathbf{W}^* , or of $\|\nabla E_0(\mathbf{W}^*)\|$ and $\|\nabla R(\mathbf{W}^*)\|$, then we can use this to estimate λ . An appealingly simple way of estimating \mathbf{W}^* is to use early stopping, because of its connection with weight decay.

Denoting the set of weights at the early stopping point by \mathbf{W}_{es} , we have

$$\hat{\lambda}_1 = \frac{\|\nabla E_0(\mathbf{W}_{es})\|}{\|\nabla R(\mathbf{W}_{es})\|}, \quad (3.15)$$

as a simple estimate for λ . A second possibility is to consider the whole set of linear equations defined by (3.13) and minimize the squared error

$$\begin{aligned} & \|\nabla E_0(\mathbf{W}_{es}) + \lambda \nabla R(\mathbf{W}_{es})\|^2 = \\ & \|\nabla E_0(\mathbf{W}_{es})\|^2 + 2\lambda \nabla E_0(\mathbf{W}_{es}) \cdot \nabla R(\mathbf{W}_{es}) + \lambda^2 \|\nabla R(\mathbf{W}_{es})\|^2 \end{aligned} \quad (3.16)$$

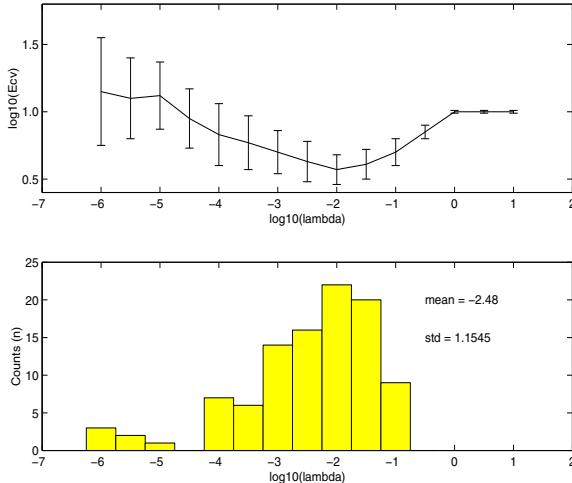


Fig. 3.1. Illustration of the procedure for estimating $\hat{\lambda}_{opt}$ on a hypothetical example. From the search we have a set of K lognormal distributions with means $\log E_{nCV,k}$ and variances $\sigma_{nCV,k}^2$, which is illustrated in the top plate. From these K distributions, we sample K error values and select the λ corresponding to the minimum error value as “winner”. This is repeated several times (100 in the figure but 10,000 times in the experiments in the text) collecting statistics on how often different λ values are winners, and the mean $\log \lambda$ is computed. This is illustrated in the bottom plate, which shows the histogram resulting from sampling 100 times. From this we get $\log \hat{\lambda}_{opt} = -2.48 \pm 1.15$, which gives us $\lambda = 10^{-2.48} = 0.003$ for training the “best” network.

with respect to λ . That is, solving the equation

$$\frac{\partial}{\partial \lambda} \{ \| \nabla E_0(\mathbf{W}_{es}) + \lambda \nabla R(\mathbf{W}_{es}) \|^2 \} = 0 \quad (3.17)$$

which gives

$$\hat{\lambda}_2 = \max \left[0, \frac{-\nabla E_0(\mathbf{W}_{es}) \cdot \nabla R(\mathbf{W}_{es})}{\| \nabla R(\mathbf{W}_{es}) \|^2} \right]. \quad (3.18)$$

The estimate is bound from below since λ must be positive.

The second estimate, $\hat{\lambda}_2$, corresponds to a linear regression without intercept term on the set of points $\{\partial_i E_0(\mathbf{W}_{es}), \partial_i R(\mathbf{W}_{es})\}$, whereas the first estimate, $\hat{\lambda}_1$, is closer to the ratio $\max[\|\partial_i E_0(\mathbf{W}_{es})\|] / \max[\|\partial_i R(\mathbf{W}_{es})\|]$. It follows from the Cauchy-Schwartz inequality that

$$\hat{\lambda}_1 \geq \hat{\lambda}_2. \quad (3.19)$$

For the specific case of weight decay, where $R(\mathbf{W}) = \|\mathbf{W}\|^2$, expressions (3.15) and (3.18) become

$$\hat{\lambda}_1 = \frac{\| \nabla E_0(\mathbf{W}_{es}) \|}{2 \|\mathbf{W}_{es}\|}, \quad (3.20)$$

$$\hat{\lambda}_2 = \max \left[0, \frac{-\nabla E_0(\mathbf{W}_{es}) \cdot \mathbf{W}_{es}}{2\|\mathbf{W}_{es}\|^2} \right]. \quad (3.21)$$

These estimates are sensitive to the particularities of the training and validation data sets used, and possibly also to the training algorithm. One must therefore average them over different validation and training sets. It is, however, still quicker to do this than to do a search since early stopping training often is several orders of magnitude faster to do than a full minimization of (3.7).

One way to view the estimates (3.15) and (3.18) is as the weight decay parameters that correspond to the early stopping point. However, our aim here is not to imitate early stopping with weight decay, but to use early stopping to estimate the weight decay parameter λ . We hope that using weight decay with this λ value will actually result in better out-of-sample performance than what we get from doing early stopping (the whole exercise becomes rather meaningless if this is not the case).

As a sidenote, we imagine that (3.15) and (3.18) could be used also to estimate weight decay parameters in cases when different weight decays are used for weights in different layers. This would then be done by considering these estimates for different groups of weights.

3.4 Experiments

3.4.1 Data Sets

We here demonstrate the performance of our algorithm on a set of five regression problems. For each problem, we vary either the number of inputs, the number of hidden units, or the amount of training data to study the effects of the numbers of parameters relative to the number of training data points. The five problems are:

Synthetic Bilinear Problem. The task is to model a bilinear function of the form

$$\phi(x_1, x_2) = x_1 x_2. \quad (3.22)$$

We use three different sizes of training data sets, $M \in \{20, 40, 100\}$, but a constant validation set size of 10 patterns. The validation patterns are in addition to the M training patterns. The test error, or generalization error, is computed by numerical integration over 201×201 data points on a two-dimensional lattice $(x_1, x_2) \in [-1, 1]^2$. The target values (but not the inputs) are contaminated with three different levels of Gaussian noise with standard deviation $\sigma \in \{0.1, 0.2, 0.5\}$. This gives a total of $3 \times 3 = 9$ different experiments on this particular problem, which we refer to as setup A1, A2, ..., and A9 below.

This allows controlled studies w.r.t. noise levels and training set sizes, while keeping the network architecture constant (2 inputs, 8 tanh hidden, and one linear output).

Predicting Puget Sound Power and Light Co. Power Load between 7 and 8 a.m. the Following Day. This data set is taken from the Puget Sound Power and Light Co's power prediction competition [3]. The winner of this competition used a set of linear models, one for each hour of the day. We have selected the subproblem of predicting the load between 7 and 8 a.m. 24 hrs. in advance. This hour shows the largest variation in power load. The training set consists of 844 weekdays between January 1985 and September 1990. Of these, 150 days are randomly selected and used for validation. We use 115 winter weekdays, from between November 1990 and March 1992, for out-of-sample testing. The inputs are things like current load, average load during the last 24 hours, average load during the last week, time of the year, etc., giving a total of 15 inputs. Three different numbers of internal units are tried on this task: 15, 10, and 5, and we refer to these experiments as B1, B2, and B3 below.

Predicting Daily Riverflow in Two Icelandic Rivers. This problem is tabulated in [22], and the task is to model tomorrow's average flow of water in one of two Icelandic rivers, knowing today's and previous days' waterflow, temperature, and precipitation. The training set consists of 731 data points, corresponding to the years 1972 and 1973, out of which we randomly sample 150 datapoints for validation. The test set has 365 data points (the year 1974). We use two different lengths of lags, 8 or 4 days back, which correspond to 24 or 12 inputs, while the number of internal units is kept constant at 12. These experiments are referred to as C1, C2, C3, and C4 below.

Predicting the Wolf Sunspots Time Series. This time series has been used several times in the context of demonstrating new regularization techniques, for instance by [15] and [26]. We try three different network architectures on this problem, always keeping 12 input units but using 4, 8, or 12 internal units in the network. These experiments are referred to as setup D1, D2, and D3 below. The training set size is kept constant at $M = 221$ (years 1700-1920), out of which we randomly pick 22 patterns for validation. We test our models under four different conditions: Single step prediction on "test set 1" with 35 data points (years 1921-1955), 4-step iterated prediction on "test set 1", 8-step iterated prediction on all 74 available test years (1921-1994), and 11-step iterated prediction on all available test years. These test conditions are coded as s1, m4, m8, and m11.

Estimating the Peak Pressure Position in a Combustion Engine. This is a data set with 4 input variables (ignition time, engine load, engine speed, and air/fuel ratio) and only 49 training data points, out of which we randomly pick 9 patterns for validation. The test set consists of 35 data points, which have been measured under slightly different conditions than the training data. We try four different numbers of internal units on this task: 2, 4, 8, or 12, and refer to these experiments as E1, E2, E3, and E4.

3.4.2 Experimental Procedure

The experimental procedure is the same for all problems: We begin by estimating λ in the “traditional” way by searching over the region $\log \lambda \in [-6.5, 1.0]$ in steps of $\Delta \log \lambda = 0.5$. For each λ value, we train 10 networks using the Rprop training algorithm³ [18]. Each network is trained until the total error (3.7) is minimized, measured by

$$\log \left[\frac{1}{100} \sum_{i=1}^{100} \frac{|\Delta E_i|}{\|\Delta \mathbf{W}_i\|} \right] < -5, \quad (3.23)$$

where the sum runs over the most recent 100 epochs, or until 10^5 epochs have passed, whichever occurs first. The convergence criterion (3.23) is usually fulfilled within 10^5 epochs. New validation and training sets are sampled for each of the 10 networks, but the different validation sets are allowed to overlap. Means and standard deviations, $\log E_{nCV,k}$ and $\sigma_{nCV,k}$, for the errors are estimated from these 10 network runs, assuming a lognormal distribution for the validation errors. Figure 3.2 shows an example of such a search for the Wolf sunspot problem, using a neural network with 12 inputs, 8 internal units, and 1 linear output.

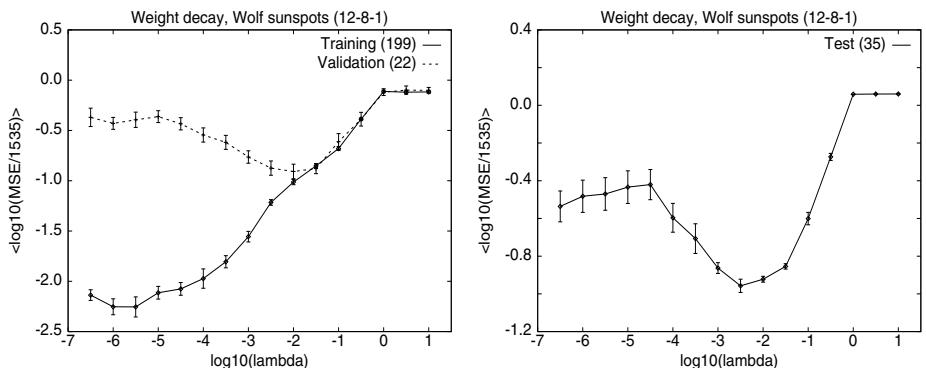


Fig. 3.2. Left panel: Training and validation errors on the Wolf sunspot time series, setup D2, plotted versus the weight decay parameter λ . Each point corresponds to an average over 10 runs with different validation and training sets. The error bars mark 95% confidence limits for the average validation and training errors, under the assumption that the errors are lognormally distributed. The objective Monte Carlo method gives $\log \hat{\lambda}_{opt} = -2.00 \pm 0.31$. Right panel: The corresponding plot for the test error on the sunspots “test set 1”. The network architecture is 12 inputs, 8 tanh internal units, and 1 linear output.

Using the objective Monte Carlo method described above, we estimate an optimal $\hat{\lambda}_{opt}$ value from this search. This value is then used to train 10 new

³ Initial tests showed that the Rprop algorithm was considerably more efficient and robust than e.g. backprop or conjugate gradients in minimizing the error. We did not, however, try true second order algorithms like Levenberg-Marquardt or Quasi-Newton.

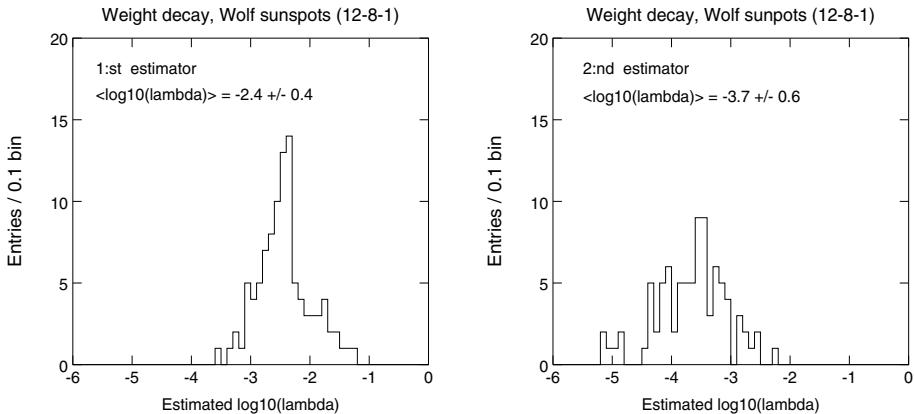


Fig. 3.3. Left panel: Histogram showing the estimated values $\hat{\lambda}_1$ for 100 different training runs, using different training and validation sets each time. Right panel: Similar histogram for $\hat{\lambda}_2$. The problem (D2) is the same as that depicted in Figure 3.2.

networks with all the training data (no validation set). The test errors for these networks are then computed using the held out test set.

A total of $16 \times 10 = 160$ network runs are thus done to select the $\hat{\lambda}_{opt}$ for each experiment. This corresponds to a few days' or a week's work, depending on available hardware and the size of the problem. Although this is in excess of what is really needed in practice (one could get away with about half as many runs in a real application) the time spent doing this is aggravating. The times needed for doing the searches described in this paper ranged from 10 up to 400 cpu-hours, depending on the problem and the computer⁴. For comparison, the early stopping experiments described below took between 10 cpu-minutes and 14 cpu-hours. There was typically a ratio of 40 between the time needed for a search and the time needed for an early stopping estimate.

We then estimate $\hat{\lambda}_1$ and $\hat{\lambda}_2$, by training 100 networks with early stopping. One problem here is that the stopping point is ill-defined, i.e. the first observed minimum in the validation error is not necessarily the minimum where one should stop. The validation error quite often decreases again beyond this point. To avoid such problems, we keep a record of the weights corresponding to the latest minimum validation error and continue training beyond that point. The training is stopped when as many epochs have passed as it took to find the validation error minimum without encountering a new minimum. The weights corresponding to the last validation error minimum are then used as the early stopping weights. For example, if the validation error has a minimum at say 250 epochs, we then wait until a total of 500 epochs have passed before deciding on that particular stopping point. From the 100 networks, we get 100 estimates for $\hat{\lambda}_1$ and $\hat{\lambda}_2$. We take the logarithm of these and compute means $\langle \log \hat{\lambda}_1 \rangle$ and $\langle \log \hat{\lambda}_2 \rangle$, and corresponding standard deviations.

⁴ A variety of computers were used for the simulations, including NeXT, Sun Sparc, DEC Alpha, and Pentium computers running Solaris.

The resulting arithmetic mean values are taken as the estimates for λ and the standard deviations are used as measures of the estimation error. The arithmetic means are then used to train 10 networks which use all the training data. Figure 3.3 shows the histograms corresponding to the problem presented in Figure 3.2.

When comparing test errors achieved with different methods, we use the Wilcoxon rank test [13], also called the Mann-Whitney test, and report differences at 95% confidence level.

3.4.3 Quality of the λ Estimates

As a first test of the quality of the estimates $\hat{\lambda}_1$ and $\hat{\lambda}_2$, we check how well they agree with the $\hat{\lambda}_{opt}$ estimate, which can be considered a “truth”. The estimates for all the problem setups are tabulated in table 3.1 and plotted in Figure 3.4.

Table 3.1. Estimates of λ for the 23 different problem setups. Code A corresponds to the synthetic problem, code B to the Power prediction, code C to the riverflow prediction, code D to the Sunspots series, and code E to the maximum pressure position problem. For the $\log \hat{\lambda}_{opt}$ column, errors are the standard deviations of the Monte Carlo estimate. For the early stopping estimates, errors are the standard deviations of the estimates.

Problem	$\log \hat{\lambda}_{opt}$	$\log \hat{\lambda}_1$	$\log \hat{\lambda}_2$
A1 ($M = 20, \sigma = 0.1$)	-2.82 ± 0.04	-2.71 ± 0.66	-3.44 ± 1.14
A2 ($M = 20, \sigma = 0.2$)	-2.67 ± 0.42	-2.32 ± 0.58	-3.20 ± 0.96
A3 ($M = 20, \sigma = 0.5$)	-0.49 ± 1.01	-1.93 ± 0.78	-3.14 ± 1.15
A4 ($M = 40, \sigma = 0.1$)	-2.93 ± 0.49	-2.85 ± 0.73	-3.56 ± 0.87
A5 ($M = 40, \sigma = 0.2$)	-2.53 ± 0.34	-2.41 ± 0.64	-2.91 ± 0.68
A6 ($M = 40, \sigma = 0.5$)	-2.43 ± 0.44	-2.13 ± 0.74	-2.85 ± 0.77
A7 ($M = 100, \sigma = 0.1$)	-3.45 ± 0.78	-3.01 ± 0.86	-3.74 ± 0.93
A8 ($M = 100, \sigma = 0.2$)	-3.34 ± 0.71	-2.70 ± 0.73	-3.33 ± 0.92
A9 ($M = 100, \sigma = 0.5$)	-3.31 ± 0.82	-2.34 ± 0.63	-3.13 ± 1.06
B1 (Power, 15 hidden)	-3.05 ± 0.21	-3.82 ± 0.42	-5.20 ± 0.70
B2 (Power, 10 hidden)	-3.57 ± 0.35	-3.75 ± 0.45	-4.93 ± 0.50
B3 (Power, 5 hidden)	-4.35 ± 0.66	-3.78 ± 0.52	-5.03 ± 0.74
C1 (Jökulsá Eystra, 8 lags)	-2.50 ± 0.10	-3.10 ± 0.33	-4.57 ± 0.59
C2 (Jökulsá Eystra, 4 lags)	-2.53 ± 0.12	-3.15 ± 0.40	-4.20 ± 0.59
C3 (Vatnsdalsá, 8 lags)	-2.48 ± 0.11	-2.65 ± 0.40	-3.92 ± 0.56
C4 (Vatnsdalsá, 4 lags)	-2.39 ± 0.55	-2.67 ± 0.45	-3.70 ± 0.62
D1 (Sunspots, 12 hidden)	-2.48 ± 0.12	-2.48 ± 0.50	-3.70 ± 0.42
D2 (Sunspots, 8 hidden)	-2.00 ± 0.31	-2.43 ± 0.45	-3.66 ± 0.60
D3 (Sunspots, 4 hidden)	-2.51 ± 0.44	-2.39 ± 0.48	-3.54 ± 0.65
E1 (Pressure, 12 hidden)	-3.13 ± 0.43	-3.03 ± 0.70	-4.69 ± 0.91
E2 (Pressure, 8 hidden)	-3.01 ± 0.52	-3.02 ± 0.64	-4.72 ± 0.82
E3 (Pressure, 4 hidden)	-3.83 ± 0.80	-3.07 ± 0.71	-4.50 ± 1.24
E4 (Pressure, 2 hidden)	-4.65 ± 0.78	-3.46 ± 1.34	-4.21 ± 1.40

The linear correlation between $\log \hat{\lambda}_1$ and $\log \hat{\lambda}_{opt}$ is 0.71, which is more than three standard deviations larger than the expected correlation between 23 random points. Furthermore, a linear regression with intercept gives the result

$$\hat{\lambda}_{opt} = 0.30 + 1.13\hat{\lambda}_1. \quad (3.24)$$

Thus, $\hat{\lambda}_1$ is a fairly good estimator of $\hat{\lambda}_{opt}$.

The linear correlation between $\hat{\lambda}_2$ and $\hat{\lambda}_{opt}$ is 0.48, more than two standard deviations from the random correlation. A linear regression gives

$$\hat{\lambda}_{opt} = -0.66 + 0.57\hat{\lambda}_2, \quad (3.25)$$

and the second estimator $\hat{\lambda}_2$ is clearly a less good estimator of $\hat{\lambda}_{opt}$.

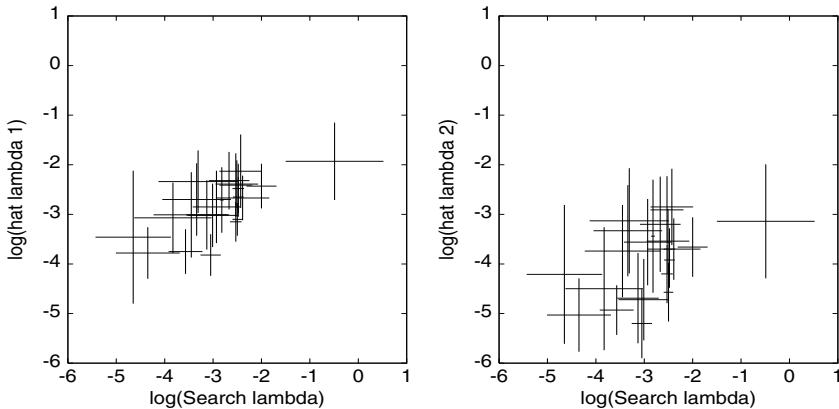


Fig. 3.4. Plot of the results in Table 3.1. Left plate: The $\hat{\lambda}_1$ estimate plotted versus $\hat{\lambda}_{opt}$. The linear correlation between $\log \hat{\lambda}_1$ and $\log \hat{\lambda}_{opt}$ is 0.71. Right plate: $\hat{\lambda}_2$ plotted versus $\hat{\lambda}_{opt}$. The linear correlation between $\log \hat{\lambda}_2$ and $\log \hat{\lambda}_{opt}$ is 0.48. The sizes of the crosses correspond to the error bars in Table 3.1.

We next compare the out-of-sample performances of these different λ estimates, which is what really matters to the practitioner. Table 3.2 lists the differences in out-of-sample performance when using the early stopping estimates or the search estimate. A “+” means that using the early stop estimate results in significantly (95% significance level) lower test error than if $\hat{\lambda}_{opt}$ is used. Similarly, a “-” means that the search estimate gives significantly lower test error than the early stopping estimates. A “0” means there is no significant difference. The conclusion from Table 3.2 is that $\hat{\lambda}_2$ is significantly worse than $\hat{\lambda}_{opt}$, but that there is no consistent difference between $\hat{\lambda}_1$ and $\hat{\lambda}_{opt}$. The two estimates are essentially equal, in terms of test error. In some cases, like the power prediction problem, it would have been beneficial to do a small search around the early stop estimate to check for a possibly better value.

The test errors for the combustion engine (setups E) are not included in Tables 3.2 (and 3.3) because the test set is too different from the training set to provide relevant results. In fact, no regularized network is significantly better than an unregularized network on this problem.

Table 3.2. Relative performance of single networks trained using the estimates $\hat{\lambda}_1$ and $\hat{\lambda}_2$, for the weight decay parameter, and the performance of single networks trained using the search estimate $\hat{\lambda}_{opt}$. The relative performances are reported as: “+” means that using $\hat{\lambda}_i$ results in a test error which is significantly lower than what the search estimate $\hat{\lambda}_{opt}$ gives, “0” means that the performances are equivalent, and “–” means that using $\hat{\lambda}_{opt}$ results in a lower test error than when using $\hat{\lambda}_i$. All results are reported for a 95% confidence level when using the Wilcoxon test. See the text on why the E results are left out.

Problem Setup	$\hat{\lambda}_1$ vs. $\hat{\lambda}_{opt}$	$\hat{\lambda}_2$ vs. $\hat{\lambda}_{opt}$
A1 ($M = 20, \sigma = 0.1$)	0	0
A2 ($M = 20, \sigma = 0.2$)	0	–
A3 ($M = 20, \sigma = 0.5$)	0	0
A4 ($M = 40, \sigma = 0.1$)	0	0
A5 ($M = 40, \sigma = 0.2$)	0	–
A6 ($M = 40, \sigma = 0.5$)	0	0
A7 ($M = 100, \sigma = 0.1$)	–	0
A8 ($M = 100, \sigma = 0.2$)	0	0
A9 ($M = 100, \sigma = 0.5$)	+	0
B1 (Power, 15 hidden)	–	–
B2 (Power, 10 hidden)	–	–
B3 (Power, 5 hidden)	+	–
C1 (Jökulsá Eystra, 8 lags)	–	–
C2 (Jökulsá Eystra, 4 lags)	0	–
C3 (Vatnsdalsá, 8 lags)	–	–
C4 (Vatnsdalsá, 4 lags)	0	–
D1.s1 (Sunspots, 12 hidden)	0	–
D2.s1 (Sunspots, 8 hidden)	+	–
D3.s1 (Sunspots, 4 hidden)	0	–
D1.m4 (Sunspots, 12 hidden)	0	–
D2.m4 (Sunspots, 8 hidden)	+	–
D3.m4 (Sunspots, 4 hidden)	+	–
D1.m8 (Sunspots, 12 hidden)	0	–
D2.m8 (Sunspots, 8 hidden)	–	–
D3.m8 (Sunspots, 4 hidden)	+	–
D1.m11 (Sunspots, 12 hidden)	0	0
D2.m11 (Sunspots, 8 hidden)	+	–
D3.m11 (Sunspots, 4 hidden)	0	–

3.4.4 Weight Decay versus Early Stopping Committees

Having trained all these early stopping networks, it is reasonable to ask if using them to estimate λ for a weight decay network is the optimal use of these networks? Another possible use is, for instance, to construct a committee [16] from them.

To test this, we compare the test errors for our regularized networks with those when using a committee of 10 networks trained with early stopping. The results are listed in Table 3.3.

Some observations from Table 3.3, bearing in mind that the set of problems is small, are: Early stopping committees seem like the better option when the problem is very noisy (setups A3, A6, and A9), and when the network does not have very many degrees of freedom (setups B3, C4, and D3). Weight decay networks, on the other hand, seem to work better than committees on problems with many degrees of freedom (setups B1 and C3), problems with low noise levels and much data (setup A7), and problems where the prediction is iterated through the network (m4, m8, and m11 setups). We emphasize, however, that these conclusions are drawn from a limited set of problems and that all problems tend to have their own set of weird characteristics.

We also check which model works best on each problem. On the power prediction, the best overall model is a large network (B1) which is trained with weight decay. On the river prediction problems, the best models are small (C2 and C4) and trained with either weight decay (Jökulsá Eystra) or early stopping and then combined into committees (Vatnsdalsá). On the sunspot problem, the best overall model is a large network (D1) trained with weight decay.

These networks are competitive with previous results on the same data sets. The performance of the power load B1 weight decay networks, using $\hat{\lambda}_{opt}$, are significantly better than what a human expert produces, and also significantly better than the results by the winner of the Puget Sound Power and Light Co. Power Load Competition [7], although the difference is small. The test results are summarized in Figure 3.5. The performance of the sunspot D1 weight decay network is comparable with the network by Weigend et al., listed in [26]. Figure 3.6 shows the performance of the D1 network trained with weight decay, $\lambda = \hat{\lambda}_1$, and compares it to the results by Weigend et al. [26]. The weight decay network produces these results using a considerably simpler λ selection method and regularization cost than the one presented in [26].

From these anecdotal results, one could be bold and say that weight decay shows a slight edge over early stopping committees. However, it is fair to say that it is a good idea to try both committees and weight decay when constructing predictor models.

It is emphasized that these results are from a small set of problems, but that these problems (except perhaps for the synthetic data) are all realistic in the sense that the datasets are small and noisy.

Table 3.3. Relative performance of single networks trained using weight decay and early stopping committees with 10 members. The relative performance of weight decay (WD) and 10 member early stopping committees are reported as: “+” means that weight decay is significantly better than committees, “0” means that weight decay and committees are equivalent, and “−” means that committees are better than weight decay. All results are reported for a 95% confidence level when using the Wilcoxon test. See the text on why the E results are left out.

Problem Setup	WD($\hat{\lambda}_{opt}$) vs. Comm.	WD($\hat{\lambda}_1$) vs. Comm.
A1 ($M = 20, \sigma = 0.1$)	0	+
A2 ($M = 20, \sigma = 0.2$)	0	0
A3 ($M = 20, \sigma = 0.5$)	−	−
A4 ($M = 40, \sigma = 0.1$)	0	0
A5 ($M = 40, \sigma = 0.2$)	+	+
A6 ($M = 40, \sigma = 0.5$)	−	−
A7 ($M = 100, \sigma = 0.1$)	+	+
A8 ($M = 100, \sigma = 0.2$)	0	0
A9 ($M = 100, \sigma = 0.5$)	−	0
B1 (Power, 15 hidden)	+	−
B2 (Power, 10 hidden)	0	−
B3 (Power, 5 hidden)	−	−
C1 (Jökulsá Eystra, 8 lags)	+	0
C2 (Jökulsá Eystra, 4 lags)	+	0
C3 (Vatnsdalsá, 8 lags)	+	+
C4 (Vatnsdalsá, 4 lags)	−	−
D1.s1 (Sunspots, 12 hidden)	0	0
D2.s1 (Sunspots, 8 hidden)	−	0
D3.s1 (Sunspots, 4 hidden)	−	−
D1.m4 (Sunspots, 12 hidden)	+	+
D2.m4 (Sunspots, 8 hidden)	+	+
D3.m4 (Sunspots, 4 hidden)	0	+
D1.m8 (Sunspots, 12 hidden)	+	+
D2.m8 (Sunspots, 8 hidden)	+	0
D3.m8 (Sunspots, 4 hidden)	0	0
D1.m11 (Sunspots, 12 hidden)	+	+
D2.m11 (Sunspots, 8 hidden)	0	+
D3.m11 (Sunspots, 4 hidden)	+	+

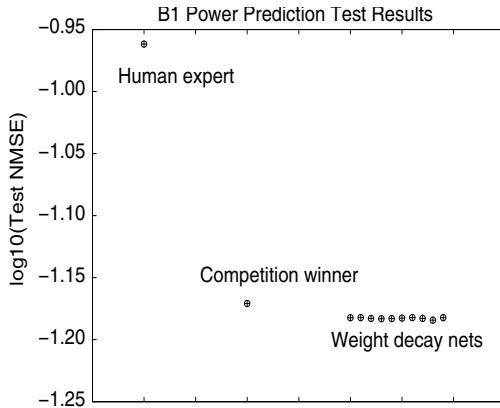


Fig. 3.5. The performance of the 10 neural networks with 15 inputs, 15 hidden units, and one output unit, trained with weight decay using $\lambda = \hat{\lambda}_{opt}$, on the power prediction problem. “Human expert” denotes the prediction result by the human expert at Puget Sound Power and Light Co., and “Competition winner” denotes the result by the model that won the Puget Sound Power and Light Co.’s Power Prediction Competition.

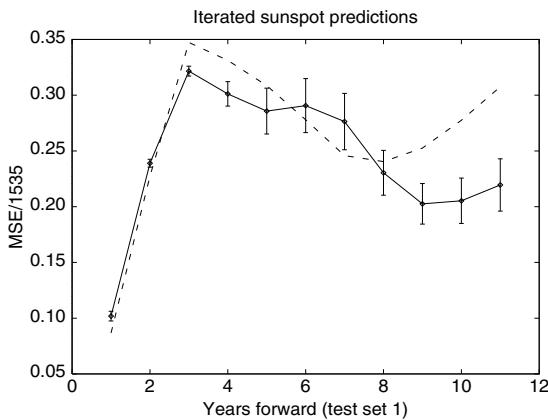


Fig. 3.6. The performance of a neural network with 12 inputs, 12 hidden units, and one output unit, trained with weight decay using $\lambda = \hat{\lambda}_1$, on iterated predictions for the sunspot problem. The error bars denote one standard deviation for the 10 trained networks. The dashed line shows the results when using the network listed in [26]. Note that these results are achieved with a simple weight decay cost and a very simple method for selecting λ , whereas [26] use weight elimination and a complicated heuristic scheme for setting λ .

3.5 Conclusions

The established connection between early stopping and weight decay regularization naturally leads to the idea of using early stopping to estimate the weight decay parameter. In this paper we have shown how this can be done and that the resulting λ results in as low test errors as achieved with the standard cross-validation method, although this varies between problems. In practical applications, this means replacing a search which may take days or weeks, with a computation that usually does not require more than a few minutes or hours. This value can also be used as a starting point for a more extensive cross-validation search.

We have also shown that using several early stopping networks to estimate λ can be smarter than combining the networks into committees. The conclusion from this is that although there is a correspondence between early stopping and weight decay under asymptotic conditions this does not mean that early stopping and weight decay give equivalent results in real life situations.

The method unfortunately only works for regularization terms that have a connection with early stopping, like quadratic weight decay or “weight decay like” regularizers where the weights are constrained towards the origin in weight space (but using e.g. a Laplacian prior instead of the usual Gaussian prior). The method does not carry over to regularizers which do not have any connection to early stopping (like e.g. Tikhonov smoothing regularizers).

Acknowledgements. David B. Rosen is thanked for a very inspiring dinner conversation during the 1996 “Machines that Learn” Workshop in Snowbird, Utah. Milan Casey Brace of Puget Sound Power and Light Co. is thanked for supplying the power load data. Financial support is gratefully acknowledged from NSF (grant CDA-9503968), Olle and Edla Ericsson’s Foundation, the Swedish Institute, and the Swedish Research Council for Engineering Sciences (grant TFR-282-95-847).

References

- [1] Abu-Mostafa, Y.S.: Hints. *Neural Computation* 7, 639–671 (1995)
- [2] Bishop, C.M.: Curvature-driven smoothing: A learning algorithm for feedforward networks. *IEEE Transactions on Neural Networks* 4(5), 882–884 (1993)
- [3] Brace, M.C., Schmidt, J., Hadlin, M.: Comparison of the forecast accuracy of neural networks with other established techniques. In: Proceedings of the First International Form on Application of Neural Networks to Power System, Seattle WA, pp. 31–35 (1991)
- [4] Buntine, W.L., Weigend, A.S.: Bayesian back-propagation. *Complex Systems* 5, 603–643 (1991)
- [5] Cheeseman, P.: On Bayesian model selection. In: The Mathematics of Generalization - The Proceedings of the SFI/CNLS Workshop on Formal Approaches to Supervised Learning, pp. 315–330. Addison-Wesley, Reading (1995)

- [6] Cybenko, G.: Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems* 2, 304–314 (1989)
- [7] Engle, R., Clive, F., Granger, W.J., Ramanathan, R., Vahid, F., Werner, M.: Construction of the puget sound forecasting model. EPRI Project # RP2919, Quantitative Economics Research Institute, San Diego, CA (1991)
- [8] Geman, S., Bienenstock, E., Doursat, R.: Neural networks and the bias/variance dilemma. *Neural Computation* 4(1), 1–58 (1992)
- [9] Girosi, F., Jones, M., Poggio, T.: Regularization theory and neural networks architectures. *Neural Computation* 7, 219–269 (1995)
- [10] Hansen, L.K., Rasmussen, C.E., Svarer, C., Larsen, J.: Adaptive regularization. In: Vlontzos, J., Hwang, J.-N., Wilson, E. (eds.) *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing IV*, pp. 78–87. IEEE Press, Piscataway (1994)
- [11] Hoerl, A.E., Kennard, R.W.: Ridge regression: Biased estimation of nonorthogonal problems. *Technometrics* 12, 55–67 (1970)
- [12] Ishikawa, M.: A structural learning algorithm with forgetting of link weights. Technical Report TR-90-7, Electrotechnical Laboratory, Information Science Division, 1-1-4 Umezono, Tsukuba, Ibaraki 305, Japan (1990)
- [13] Kendall, M.G., Stuart, A.: *The Advanced Theory of Statistics*, 3rd edn. Hafner Publishing Co., New York (1972)
- [14] Moody, J.E., Rögnvaldsson, T.S.: Smoothing regularizers for projective basis function networks. In: *Advances in Neural Information Processing Systems 9*. MIT Press, Cambridge (1997)
- [15] Nowlan, S., Hinton, G.: Simplifying neural networks by soft weight-sharing. *Neural Computation* 4, 473–493 (1992)
- [16] Perrone, M.P., Cooper, L.C.: When networks disagree: Ensemble methods for hybrid neural networks. In: *Artificial Neural Networks for Speech and Vision*, pp. 126–142. Chapman and Hall, London (1993)
- [17] Plaut, D., Nowlan, S., Hinton, G.: Experiments on learning by backpropagation. Technical Report CMU-CS-86-126, Carnegie Mellon University, Pittsburg, PA (1986)
- [18] Riedmiller, M., Braun, H.: A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In: Ruspini, H. (ed.) *Proc. of the IEEE Intl. Conference on Neural Networks*, San Fransisco, California, pp. 586–591 (1993)
- [19] Ripley, B.D.: *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge (1996)
- [20] Sjöberg, J., Ljung, L.: Overtraining, regularization, and searching for minimum with application to neural nets. *Int. J. Control* 62(6), 1391–1407 (1995)
- [21] Tikhonov, A.N., Arsenin, V.Y.: *Solutions of Ill-Posed problems*. V. H. Winston & Sons, Washington D.C. (1977)
- [22] Tong, H.: *Non-linear Time Series: A Dynamical System Approach*. Clarendon Press, Oxford (1990)
- [23] Utans, J., Moody, J.E.: Selecting neural network architectures via the prediction risk: Application to corporate bond rating prediction. In: *Proceedings of the First International Conference on Artificial Intelligence Applications on Wall Street*. IEEE Computer Society Press, Los Alamitos (1991)
- [24] Wahba, G., Gu, C., Wang, Y., Chappell, R.: Soft classification, a.k.a. risk estimation, via penalized log likelihood and smoothing spline analysis of variance. In: *The Mathematics of Generalization - The Proceedings of the SFI/CNLS Workshop on Formal Approaches to Supervised Learning*, pp. 331–359. Addison-Wesley, Reading (1995)

- [25] Wahba, G., Wold, S.: A completely automatic french curve. *Communications in Statistical Theory & Methods* 4, 1–17 (1975)
- [26] Weigend, A., Rumelhart, D., Hubermann, B.: Back-propagation, weight-elimination and time series prediction. In: Sejnowski, T., Hinton, G., Touretzky, D. (eds.) *Proc. of the Connectionist Models Summer School*. Morgan Kaufmann Publishers, San Mateo (1990)
- [27] Williams, P.M.: Bayesian regularization and pruning using a Laplace prior. *Neural Computation* 7, 117–143 (1995)

Controlling the Hyperparameter Search in MacKay's Bayesian Neural Network Framework^{*}

Tony Plate

School of Mathematical and Computing Sciences,
 Victoria University, Wellington, New Zealand
 tap@mcs.vuw.ac.nz
<http://www.mcs.vuw.ac.nz/~tap/>

Abstract. In order to achieve good generalization with neural networks overfitting must be controlled. Weight penalty factors are one common method of providing this control. However, using weight penalties creates the additional search problem of finding the optimal penalty factors. MacKay [5] proposed an approximate Bayesian framework for training neural networks, in which penalty factors are treated as hyperparameters and found in an iterative search. However, for classification networks trained with cross-entropy error, this search is slow and unstable, and it is not obvious how to improve it. This paper describes and compares several strategies for controlling this search. Some of these strategies greatly improve the speed and stability of the search. Test runs on a range of tasks are described.

4.1 Introduction

Neural networks can provide useful flexible statistical models for non-linear regression and classification. However, as with all such models, the flexibility must be controlled to avoid overfitting. One way of doing this in neural networks is to use weight penalty factors (regularization parameters). This creates the problem of finding the values of the penalty factors which will maximize performance on new data. As various researchers have pointed out, including MacKay [5], Neal [10] and Bishop [1], it is generally advantageous to use more than one penalty factor, in order to differentially penalize weights between different layers of the network. However, doing this makes it computationally infeasible to choose optimal penalty factors by k-fold cross validation.

MacKay [5] describes a Bayesian framework for training neural networks and choosing optimal penalty factors (which are hyperparameters in his framework). In this framework, we choose point estimates of hyperparameters to maximize the “evidence” of the network. Parameters (i.e., weights) can be assigned into different

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

groups, and each controlled by a separate hyperparameter. This allows weights between different layers to be penalized differently. MacKay [6, 8] and Neal [10] have shown that it also provides a way of implementing “Automatic Relevance Detection” (ARD), in which connections emerging from different units in the input layer are assigned to different regularization groups. The idea is that hyperparameters controlling weights for irrelevant inputs should become large, driving those weights to zero, while hyperparameters for relevant inputs stabilize at small to moderate values. This can help generalization by causing the network to ignore irrelevant inputs and also makes it possible to see at a glance which inputs are important.

In this framework the search for an optimal network has two levels. The inner level is a standard search for weights which minimize error on the training data, with fixed hyperparameters. The outer level is a search for hyperparameters which maximize the evidence. For the Bayesian theory to apply, the inner level search should be allowed to converge to a local minima at each step of the outer level search. However, this can be expensive and slow. Problems with speed and stability of the search seem especially severe with classification networks trained with cross-entropy error.

This paper describes experiments with different control strategies for updating the hyperparameters in the outer level search. These experiments show that the simple “let it run to convergence and then update” strategy often does not work well, and that other strategies can generally work better. In previous work, the current author successfully employed one of these strategies in an application of neural networks to epidemiological data analysis [11]. The experiments reported here confirm the necessity for update strategies and also demonstrate that although the strategy used in this previous work is reasonably effective in some situations, there are simpler and better strategies which work in a wider range of situations. These experiments also furnish data on the relationship between the evidence and the generalization error. This data confirms theoretical expectations about when the evidence should and should not be a good indication of generalization error.

In the second section of this chapter, the update formulas for hyperparameters are given. Network propagation and weight update formulas are not given, as they are well known and available elsewhere, e.g., in Bishop [1]. Different control strategies for the outer level hyperparameter search are described in the third section. In the fourth section, the simulation experiments are described, and the results are reported in the fifth section. The experimental relationships between evidence and generalization error are reported in the sixth section.

4.2 Hyperparameter Updates

The update formulas for the hyperparameters (weight penalty factors) in the outer level search are quite simple. Before describing them we need some terminology. For derivations and background theory see Bishop [1], MacKay [5, 7], or Thodberg [13].

- n is the total number of weight in the network.
- w_i the value of the i th weight.

- K is the number of hyperparameters.
- \mathcal{I}_c is the set of indices of the weights in the c th hyperparameter group.
- α_c is the value of the hyperparameter controlling the c th hyperparameter group; it specifies the prior distribution on the weights in that group. $\alpha_{[i]}$ denotes the value of the hyperparameter controlling the group to which weight i belongs.
- n_c is the number of weights in the c th hyperparameter group.
- C is the weight cost (penalty term) for the network: $C = \frac{1}{2} \sum_{i=1}^n \alpha_{[i]} w_i^2$.
- m is the total number of training examples.
- y^j and t^j are the network outputs and target values, respectively, for the j th training example.
- E is the error term of the network. For the classification networks described here, the modified cross-entropy (Bishop [1], p.232) is used:

$$E = - \sum_{j=1}^m \left\{ t^j \log \frac{y^j}{t^j} + (1 - t^j) \log \frac{1 - y^j}{1 - t^j} \right\}.$$

Note that all graphs and tables of test set performance use the “deviance”, which is twice the error.

- \mathbf{H} is the Hessian of the network (the second partial derivatives of the sum of the error and weight cost). h_{ij} denotes the ij th element of this matrix, and h_{ij}^{-1} denotes the ij th element of \mathbf{H}^{-1} :

$$h_{ij} = \frac{\partial^2(E + C)}{\partial w_i \partial w_j}.$$

\mathbf{H}^E is the matrix of second partial derivatives of just the error, and \mathbf{H}^C is the matrix of second partial derivatives of just the weight cost.

- $\text{Tr}(\mathbf{H}^{-1})$ is the trace of the inverse of \mathbf{H} : $\text{Tr}(\mathbf{H}^{-1}) = \sum_{i=1}^n h_{ii}^{-1}$.
- $\text{Tr}_c(\mathbf{H}^{-1})$ is the trace of the inverse Hessian for just those elements of the c th regularization group: $\text{Tr}_c(\mathbf{H}^{-1}) = \sum_{i \in \mathcal{I}_c} h_{ii}^{-1}$.
- γ_c is a derived parameter which can be seen as an estimate of the number of well-determined parameters in the c th regularization group, i.e., the number of parameters determined by the data rather than by the prior.

The overall training procedure is shown in Figure 4.1.

The updates for the hyperparameters α_c depend on the estimate γ_c (the number of well-determined parameters in group c) which is calculated as follows (Eqn 27 in [7]; derivable from Eqn 10.140 in [1]):

$$\gamma_c = n_c - \alpha_c \text{Tr}_c(\mathbf{H}^{-1}). \quad (4.1)$$

If a Gaussian distribution is a reasonable approximation to the posterior weight distribution, γ_c should be between 0 and n_c . Furthermore, we expect each parameter in group c to contribute between 0 and 1 to γ_c . Hence, we expect h_{ii}^{-1} to always be in the range $[0, 1/\alpha_{[i]}]$.

```

set the  $\alpha_c$  to initial values
set  $w_i$  to initial random values
repeat
  repeat
    make an optimization step for weights to minimize  $E + C$ 
  until finished weight optimization
  re-estimate the  $\alpha_c$ 
until finished max number of passes through training data

```

Fig. 4.1. The training procedure

The updates for the α_c is as follows (Eqn 22 in [7]; Eqn 10.74 in [1]):

$$\alpha'_c = \frac{\gamma_c}{\sum_{i \in \mathcal{I}_c} w_i^2} \quad (4.2)$$

MacKay [7] remarks that this formula can be seen as matching the prior to the data: $1/\alpha_c$ is an estimate of the variance for the weights in group c , taking into account the effective number of well determined parameters (effective degrees of freedom) in that group.

4.2.1 Difficulties with Using the Update Formulas

The difficulties with using these update formulas arise when the assumption that the error plus cost surface is a quadratic bowl is false. This assumption can fail in two ways: the error plus cost surface may not be quadratic, or it may not be a bowl (i.e., the Hessian is not positive definite). In either of these situations, it is possible for γ_c to be out of the range $[0, n_c]$. To illustrate, consider a single diagonal element of the Hessian in the situation where off-diagonal elements are zero:

$$\mathbf{H} = \begin{bmatrix} \ddots & 0 \\ h_{ii} & \ddots \\ 0 & \ddots \end{bmatrix} = \begin{bmatrix} \ddots & 0 \\ h_{ii}^E + \alpha_{[i]} & \ddots \\ 0 & \ddots \end{bmatrix}$$

Since the off-diagonal elements are zero, the inverse Hessian is simple to write down:

$$\mathbf{H}^{-1} = \begin{bmatrix} \ddots & 0 \\ \frac{1}{h_{ii}^E + \alpha_{[i]}} & \ddots \\ 0 & \ddots \end{bmatrix}$$

Suppose the i parameter is in the c th regularization group, by itself. Then the number of well-determined parameters in this group is given by:

$$\gamma_{[i]} = 1 - \alpha_{[i]} h_{ii}^{-1} = 1 - \frac{\alpha_{[i]}}{h_{ii}^E + \alpha_{[i]}} = \frac{h_{ii}^E}{h_{ii}^E + \alpha_{[i]}} \quad (4.3)$$

If h_{ii}^E is positive, $\gamma_{[i]}$ will be between 0 and 1. $\gamma_{[i]}$ will be large if h_{ii}^E is large relative to $\alpha_{[i]}$, which means that w_i is well determined by the data, i.e., small moves of w_i will make a large difference to E . $\gamma_{[i]}$ will be small if h_{ii}^E is small relative to $\alpha_{[i]}$, which means that w_i is poorly determined by the data.

The expectation that h_{ii}^{-1} is in the range $[0, 1/\alpha_c]$ (and hence contributes between 0 and 1 well determined parameter to $\gamma_{[i]}$) can fail even if the model is at a local minima of $E + C$. Being at a local minimum of $E + C$ does not guarantee that h_{ii}^E will be positive: it is possible for the hyperparameter to “pin” the weight value to a convex portion of a non-quadratic E surface. Consider the case where the Hessian is diagonal and positive definite, but h_{ii}^E is negative. From Eqn 4.3, we can see that h_{ii}^{-1} can make a negative contribution¹ to γ_c , which makes little sense in terms of “numbers of well-determined parameters”. This situation² is illustrated in Figure 4.2: at the minimum of the $E + C$ function the E function is convex ($\frac{d^2 E}{dw^2}$ is negative). Here, negative degrees of freedom would be calculated under the (incorrect) assumption that error plus cost is quadratic. This is important for neural networks, because even if sum-squared error is used, non-linearities in the sigmoids can cause the the error plus cost function to be not a quadratic function of weights.

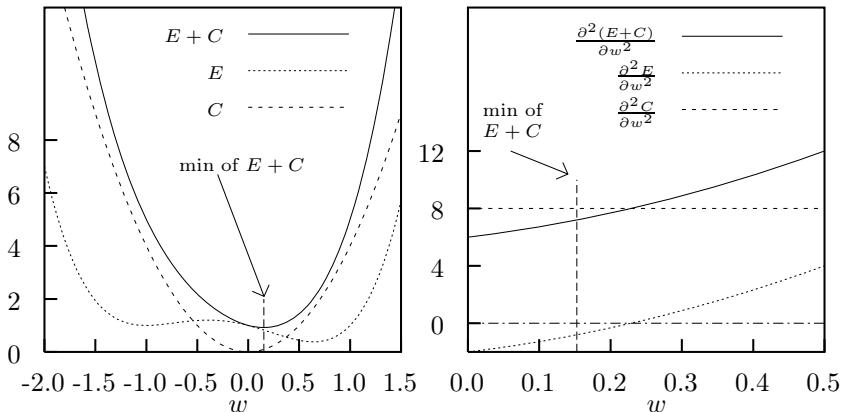


Fig. 4.2. In minimizing $E + C$, a weight cost function C can pin a weight value to a convex portion of the error surface E . The plot on the left shows the surfaces, the plot on the right shows the derivatives in the region of the minimum.

If the model is not at a local minimum of $E + C$ all bets are off. \mathbf{H} may not even be positive definite (i.e., the Hessian of a quadratic bowl), and if this is the case it is almost certain that some h_{ii}^{-1} will be out of the range $[0, 1/\alpha_{[i]}]$. Even

¹ With general matrices it is possible that $h_{ii}^{-1} < -\alpha_{[i]}$, in which case the contribution will be an unbounded positive number.

² In Figure 4.2, $E = w(w - 1)(w + 1)^2 + 1$, $C = 4w^2$, $\left.\frac{d(E+C)}{dw}\right|_{0.152645} \approx 0$, and $\left.\frac{d^2 E}{dw^2}\right|_{0.152645} \approx -0.8036$.

if \mathbf{H} is positive definite, and \mathbf{H}^E is also positive definite, it can still be the case that some h_{ii}^{-1} are out-of-bounds, and thus make contributions of less than zero or more than one “well-determined parameter” each.

These difficulties leave us with two problems:

1. When should the hyperparameters be updated?
2. What should be done when h_{ii}^{-1} is not in $[0, 1/\alpha_{[i]}]$ (i.e., is “out-of-bounds”).

Bishop [1] suggests updating hyperparameters after every few weight updates. Thodberg [13] suggests updating the α_c after every weight update, but only recalculating the γ_c occasionally (at five evenly-spaced intervals throughout the whole process). While updating hyperparameters after every weight update is not feasible when using conjugate gradient or other second-order training methods, common practice seems to be more or less in line with Thodberg’s recommendations: train the network more or less to convergence before each hyperparameter update. However, this strategy can result in an extremely slow overall search. Furthermore, training to convergence does not eliminate the problem of out-of-bounds h_{ii}^{-1} values. In the remainder of this chapter, various strategies for choosing when to update hyperparameters, and for dealing with out-of-bounds h_{ii}^{-1} values are described and compared in experiments.

4.3 Control Strategies

The strategies tested here fall into three groups: strategies for when to update hyperparameters, strategies for dealing with out-of-bounds h_{ii}^{-1} values, and special strategies for dealing with exceptional cases. For each task, searches were run a fixed number of minimization steps, with different combinations of control strategies. Each step of the inner minimization loop was a step of a conjugate gradient method, and could involve a number of passes through the training data, though the average was just over two.

4.3.1 Choosing When to Update Hyperparameters

Four different strategies for choosing when to break out of the inner loop and update hyperparameters were employed:

rare: Update at 10 evenly spaced intervals.

medium: Update at 30 evenly spaced intervals.

often: Update at 100 evenly spaced intervals.

patience: Update when the improvement in the last n steps was less than 1% of the improvement since the start of the inner loop, or when the improvement in the last n steps was less than 0.01% of the null error (the minimum error that can be achieved with a constant output value). At least n steps of the inner loop are taken.

Convergence was difficult to test when using a number of different datasets and networks. One standard way of detecting convergence is to test whether the ratio $|\bar{g}|/|\bar{w}|$ is less than some threshold ($|\bar{g}|$ is the Euclidean length of the vector of weight derivatives, and $|\bar{w}|$ is the Euclidean length of the vector of weights.) However, the appropriate threshold varied greatly among different tasks. In any case, with all strategies, if a convergence test was met ($|\bar{g}|/|\bar{w}| < 10^{-6}$), the inner loop terminated, and the hyperparameters were updated. This did not occur often in the experiments described here.

The “patience” strategy is intended to be a surrogate for convergence: the inner loop runs out of patience when the improvement achieved in the last n steps is minuscule. With “patience”, the inner loop is guaranteed to terminate if the error is bounded below. In practice, the inner loop runs out of patience reasonable quickly: the update rate is somewhere between “rare” and “medium” depending on the difficulty of the optimization problem.

4.3.2 Dealing with Out-of-Bounds Estimates of Numbers of Well-Determined Parameters

In each of the experiments, one of the following strategies was used to deal with out-of-bounds h_{ii}^{-1} values. In describing these strategies, h_{ii}^{-1}' or γ_c' are used to denote values which are used instead of the originally calculated ones.

none: This strategy allows γ_c to take on unreasonably large values, but not negative ones (h_{ii}^{-1} values are not checked):

$$\gamma_c' = \begin{cases} 0 & \text{if } \gamma_c < 0 \\ \gamma_c & \text{otherwise} \end{cases}$$

group: This strategy forces the total number of well-determined parameters in a regularization group to be reasonable:

$$\gamma_c' = \begin{cases} 0 & \text{if } \gamma_c < 0 \\ n_c & \text{if } \gamma_c > n_c \\ \gamma_c & \text{otherwise} \end{cases}$$

trim: This strategy forces the contribution of each h_{ii}^{-1} to be reasonable. If h_{ii}^{-1} is out-of-bounds, it is assumed to represent zero well-determined parameters:

$$h_{ii}^{-1}' = \begin{cases} 1/\alpha_{[i]} & \text{if } h_{ii}^{-1} \text{ is not in } [0, 1/\alpha_{[i]}] \\ h_{ii}^{-1} & \text{otherwise} \end{cases}$$

snip: This strategy forces the contribution of each h_{ii}^{-1} to be reasonable. If h_{ii}^{-1} is out-of-bounds, it is assumed to represent one well-determined parameter:

$$h_{ii}^{-1}' = \begin{cases} 0 & \text{if } h_{ii}^{-1} \text{ is not in } [0, 1/\alpha_{[i]}] \\ h_{ii}^{-1} & \text{otherwise} \end{cases}$$

useold: This strategy forces the contribution of each h_{ii}^{-1} to be reasonable. If h_{ii}^{-1} is out-of-bounds the last good estimate of the well-determinedness of parameter i is used:

$$\gamma'_c = \sum_{i=1}^{n_c} \begin{cases} 1 - \alpha_c^* h_{ii}^{-1*} & \text{if } h_{ii}^{-1} \text{ is not in } [0, 1/\alpha_{[i]}] \\ 1 - \alpha_c h_{ii}^{-1} & \text{otherwise} \end{cases},$$

where α_c^* and h_{ii}^{-1*} are most recent values such that h_{ii}^{-1} is in $[0, 1/\alpha_{[i]}]$, or if there are no such values, $h_{ii}^{-1*} = 0$.

cond: This strategy only updates α_c , using γ_c or a snipped version of γ_c , under the following conditions:

- (a) all eigenvalues of \mathbf{H} are positive, **and**
- (b) for all $i \in \mathcal{I}_c$, h_{ii}^{-1} is in $[0, 1/\alpha_{[i]}]$, **or** a snipped version of γ_c will result in a change in α_c in the same direction as the last change when all the eigenvalues of \mathbf{H} were positive and all the h_{ii}^{-1} in group c were in bounds, **and** there have not been more than five such changes since all of the h_{ii}^{-1} for group c have been in range.

cheap: This is Mackay's [7] "cheap and cheerful" method, in which all parameters are assumed to be well determined, i.e.,

$$\gamma'_c = n_c.$$

The advantage of this method is that Hessian need not be calculated. Mackay remarks that this method can be expected to perform poorly when there are a large number of poorly determined parameters.

4.3.3 Further Generally Applicable Strategies

Several further strategies which could be combined with any of the ones already mentioned were also employed:

nza: (no zero alphas) Do not accept an updated alpha value of zero (retain the old value).

limit: Limit the the change in an alpha value to have a magnitude of no more than 10, i.e., round α'_c to be in the interval $[0.1\alpha_c, 10\alpha_c]$.

omit: If there are any h_{ii}^{-1} not in $[0, 1/\alpha_{[i]}]$, omit the corresponding rows and columns from \mathbf{H} to give the smaller matrix \mathbf{H}' , and use the diagonal elements of \mathbf{H}'^{-1} for h_{ii}^{-1} . The idea is to omit troublesome components of the model so that they do not interfere with estimates of well-determinedness for well-behaved parameters. Strategies from the previous section are used to assign well-determinedness values for parameters with out-of-bound h_{ii}^{-1} values in \mathbf{H}^{-1} or \mathbf{H}'^{-1} .

4.4 Experimental Setup

The experiments reported here used seven different test functions, based on the five 2-dimensional functions used by Hwang et al. [4] and Roosen and Hastie [12]:

linear function:

$$f_0(x_1, x_2) = (2x_1 + x_2)/0.6585$$

simple interaction:

$$f_1(x_1, x_2) = 10.391((x_1 - 0.4)(x_2 - 0.6) + 0.36)$$

radial function:

$$\begin{aligned} f_2(x_1, x_2) = & 24.234((x_1 - 0.5)^2 + (x_2 - 0.5)^2)(0.75 - ((x_1 - 0.5)^2 \\ & + (x_2 - 0.5)^2)) \end{aligned}$$

harmonic function:

$$\begin{aligned} f_3(x_1, x_2) = & 42.659(0.1 + (x_1 - 0.5)(0.05 - 10(x_1 - 0.5)^2(x_2 - 0.5)^2 \\ & + (x_1 - 0.5)^4 + 5(x_2 - 0.5)^4)) \end{aligned}$$

additive function:

$$\begin{aligned} f_4(x_1, x_2) = & 1.3356(1.5(1 - x_1) + e^{(2x_1 - 1)} \sin(3\pi(x_1 - 0.6)^2) \\ & + e^{3(x_2 - 0.5)} \sin(4\pi(x_2 - 0.9)^2)) \end{aligned}$$

complicated interaction:

$$f_5(x_1, x_2) = 1.9(1.35 + e^{x_1} \sin(13(x_1 - 0.6)^2)e^{-x_2} \sin(7x_2))$$

interaction plus linear:

$$f_6(x_1, x_2) = 0.83045(f_0(x_1, x_2) + f_1(x_1, x_2))$$

Training data for the binary classification tasks was generated by choosing 250 x_1^j and x_2^j points from a uniform distribution over [0, 1]. Another 250 s^j points were chosen from the uniform [0, 1] distribution to determine whether the target should be 0 or 1. The {0, 1} target t_i^j for case j for function i depended on the probability p_i^j of a 1 calculated as the sigmoid of the function value (with the mean of the function subtracted):

$$\begin{aligned} p_i^j &= \frac{1}{1 + e^{-(f_i(x_1^j, x_2^j) - \mu_i)}} \\ t_i^j &= \begin{cases} 0 & \text{if } p_i^j < s^j \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

where μ_i is the mean of each function over the unit square ($\mu_0 = 2.28, \mu_1 = 3.6, \mu_2 = 2.3, \mu_3 = 4.4, \mu_4 = 2.15, \mu_5 = 2.7, \mu_6 = 4.92$).

A further 5 random distractor points chosen from a uniform [0, 1] distribution were concatenated with each (x_1^j, x_2^j) pair, to give a 7 dimensional function-approximation task. These extra points were added so that in order for the neural network to generalize well it would be necessary that the automatic relevance determination set the weights from irrelevant inputs to zero (by driving the α 's for those weights to high values).

For testing, the probabilities were used as targets, rather than stochastic binary values. This was done to reduce the noise in measuring the test error. The test set inputs consisted of 400 (x_1, x_2) points from the uniform grid over $[0, 1]$, and 400 vectors of distractors chosen from a uniform distribution over $[0, 1]$. In order that test and training errors be comparable the test error was calculated as the expected error over the test points, assuming actual targets had been chosen randomly with the given target probabilities. The expected error for one test case was calculated as follows:

$$E_i^j = - \left(p_i^j \log y_i^j + (1 - p_i^j) \log(1 - y_i^j) \right)$$

where y_i^j was the prediction of the network for test case j .

The seven learning tasks are called I0 through I6 (the “I” is for “impure” in Roosen and Hastie’s terminology [12].) It should be noted that these learning tasks are much harder than the tasks used by Hwang et al. [4], as the binomial outputs make the training data much noisier, and irrelevant inputs are present.

4.4.1 Targets for “Good” Performance

Various simple modeling strategies were applied to the data to give an indication of what level of test error performance was achievable, and to demonstrate how important it was to ignore the distractor inputs. Three different models were tried: linear and quadratic logistic models, and generalized additive models (GAMs) [3], each with and without the distractor inputs. No attempt was made to prevent overfitting, as the intention of these models was to show how much overfitting can occur if distractor are not ignored.

- null:** No inputs are used. The predicted output is the average of the targets (1 parameter).
- true:** The actual function value. This is included to indicate the level of noise in the data.
- lin:** A logistic (linear) model fit using only x_1 and x_2 as inputs (3 parameters).
- lin.D:** A logistic (linear) model fit using all inputs, including the distractors (8 parameters).
- quad:** A logistic model (with quadratic terms) fit using only x_1 and x_2 as inputs (6 parameters).
- quad.D:** A logistic model (with quadratic terms) fit using all inputs, including the distractors (36 parameters).
- gam:** A generalized additive model fit using only x_1 and x_2 as inputs, with three degrees of freedom for each dimension (approx 7 parameters).
- gam.D:** A generalized additive model fit using all inputs, with three degrees of freedom for each dimension (approx 22 parameters).
- T:** The target for “good” network performance.

The test set deviances for the various models and tasks are shown in Table 4.1, ordered by error for each task. Some tasks are easy, while others are very difficult

(finding good solutions for task I3 appears to extremely difficult, as Roosen and Hastie [12] also discovered.) Targets for “good” neural network performance were derived from the errors achieved by any model. The targets were chosen to be achievable by neural network models and yet lower than the test set deviance achieved by any of the above simple models using all the inputs (except for I3, which neural networks had great difficulty with).

Table 4.1. Test set deviances (twice the error) for various models. Names ending in “.D” are those of models which used both the distractor and relevant inputs. The number in parentheses beside the target for good network performance (T) is the number of networks which achieved this target at the end of training (out of 540).

I0	I1	I2	I3	I4	I5	I6
true 478	true 488	true 475	true 495	true 480	true 491	true 478
lin 487	quad 495	quad 494	gam 546	gam 498	gam 539	quad 485
T (165) 493	T (258) 520	gam 507	T (7) 555	T (114) 530	T (58) 545	T (233) 500
lin.D 498	lin 545	T (44) 530	lin 556	quad 533	quad 547	lin 517
quad 501	gam 548	gam.D 551	lin.D 557	gam.D 537	lin 547	lin.D 523
gam 502	null 556	null 555	null 558	lin 554	lin.D 555	gam 529
gam.D 519	lin.D 558	lin 561	quad 566	null 556	null 555	quad.D 540
null 555	quad.D 595	quad.D 575	gam.D 574	lin.D 558	gam.D 572	gam.D 550
quad.D 556	gam.D 608	lin.D 577	quad.D 669	quad.D 597	quad.D 620	null 557

4.4.2 Network Architecture and Training

Standard feed-forward networks were used, with details as follows. All networks had 3 to 15 hidden units, which computed the tanh function (a symmetric sigmoid). Inputs were in the range 0 to 1. The output unit computed the logistic function of its total input. Weights were initialized to random values drawn from a Gaussian distribution with variance 0.5.

Networks had nine hyperparameters (weight penalties): one for the weights for each input, one for hidden unit biases, and one for hidden to output weights. There was no penalty on the output bias. All hyperparameters were initialized to 0.5. Hyperparameters were allowed to increase to maximum value of 10,000.

Networks were trained using a conjugate algorithm for the number of steps specified in Table 4.2. These numbers of steps were chosen give ample time for reasonably good methods to converge on some solution (the harder problems required more steps). Each step of the conjugate gradient algorithm involved one or more passes through the training set (the average was just over two). Training was terminated if the total number of passes through the training set exceeded 2.3 times the maximum allowed number of conjugate gradient steps.

The Hessian was calculated using the exact analytical method described in Buntine and Weigend [2]. This requires $h + 1$ passes through the training data, where h is the number of hidden units. This is usually far faster than a finite-differences method, which requires $n + 1$ for the forward differences method and $2n + 1$ passes for the more accurate central differences method. The total number of floating point operations involved in the exact calculation of the Hessian is dominated by the update of the Hessian matrix (Eqn 15c in [2]). For a network with one output unit it is approximately $3.5(h + 1)Nn^2$ (there are 7 operations

in each Hessian element update, but only half the elements need be computed as the matrix is symmetric). Eigendecomposition and inversion of the Hessian takes approximately $4/3n^3$ operations, but this is usually small compared to the calculation of the Hessian matrix. As long as hyperparameters are not updated too frequently, the time taken by Hessian evaluation and inversion is generally not an excessive amount on top of the time taken by the standard weight-training part of the procedure. For example, in the easier tasks (I0, I1, and I6) with the most frequent Hessian calculations (the “often” updates: 100 during), approximately one-third of the computation time was spent in Hessian calculations. In the more difficult tasks, relatively less time was spent in Hessian calculations because the training times were longer.

Thirty six different combinations of the hyperparameter update strategies discussed in Section 4.3 were tested. Training on each problem was repeated five times with different initial weights. The same five sets of random initial weights were used for each strategy. This means that there were a total of 160 attempts to train each sized network for each problem.

Table 4.2. Number of conjugate gradient steps allowed for different sized networks on the different tasks

Task	Number of hidden units			
	3	5	10	15
I0, I1, I6	1800	3000	6000	–
I2, I3, I4	3000	6000	12000	–
I5	–	6000	12000	30000

4.5 Effectiveness of Control Strategies

Effectiveness of various combinations of control strategies was judged by whether or not the test error at the end of training was acceptable (using the deviance targets in Table 4.1). Good performance on any of the tasks was not possible without setting the hyperparameters in the appropriate ranges: low for the weights coming from the x_1 and x_2 inputs, and high for the weights coming from the distractor inputs. Figure 4.3 shows example plots of hyperparameter values versus test deviance for networks trained on task I6 (with jitter added to make dense clouds of points visible). Task I6 was a reasonably easy task: nearly all networks ended with appropriate low values for the relevant-input hyperparameters. Finding appropriate high values for distractor hyperparameters was more difficult, and those networks which did not perform well. All such plots had the same tendencies as those in Figure 4.3: low test set deviance was achieved only by networks with low values for relevant-input hyperparameters and high values for distractor hyperparameters. Some of the other tasks were more difficult, e.g., I5, and poor search strategies would set relevant-input hyperparameters to high values, resulting in poor performance from ignoring the relevant inputs.

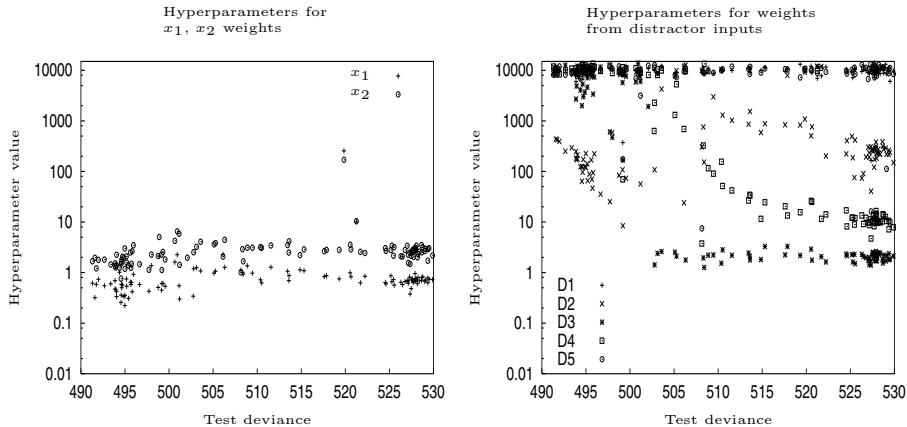


Fig. 4.3. Final hyperparameter values versus test set deviances for networks with 5 hidden units trained on task I6

The number of successes for each of the 36 (combinations of) strategies on each task is shown in Table 4.3. Each asterisk represents one success, i.e., a network which ended up with test set performance lower than the target for good performance (Table 4.1.) The maximum number of successes for any cell is 15, as 5 random starts were used for each of three different sized networks. The row totals are out of 105, and the column totals are out of 540. The grand total (987 successes) is out of 3780.

The clear overall best strategy is “snip+often”. The special strategies “nza” and “limit” seem to help a little. These conclusions are strengthened by examining plots of the evolution of test deviance during training. Figures 4.4 and 4.5 show plots of test deviance during training for networks with 5 hidden units on task I6, and for networks with 10 hidden units on task I4. There are five lines in each plot because five random starts were used for each network. The ideal network has a test deviance which descends rapidly and then stays below the target performance (the dotted line) – this shows that the strategy quickly found good values for hyperparameters (the tasks were set up so that it was not possible to achieve low test deviance without having appropriate values for the hyperparameters). Lines which flatten out at around 557 are for networks whose hyperparameters have all been set to high values, so that the network ignores all inputs and thus has the same performance as the null model (see Table 4.1). The search was terminated early for many of these networks, because no hyperparameters or weights were changing. Note that few of the networks with 10 hidden units reached the target for good performance on task I4 though networks with 5 hidden units did much better. This indicates that the redundancy in these networks could not be effectively controlled.

Table 4.3. Number of successes for each strategy on each task

	total	I0	I1	I2	I3	I4	I5	I6
cheap+often	4		***			*		
group+medium	10	***	***		*			***
group+patience	10	*	***		*			***
none+patience	10		***			*		
none+rare	11		***			**		***
group+rare	12		***			***		***
snip+rare	14	**	**			***		***
trim+often	16	****	****	****				
trim+patience	17	***	*****	*		*	*	***
none+medium	18	***	*****	*	*	**		***
omit+snip+patience	18		*****			*****		***
omit+snip+patience+limit	18		*****			*****		***
trim+medium	20	****	****	***			***	***
trim+rare	20	**	****			***	***	***
useold+patience	20	***	****	*	*	**	*	***
omit+useold+patience	21	***	*****			***	**	***
omit+useold+patience+limit	22	***	*****			***	***	***
cheap+medium	22	****	****					*****
cheap+patience	23		****		*			*****
none+often	26	*****	*****	*		***	**	***
snip+medium	26	*****	*****			***	**	***
snip+patience	26	*****	*****	*		***	***	***
omit+trim+patience	27	*****	*****	*		**	**	***
snip+patience+nza+limit	27	*****	*****	*		***	***	***
omit+useold+often	28	*****	*****	***		**		***
omit+useold+often+limit	29	*****	*****	*		***		***
snip+patience+nza	29	*****	*****	*		***	***	***
group+often	31	*****	*****	*		*	**	*****
cond+often+limit+nza	32	*****	*****			*****		*****
omit+snip+often+limit	32	*****	*****			*****		*****
useold+often	32	*****	*****	***		**		*****
cheap+rare	34	*****	*****	***				*****
omit+snip+often	36	**	*****			*****		*****
snip+often	50	*****	*****	**	**	*****	*****	*****
snip+often+nza	54	*****	*****	**	**	*****	*****	*****
snip+often+nza+limit	54	*****	*****	*		*****	*****	*****
	987	165	258	44	7	114	58	233

Update strategies other than “snip” tend to be very unstable. Frequent updates seem essential for achieving good final performance within a reasonable amount of time. The more complex strategies for getting or retaining good estimates of γ ’s, i.e., omit and useold, seemed to be of little benefit.

Finding good hyperparameter values is difficult. If updates are too frequent, α ’s can rise uncontrollably or become unstable. If updates are too infrequent, the search is too slow. Uncontrollable rises in α ’s can occur when the network has not started using the weights it needs, and those weights are small. In these circumstances α is overestimated, which forces weights to become smaller in a runaway-feedback process. Instability results because of the feedback between γ and α : a change in γ causes a same direction change in α , and a change in α causes an opposite direction change in γ . Thus, if γ is overestimated, this leads to an overestimation in α , which lead to a lower reestimate of γ , which in turn can lead to a lower reestimate of α . While this process sometimes results in stable self-correcting behavior, other times it results in uncontrolled oscillations (which is what is happening with the “none+often” strategy: second row, third column in Figures 4.4 and 4.5). In general, while it is slow to raise an α from a value that is too low, it is more difficult to lower an α from a value which is too high (because the weights are forced to zero immediately). The reasons for the differing behavior of the various strategies appear to be as follows:

none, group: Frequently give out-of-bounds or very poor estimates for γ , and cause instability in the search.

cond: Results in fairly stable behavior, but often does not find the optimal values for all α 's because it stops updating them.

cheap: Overestimates γ and kills weights too quickly.

trim: Sometimes underestimates γ because it assumes zero degrees of freedom when the γ estimates from the Hessian are out-of-bounds, and thus reduces α values which should be large, resulting in instability.

snip: Sometimes overestimates γ because it assumes one degree of freedom for a parameter whose γ estimate from the Hessian is out-of-bounds. However, this keeps α values high without raising them too much: overestimation of γ appears to be much less harmful than underestimation.

Update frequency was also important. With good α calculation strategies, updating frequently (i.e., “often”) gave fastest convergence to good α values and best overall performance. Waiting for some degree of convergence in the conjugate gradient search before updating α values (i.e., “patience”) was of no benefit at all.

4.6 Relationship of Test Set Error to Evidence

If we have trained a number of networks, we often want to know which will have the lowest generalization error. The “evidence” value calculated for each network can be used choose networks which will perform well. The evidence is the log likelihood of the data given the values of the hyperparameters, integrated over weight values based on the assumption of a Gaussian distribution for the posterior of the weights. The evidence for a network evaluated with cross-entropy error is as follows (Eqn 10.67 in [1]; Eqn 30 in [7]):

$$\ln p(D|\alpha) = -\frac{1}{2} \sum_{i=1}^n \alpha_{[i]} w_i^2 - E - \frac{1}{2}|H| + \sum_c \frac{n_c}{2} \ln \alpha_c - \frac{N}{2} \ln(2\pi) \quad (4.4)$$

Whether or not the evidence is a good guide to which networks will perform well is questionable, as various assumptions on which evidence calculations are based are often violated in particular networks. The simulations performed offer a good opportunity to examine how accurately high evidence indicates good test set performance.

The evidence is particularly sensitive to low eigenvalues in the Hessian of the network. The validity of the evidence value is doubtful in cases where the Hessian has low or negative eigenvalues. Bishop [1] recommends omitting eigenvalues which are lower than some threshold from the calculation of the evidence.

Figure 4.6 shows plots of test deviances that would be achieved by selecting twenty networks based on their evidence values. Two different methods of calculating the evidence were used: (a) ignore negative eigenvalues of the Hessian, and (b) if an eigenvalue is lower than the smallest non-zero α_c , replace it with that α_c (“clipped evidence”). Because the evidence is sensitive to low eigenvalues, four different filters were applied to networks: (a) use all networks; (b) throw out networks

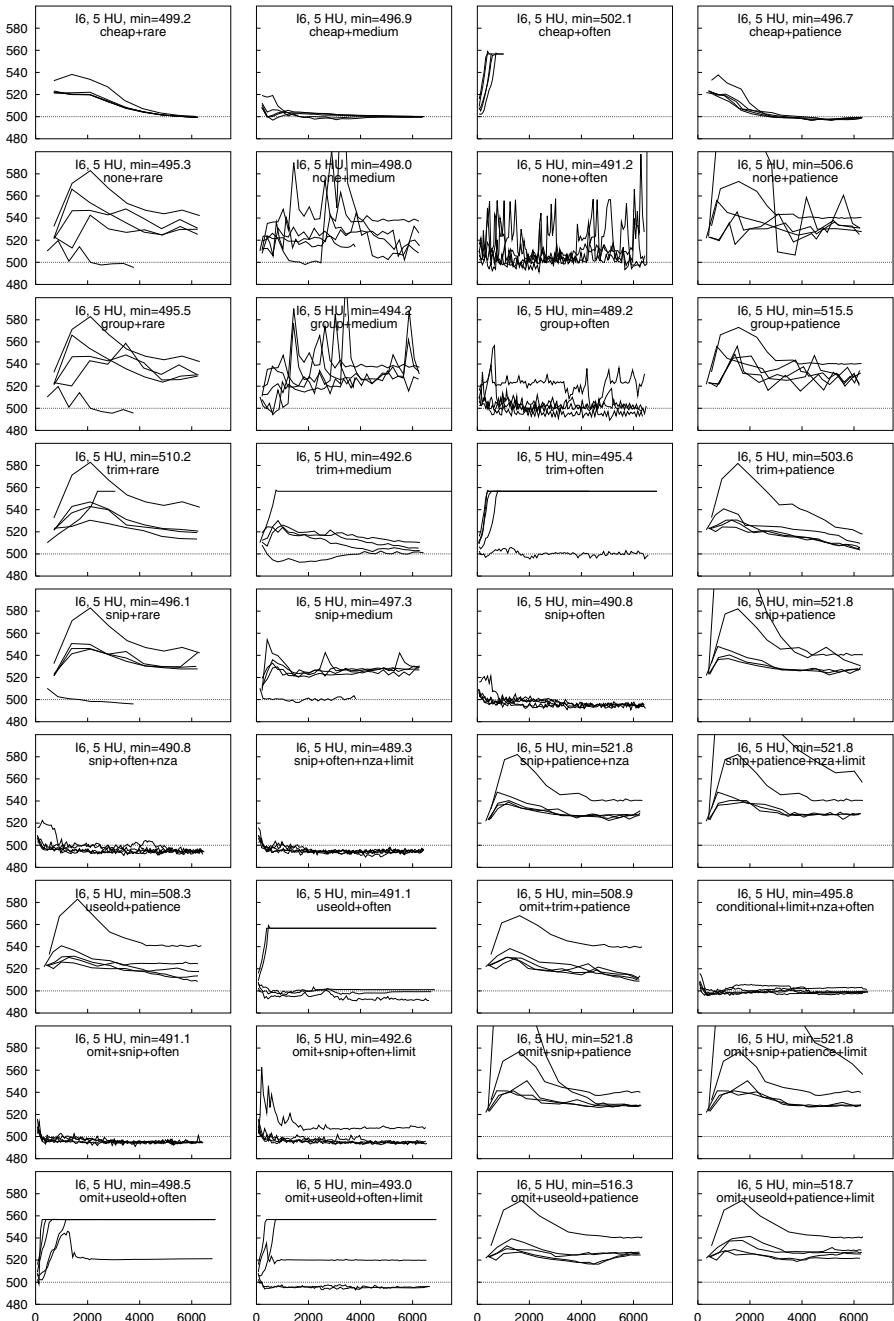


Fig. 4.4. Test set deviance during training for networks with 5 hidden units on task I6. The horizontal axis is the number of passes through the training set. The dotted line is the target for good performance. The minimum value quoted is the minimum deviance at any point in training.

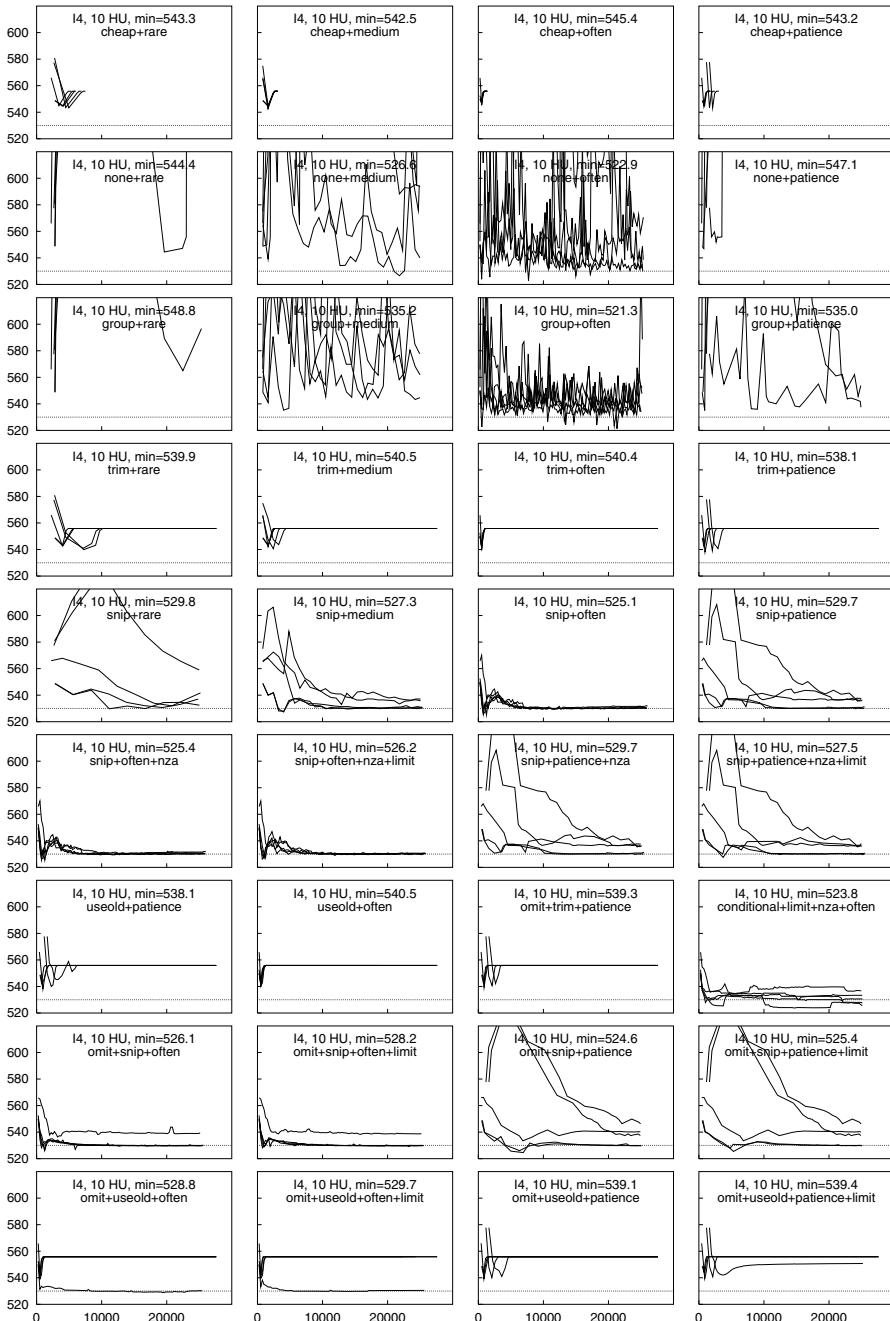


Fig. 4.5. Test set deviance during training for networks with 10 hidden units on task I4. The horizontal axis is the number of passes through the training set. The dotted line is the target for good performance. The minimum value quoted is the minimum deviance at any point in training.

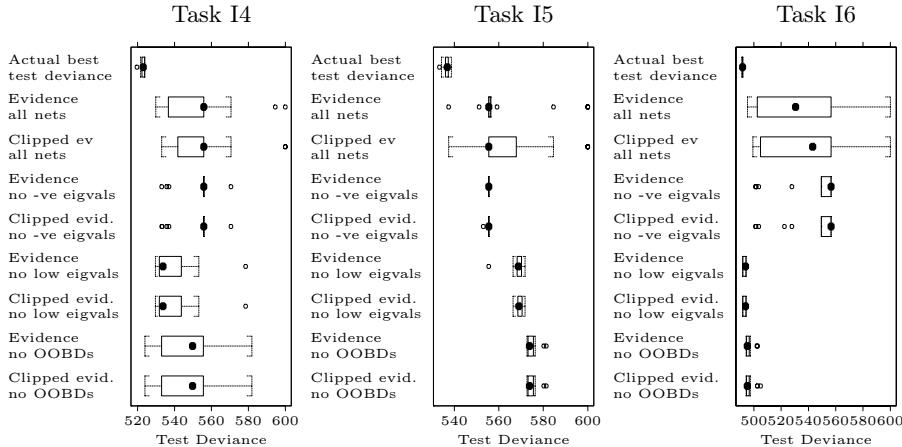


Fig. 4.6. Distribution of test errors for the 20 “best” nets selected according to evidence and network diagnostics

with negative eigenvalues; (c) throw out networks with eigenvalues lower than the smallest α_c ; and (d) throw out networks with out-of-bounds h_{ii}^{-1} values.

The first row in each plot of Figure 4.6 shows the distribution of the actual best 20 test deviances (this would normally be impossible to calculate, but we can calculate it here because we have artificial tasks.) If evidence were a perfect predictor of test set performance, the 20 networks with highest evidence would be the same 20 networks, but it is not, as the rest of the figure shows. The remaining eight rows show the distribution of test errors for nets with the highest evidence, with evidence calculated in both of the ways described above. Lower pairs of rows rules more nets out of consideration, based on diagnostics of the Hessian and inverse Hessian.

The two different ways of calculating the evidence did not make any discernible difference. The box-and-whisker plots show the median as a solid circle. The box shows the upper and lower quartiles (i.e., 50% of points are in the box). The whiskers show the furthest points within 1.5 times the interquartile range (the length of the box) of the quartiles. Points outside the whiskers are plotted individually with hollow circles.

Plots of the evidence versus test deviance show a strong correlation, which seems strongest and most reliable for networks which have no low eigenvalues. However, the plots in Figure 4.6 show that using the evidence to select networks is not always reliable. Networks with high evidence but with poorly conditioned Hessians have a wide range of test error, but some networks with the lowest test error have poorly conditioned Hessians. This means that allowing networks with poorly conditioned Hessians resulted in choosing too many networks with high test error, whereas rejecting networks with poorly conditioned Hessians rejected too many of the networks with low test error. This seemed to be more of a problem for the more difficult tasks. On the easier tasks (e.g., I6), networks with high evidence and no low eigenvalues had very low test deviance.

4.7 Conclusion

For classification networks, the search for optimal hyperparameters can be slow and unstable. However, the search can be improved by using the strategies summarized below. Similar experiments with regression networks (with linear output units and sum-square errors, on tasks with continuous targets and Gaussian noise) revealed that the hyperparameter search in such tasks is generally faster and more stable than in classification tasks. Only when the tasks had very high noise in the training cases (with variance of 4 or greater) did the hyperparameter search become at all difficult. Under these conditions, none of the strategies tried was clearly superior.

One of the main causes of instability in the hyperparameter search is low eigenvalues in the Hessian. In turn, one of the main causes of this is redundancies in the network. It is easily verified that the Hessian for a network with redundancies can have zero or close to zero eigenvalues – the only thing that prevents the eigenvalues being zero is non-zero hyperparameters. In neural networks both additive (parallel) and multiplicative (serial) redundancies can occur. Larger networks are more likely to have redundancies. Hence we could expect the search to be more stable for smaller networks, and this is what was observed (though of course some small networks did not perform well as they did not have sufficient capacity to model the task).

The initial hyperparameter and weight values are important. If the initial hyperparameters are too high, they can force all the weights to zero before the network has a chance to learn anything. If the initial hyperparameters are too low the network can get trapped in an overfitting mode. Thodberg [13] makes the reasonable suggestion that the initial hyperparameters should be set so that the weight cost is 10% of the error at the beginning of training.

The results described in this chapter lead to the following recommendations for hyperparameter updates:

- update hyperparameters frequently
- if any h_{ii}^{-1} values are out-of-bounds, replace them with zero in the calculation of $\gamma_{[i]}$ (so that each out-of-bounds h_{ii}^{-1} contributes one well-determined parameter)
- ignore updates for α_c which suggest a zero value
- limit changes in α_c to have a maximum magnitude of 10.
- ignore negative eigenvalues in calculations of the evidence
- regard evidence values as unreliable for networks with eigenvalues lower than the lowest α_c

Those interested in Bayesian approaches to neural network modeling should also consider Neal's [9] Markov-chain Monte Carlo methods. Although these Monte Carlo methods sometimes require longer computation times than methods based on Mackay's approximate Bayesian framework, they have some theoretical advantages and also require less tweaking.

References

- [1] Bishop, C.: Neural Networks for Pattern Recognition. Oxford University Press (1995)
- [2] Buntine, W.L., Weigend, A.S.: Computing second derivatives in feed-forward networks: A review. *IEEE Transactions on Neural Networks* 5(3), 480–488 (1994)
- [3] Hastie, T.J., Tibshirani, R.J.: Generalized additive models. Chapman and Hall, London (1990)
- [4] Hwang, J.N., Lay, S.-R., Maechler, M., Martin, R.D., Schimert, J.: Regression modeling in back-propagation and projection pursuit learning. *IEEE Transactions on Neural Networks* 5(3), 342–353 (1994)
- [5] MacKay, D.J.C.: A practical Bayesian framework for backpropagation networks. *Neural Computation* 4(3), 448–472 (1992)
- [6] MacKay, D.J.C.: Bayesian methods for backpropagation networks. In: Domany, E., van Hemmen, J.L., Schulten, K. (eds.) *Models of Neural Networks III*, ch. 6, Springer, New York (1994)
- [7] MacKay, D.J.C.: Probable networks and plausible predictions - a review of practical Bayesian methods for supervised neural networks. *Network: Computation in Neural Systems* 6, 469–505 (1995)
- [8] MacKay, D.J.C.: Bayesian non-linear modelling for the 1993 energy prediction competition. In: Heidbreder, G. (ed.) *Maximum Entropy and Bayesian Methods*, Santa Barbara 1993, pp. 221–234. Kluwer, Dordrecht (1996)
- [9] Neal, R.M.: Monte carlo implementation of gaussian process models for bayesian regression and classification. Technical Report TR9702, Dept. of Statistics, University of Toronto (1997), Software, <http://www.cs.utoronto.ca/~radford/>
- [10] Neal, R.M.: Bayesian Learning for Neural Networks. Springer, New York (1996)
- [11] Plate, T., Bert, J., Grace, J., Band, P.: A comparison between neural networks and other statistical techniques for modeling the relationship between tobacco and alcohol and cancer. In: Mozer, M.C., Jordan, M.I., Petsche, T. (eds.) *Advances in Neural Information Processing* 9 (NIPS 1996). MIT Press (1997)
- [12] Roosen, C., Hastie, T.: Logistic response projection pursuit. Technical report, AT&T Bell Laboratories (1993)
- [13] Thodberg, H.H.: A review of bayesian neural networks with an application to near infrared spectroscopy. *IEEE Transactions on Neural Networks* 7(1), 56–72 (1996)

Adaptive Regularization in Neural Network Modeling^{*}

Jan Larsen¹, Claus Svarer², Lars Nonboe Andersen¹, and Lars Kai Hansen¹

¹ Department of Mathematical Modeling, Building 321

Technical University of Denmark

DK-2800 Lyngby, Denmark

² Neurobiology Research Unit

Department of Neurology, Building 9201

Copenhagen University Hospital

Blegdamsvej 9

DK-2100 Copenhagen, Denmark

{jl,lna,lkhansen}@imm.dtu.dk, csvarer@pet.rh.dk

<http://eivind.imm.dtu.dk>, <http://neuro.pet.rh.dk>

Abstract. In this paper we address the important problem of optimizing regularization parameters in neural network modeling. The suggested optimization scheme is an extended version of the recently presented algorithm [25]. The idea is to minimize an empirical estimate – like the cross-validation estimate – of the generalization error with respect to regularization parameters. This is done by employing a simple iterative gradient descent scheme using virtually no additional programming overhead compared to standard training. Experiments with feed-forward neural network models for time series prediction and classification tasks showed the viability and robustness of the algorithm. Moreover, we provided some simple theoretical examples in order to illustrate the potential and limitations of the proposed regularization framework.

5.1 Introduction

Neural networks are flexible tools for time series processing and pattern recognition. By increasing the number of hidden neurons in a 2-layer architecture any relevant target function can be approximated arbitrarily close [19]. The associated risk of overfitting on noisy data is of major concern in neural network design, which find expression in the ubiquitous bias-variance dilemma, see e.g., [9].

The need for regularization is two-fold: First, it remedies numerical problems in the training process by smoothing the cost function and by introducing curvature in low (possibly zero) curvature regions of cost function. Secondly, regularization is a tool for reducing variance by introducing extra bias. The overall

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

objective of architecture optimization is to minimize the generalization error. The architecture can be optimized *directly* by stepwise selection procedures (including pruning techniques) or *indirectly* using regularization. In general, one would prefer a hybrid scheme; however, a very flexible regularization may substitute the need for selection procedures. The numerical experiments we consider mainly hybrid pruning/adaptive regularization schemes.

The trick presented in this communication addresses the problem of adapting regularization parameters.

The trick consists in formulating a simple iterative gradient descent scheme for adapting the regularization parameters aiming at minimizing the generalization error.

We suggest to use an empirical estimate¹ of the generalization error, viz. the K -fold cross-validation [8], [39]. In [25] and [3] the proposed scheme was studied using the hold-out validation estimator.

In addition to empirical estimators for the generalization error a number of *algebraic* estimators like FPE [1], FPER [23], GEN [21], GPE [31] and NIC [33] have been developed in recent years. These estimates, however, depend on a number of statistical assumptions which can be quite hard to justify. In particular, they are $o(1/N_t)$ estimators where N_t is the number of training examples. However, for many practical modeling set-ups it is hard to meet the large training set assumption.

In [14] properties of adaptive regularization is studied in the simple case of estimating the mean of a random variable using an algebraic estimate of the average² generalization error and [15] proposed an adaptive regularization scheme for neural networks based on an algebraic estimate. However, experiments indicate that this scheme has a drawback regarding robustness. In addition, the requirement of a large training set may not be met.

The Bayesian approach to adapt regularization parameters is to minimize the so-called evidence [5, Ch. 10], [30]. The evidence, however, does not in a simple way relate to the generalization error which is our primary object of interest.

Furthermore [2] and [38] consider the use of a validation set to tune the amount of regularization, in particular when using the early-stop technique.

Section 5.2 considers training and empirical generalization assessment. In Section 5.3 the framework for optimization of regularization parameters is presented. The experimental section 5.4 deals with examples of feed-forward neural networks models for classification and time series prediction. Further, in order to study the theoretical potential/limitations of the proposed framework, we include some simulations on a simple toy problem.

5.2 Training and Generalization

Suppose that the neural network is described by the vector function $\mathbf{f}(\mathbf{x}; \mathbf{w})$ where \mathbf{x} is the input vector and \mathbf{w} is the vector of network weights and thresholds

¹ For further discussion on empirical generalization assessment, see e.g., [24].

² Average w.r.t. to different training sets.

with dimensionality m . The objective is to use the network model for approximating the true conditional input-output distribution $p(\mathbf{y}|\mathbf{x})$, or some moments hereof. For regression and signal processing problems we normally model the conditional expectation $E\{\mathbf{y}|\mathbf{x}\}$.

Assume that we have available a data set $\mathcal{D} = \{\mathbf{x}(k); \mathbf{y}(k)\}_{k=1}^N$ of N input-output examples. In order to both train and empirically estimate the generalization performance we follow the idea of K -fold cross-validation [8], [39] and split the data set into K randomly chosen disjoint sets of approximately equal size, i.e., $\mathcal{D} = \cup_{j=1}^K \mathcal{V}_j$ and $\forall i \neq j : \mathcal{V}_i \cap \mathcal{V}_j = \emptyset$. Training and validation is replicated K times, and in the j 'th run training is done on the set $\mathcal{T}_j = \mathcal{D} \setminus \mathcal{V}_j$ and validation is performed on \mathcal{V}_j .

The cost function, $C_{\mathcal{T}_j}$, for network training on \mathcal{T}_j , is supposed to be the sum of a loss function (or training error), $S_{\mathcal{T}_j}(\mathbf{w})$, and a regularization term $R(\mathbf{w}, \boldsymbol{\kappa})$ parameterized by a set of regularization parameters $\boldsymbol{\kappa}$, i.e.,

$$C_{\mathcal{T}_j}(\mathbf{w}) = S_{\mathcal{T}_j}(\mathbf{w}) + R(\mathbf{w}, \boldsymbol{\kappa}) = \frac{1}{N_{tj}} \sum_{k=1}^{N_{tj}} \ell(\mathbf{y}(k), \hat{\mathbf{y}}(k); \mathbf{w}) + R(\mathbf{w}, \boldsymbol{\kappa}) \quad (5.1)$$

where $\ell(\cdot)$ measures the distance between the output $\mathbf{y}(k)$ and the network prediction $\hat{\mathbf{y}}(k) = \mathbf{f}(\mathbf{x}(k); \mathbf{w})$. In section 5.4 we will consider log-likelihood and the square error loss function $\ell = |\mathbf{y} - \hat{\mathbf{y}}|^2$. $N_{tj} \equiv |\mathcal{T}_j|$ defines the number of training examples in \mathcal{T}_j and k is used to index the k 'th example $[\mathbf{x}(k), \mathbf{y}(k)]$. Training provides the estimated weight vector $\hat{\mathbf{w}}_j = \arg \min_{\mathbf{w}} C_{\mathcal{T}_j}(\mathbf{w})$.

The j 'th validation set \mathcal{V}_j consist of $N_{vj} = N - N_{tj}$ examples and the validation error³ of the trained network reads

$$S_{\mathcal{V}_j}(\hat{\mathbf{w}}_j) = \frac{1}{N_{vj}} \sum_{k=1}^{N_{vj}} \ell(\mathbf{y}(k), \hat{\mathbf{y}}(k); \hat{\mathbf{w}}_j) \quad (5.2)$$

where the sum runs over the N_{vj} validation examples. $S_{\mathcal{V}_j}(\hat{\mathbf{w}}_j)$ is thus an estimate of the generalization error, defined as the expected loss,

$$G(\hat{\mathbf{w}}_j) = E_{\mathbf{x}, \mathbf{y}}\{\ell(\mathbf{y}, \hat{\mathbf{y}}; \hat{\mathbf{w}}_j)\} = \int \ell(\mathbf{y}, \hat{\mathbf{y}}; \hat{\mathbf{w}}_j) \cdot p(\mathbf{x}, \mathbf{y}) d\mathbf{x} d\mathbf{y} \quad (5.3)$$

where $p(\mathbf{x}, \mathbf{y})$ is the unknown joint input-output probability density. Generally, $S_{\mathcal{V}_j}(\hat{\mathbf{w}}_j) = G(\hat{\mathbf{w}}_j) + O(1/\sqrt{N_{vj}})$ where $O(\cdot)$ is the Landau order function⁴. Thus we need large N_{vj} to achieve an accurate estimate of the generalization error. On the other hand, this leaves only few data for training thus the true generalization $G(\hat{\mathbf{w}}_j)$ increases. Consequently there exist a trade-off among the two conflicting aims which calls for finding an optimal split ratio. The optimal split ratio⁵ is an interesting open and difficult problem since it depends on the total algorithm in which the validation error enters. Moreover, it depends on the learning curve⁶ [17].

³ That is, the loss function on the validation set.

⁴ If $h(x) = O(g(x))$ then $|h(x)|/|g(x)| < \infty$ for $x \rightarrow 0$.

⁵ For more elaborations on the split of data, see e.g., [2], [20], [24] and [26].

⁶ Defined as the average generalization error as a function of the number of training examples.

The final K -fold cross-validation estimate is given by the average validation error estimates,

$$\hat{\Gamma} = \frac{1}{K} \sum_{j=1}^K S_{\mathcal{V}_j}(\hat{\mathbf{w}}_j). \quad (5.4)$$

$\hat{\Gamma}$ is an estimate of the *average* generalization error over all possible training sets of size N_{tj} ,

$$\Gamma = E_{\mathcal{T}}\{G(\hat{\mathbf{w}}_j)\}. \quad (5.5)$$

$\hat{\Gamma}$ is an unbiased estimate of Γ if the data of \mathcal{D} are independently distributed⁷, see e.g., [16].

The idea is now to optimize the amount of regularization by minimizing $\hat{\Gamma}$ w.r.t. the regularization parameters κ . An algorithm for this purpose is described in Section 5.3. Furthermore, we might consider optimizing regularization using the hold-out validation estimate corresponding to $K = 1$. In this case one has to choose a split ratio. Without further ado, we will recommend a 50/50 splitting.

Suppose that we found the optimal κ using the cross-validation estimate. Replications of training result in K different weight estimates $\hat{\mathbf{w}}_j$ which might be viewed as an ensemble of networks. In [16] we showed under certain mild conditions that when considering a $o(1/N)$ approximation, the average generalization error of the ensemble network $f_{\text{ens}}(\mathbf{x}) = \sum_{j=1}^K \beta_j \cdot f(\mathbf{x}, \hat{\mathbf{w}}_j)$ equals that of the network trained on all examples in \mathcal{D} where β_j weights the contribution from the j 'th network and $\sum_j \beta_j = 1$. If K is a divisor in N then $\forall j, \beta_j = 1/K$, otherwise $\beta_j = (N - N_{vj})/N(K - 1)$. Consequently, one might use the ensemble network to compensate for the increase in generalization error due to only training on $N_{tj} = N - N_{vj}$ data. Alternatively, one might retrain on the full data set \mathcal{D} using the optimal κ . We use the latter approach in the experimental section.

A minimal necessary requirement for a procedure which estimates the network parameters on the training set and optimizes the amount of regularization from a cross-validation set is: the generalization error of the regularized network should be smaller than that of the unregularized network trained on the full data set \mathcal{D} . However, this is not always the case, and is the quintessence of various “no free lunch” theorems [12], [44], [46]:

- If the regularizer is parameterized using many parameters, κ , there is a potential risk of over-fitting on the cross-validation data. A natural way to avoid this situation is to limit the number of regularization parameters. Another recipe is to impose constraints on κ (hyper regularization).
- The specific choice of the regularizers functional form impose prior constraints on the functions to be implemented by the network⁸. If the prior information is mismatched to the actual problem it might be better not to use regularization.
- The de-biasing procedure described above which compensate for training only on $N_{tj} < N$ examples might fail to yield better performance since the

⁷ That is, $[\mathbf{x}(k_1), \mathbf{y}(k_1)]$ is independent of $[\mathbf{x}(k_2), \mathbf{y}(k_2)]$ for all $k_1 \neq k_2$.

⁸ The functional constraints are through the penalty imposed on the weights.

weights now are optimized using all data, including those which were left out exclusively for optimizing regularization parameters.

- The split among training/validation data, and consequently the number of folds, K , may not be chosen appropriately.

These problems are further addressed in Section 5.4.1.

5.3 Adapting Regularization Parameters

The choice of regularizer may be motivated by

- the fact that the minimization of the cost function is normally an ill-posed task. Regularization smoothens the cost function and thereby facilitates the training. The weight decay regularizer⁹, originally suggested by Hinton in the neural networks literature, is a simple way to accomplish this task, see e.g., [35].
- a priori knowledge of the weights, e.g., in terms of a prior distribution (when using a Bayesian approach). In this case the regularization term normally plays the role of a log-prior distribution. Weight decay regularization may be viewed as a Gaussian prior, see e.g., [2]. Other types of priors, e.g., the Laplacian [13], [43] and soft weight sharing [34] has been considered. Moreover, priors have been developed for the purpose of restricting the number of weights (pruning), e.g., the so-called weight elimination [42].
- a desired characteristics of the functional mapping performed by the network. Typically, a smooth mapping is preferred. Regularizers which penalizes curvature of the mapping has been suggested in [4], [7], [32], [45], [10].

In the experimental section we consider weight decay regularization and some generalizations hereof. Without further ado, weight decay regularization has proven to be useful in many neural network applications.

The standard approach for estimation of regularization parameters is more and less systematic search and evaluation of the cross-validation error. However, this is not viable for multiple regularization parameters. On the other hand, as will be demonstrated, it is possible to derive an optimization algorithm based on gradient descent.

Consider a regularization term $R(\mathbf{w}, \boldsymbol{\kappa})$ which depends on q regularization parameters contained in the vector $\boldsymbol{\kappa}$. Since the estimated weights $\hat{\mathbf{w}}_j = \arg \min_{\mathbf{w}} C_{\mathcal{T}_j}(\mathbf{w})$ are controlled by the regularization term, we may in fact consider the cross-validation error (5.4) as an *implicit function* of the regularization parameters, i.e.,

$$\widehat{\Gamma}(\boldsymbol{\kappa}) = \frac{1}{K} \sum_{j=1}^K S_{\mathcal{V}_j}(\hat{\mathbf{w}}_j(\boldsymbol{\kappa})) \quad (5.6)$$

⁹ Also known as ridge regression.

where $\hat{\mathbf{w}}_j(\boldsymbol{\kappa})$ is the $\boldsymbol{\kappa}$ -dependent vector of weights estimated from training set \mathcal{T}_j . The optimal regularization can be found by using gradient descent¹⁰,

$$\boldsymbol{\kappa}_{(n+1)} = \boldsymbol{\kappa}_{(n)} - \eta \frac{\partial \hat{\Gamma}}{\partial \boldsymbol{\kappa}}(\hat{\mathbf{w}}(\boldsymbol{\kappa}_{(n)})) \quad (5.7)$$

where $\eta > 0$ is a step-size (learning rate) and $\boldsymbol{\kappa}_{(n)}$ is the estimate of the regularization parameters in iteration n .

Suppose the regularization term is linear in the regularization parameters,

$$R(\mathbf{w}, \boldsymbol{\kappa}) = \boldsymbol{\kappa}^\top \mathbf{r}(\mathbf{w}) = \sum_{i=1}^q \kappa_i r_i(\mathbf{w}) \quad (5.8)$$

where κ_i are the regularization parameters and $r_i(\mathbf{w})$ the associated regularization functions. Many suggested regularizers are linear in the regularization parameters, this includes the popular weight decay regularization as well as regularizers imposing smooth functions such as the Tikhonov regularizer [4], [2] and the smoothing regularizer for neural networks [32], [45]. However, there exist exceptions such as weight-elimination [42] and soft weight sharing [34]. In this case the presented method needs few modifications.

Using the results of the Appendix, the gradient of the cross-validation error equals

$$\frac{\partial \hat{\Gamma}}{\partial \boldsymbol{\kappa}}(\boldsymbol{\kappa}) = \frac{1}{K} \sum_{j=1}^K \frac{\partial S_{\mathcal{V}_j}}{\partial \boldsymbol{\kappa}}(\hat{\mathbf{w}}_j), \quad (5.9)$$

$$\frac{\partial S_{\mathcal{V}_j}}{\partial \boldsymbol{\kappa}}(\hat{\mathbf{w}}_j) = -\frac{\partial \mathbf{r}}{\partial \mathbf{w}^\top}(\hat{\mathbf{w}}_j) \cdot \mathbf{J}_j^{-1}(\hat{\mathbf{w}}_j) \cdot \frac{\partial S_{\mathcal{V}_j}}{\partial \mathbf{w}}(\hat{\mathbf{w}}_j). \quad (5.10)$$

where $\mathbf{J}_j = \partial^2 C_{\mathcal{T}_j} / \partial \mathbf{w} \partial \mathbf{w}^\top$ is the Hessian of the cost function. As an example, consider the case of weight decay regularization with separate weight decays for two group of weights, e.g., the input-to-hidden and hidden-to output weights, i.e.,

$$R(\mathbf{w}, \boldsymbol{\kappa}) = \kappa^I \cdot |\mathbf{w}^I|^2 + \kappa^H \cdot |\mathbf{w}^H|^2 \quad (5.11)$$

where $\boldsymbol{\kappa} = [\kappa^I, \kappa^H]$, $\mathbf{w} = [\mathbf{w}^I, \mathbf{w}^H]$ with \mathbf{w}^I , \mathbf{w}^H denoting the input-to-hidden and hidden-to output weights, respectively. The gradient of the validation error then yields,

$$\frac{\partial S_{\mathcal{V}_j}}{\partial \kappa^I}(\hat{\mathbf{w}}_j) = -2(\hat{\mathbf{w}}_j^I)^\top \cdot \mathbf{g}_j^I, \quad \frac{\partial S_{\mathcal{V}_j}}{\partial \kappa^H}(\hat{\mathbf{w}}_j) = -2(\hat{\mathbf{w}}_j^H)^\top \cdot \mathbf{g}_j^H \quad (5.12)$$

where \mathbf{g}_j is the vector

$$\mathbf{g}_j = [\mathbf{g}_j^I, \mathbf{g}_j^H] = \mathbf{J}_j^{-1}(\hat{\mathbf{w}}_j) \cdot \frac{\partial S_{\mathcal{V}_j}}{\partial \mathbf{w}}(\hat{\mathbf{w}}_j). \quad (5.13)$$

In summary, the algorithm for adapting regularization parameters consists of the following 8 steps:

¹⁰ We have recently extended this algorithm incorporating second order information via the Conjugate Gradient technique [11].

1. Choose the split ratio; hence, the number of folds, K .
2. Initialize κ and the weights of the network¹¹.
3. Train the K networks with fixed κ on \mathcal{T}_j to achieve $\hat{w}_j(\kappa)$, $j = 1, 2, \dots, K$. Calculate the validation errors $S_{\mathcal{V}_j}$ and the cross-validation estimate $\hat{\Gamma}$.
4. Calculate the gradients $\partial S_{\mathcal{V}_j} / \partial \kappa$ and $\partial \hat{\Gamma} / \partial \kappa$ cf. (5.9) and (5.10). Initialize the step-size η .
5. Update κ using (5.7).
6. Retrain the K networks from the previous weight estimates and recalculate the cross-validation error $\hat{\Gamma}$.
7. If no decrease in cross-validation error then perform a bisection of η and go to step 5; otherwise, continue.
8. Repeat steps 4–7 until the relative change in cross-validation error is below a small percentage or, e.g., the 2-norm of the gradient $\partial \hat{\Gamma} / \partial \kappa$ is below a small number.

Compared to standard neural network training the above algorithm does generally not lead to severe computational overhead. First of all, the standard approach of tuning regularization parameters by, more or less systematic search, requires a lot of training sessions. The additional terms to be computed in the adaptive algorithm are: 1) the derivative of the regularization functions w.r.t. the weights, $\partial r / \partial \mathbf{w}$, 2) the gradient of the validation errors, $\partial S_{\mathcal{V}_j} / \partial \mathbf{w}$, and 3) the inverse Hessians, \mathbf{J}_j^{-1} . The first term is often a simple function of the weights¹² and computationally inexpensive. In the case of feed-forward neural networks, the second term is computed by one pass of the validation examples through a standard back-propagation algorithm. The third term is computationally more expensive. However, if the network is trained using a second order scheme, which requires computation of the inverse Hessian¹³, there is no computational overhead.

The adaptive algorithm requires of the order of $K \cdot itr_{\kappa} \cdot itr_{\eta}$ weight retrainings. Here itr_{κ} is the number of iterations in the gradient descent scheme for κ and itr_{η} is the average number of bisections of η in step 7 of the algorithm. In the experiments carried out the number of retrainings is approx. 100–300 times K . Recall, since we keep on retraining from the current weight estimate, the number of training epochs is generally small.

The number of weight retrainings is somewhat higher than that involved when optimizing the network by using a pruning technique like validation set based Optimal Brain Damage (vOBD) [25], [27]. vOBD based on K -fold cross-validation requires of the order of $K \cdot m$ retrainings, where $m = \dim(\mathbf{w})$. The adaptive regularization algorithm is easily integrated with the pruning algorithm as demonstrated in the experimental section.

¹¹ In Sec. 5.4.1 a practical initialization procedure for κ is described.

¹² For weight decay, it is $2\mathbf{w}$.

¹³ Often the computations are reduced by using a Hessians approximation, e.g., the Gauss–Newton approximation. Many studies have reported significant training speed-up by using second order methods, see e.g., [22], [35].

5.4 Numerical Experiments

5.4.1 Potentials and Limitations in the Approach

The purpose of the section is to demonstrate the potential and limitations of the suggested adaptive regularization framework. We consider the simple linear data generating *system*, viz. estimating the mean of a Gaussian variable,

$$y(k) = w^\circ + \varepsilon(k) \quad (5.14)$$

where w° is the true mean and the noise $\varepsilon(k) \sim \mathcal{N}(0, \sigma_\varepsilon^2)$.

We employ 2-fold cross-validation, i.e., $\mathcal{D} = \mathcal{T}_1 \cup \mathcal{T}_2$, where \mathcal{T}_j , $j = 1, 2$ denote the two training sets in the validation procedure containing approximately half the examples¹⁴. The linear model $y(k) = w + e(k)$ is trained using the mean square cost function augmented by simple weight decay, as shown by

$$C_{\mathcal{T}_j}(w) = \frac{1}{N_{tj}} \sum_{k=1}^{N_{tj}} (y(k) - w)^2 + \kappa \cdot w^2 \quad (5.15)$$

where k runs over examples of the data set in question. The estimated weights are $\hat{w}_j = \bar{y}_j / (1 + \kappa)$ where $\bar{y}_j = N_{tj}^{-1} \sum_{k=1}^{N_{tj}} y(k)$ are the estimated mean. For this simple case, the minimization of the cross-validation error given by,

$$\hat{\Gamma}(\kappa) = \frac{1}{2} \sum_{j=1}^2 S_{\mathcal{V}_j}(\hat{w}_j(\kappa)), \quad S_{\mathcal{V}_j}(\hat{w}_j(\kappa)) = \frac{1}{N_{vj}} \sum_{k=1}^{N_{vj}} (y(k) - \hat{w}_j)^2, \quad (5.16)$$

can be done exactly. The optimal κ is given by

$$\kappa_{\text{opt}} = \frac{\bar{y}_1^2 + \bar{y}_2^2}{2\bar{y}_1\bar{y}_2} - 1. \quad (5.17)$$

Assuming N to be even, the ensemble average of the estimated weights¹⁵, $\hat{w}_j(\kappa_{\text{opt}})$, leads to the final estimate

$$\hat{w}_{\text{reg}} = \frac{1}{2} (\hat{w}_1(\kappa_{\text{opt}}) + \hat{w}_2(\kappa_{\text{opt}})) = \frac{\bar{y}_1\bar{y}_2(\bar{y}_1 + \bar{y}_2)}{\bar{y}_1^2 + \bar{y}_2^2}. \quad (5.18)$$

Notice two properties: First, the estimate is self-consistent as $\lim_{N \rightarrow \infty} \hat{w}_{\text{reg}} = \lim_{N \rightarrow \infty} \hat{w}_{\mathcal{D}} = w^\circ$ where $\hat{w}_{\mathcal{D}} = N^{-1} \sum_{k=1}^N y(k) = (\bar{y}_1 + \bar{y}_2)/2$ is the unregularized estimate trained on all data. Secondly, it is easy to verify that $\bar{y}_j \sim \mathcal{N}(w^\circ, 2\sigma_\varepsilon^2/N)$. That is, if the *normalized true weight* $\theta \equiv w^\circ/A$ where $A = \sqrt{2/N} \cdot \sigma_\varepsilon$ is large then $\bar{y}_j \approx w^\circ$ which means, $\hat{w}_{\text{reg}} \approx \hat{w}_{\mathcal{D}}$.

¹⁴ That is, $N_{t1} = \lfloor N/2 \rfloor$ and $N_{t2} = N - N_{t1}$. Note that these training sets are also the two validation sets, $\mathcal{V}_1 = \mathcal{T}_2$, and vice versa.

¹⁵ The ensemble average corresponds to retraining on all data using κ_{opt} . The weighting of the two estimates is only valid for N even (see Sec. 5.2 for the general case).

The objective is now to test whether using \hat{w}_{reg} results in lower generalization error than employing the unregularized estimate $\hat{w}_{\mathcal{D}}$. The generalization error associated with using the weight w is given by

$$G(w) = \sigma_{\varepsilon}^2 + (w - w^{\circ})^2. \quad (5.19)$$

Further define the generalization error improvement,

$$Z = G(\hat{w}_{\mathcal{D}}) - G(\hat{w}_{\text{reg}}) = (\hat{w}_{\mathcal{D}} - w^{\circ})^2 - (\hat{w}_{\text{reg}} - w^{\circ})^2. \quad (5.20)$$

Note that Z merely is a function of the random variables \bar{y}_1 , \bar{y}_2 and the true weight w° , i.e., it suffices to get samples of \bar{y}_1 , \bar{y}_2 when evaluating properties of Z . Define the normalized variables

$$\tilde{y}_j = \frac{\bar{y}_j}{A} \sim \mathcal{N}\left(\frac{w^{\circ}}{\sigma_{\varepsilon}} \cdot \sqrt{\frac{N}{2}}, 1\right) = \mathcal{N}(\theta, 1). \quad (5.21)$$

It is easily shown that the normalized generalization error improvement Z/A^2 is a function of \tilde{y}_1 , \tilde{y}_2 and θ ; hence, the distribution of Z/A^2 is parameterized solely by θ .

As a quality measure we consider the *probability of improvement* in generalization error given by $\text{Prob}\{Z > 0\}$. Note that $\text{Prob}\{Z > 0\} = 1/2$ corresponds to equal preference of the two estimates. The probability of improvement depends only on the normalized weight θ since $\text{Prob}\{Z > 0\} = \text{Prob}\{Z/A^2 > 0\}$.

Moreover, we consider the *relative generalization error improvement*, defined as

$$\text{RGI} = 100\% \cdot \frac{Z}{G(\hat{w}_{\mathcal{D}})}. \quad (5.22)$$

In particular, we focus on the probability that the relative improvement in generalization is bigger than¹⁶ x , i.e., $\text{Prob}(\text{RGI} > x)$. Optimally $\text{Prob}(\text{RGI} > x)$ should be close to 1 for $x \leq 0\%$ and slowly decaying towards zero for $0\% < x \leq 100\%$. Using the notation $\tilde{w}_{\text{reg}} = \hat{w}_{\text{reg}}/A$, $\tilde{w}_{\mathcal{D}} = \hat{w}_{\mathcal{D}}/A$, RGI can be written as

$$\text{RGI} = 100\% \cdot \frac{(\tilde{w}_{\mathcal{D}} - \theta)^2 - (\tilde{w}_{\text{reg}} - \theta)^2}{N/2 + (\tilde{w}_{\mathcal{D}} - \theta)^2}. \quad (5.23)$$

Thus, the distribution of RGI is parameterized by θ and N .

The quality measures are computed by generating Q independent realizations of \tilde{y}_1 , \tilde{y}_2 , i.e., $\{\tilde{y}_1^{(i)}, \tilde{y}_2^{(i)}\}_{i=1}^Q$. E.g., the probability of improvement is estimated by $P_{\text{imp}} = Q^{-1} \sum_{i=1}^Q \mu(Z^{(i)})$ where $\mu(x) = 1$ for $x > 0$, and zero otherwise.

The numerical results of comparing \hat{w}_{reg} to the unregularized estimate $\hat{w}_{\mathcal{D}}$ is summarized in Fig. 5.1.

¹⁶ Note that, $\text{Prob}(\text{RGI} > 0) = \text{Prob}(Z > 0)$.

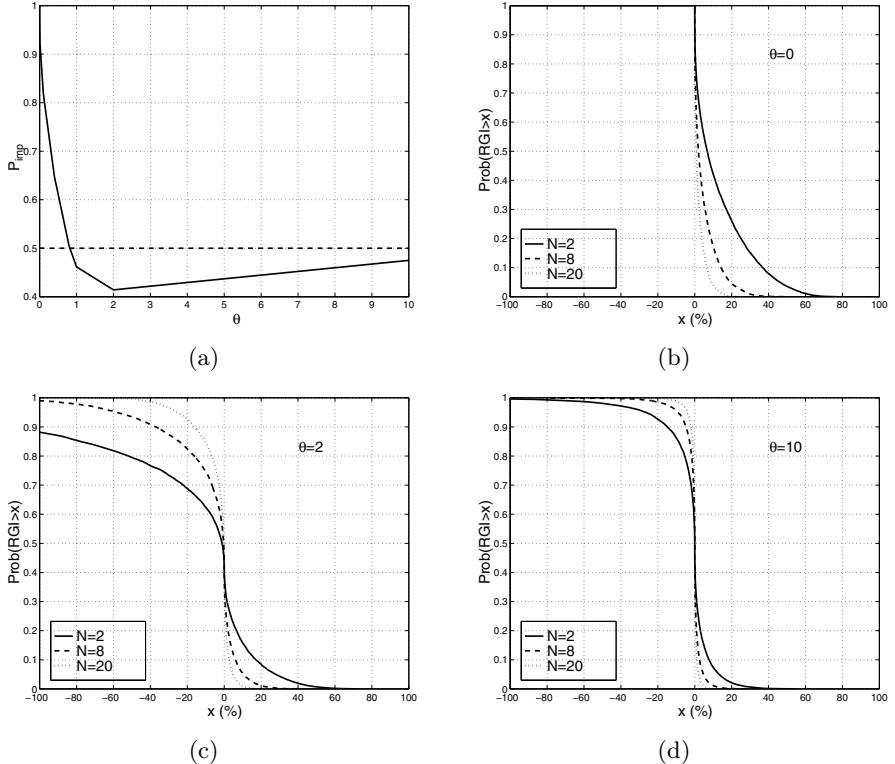


Fig. 5.1. Result of comparing the optimally regularized estimate \widehat{w}_{reg} of the mean of a Gaussian variable to the unregularized estimate $\widehat{w}_{\mathcal{D}}$. The results are based on $Q = 10^5$ independent realizations. The probability of improvement P_{imp} , shown in panel (a), is one for when the normalized true weight $\theta = \sqrt{N/2} \cdot w^\circ / \sigma_\varepsilon = 0$, and above 0.5 for $\theta \lesssim 0.8$. That is, when the prior information of the weight decay regularizer is correct (true weight close to zero), when N is small or when σ_ε is large. As θ becomes large P_{imp} tends to 0.5 due to the fact that $\tilde{w} \approx \widehat{w}_{\mathcal{D}}$. Panel (b)–(d) display $\text{Prob}(\text{RGI} > x)$ for $\theta \in \{0, 2, 10\}$. The ideal probability curve is 1 for $x < 0$ and a slow decay towards zero for $x > 0$. The largest improvement is attained for small θ and small N . Panel (c) and (d) indicate that small N gives the largest probability for $x > 0$; however, also the smallest probability for negative x . That is, a higher chance of getting a good improvement also increases the chance of deterioration. Notice, even though $P_{\text{imp}} < 0.5$ for $\theta = 2, 10$ there is still a reasonable probability of getting a significant improvement.

5.4.2 Classification

We test the performance of the adaptive regularization algorithm on a vowel classification problem. The data are based on the Peterson and Barney database [36]. The classes are vowel sounds characterized by the first four formant frequencies. 76 persons (33 male, 28 female and 15 children) have pronounced $c = 10$ different vowels (IY IH EH AE AH AA AO UH UW ER) two times. This results

in a data base of totally 1520 examples. The database is the verified database described in [41] where all data¹⁷ are used, including examples where utterance failed of unanimous identification in the listening test (26 listeners). All examples were included to make the task more difficult.

The regularization was adapted using a hold-out validation error estimator, thus the examples were split into a data set, \mathcal{D} , consisting of $N = 760$ examples (16 male, 14 female and 8 children) and an independent test set of the remaining 760 examples. The regularization was adapted by splitting the data set \mathcal{D} equally into a validation set of $N_v = 380$ examples and a training set of $N_t = 380$ examples (8 male, 7 female and 4 children in each set).

We used a feed-forward 2-layer neural network with hyperbolic tangent neurons in the hidden layer and modified SoftMax normalized outputs, \hat{y}_i , see e.g., [2], [18], [3]. Thus, the outputs estimates the posterior class probabilities $p(\mathcal{C}_i|\mathbf{x})$, where \mathcal{C}_i denotes the i 'th class, $i = 1, 2, \dots, c$. Bayes rule (see e.g., [2]) is used to assign \mathcal{C}_i to input \mathbf{x} if $i = \arg \max_j p(\mathcal{C}_j|\mathbf{x})$. Suppose that the network weights are given by $\mathbf{w} = [\mathbf{w}^I, \mathbf{w}_{\text{bias}}^I, \mathbf{w}^H, \mathbf{w}_{\text{bias}}^H]$ where $\mathbf{w}^I, \mathbf{w}^H$ are input-to-hidden and hidden-to-output weights, respectively, and the bias weights are assembled in $\mathbf{w}_{\text{bias}}^I$ and $\mathbf{w}_{\text{bias}}^H$. Suppose that the targets $y_i(k) = 1$ if $\mathbf{x}(k) \in \mathcal{C}_i$, and zero otherwise. The network is optimized using a log-likelihood loss function augmented by a weight decay regularizer using 4 regularization parameters,

$$C(\mathbf{w}) = \frac{1}{N_t} \sum_{k=1}^{N_t} \sum_{i=1}^c y_i(k) \log(\hat{y}_i(k, \mathbf{w})) + \kappa^I \cdot |\mathbf{w}^I|^2 + \kappa_{\text{bias}}^I \cdot |\mathbf{w}_{\text{bias}}^I|^2 + \kappa^H \cdot |\mathbf{w}^H|^2 + \kappa_{\text{bias}}^H \cdot |\mathbf{w}_{\text{bias}}^H|^2. \quad (5.24)$$

We further define unnormalized weight decays as $\alpha \equiv \kappa \cdot N_t$. This regularizer is motivated by the fact that the bias, input and hidden layer weights play a different role, e.g., the input, hidden and bias signals normally have different scale (see also [2, Ch. 9.2]).

The simulation set-up is:

- Network: 4 inputs, 5 hidden neurons, 9 outputs¹⁸.
- Weights are initialized uniformly over $[-0.5, 0.5]$, regularization parameters are initialized at zero. One step in a gradient descent training algorithm (see e.g., [29]) is performed and the weight decays are re-initialized at $\lambda_{\text{max}}/10^2$, where λ_{max} is the max. eigenvalue of the Hessian matrix of the cost function. This initialization scheme is motivated by the following observations:
 - Weight decays should be so small that they do not reduce the approximation capabilities of the network significantly.
 - They should be so large that the algorithm is prevented from being trapped in a local optimum and numerical instabilities are eliminated.

¹⁷ The database can be retrieved from <ftp://eivind.imm.dtu.dk/dist/data/vowel/PetersonBarney.tar.Z>

¹⁸ We only need 9 outputs since the posterior class probability of the 10th class is given by $1 - \sum_{j=1}^9 p(\mathcal{C}_j|\mathbf{x})$.

- Training is now done using a Gauss-Newton algorithm (see e.g., [29]). The Hessian is inverted using the Moore-Penrose pseudo inverse ensuring that the eigenvalue spread¹⁹ is less than 10^8 .
- The regularization step-size η is initialized at 1.
- When the adaptive regularization scheme has terminated 3% of the weights are pruned using a validation set based version of the Optimal Brain Damage (vOBD) recipe [25], [27].
- Alternation between pruning and adaptive regularization continues until the validation error has reached a minimum.
- Finally, remaining weights are retrained on all data using the optimized weight decay parameters.

Table 5.1. Probability of misclassification (pmc) and log-likelihood cost function (without reg. term, see (5.24)) for the classification example. The neural network averages and standard deviations are computed from 10 runs. In the case of small fixed regularization, weight decays were set at initial values equal to $\lambda_{\max}/10^6$ where λ_{\max} is the largest eigenvalue of the Hessian matrix of the cost function. Optimal regularization refers to the case of optimizing 4 weight decay parameters. Pruning refers to validation set based OBD. KNN refers to k -nearest-neighbor classification.

Probability of Misclassification (pmc)

	NN small fixed reg.	NN opt. reg.+prun.	KNN ($k = 9$)
Training Set	0.075 ± 0.026	0.107 ± 0.008	0.150
Validation Set	0.143 ± 0.014	0.115 ± 0.004	0.158
Test Set	0.146 ± 0.010	0.124 ± 0.006	0.199
Test Set (train. on all data)	0.126 ± 0.010	0.119 ± 0.004	0.153

Log-likelihood Cost Function

	NN small fixed reg.	NN opt. reg.+prun.
Training Set	0.2002 ± 0.0600	0.2881 ± 0.0134
Validation Set	0.7016 ± 0.2330	0.3810 ± 0.0131
Test Set	0.6687 ± 0.2030	0.3773 ± 0.0143
Test Set (train. on all data)	0.4426 ± 0.0328	0.3518 ± 0.0096

Table 5.1 reports the average and standard deviations of the probability of misclassification (pmc) and log-likelihood cost function over 10 runs for pruned networks using the optimal regularization parameters. Note that retraining on

¹⁹ Eigenvalue spread should not be larger than the square root of the machine precision [6].

the full data set decreases the test pmc slightly on the average. In fact, improvement was noticed in 9 out of 10 runs. The table further shows the gain of the combined adaptive regularization/pruning algorithm relative to using a small fixed weight decay. However, recall, cf. Sec. 5.4.1, that the actual gain is *very* dependent on the noise level, data set size, etc. The objective is not to demonstrate high gain for a specific problem, rather to demonstrate that algorithm runs fairly robust in a classification set-up. For comparison we used a k -nearest-neighbor (KNN) classification (see e.g., [2]) and found that $k = 9$ neighbors was optimal by minimizing pmc on the validation set. The neural network performed significantly better. Contrasting the obtained results to other work is difficult. In [37] results on the Peterson-Barney vowel problem are reported, but their data are not exactly the same; only the first 2 formant frequencies were used. Furthermore, different test sets have been used for the different methods presented. The best result reported [28] is obtained by using KNN and reach $pmc = 0.186$ which is significantly higher than our results.

Fig. 5.2 shows the evolution of the adaptive regularization as well as the pruning algorithm.

5.4.3 Time Series Prediction

We tested the performance of the adaptive regularization schemes on the Mackey-Glass chaotic time series prediction problem, see e.g., [22], [40]. The goal is to predict the series 100 steps ahead based on previous observations. The feed-forward net configuration is an input lag-space $\mathbf{x}(k) = [x(k), x(k-6), x(k-12), x(k-18)]$ of 4 inputs, 25 hidden hyperbolic tangent neurons, and a single linear output unit $\hat{y}(k)$ which predicts $y(k) = x(k+100)$. The cost function is the squared error, $N_t^{-1} \sum_{k=1}^{N_t} (y(k) - \hat{y}(k, \mathbf{w}))^2$, augmented by a weight decay regularizer using 4 different weight decays as described in Section 5.4.2.

The simulation set-up is:

- The data set, \mathcal{D} , has $N = 500$ examples and an independent test has 8500 examples.
- The regularization parameters are optimized using a hold-out validation error with an even split²⁰ of the data set into training and validation sets each having 250 examples.
- Weight decays are initialized at zero and one Gauss-Newton iteration is performed, then weight decays were re-initialized at $\lambda_{\max}/10^6$, where λ_{\max} is the max. eigenvalue of the Hessian matrix of the cost function.
- The network is trained using a Gauss-Newton training scheme. The Hessian is inverted using the Moore-Penrose pseudo inverse ensuring that the eigenvalue spread is less than 10^8 .
- The regularization step-size η is initialized at 10^{-2} .

²⁰ The sensitivity to different splits are considered in [25].

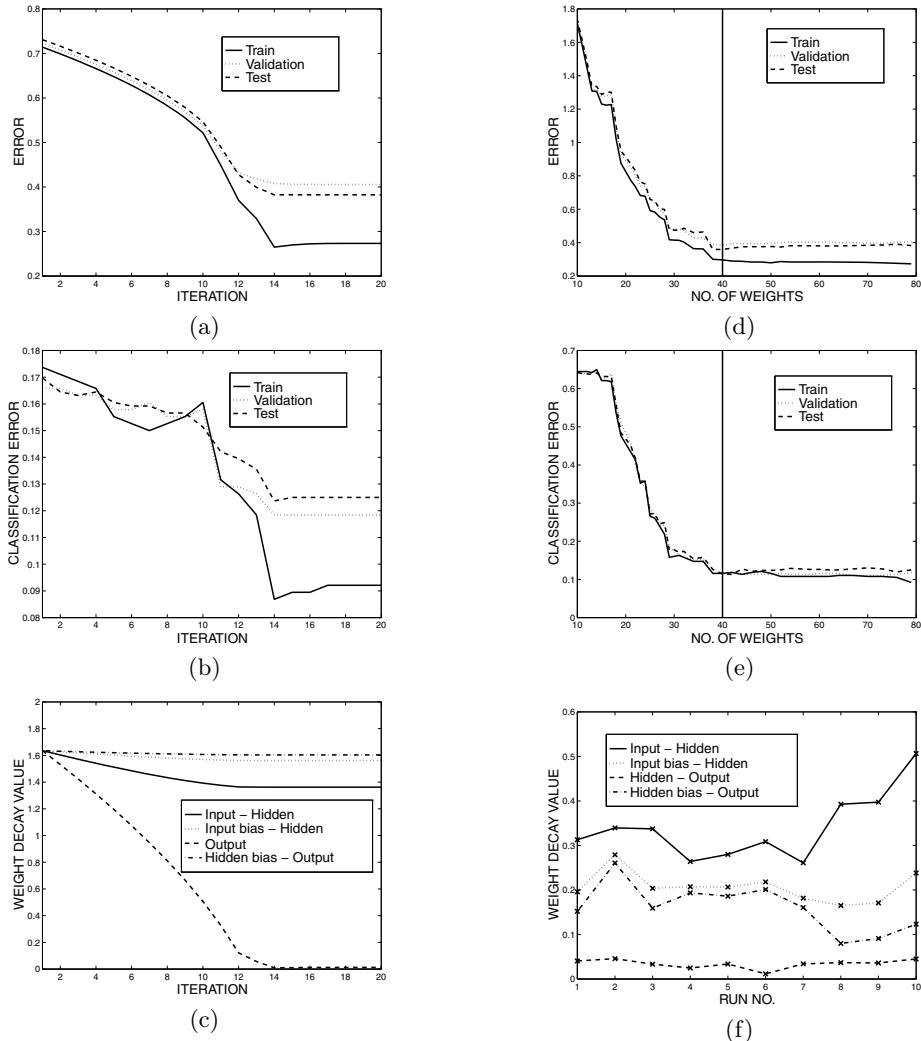


Fig. 5.2. Classification example. Panels (a), (b) and (c) show the evolution of the adaptive regularization algorithm in a typical run (fully connected network). The weight decays are optimized aiming at minimizing the validation error in panel (a). Note that also the test error decreases. This tendency is also evident in panel (b) displaying *pmc* even though a small increase noticed. In panel (c) the convergence unnormalized weight decays, $\alpha = \kappa \cdot N_t$, are depicted. Panels (d) and (e) show the evolution of errors and *pmc* during the pruning session. The optimal network is chosen as the one with minimal validation error, as indicated by the vertical line. There is only a marginal effect of pruning in this run. Finally, in panel (f), the variation of the optimal (end of pruning) α 's in different runs is demonstrated. A clear similarity over runs is noticed.

Table 5.2. Normalized squared error performance for the time series prediction examples. All figures are in units of $10^{-3}\hat{\sigma}_x^2$ and averages and standard deviations are computed from 10 runs. In the case of small fixed regularization, weight decays were set at initial values equal to $\lambda_{\max}/10^6$ where λ_{\max} is the largest eigenvalue of the Hessian matrix of the cost function. Optimal regularization refers to the case of optimizing 4 weight decay parameters. Pruning refers to validation set based OBD.

	NN small fixed reg.	NN small fixed reg.+prun.	NN opt. reg.+prun.
Training Set	0.17 ± 0.07	0.12 ± 0.04	0.10 ± 0.07
Validation Set	0.53 ± 0.26	0.36 ± 0.07	0.28 ± 0.14
Test Set	1.91 ± 0.68	1.58 ± 0.21	1.29 ± 0.46
Test Set (train. on all data)	1.33 ± 0.43	1.34 ± 0.26	1.17 ± 0.48

- Alternation between adapting the 4 weight decays and validation set based pruning [25].
- The pruned network is retrained on all data using the optimized weight decay parameters.

Table 5.2 reports the average and standard deviations of the normalized squared error (i.e., the squared error normalized with the estimated variance of $x(k)$, denoted $\hat{\sigma}_x^2$) over 10 runs for optimal regularization parameters. Retraining on the full data set decreases the test error somewhat on the average. Improvement was noticed in 10 out of 10 runs. We tested 3 different cases: small fixed regularization, small fixed regularization assisted by pruning and combined adaptive regularization/pruning. It turns that pruning alone does not improve performance; however, supplementing by adaptive regularization gives a test error reduction.

We furthermore tried a flexible regularization scheme, viz. individual weight decay where $R(\mathbf{w}, \kappa) = \sum_{i=1}^m \kappa_i w_i^2$ and $\kappa_i \geq 0$ are imposed. In the present case it turned out that the flexible regularizer was not able to outperform the joint adaptive regularization/pruning scheme; possibly due to training and validation set sizes.

Fig. 5.3 demonstrates adaptive regularization and pruning in a typical case using 4 weight decays.

5.5 Conclusions

In this paper it was suggested to adapt regularization parameters by minimizing the cross-validation error or a simple hold-out validation error. We derived a simple gradient descent scheme for optimizing regularization parameters which has a small programming overhead and an acceptable computational overhead compared to standard training. Numerical examples with a toy linear model showed limitations and advantages of the adaptive regularization approach. Moreover, numerical experiments on classification and time series prediction problems successfully demonstrated the functionality of the algorithm. Adaptation of regularization parameters resulted in lower generalization error; however, it should

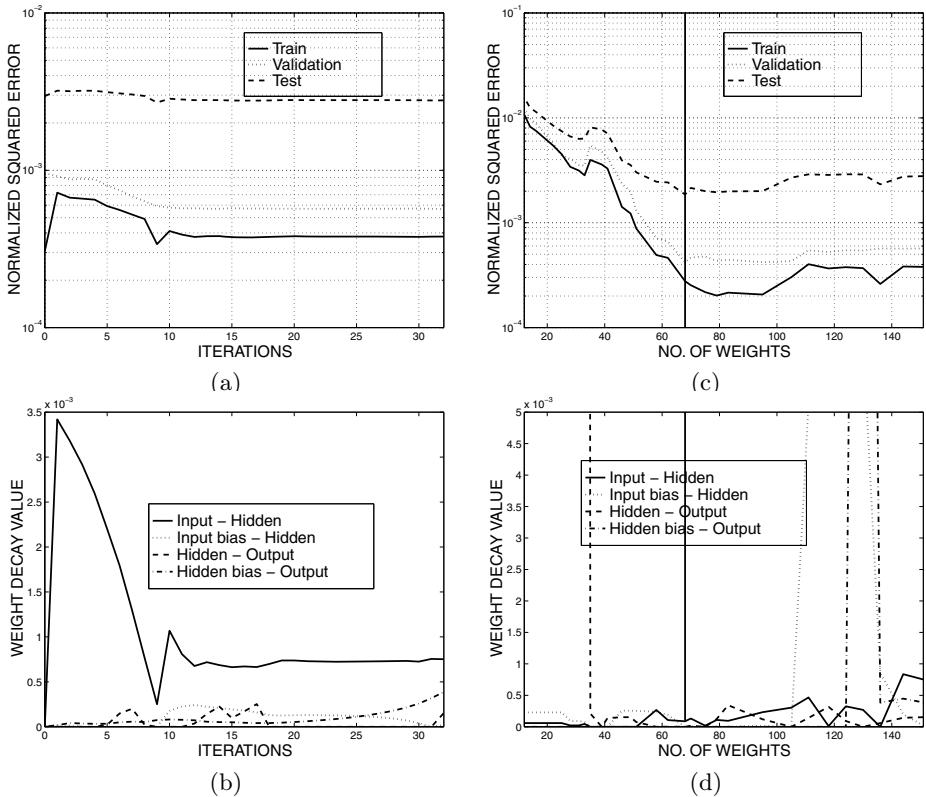


Fig. 5.3. Time series prediction example. Panels (a) and (b) show a typical evolution of errors and unnormalized weight decay values α when running the adaptive regularization algorithm using 4 weight decays. The normalized validation error drops approx. a factor of 2 when adapting weight decays. It turns out that some regularization of the input-to-hidden and output bias weights are needed whereas the other weights essentially requires no regularization²². In panel (c) and (d) it is demonstrated that pruning reduces the test error slightly. The optimal network is chosen as the one with minimal validation error, as indicated by the vertical line.

be emphasized that the actual yield is very dependent on the problem and the choice of the regularizers functional form.

Acknowledgments. This research was supported by the Danish Natural Science and Technical Research Councils through the Computational Neural Network Center. JL furthermore acknowledge the Radio Parts Foundation for financial support. Mads Hintz-Madsen and Morten With Pedersen are acknowledged for stimulating discussions.

²² Recall that if a weight decay κ is below $\lambda_{\max}/10^8$ it does not influence the Moore-Penrose pseudo inversion of the Hessian.

Appendix

Assume that the regularization term is linear in the regularization parameters, i.e.,

$$R(\mathbf{w}, \boldsymbol{\kappa}) = \boldsymbol{\kappa}^\top \mathbf{r}(\mathbf{w}) = \sum_{i=1}^q \kappa_i r_i(\mathbf{w}) \quad (5.25)$$

The gradient of the cross-validation error (5.4) is

$$\frac{\partial \hat{\Gamma}}{\partial \boldsymbol{\kappa}}(\boldsymbol{\kappa}) = \frac{1}{K} \sum_{j=1}^K \frac{\partial S_{\mathcal{V}_j}}{\partial \boldsymbol{\kappa}}(\hat{\mathbf{w}}_j(\boldsymbol{\kappa})) \quad (5.26)$$

Using the chain rule the gradient vector of the validation error, $S_{\mathcal{V}_j}$, can be written as

$$\frac{\partial S_{\mathcal{V}_j}}{\partial \boldsymbol{\kappa}}(\hat{\mathbf{w}}_j(\boldsymbol{\kappa})) = \frac{\partial \mathbf{w}^\top}{\partial \boldsymbol{\kappa}}(\hat{\mathbf{w}}_j(\boldsymbol{\kappa})) \cdot \frac{\partial S_{\mathcal{V}_j}}{\partial \mathbf{w}}(\hat{\mathbf{w}}_j(\boldsymbol{\kappa})) \quad (5.27)$$

where $\partial \mathbf{w}^\top / \partial \boldsymbol{\kappa}$ is the $q \times m$ derivative matrix of the estimated weights w.r.t. the regularization parameters and $m = \dim(\mathbf{w})$. In order to find this derivative matrix, consider the gradient of the cost function w.r.t. to the weights as a function of $\boldsymbol{\kappa}$ and use the following expansion around the current estimate $\boldsymbol{\kappa}_{(n)}$,

$$\frac{\partial C_{\mathcal{T}_j}}{\partial \mathbf{w}}(\boldsymbol{\kappa}) = \frac{\partial C_{\mathcal{T}_j}}{\partial \mathbf{w}}(\boldsymbol{\kappa}_{(n)}) + \frac{\partial^2 C_{\mathcal{T}_j}}{\partial \mathbf{w} \partial \boldsymbol{\kappa}^\top}(\boldsymbol{\kappa}_{(n)}) \cdot (\boldsymbol{\kappa} - \boldsymbol{\kappa}_{(n)}) + o(|\boldsymbol{\kappa} - \boldsymbol{\kappa}_{(n)}|). \quad (5.28)$$

Requiring $\hat{\mathbf{w}}(\boldsymbol{\kappa}_{(n+1)})$ in the next iteration to be an optimal weight vector, i.e., $\partial C_{\mathcal{T}_j} / \partial \mathbf{w}(\boldsymbol{\kappa}_{(n+1)}) = \mathbf{0}$ implies

$$\frac{\partial^2 C_{\mathcal{T}_j}}{\partial \mathbf{w} \partial \boldsymbol{\kappa}^\top}(\hat{\mathbf{w}}(\boldsymbol{\kappa}_{(n)})) = \mathbf{0}. \quad (5.29)$$

Recall that $\partial C_{\mathcal{T}_j} / \partial \mathbf{w}(\boldsymbol{\kappa}_{(n)}) = \mathbf{0}$ by assumption. (5.29) can be used for determining $\partial \mathbf{w}^\top / \partial \boldsymbol{\kappa}$. Recognizing that the cost function $C_{\mathcal{T}_j}(\hat{\mathbf{w}}(\boldsymbol{\kappa})) = S_{\mathcal{T}_j}(\hat{\mathbf{w}}(\boldsymbol{\kappa})) + R(\hat{\mathbf{w}}(\boldsymbol{\kappa}), \boldsymbol{\kappa})$ depends *implicitly* (thorough $\hat{\mathbf{w}}(\boldsymbol{\kappa})$) and *explicitly* on $\boldsymbol{\kappa}$ it is possible, by using (5.25), to derive the following relation²³:

$$\frac{\partial \mathbf{w}^\top}{\partial \boldsymbol{\kappa}}(\hat{\mathbf{w}}_j) = -\frac{\partial \mathbf{r}}{\partial \mathbf{w}^\top}(\hat{\mathbf{w}}_j) \cdot \mathbf{J}_j^{-1}(\hat{\mathbf{w}}_j) \quad (5.30)$$

where $\mathbf{J}_j = \partial^2 C_{\mathcal{T}_j} / \partial \mathbf{w} \partial \mathbf{w}^\top$ is the Hessian of the cost function which e.g., might be evaluated using the Gauss-Newton approximation [29]. Finally, substituting (5.30) into (5.27) gives

$$\frac{\partial S_{\mathcal{V}_j}}{\partial \boldsymbol{\kappa}}(\hat{\mathbf{w}}_j) = -\frac{\partial \mathbf{r}}{\partial \mathbf{w}^\top}(\hat{\mathbf{w}}_j) \cdot \mathbf{J}_j^{-1}(\hat{\mathbf{w}}_j) \cdot \frac{\partial S_{\mathcal{V}_j}}{\partial \mathbf{w}}(\hat{\mathbf{w}}_j) \quad (5.31)$$

$\partial S_{\mathcal{V}_j} / \partial \mathbf{w}$ is found by ordinary back-propagation on the validation set while $\partial \mathbf{r} / \partial \mathbf{w}^\top$ is calculated from the specific assumptions on the regularizer.

²³ For convenience, here $\hat{\mathbf{w}}$'s explicit $\boldsymbol{\kappa}$ -dependence is omitted.

References

- [1] Akaike, H.: Fitting Autoregressive Models for Prediction. *Annals of the Institute of Statistical Mathematics* 21, 243–247 (1969)
- [2] Amari, S., Murata, N., Müller, K.R., Finke, M., Yang, H.: Asymptotic Statistical Theory of Overtraining and Cross-Validation. Technical report METR 95-06 and *IEEE Transactions on Neural Networks* 8(5), 985–996 (1995)
- [3] Nonboe Andersen, L., Larsen, J., Hansen, L.K., Hintz-madsen, M.: Adaptive Regularization of Neural Classifiers. In: Principe, J., et al. (eds.) *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing VII*, pp. 24–33. IEEE, Piscataway (1997)
- [4] Bishop, C.M.: Curvature-Driven Smoothing: A Learning Algorithm for Feedforward Neural Networks. *IEEE Transactions on Neural Networks* 4(4), 882–884 (1993)
- [5] Bishop, C.M.: *Neural Networks for Pattern Recognition*. Oxford University Press, Oxford (1995)
- [6] Dennis, J.E., Schnabel, R.B.: *Numerical Methods for Unconstrained Optimization and Non-linear Equations*. Prentice-Hall, Englewood Cliffs (1983)
- [7] Drucker, H., Le Cun, Y.: Improving Generalization Performance in Character Recognition. In: Juang, B.H., et al. (eds.) *Neural Networks for Signal Processing: Proceedings of the 1991 IEEE-SP Workshop*, pp. 198–207. IEEE, Piscataway (1991)
- [8] Geisser, S.: The Predictive Sample Reuse Method with Applications. *Journal of the American Statistical Association* 50, 320–328 (1975)
- [9] Geman, S., Bienenstock, E., Doursat, R.: Neural Networks and the Bias/Variance Dilemma. *Neural Computation* 4, 1–58 (1992)
- [10] Girosi, F., Jones, M., Poggio, T.: Regularization Theory and Neural Networks Architectures. *Neural Computation* 7(2), 219–269 (1995)
- [11] Goutte, C., Larsen, J.: Adaptive Regularization of Neural Networks using Conjugate Gradient. In: *Proceedings of ICASSP 1998*, Seattle, USA, vol. 2, pp. 1201–1204 (1998)
- [12] Goutte, C.: Note on Free Lunches and Cross-Validation. *Neural Computation* 9(6), 1211–1215 (1997)
- [13] Goutte, C.: Regularization with a Pruning Prior. *Neural Networks* (1997) (to appear)
- [14] Hansen, L.K., Rasmussen, C.E.: Pruning from Adaptive Regularization. *Neural Computation* 6, 1223–1232 (1994)
- [15] Hansen, L.K., Rasmussen, C.E., Svarer, C., Larsen, J.: Adaptive Regularization. In: Vlontzos, J., Hwang, J.-N., Wilson, E. (eds.) *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing IV*, pp. 78–87. IEEE, Piscataway (1994)
- [16] Hansen, L.K., Larsen, J.: Linear Unlearning for Cross-Validation. *Advances in Computational Mathematics* 5, 269–280 (1996)
- [17] Hertz, J., Krogh, A., Palmer, R.G.: *Introduction to the Theory of Neural Computation*. Addison-Wesley Publishing Company, Redwood City (1991)
- [18] Hintz-Madsen, M., With Pedersen, M., Hansen, L.K., Larsen, J.: Design and Evaluation of Neural Classifiers. In: Usui, S., Tohkura, Y., Katagiri, S., Wilson, E. (eds.) *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing VI*, pp. 223–232. IEEE, Piscataway (1996)

- [19] Hornik, K.: Approximation Capabilities of Multilayer Feedforward Networks. *Neural Networks* 4, 251–257 (1991)
- [20] Kearns, M.: A Bound on the Error of Cross Validation Using the Approximation and Estimation Rates, with Consequences for the Training-Test Split. *Neural Computation* 9(5), 1143–1161 (1997)
- [21] Larsen, J.: A Generalization Error Estimate for Nonlinear Systems. In: Kung, S.Y., et al. (eds.) *Proceedings of the 1992 IEEE-SP Workshop on Neural Networks for Signal Processing*, vol. 2, pp. 29–38. IEEE, Piscataway (1992)
- [22] Larsen, J.: Design of Neural Network Filters, Ph.D. Thesis, Electronics Institute, Technical University of Denmark (1993),
ftp://eivind.imm.dtu.dk/dist/PhD_thesis/jlarsen.thesis.ps.Z
- [23] Larsen, J., Hansen, L.K.: Generalization Performance of Regularized Neural Network Models. In: Vlontzos, J., et al. (eds.) *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing IV*, pp. 42–51. IEEE, Piscataway (1994)
- [24] Larsen, J., Hansen, L.K.: Empirical Generalization Assessment of Neural Network Models. In: Girosi, F., et al. (eds.) *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing V*, pp. 30–39. IEEE, Piscataway (1995)
- [25] Larsen, J., Hansen, L.K., Svarer, C., Ohlsson, M.: Design and Regularization of Neural Networks: The Optimal Use of a Validation Set. In: Usui, S., Tohkura, Y., Katagiri, S., Wilson, E. (eds.) *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing VI*, pp. 62–71. IEEE, Piscataway (1996)
- [26] Larsen, J., et al.: Optimal Data Set Split Ratio for Empirical Generalization Error Estimates (in preparation)
- [27] Le Cun, Y., Denker, J.S., Solla, S.A.: Optimal Brain Damage. In: Touretzky, D.S. (ed.) *Proceedings of the 1989 Conference on Advances in Neural Information Processing Systemsshers*, vol. 2, pp. 598–605. Morgan Kaufmann Publishers, San Mateo (1990)
- [28] Lowe, D.: Adaptive Radial Basis Function Nonlinearities and the Problem of Generalisation. In: Proc. IEE Conf. on Artificial Neural Networks, pp. 171–175 (1989)
- [29] Ljung, L.: *System Identification: Theory for the User*. Prentice-Hall, Englewood Cliffs (1987)
- [30] MacKay, D.J.C.: A Practical Bayesian Framework for Backprop Networks. *Neural Computation* 4(3), 448–472 (1992)
- [31] Moody, J.: Prediction Risk and Architecture Selection for Neural Networks. In: Cherkassky, V., et al. (eds.) *From Statistics to Neural Networks: Theory and Pattern Recognition Applications*, vol. 136. Springer-Verlag Series F, Berlin (1994)
- [32] Moody, J., Rögnvaldsson, T.: Smoothing Regularizers for Projective Basis Function Networks. In: *Proceedings of the 1996 Conference on Advances in Neural Information Processing Systems*, vol. 9. MIT Press, Cambridge (1997)
- [33] Murata, N., Yoshizawa, S., Amari, S.: Network Information Criterion — Determining the Number of Hidden Units for an Artificial Neural Network Model. *IEEE Transactions on Neural Networks* 5(6), 865–872 (1994)
- [34] Nowlan, S., Hinton, G.: Simplifying Neural Networks by Soft Weight Sharing. *Neural Computation* 4(4), 473–493 (1992)
- [35] With Pedersen, M.: Training Recurrent Networks. In: *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing VII*. IEEE, Piscataway (1997)
- [36] Peterson, G.E., Barney, H.L.: Control Methods Used in a Study of the Vowels. *JASA* 24, 175–184 (1952)
- [37] Shadafan, R.S., Niranjan, M.: A Dynamic Neural Network Architecture by Sequential Partitioning of the Input Space. *Neural Computation* 6(6), 1202–1222 (1994)

- [38] Sjöberg, J.: Non-Linear System Identification with Neural Networks, Ph.D. Thesis no. 381, Department of Electrical Engineering, Linköping University, Sweden (1995)
- [39] Stone, M.: Cross-validatory Choice and Assessment of Statistical Predictors. *Journal of the Royal Statistical Society B* 36(2), 111–147 (1974)
- [40] Svarer, C., Hansen, L.K., Larsen, J., Rasmussen, C.E.: Designer Networks for Time Series Processing. In: Kamm, C.A., et al. (eds.) *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing*, vol. 3, pp. 78–87. IEEE, Piscataway (1993)
- [41] Watrous, R.L.: Current Status of PetersonBarney Vowel Formant Data. *JASA* 89, 2459–2460 (1991)
- [42] Weigend, A.S., Huberman, B.A., Rumelhart, D.E.: Predicting the Future: A Connectionist Approach. *International Journal of Neural Systems* 1(3), 193–209 (1990)
- [43] Williams, P.M.: Bayesian Regularization and Pruning using a Laplace Prior. *Neural Computation* 7(1), 117–143 (1995)
- [44] Wolpert, D.H., Macready, W.G.: The Mathematics of Search. Technical Report SFI-TR-95-02-010, Santa Fe Instute (1995)
- [45] Wu, L., Moody, J.: A Smoothing Regularizer for Feedforward and Recurrent Neural Networks. *Neural Computation* 8(3) (1996)
- [46] Zhu, H., Rohwer, R.: No Free Lunch for Cross Validation. *Neural Computation* 8(7), 1421–1426 (1996)

6

Large Ensemble Averaging^{*}

David Horn¹, Ury Naftaly¹, and Nathan Intrator²

¹ School of Physics and Astronomy

² School of Mathematical Sciences

Raymond and Beverly Sackler Faculty of Exact Sciences

Tel Aviv University, Tel Aviv 69978, Israel

horn@neuron.tau.ac.il

<http://neuron.tau.ac.il/~horn/>

Abstract. Averaging over many predictors leads to a reduction of the variance portion of the error. We present a method for evaluating the mean squared error of an infinite ensemble of predictors from finite (small size) ensemble information. We demonstrate it on ensembles of networks with different initial choices of synaptic weights. We find that the optimal stopping criterion for large ensembles occurs later in training time than for single networks. We test our method on the sunspots data set and obtain excellent results.

6.1 Introduction

Ensemble averaging has been proposed in the literature as a means to improve the generalization properties of a neural network predictor[3, 11, 7]. We follow this line of thought and consider averaging over a set of networks that differ from one another just by the initial values of their synaptic weights.

We introduce a method to extract the performance of large ensembles from that of finite size ones. This is explained in the next section, and is demonstrated on the sunspots data set. Ensemble averaging over the initial conditions of the neural networks leads to a lower prediction error, which is obtained for a later training time than that expected from single networks. Our method outperforms the best published results for the sunspots problem [6].

The theoretical setting of the method is provided by the bias/variance decomposition. Within this framework, we define a particular bias/variance decomposition for networks differing by their initial conditions only. While the bias of the ensemble of networks with different initial conditions remains unchanged, the variance error decreases considerably.

6.2 Extrapolation to Large-Ensemble Averages

The training procedure of neural networks starts out with some choice of initial values of the connection weights. We consider ensembles of networks that differ

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

from one another just by their initial values and average over them. Since the space of initial conditions is very large we develop a technique which allows us to approximate averaging over the whole space.

Our technique consists of constructing groups of a fixed number of networks, Q . All networks differ from one another by the random choice of their initial weights. For each group we define our predictor to be the average of the output of all Q networks. Choosing several different groups of the same size Q , and averaging over their predictions for the test set, defines the finite size average that is displayed in Fig. 1. Then we perform a parametric estimate of the limit $Q \rightarrow \infty$. A simple regression in $1/Q$ suffices to obtain this limit in the sunspots problem, as shown in Fig. 2. In general one may encounter a more complicated inverse power behavior, indicating correlations between networks with different initial weights.

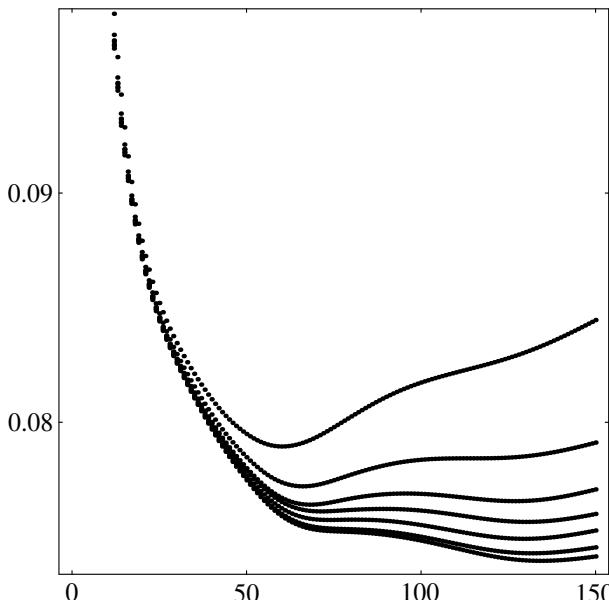


Fig. 6.1. ARV of test set

Prediction error (ARV) is plotted *vs.* training time in kilo epochs (KE). The curves correspond to different choices of group sizes: $Q = 1, 2, 4, 5, 10, 20$ from top to bottom. The lowest curve is the extrapolation to $Q \rightarrow \infty$.

6.2.1 Application to the Sunspots Problem

Yearly sunspot statistics have been gathered since 1700. These data have been extensively studied and have served as a benchmark in the statistical literature [9, 10, 4]. Following previous publications [10, 6, 8] we choose the training set to

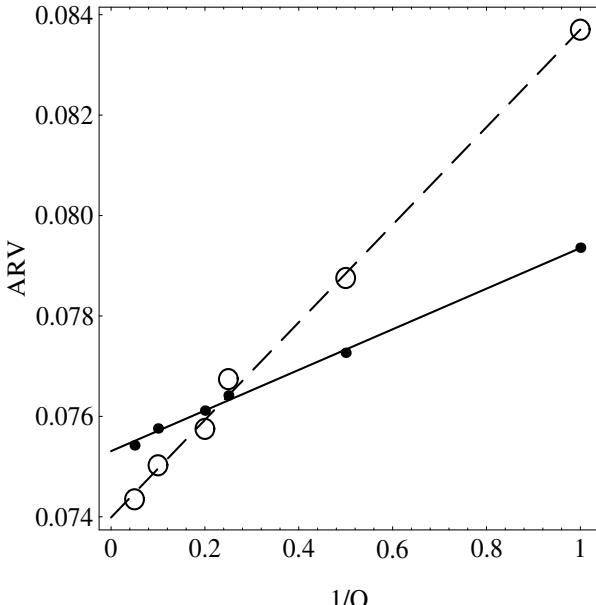


Fig. 6.2. Extrapolation method used for extracting the $Q \rightarrow \infty$ prediction

The results for different ensemble size Q at two different training periods, $t = 70\text{KE}$ (dots) and 140KE (circles) lie on straight lines as a function of $1/Q$. For each curve, the first three points from the right represent ensemble sizes of 1, 2, and 4 respectively. While the three points of 140KE all lie above the corresponding ones of 70KE , an extrapolation to larger ensemble sizes suggests that the overall performance will be better for 140KE as is observed from the fit to the line.

contain the period between 1701 and 1920, and the test-set to contain the years 1921 to 1955. Following [10], we calculate the prediction error according to the average relative variance (ARV) of the data set S :

$$\text{ARV} = \frac{\sum_{k \in S} (y_k - f(\mathbf{x}_k))^2}{\sum_{k \in S} (y_k - E[y_k])^2} \quad (6.1)$$

$y_k(\mathbf{x}_k)$ are the data values and $f(\mathbf{x}_k)$ is the predictor. In our time series problem, for any given time point $t = k$, the input vector \mathbf{x}_k has component values taken from the series at times $t - 1, t - 2, \dots, t - 12$ (as in [10]). The denominator in Eq. 1 is $\sigma^2 = 1535$ for the training set. The same value is used for the test set. We use neural networks with 12 inputs, one sigmoidal hidden layer consisting of 4 units and a linear output. They are then enlarged to form recurrent networks (SRN) [1] in which the input layer is increased by adding to it the hidden layer of the previous point in the time series. The learning algorithm consists of back propagation applied to an error function which is the MSE of the training set. A

validation set containing 35 randomly chosen points was left out during training to serve for performance validation.

Fig. 6.1 displays our results on the test set as a function of the number of training epochs. We observe a descending order of $Q = 1, 2, 4, 5, 10, 20$ followed by the extrapolation to $Q \rightarrow \infty$. All of these curves correspond to averages over groups of size Q , calculated by running 60 networks. To demonstrate how the extrapolation is carried out we display in Fig. 6.2 the points obtained for $t = 70$ and $t = 140$ KE as a function of $\frac{1}{Q}$. It is quite clear that a linear extrapolation is very satisfactory. Moreover, the results for $Q = 20$ are not far from the extrapolated $Q \rightarrow \infty$ results. Note that the minimum of the $Q \rightarrow \infty$ curve in Fig. 1 occurs at a much higher training time than that of the $Q = 1$ single network curve. This is also evident from the crossing of the $t = 70$ and $t = 140$ KE lines on Fig. 2. An important conclusion is that the stopping criterion for ensemble training (to be applied, of course, to every network in the group) is very different from that of single network training.

6.2.2 Best Result

The curves shown in Fig. 1 were obtained with a learning rate of 0.003. Lower learning rates lead to lower errors. In that case the effect of ensemble averaging is not as dramatic. We obtained our best result by changing our input vector into the six dimensional choice of Pi & Peterson [8] that consists of $x_{t-1}, x_{t-2}, x_{t-3}, x_{t-4}, x_{t-9}$ and x_{t-10} . Using a learning rate of 0.0005 on the otherwise unchanged SRN described above, we obtain the minimum of the prediction error at 0.0674, which is better than any previously reported result.

6.3 Theoretical Analysis

The theoretical setting of the method is provided by the bias/variance decomposition. Within this framework, we define a particular bias/variance decomposition for networks differing by their initial conditions only. This is a particularly useful subset of the general set of all sources of variance.

The performance of an estimator is commonly evaluated by the Mean Square Error (MSE) defined as

$$\text{MSE}(f) \equiv E \left[(y - f(\mathbf{x}))^2 \right] \quad (6.2)$$

where the average is over test sets for the predictor f , and y are the target values of the data in \mathbf{x} . Assuming the expectation E is taken with respect to the true probability of \mathbf{x} and y , the MSE can be decomposed into

$$E \left[(y - f(\mathbf{x}))^2 \right] = E \left[(y - E[y|\mathbf{x}])^2 \right] + E \left[(f(\mathbf{x}) - E[y|\mathbf{x}])^2 \right]. \quad (6.3)$$

The first RHS term represents the variability or the noise in the data and is independent of the estimator f . It suffices therefore to concentrate on the second

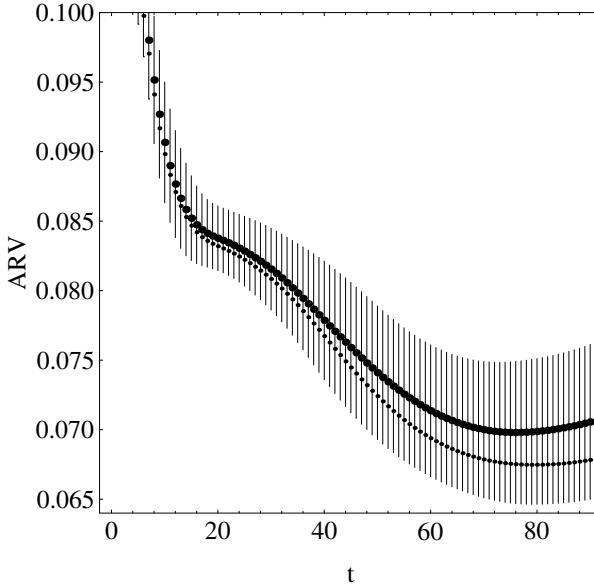


Fig. 6.3. Our best results for the test set of the sunspots problem. Plotted here are $Q = 1$ results for various choices of initial conditions, represented by their averages with error-bars extending over a standard deviation, and $Q = 20$ results (the thinner points), as a function of training time in K-epochs. The network is based on the Pi & Peterson variables, and the learning rate is 0.0005.

term. Any given predictor $f(\mathbf{x})$ is naturally limited by the set of data on which it is trained. Considering a typical error one may average over all data space [2] and decompose this error into Bias and Variance components:

$$E_{\mathcal{D}} \left[(f(\mathbf{x}) - E[y|\mathbf{x}])^2 \right] = B_{\mathcal{D}} + V_{\mathcal{D}} \quad (6.4)$$

where

$$B_{\mathcal{D}}(f(\mathbf{x})) = (E_{\mathcal{D}} [f(\mathbf{x})] - E[y|\mathbf{x}])^2 \quad (6.5)$$

$$V_{\mathcal{D}}(f(\mathbf{x})) = E_{\mathcal{D}} \left[(f(\mathbf{x}) - E_{\mathcal{D}} [f(\mathbf{x})])^2 \right]. \quad (6.6)$$

In our application we use as our predictor $E_{\mathcal{I}}[f(\mathbf{x})]$ where the subscript \mathcal{I} denotes the space of initial weights of the neural network that serves as $f(\mathbf{x})$. To understand the effect of averaging over initial weights let us construct a space \mathcal{R} by the direct product of \mathcal{D} and \mathcal{I} . It can then be shown that

$$B_{\mathcal{R}}(f(\mathbf{x})) = B_{\mathcal{D}}(E_{\mathcal{I}}[f(\mathbf{x})]), \quad (6.7)$$

and

$$V_{\mathcal{R}}(f(\mathbf{x})) \geq V_{\mathcal{D}}(E_{\mathcal{I}}[f(\mathbf{x})]). \quad (6.8)$$

This means that using $E_{\mathcal{I}}[f(\mathbf{x})]$ as the predictor, the characteristic error has reduced variance but unchanged bias.

The bias term may also be represented as $B_{\mathcal{R}} = E_{\mathcal{D}}[B_{\mathcal{I}}]$ where

$$B_{\mathcal{I}}(f(\mathbf{x})) = (E_{\mathcal{I}}[f(\mathbf{x})] - E[y|\mathbf{x}])^2. \quad (6.9)$$

$B_{\mathcal{I}}$ is unaffected when $f(\mathbf{x})$ is replaced by its average. The analogously defined variance term, $V_{\mathcal{I}}$, gets eliminated by such averaging. In other words, by averaging over all \mathcal{I} we eliminated all variance due to the choice of initial weights. The difference between the $Q = 1$ and $Q \rightarrow \infty$ curves in Fig. 1 represents this reduction of variance.

To understand the Q dependence of Fig. 2 consider an average defined by $\bar{f}(x)$ over functions that represent independent identically distributed random variables. It can then be shown that

$$B(\bar{f}) = \overline{B(f)} \quad V(\bar{f}) = \overline{V(f)}/Q. \quad (6.10)$$

Hence we can interpret the $1/Q$ behavior displayed in Fig. 2 as a demonstration that the choice of initial conditions in that analysis acts as effective random noise. In general this is not necessarily the case, since networks with different initial conditions may have non-trivial correlations. For a more thorough discussion of this and other points see [5].

In conclusion, we see that averaging over networks with different initial weights is helpful in reducing the prediction error by eliminating the variance induced by initial conditions. Performing this average over groups of finite size Q one finds out from the Q dependence if the errors induced by initial conditions are correlated or not. Moreover, one may estimate the Q needed to eliminate this source of variance.

References

- [1] Elman, J.L., Zipser, D.: Learning the Hidden Structure of Speech. *J. Acoust. Soc. Amer.* 83, 1615–1626 (1988)
- [2] Geman, S., Bienenstock, E., Doursat, R.: Neural networks and the bias/variance dilemma. *Neural Comp.* 4(1), 1–58 (1992)
- [3] Lincoln, W.P., Skrzypek, J.: Synergy of clustering multiple back propagation networks. In: Touretzky, D.S. (ed.) *Advances in Neural Information Processing Systems*, vol. 2, pp. 650–657. Morgan Kaufmann, SanMateo (1990)
- [4] Morris, J.: Forecasting the sunspot cycle. *J. Roy. Stat. Soc. Ser. A* 140, 437–447 (1977)
- [5] Naftaly, U., Intrator, N., Horn, D.: Optimal Ensemble Averaging of Neural Networks. *Network, Comp. Neural Sys.* 8, 283–296 (1997)
- [6] Nowlan, S.J., Hinton, G.E.: Simplifying neural networks by soft weight-sharing. *Neural Computation* 4, 473–493 (1992)
- [7] Perrone, P.M.: Improving regression estimation: averaging methods for variance reduction with extensions to general convex measure optimization. PhD thesis, Brown University, Institute for Brain and Neural Systems (1993)

- [8] Pi, H., Peterson, C.: Finding the Embedding Dimension and Variable Dependencies in Time Series. *Neural Comp.* 6, 509–520 (1994)
- [9] Priestley, M.B.: *Spectral Analysis and Time Series*. Academic Press (1981)
- [10] Weigend, A.S., Huberman, B.A., Rumelhart, D.: Predicting the future: A connectionist approach. *Int. J. Neural Syst.* 1, 193–209 (1990)
- [11] Wolpert, D.H.: Stacked generalization. *Neural Networks* 5, 241–259 (1992)

Improving Network Models and Algorithmic Tricks^{*}

Preface

This section contains 5 chapters presenting easy to implement tricks which modify either the architecture and/or the learning algorithm so as to enhance the network's modeling ability. Better modeling means better solutions in less time.

In chapter 7, Gary Flake presents a trick that gives an MLP the additional power of an RBF. Trivial to implement, one simply adds extra inputs whose values are the **square of the original inputs** (p. 144). While adding higher order terms as inputs is not a new idea, this chapter contributes new insight by providing (1) a good summary of previous work, (2) simple clear examples illustrating this trick, (3) a theoretical justification showing that one need only add the higher order squared terms, and (4) a thorough comparison with numerous other network models. The need for *only* the squared terms is significant because it means that we gain this extra power without having the number of inputs grow excessively large. We remark that this idea can be extended further by including relevant features other than squared inputs e.g. by using kernel PCA [2] to obtain the non-linear features.

Rich Caruana in chapter 8 presents **multi-task learning (MTL)** (p. 163) where extra outputs are added to a network to predict tasks separate but related to the primary task. To introduce the trick, the chapter begins with an example and detailed discussion of a simple boolean function of binary inputs. The author then presents several of what one might use as extra outputs in practice. These include, among others: (1) features that are available only **after predictions** must be made, but which are available offline at *training* time (p. 170), (2) the same task but with a **different metric** (p. 175), and (3) the same task but with **different output representations** (p. 176). Empirical results are presented for (1) mortality rankings for pneumonia where the extra outputs are test results not available when the patient first enters the hospital but which are available a posteriori to complement the training data, and (2) a vehicle steering task where other outputs include location of centerline and road edges, etc. The last part of the chapter is devoted to topics for implementing MTL effectively, such as size of hidden layers (p. 181), early stopping (p. 181), and learning rates (p. 185).

The next chapter by Patrick van der Smagt and Gerd Hirzinger presents a trick to reduce the problem of ill-conditioning in the Hessian. If a unit has a very small outgoing weight then the influence of the incoming weights to that unit will be severely diminished (see chapter 1 for other sources of Hessian ill-conditioning). This results in flat spots in the error surface, which translates into

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

slow training (see also [1]). The trick is to add linear **shortcut connections** (p. 196) from the input to the output nodes to create what the authors refer to as a **linearly augmented feed-forward network**. These connections **share** the weights with the input to hidden connections so that no new weight parameters are added. This trick enhances the sensitivity of the network to those incoming weights thus removing or reducing the flat spots in the error surface. The improvement in the quality of the error surface is illustrated in a toy example. Simulations with data from a robot arm are also shown.

The trick discussed by Nicol Schraudolph in chapter 10 is to center the various factors comprising the neural network's gradient (p. 208): input and hidden unit activities (see chapter 1), error signals, and the slope of the hidden units' nonlinear activation functions (p. 208). To give an example: *activity centering* (p. 207) is done by simply transforming the values of the components x_i into

$$\check{x}_i = x_i - \langle x_i \rangle,$$

where $\langle \cdot \rangle$ denotes averaging over training samples. All different centering strategies can be implemented efficiently for a stochastic, batch or mini batch learning scenario (p. 209). He also uses **shortcut connections** (p. 208) but quite differently from the previous chapter: the shortcut connections contain new weights (not shared) which complement slope centering by carrying the linear component of the signal, making it possible for the rest of the network to concentrate on the nonlinear component of the problem. So, with respect to shortcut connections, the approaches in chapters 9 and 10 appear complementary. Centering gives a nice speed-up without much harm to the generalization error, as seen in the simulations on toy and vowel data (p. 211).

In chapter 11, Tony Plate presents a trick requiring only minimal memory overhead that reduces **numerical round-off** in backpropagation networks. Round-off error can occur in the standard method for computing the derivative of the logistic function since it requires calculating the product

$$y(1 - y)$$

where y is the output of either a hidden or output unit. When the value of y is close to 1 then the limited precision of single or even double precision floating point numbers can result in the product being zero. This may not be a serious problem for on-line learning but can cause significant problems for networks using batch mode, particularly when second order methods are used. Such round-off can occur in other types of units as well. This chapter provides formulas for reducing such round-off errors in the computation of

- derivatives of the error for logistic units or tanh units (p. 226 and 229)
- derivatives in a one-of- k classification problem with cross-entropy error and softmax (p. 228)
- derivatives and errors in a two-class classification problem using a single logistic output unit with cross entropy error and 0/1 targets (p. 228).

References

- [1] Hochreiter, S., Schmidhuber, J.: Flat minima. *Neural Computation* 9(1), 1–42 (1997)
- [2] Schölkopf, B., Smola, A., Müller, K.-R.: Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation* 10, 1299–1319 (1998)

Square Unit Augmented, Radially Extended, Multilayer Perceptrons^{*}

Gary William Flake

Siemens Corporate Research, Inc.

755 College Road East

Princeton, NJ 08540

flake@scr.siemens.com

<http://mitpress.mit.edu/books/FLA0H/cbnhtml/author.html>

Abstract. Consider a multilayer perceptron (MLP) with d inputs, a single hidden sigmoidal layer and a linear output. By adding an additional d inputs to the network with values set to the square of the first d inputs, properties reminiscent of higher-order neural networks and radial basis function networks (RBFN) are added to the architecture with little added expense in terms of weight requirements. Of particular interest, this architecture has the ability to form localized features in a d -dimensional space with a single hidden node but can also span large volumes of the input space; thus, the architecture has the localized properties of an RBFN but does not suffer as badly from the curse of dimensionality. I refer to a network of this type as a SQuare Unit Augmented, Radially Extended, MultiLayer Perceptron (SQUARE-MLP or SMLP).

7.1 Introduction and Motivation

When faced with a new and challenging problem, the most crucial decision that a neural network researcher must make is in the choice of the model class to pursue. Several different types of neural architectures are commonly found in most model building tool-boxes, with two of the more familiar, radial basis function networks (RBFNs) [15, 16, 13, 1, 17] and multilayer perceptrons (MLPs) [25, 20], exemplifying the differences found between global and local model types. Specifically, an MLP is an example of a global model that builds approximations with features that alter the entire input-output response, while an RBFN is a local model that uses features confined to a finite region in the input-space. This single difference between the two model types has major implications for many architectural and algorithmic issues. While MLPs are slow learners, have low memory retention, typically use homogeneous learning rules and are relatively less troubled by the curse of dimensionality, RBFNs are nearly opposite in every way:

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998)

they are fast learners, have high memory retention, typically use heterogeneous learning rules and are greatly troubled by the curse of dimensionality.

Because of these differences, it is often tempting to use one or more heuristics to make the choice, e.g., RBFNs (MLPs) for low (high) dimensional problems, or RBFNs (MLPs) for continuous function approximation (pattern classification) problems. While these rules-of-thumb are often sufficient for simple problems there are many exceptions that defy the rules (e.g., see [12]). Moreover, more challenging problems from industrial settings often have high-dimensional input-spaces that are locally well-behaved and may not be clearly defined as either function approximation or pattern classification problems. This means that choosing the best architectures for a particular problem can be a nontrivial problem in itself.

Ironically, a good compromise to this dilemma has been known for quite some time but only recently has the elegance of the trick been appreciated. For years, researchers have commonly used augmented MLPs by adding the squares of the inputs as auxiliary inputs. The justification for this has always been fairly casual and has usually boiled down to the argument that using this trick couldn't possibly hurt. But as it turns out, an MLP augmented in this way with n hidden nodes can almost perfectly approximate an RBFN with n basis functions. The "almost" comes from the fact that the radial basis function in the augmented MLP is not Gaussian but quasi-Gaussian (which is an admittedly undefined term that I simply use to mean "so close to Gaussian that it really doesn't matter."). This means that an MLP augmented with the squares of its inputs can easily form local features with a single hidden node but can also span vast regions of the input-space, thereby effectively ignoring inputs when needed. Thus, the best of both architectural approaches is retained by using this amazingly simple trick.

The remainder of this chapter is divided into 5 more sections. Section 7.2 contains a description of the trick and briefly gives a comparison of the proposed architecture to other classes of well-known models. In Section 7.3, a function approximation problem and a pattern classification problem are used as examples to demonstrate the effectiveness of the trick. Afterwards, a well-known and challenging vowel classification problem is studied in greater detail. Section 7.4 theoretically justifies the trick by showing the equivalence of the resulting architecture and RBFNs, while Section 7.5 gives a more intuitive justification for the proposed trick by illustrating the types of surfaces and boundaries that can be formed by a single node with auxiliary square inputs. Finally, in Section 7.6, I give my conclusions.

7.2 The Trick: A SQUARE-MLP

The proposed trick involves only a simple modification to the standard MLP architecture: the input layer of an MLP is augmented with an extra set of inputs that are coupled to the squares of the original inputs. This trick can be implemented in at least two different ways. The first technique is to simply augment a data set with the extra inputs. Thus, if one had a set with each input pattern having d components, then a new data set can be made from the original that

has $2d$ inputs with the extra d inputs set equal to the squares of the original inputs. Implementing the trick in this way is expensive from a memory point of view but has the advantage that not a single line of new source code need be written to try it out. Moreover, this allows the trick to be tried even on a commercial simulator where one may not have access to source code.

The second way to implement the trick is to explicitly code the actual changes into the architecture:

$$y = \sum_i w_i g \left(\sum_j u_{ij} x_j + \sum_k v_{ik} x_k^2 + a_i \right) + b, \quad (7.1)$$

with $g(x) = \tanh(x)$ or $1/(1+\exp(-x))$. I call such a network a SQuare Unit Augmented, Radially Extended, MultiLayer Perceptron (SQUARE-MLP or SMLP). The “square unit augmented” portion of the name comes from the newly added $v_{ik} x_k^2$ terms. The reason behind the “radially extended” portion of the name will become clear in Sections 7.4 and 7.5. All experiments in Section 7.3 use the architecture described by Equation 7.1. The history of this trick is rather difficult to trace primarily because it is such a trivial trick; however, a brief list of some related ideas is presented below.

Engineering and Statistics. Very early related ideas have been pursued in the statistics community in the form of polynomial regression and Volterra filters in the engineering community [24, 22]. However, in both of these related approaches the model output is always linear in the polynomial terms, which is not the case with the SMLP architecture or in the other neural architectures discussed below.

Sigma-Pi Networks. Some neural network architectures which are much more complicated than Equation 7.1 have the SMLP as a special case. Perhaps the earliest reference to a similar idea in the neural network literature can be traced back to Sigma-Pi networks [20], which extends an MLP’s linear net input function with a summation of products, $\sum_i w_{ji} \prod_k x_{ik}$. One could imagine a multilayer Sigma-Pi network that manages to compute the squares of the x_{ik} terms prior to them being passed through to a sigmoidal activation function. This would be a rather clumsy way of calculating the squares of the inputs, but it is possible to do it, nonetheless.

Higher-Order Networks. Perhaps the closest example is the higher-order network, proposed by Lee *et al.* [14], which is similar to Equation 7.1 but uses the full quadratic net input function:

$$y = \sum_i w_i g \left(\sum_j u_{ij} x_j + \sum_k \sum_l v_{ikl} x_k x_l + a_i \right) + b. \quad (7.2)$$

With v_{ikl} set to zero when $k \neq l$ a SMLP is recovered. Thus, a SMLP is actually a higher-order network with a diagonal quadratic term. Higher-order networks have been shown to be very powerful extensions of MLPs. They can form both

local and global features but only at the cost of squaring the number of weights for each hidden node. The memory requirements become an even greater issue when more sophisticated optimization routines are applied to an architecture such as Newton's or quasi-Newton methods which require memory proportional to the square of the number of weights in the network.

Functional Link Networks. Another related architecture is the functional-link network [18], which is similar to a standard MLP but explicitly augments the network with the results of scalar functions applied to the inputs. For example, in some applications it may be known in advance that the desired output of the network is a function of the sine and cosine of one or more inputs (e.g., the inputs may correspond to angles of a robot arm). In this case, one would do well to include these values explicitly as inputs into the network instead of forcing the network to learn a potentially difficult-to-model concept. Functional-link networks may use any scalar function that, in the end, essentially performs a type of preprocessing on the data. Usually, expert knowledge is used to determine which extra scalar functions are to be incorporated into the network; that is, there is no general technique for choosing the best preprocessor functions *a priori*. However, given a set of nonlinear transformations on the input data one can perform principal component analysis (PCA) on the nonlinear feature space to select a subset that carries the most variance. A computationally feasible version of this technique has been proposed in [23], which they refer to as kernel PCA. In any event, by using the square function to augment a functional-link network, the SMLP is once again recovered.

While I have tried to assign proper credit, it is generally accepted that the basic idea of adding the squares of the inputs to a model is at least as old as the sage advice “preprocessing is everything.”

7.3 Example Applications

The following three examples demonstrate problem domains in which an SMLP can conceivably outperform an MLP or an RBFN. All of the examples are well-known benchmarks. In each case, the output response of the models must form local features while simultaneously spanning a large region of the input-space. In general, the MLPs will have difficulty forming the local features, while the RBFNs will have trouble spanning the flat regions of the input space.

7.3.1 Hill-Plateau Function Approximation

The first problem is an admittedly contrived example that was chosen precisely because it is difficult for both MLPs and RBFNs. The “Hill-Plateau” surface [21], displayed in Figure 7.1, has a single local bump on a sigmoidal ridge. Training data for this problem consists of a two-dimensional uniform sampling on a 21×21 grid of the region shown in the figure while the testing data comes from a finer 41×41 grid.

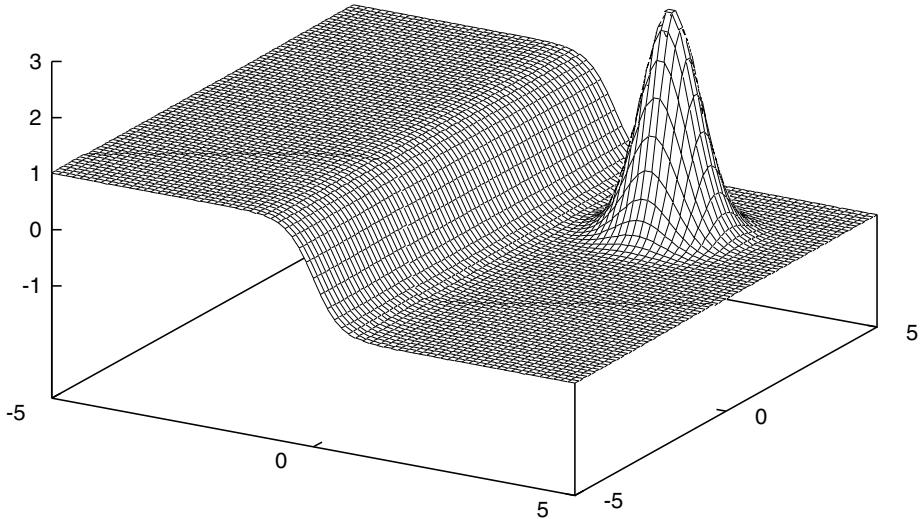


Fig. 7.1. A hill-plateau exemplifies the differences between local and global architectures. MLPs can easily form the plateau but have a hard time on the hill, while RBFNs trivially form the hill but are troubled by the plateau.

Table 7.1. Best of twenty runs for the Hill-Plateau surface

Model	# of Nodes	# of Weights	Test RMSE
RBFN	2	9	0.333406
	3	13	0.071413
	4	17	0.042067
	5	21	0.002409
MLP	2	9	0.304800
	3	13	0.015820
	4	17	0.001201
SMLP	2	13	0.000025

Besides a standard MLP, a normalized RBFN (NRBFN) was used for this problem, which is described by the two equations:

$$y = \frac{\sum_i w_i r_i(\mathbf{x})}{\sum_j r_j(\mathbf{x})} + a \quad \text{and} \quad (7.3)$$

$$r_i(\mathbf{x}) = \exp(-||\mathbf{x} - \mathbf{c}_i||^2 / \sigma_i^2), \quad (7.4)$$

with \mathbf{c}_i and σ_i being the i th basis center and width, respectively. To train the NRBFNs the centers were first clustered in the input-space of the training patterns with the k -means clustering algorithm. The width of each basis function was then set proportional to the distance to the nearest neighbor. Afterwards, the least-mean-square solution of the linear terms, w_i and a , were solved for

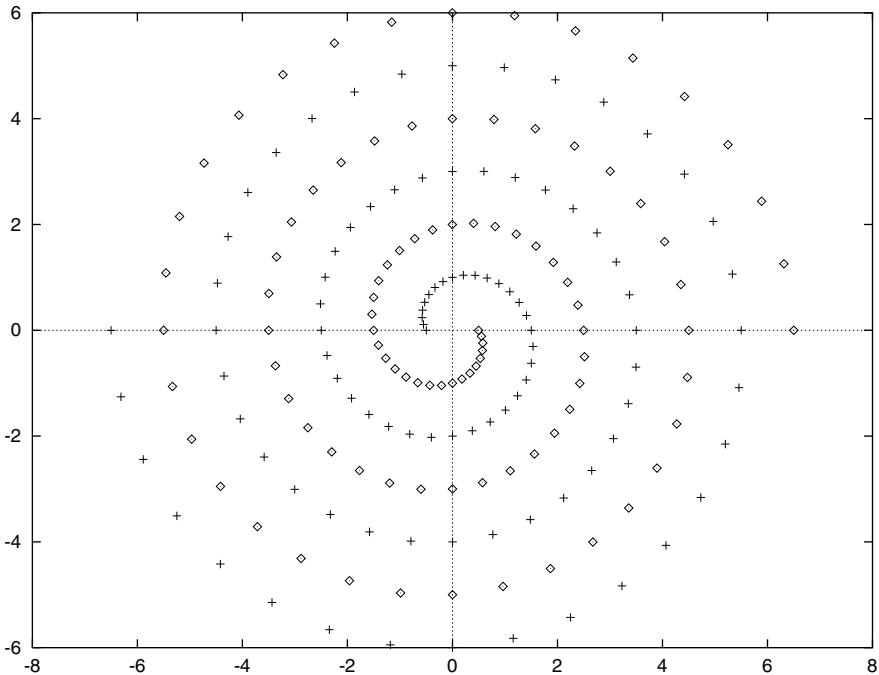


Fig. 7.2. Data for the Two-Spiral problem

exactly using a singular value decomposition to compute the pseudo-inverse. All of this formed the initial set of weights for the quasi-Newton's method (BFGS) optimization routine which was used on all weights simultaneously for up to 200 epochs.

For the MLPs, a single hidden layer with the $\tanh(x)$ activation function and a linear output was used. All weights were initially set to uniform random values in a -0.1 to 0.1 range. All weights were then trained with quasi-Newton method (BFGS) for up to 200 epochs. The SMLPs were setup and trained exactly as the MLPs.

Table 7.1 shows the results for all three architectures. As can be seen, both the RBFNs and the MLPs have a fair amount of difficulty with this task, even though the training data is noise free and the training procedures are fairly sophisticated. Contrary to this, the SMLP manages to nail the surface with only two hidden nodes. Moreover, the testing error is orders of magnitude better than the best results from the other two architectures.

7.3.2 Two-Spirals Classification

The two-spirals classification problem is a well-known benchmark that is extremely challenging for all neural network architectures; additionally, virtually no results have been reported for RBFNs as the problem is such that local models would

have to memorize the training data with many basis functions (> 100) in order to come even close to solving it. Thus, this is an example of a problem that RBFNs are not even viable candidates, which is why they are not considered further. Figure 7.2 shows the data for the two-spirals problem, which consists of 194 points on the x - y -plane that belong to one of two spirals, each of which rotates around the origin three times.

Previous results by other researchers for this problem have mostly focused on traditional MLPs and MLPs with shortcut connections. The best reported results for 100% classification accuracy are summarized below:

- (Lang & Witbrock [9]): 2-5-5-5-1 MLP with shortcut connections and 138 total weights. Average convergence time of 20,000 batched backpropagation epochs.
- (Lang & Witbrock [9]): 2-5-5-5-1 MLP with shortcut connections, 138 total weights, and cross-entropy error function. Average convergence time of 12,000 batched backpropagation epochs.
- (Lang & Witbrock [9]): 2-5-5-5-1 MLP with shortcut connections and 138 total weights. Average of 7,900 quickprop [3] epochs.
- (Frostrom [unpublished]): 2-20-10-1 MLP with no shortcut connections and 281 weights. Required 13,900 batched backpropagation with momentum epochs.
- (Fahlman and Lebiere [4]): Cascade-Correlation MLP using 12 to 19 hidden units (15 average) and an average of 1700 quickprop epochs. Because of the cascade correlation topology, these networks used between 117 and 250 weights.

As these results show, the two-spiral problem is exceptionally difficult, requiring both complicated network topologies and long training times.

Compared to the results above, the SMLP architecture seems to be very well-suited to this problem. An SMLP with 15 hidden hyperbolic tangent units (for a total of only 91 weights) was trained with a conjugate gradient optimization routine. In ten out of ten trials, the SMLP solved the two-spirals problem with an average of 2500 training epochs (but as few as 800). Notice that the architecture for the SMLP, both topologically and in the number of weights, is much simpler than those used in the studies with the MLPs. As a result, the optimization algorithms can be much more efficient. This is a case of the representation power of an architecture simplifying the learning, thereby making weight optimization a faster process.

Although it was not always possible to consistently train a simpler SMLP to solve this problem, an SMLP with 10 hidden nodes (and only 61 weights) succeeded on three separate trials, taking an average of 1500 epochs. The output response surface of this SMLP is shown in Figure 7.3. In Section 7.5 we will examine the different types of surfaces that can be formed by a single SMLP hidden node. We shall see that the the SMLP's ability to easily form local and global features is crucial to its ability to rapidly solve the two-spiral problem.

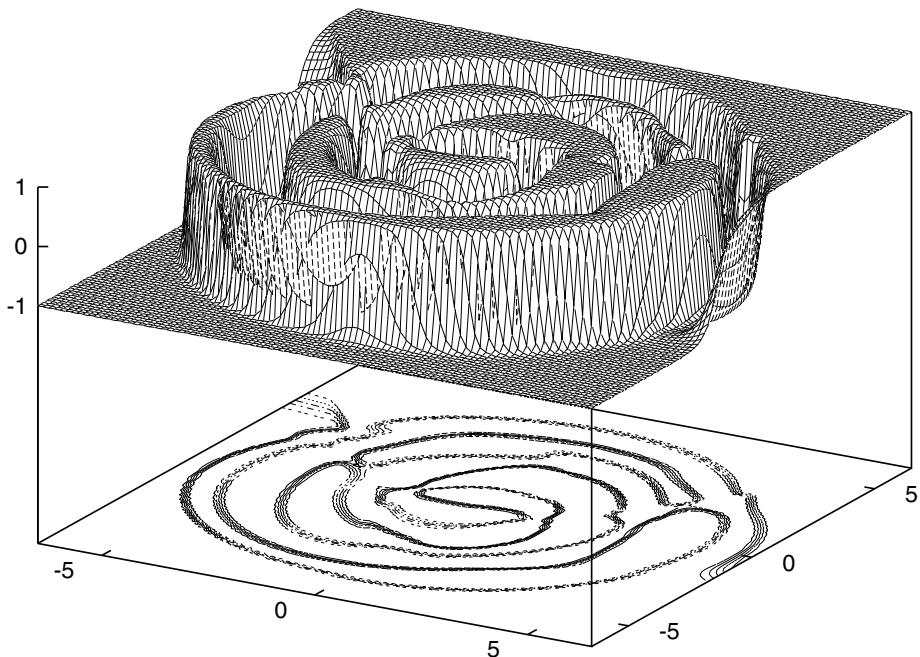


Fig. 7.3. SMLP reconstructed Two-Spiral surface from only ten hidden nodes

7.3.3 Vowel Classification

The Deterding vowel recognition data set [2] is another widely studied benchmark that is much more difficult than the two earlier problems and is also more indicative of the type of problem that a neural network practitioner could be faced with. The data consists of auditory features of steady state vowels spoken by British English speakers. There are 528 training patterns and 462 test patterns with each pattern consisting of 10 features and belonging to exactly one of 11 classes that correspond to the spoken vowel. The speakers are of both genders, making this a very interesting problem.

All results for this section use an architecture with 10 inputs, a varying number of hidden units, and 11 outputs. Some results from previous studies are summarized in Table 7.2. Some of the earlier studies are somewhat anecdotal in that they used either a single experiment or only a few experiments but they are informative as they demonstrate what the sophisticated neural network practitioner could expect to achieve on this problem with a wide number of architectures. Interestingly, Robinson's results show that a nearest neighbor classifier is very difficult to beat for this problem. With a 56% correct classification rate, a nearest neighbor classifier outperforms all of Robinson's neural solutions. However, nearest neighbor approaches require vast amounts of data to be stored as a look-up table, so this is not a particularly encouraging result.

The best score, reported by Hastie and Tibshirani [7], was achieved with a Discriminant Adaptive Nearest Neighbor (DANN) classifier. The score of 61.7% was from the best classifier found in a number of simulation studies; hence, this score represents the best known prior result as found by an expert attempting multiple solutions.

Table 7.3 lists the results of the experiments done specifically for this work. Five different model/optimization combinations are shown in the table, and each row in the table corresponds to fifty separate trials started with different random initial conditions. For the major entries labeled as “trained” the weights of the model were determined by conjugate gradient for models with more than 1,000 weights and quasi-Newton’s method for models with fewer than 1,000 weights. The optimization routines were set to minimize an error function of the form $E = e^2 + \lambda \|\mathbf{w}\|^2$ where e is the difference between the actual and desired outputs and λ is a weight decay term that penalizes large weights. λ was set to 10^{-4} for all experiments. Values of λ equal to 10^{-3} and 10^{-5} consistently gave worse results for all architectures, so 10^{-4} was a fair compromise.¹ All MLP and SMLP architectures used a $\tanh(x)$ activation function, while the RBFN is the same as described in Section 7.3.1 but is unnormalized (The normalized RBFN gave consistently inferior results). The optimization routines were always allowed to run until convergence (change in error measure is less 10^{-20}) unless otherwise noted.

The weights in the RBFN and NRBFN models were “solved” with a three step process: 1) set the centers to the cluster centers generated by the k -means clustering algorithm applied to the input vectors of the training data; 2) set the widths proportional to the distance to the nearest neighbor of each center; and 3) solve the remaining linear weights as a least mean square problem with a matrix pseudo-inverse.

The SMLP architecture can be solved for in a manner similar to how the RBFN and NRBFN networks are solved. The details of this procedure are covered in Section 7.4, but it suffices to say at this point that the procedure is nearly identical with the exception that the weights corresponding to the centers and the widths must be slightly transformed.

Interestingly, the SMLP can use this “solve” procedure to compute an initial set of weights for an SMLP that is then trained with a gradient-based method. This has the effect of predisposing the SMLP to a very good initial solution that can be refined by the gradient-based optimization routines. The row in Table 7.3 with the parenthetical label “hybrid” corresponds to SMLPs trained in this way.

Three of the columns in Table 7.3 show three different ways of measuring success for the various models, of which the only statistically significant measure is the column labeled “% Correct (Average),” which is the average test set score achieved after the optimization procedure halted on the training data. The scores reported under the column heading “% Correct (Best)” correspond to the best

¹ Note that since the SMLP has an extra set of weights, care must be taken to control the capacity and avoid over-fitting the data.

Table 7.2. Previous result on the vowel data as summarized in [19], [12], [5] [8], [6], and [7]. All entries are either deterministic techniques or are the best scores reported, unless the score appears with a “*,” in which case the score represents an average over multiple runs.

Model	Number of Hidden	Number of Weights	Percent Correct
Single-Layer Perceptron	—	11	33
Multilayer Perceptron [19]	11	253	44
	22	495	45
	88	1,947	51
Multilayer Perceptron [12] (with renormalization)	5	121	50.1*
	10	231	57.5*
	20	451	50.6*
Stochastic Network [5] (FF-R classifier)	8	297	54*
	16	473	56*
	32	825	57.9*
Radial Basis Function	88	1,936	48
	528	11,616	53
Gaussian Node Network	11	253	47
	22	495	54
	88	1,947	53
	528	11,627	55
Square Node Network (not an SMLP)	11	253	50
	22	495	51
	88	1,947	55
Modified Kanerva Model	88	968	43
	528	5808	50
Local Approximation	2	5808	50.0
	3	5808	52.8
	5	5808	53.0
	10	5808	48.3
	20	5808	45.0
Nearest Neighbor	—	(5,808)	56
Linear Discriminant Analysis	—	715	44
Softmax	—	-?-	33
Quadratic Discriminant Analysis	—	-?-	47
CART	—	-?-	44
CART (linear comb. splits)	—	-?-	46
FDA / BRUTO	—	-?-	56
Softmax / BRUTO	—	-?-	50
FDA / MARS (degree 1)	—	-?-	55
FDA / MARS (degree 2)	—	-?-	58
Softmax / MARS (degree 1)	—	-?-	52
Softmax / MARS (degree 2)	—	-?-	50
LOCOCODE / Backprop (30 inputs)	11	473	58*
DANN	—	-?-	61.7

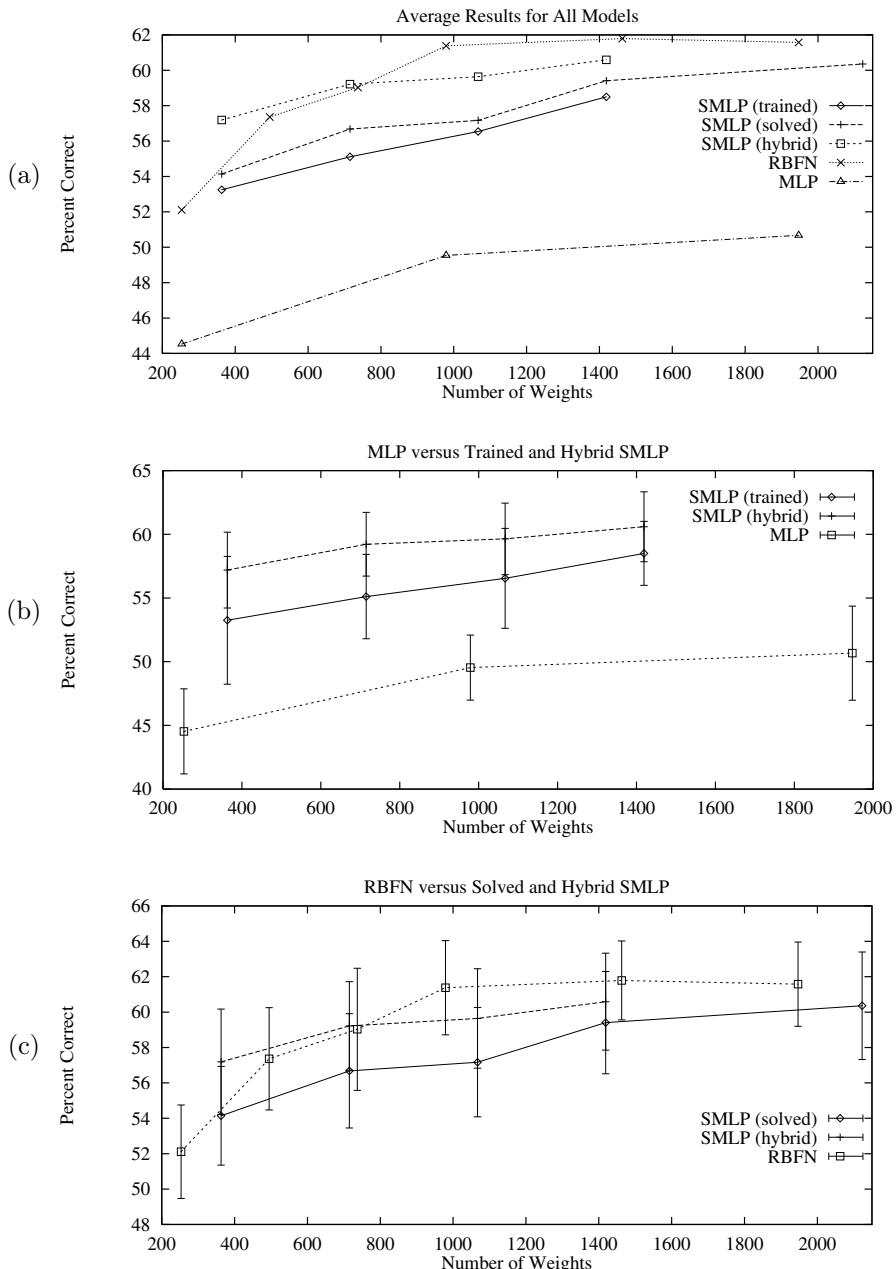


Fig. 7.4. The results from Table 7.3 shown in a graphical format: (a) the average results shown for all models types; (b) the average of the SMLP and MLP models with error bars shown; (c) the average of the SMLP and RBFN models with error bars shown

Table 7.3. Results from this study: All averages are computed from 50 trials with any result less than 33% (the score of a perceptron) being discarded as non-convergent. See the text for an explanation of the terms “Cheating,” “Best,” “and “Average”.

Model	# of Hidden	# of Weights	% Correct (Cheating)	% Correct (Best)	% Correct (Average)	Standard Deviation
MLP (trained)	11	253	53.24	51.08	44.53	3.34
	44	979	58.00	54.76	49.54	2.55
	88	1947	57.57	57.14	50.67	3.69
SMLP (trained)	11	363	63.63	60.82	53.25	5.02
	22	715	64.93	63.63	55.11	3.31
	33	1067	65.15	65.15	56.54	3.92
	44	1419	66.66	65.15	58.50	2.51
RBFN (solved)	11	253	—	56.92	52.11	2.64
	22	495	—	63.20	57.36	2.89
	33	737	—	66.88	59.03	3.45
	44	979	—	67.53	61.38	2.66
	66	1463	—	65.80	61.79	2.23
	88	1947	—	67.09	61.58	2.38
SMLP (solved)	11	363	—	58.87	54.14	2.79
	22	715	—	63.63	56.68	3.23
	33	1067	—	63.85	57.17	3.09
	44	1419	—	67.31	59.41	2.89
	66	2123	—	68.39	60.36	3.04
	88	2827	—	67.09	60.30	2.57
SMLP (hybrid)	11	363	66.66	63.85	57.19	2.98
	22	715	66.88	63.41	59.22	2.50
	33	1067	66.45	64.71	59.64	2.81
	44	1419	68.18	66.88	60.59	2.74

final test score achieved from the 50 runs, while the “% Correct (Cheating)” is the best test score achieved at any time during the training by any of the models in the fifty runs. Since the “solved” models have their weights computed in a single step, the “cheating” score only has meaning for the iterative techniques. One way of interpreting the “cheating” score is that it is the best score that could be achieved if one had a perfect cross validation set to use for the purpose of early stopping.

Some of the information in Table 7.3 is graphically summarized in Figure 7.3.3 and can, therefore, be better appreciated. In every case the SMLPs and the RBFNs outperform the MLPs by a statistically significant margin. However, the difference between the SMLPs and the RBFNs is much narrower, with the RBFNs and the hybrid SMLPs being nearly identical performance-wise. Also note that the hybrid training scheme appears to offer some improvement over both the trained and the solved SMLPs.

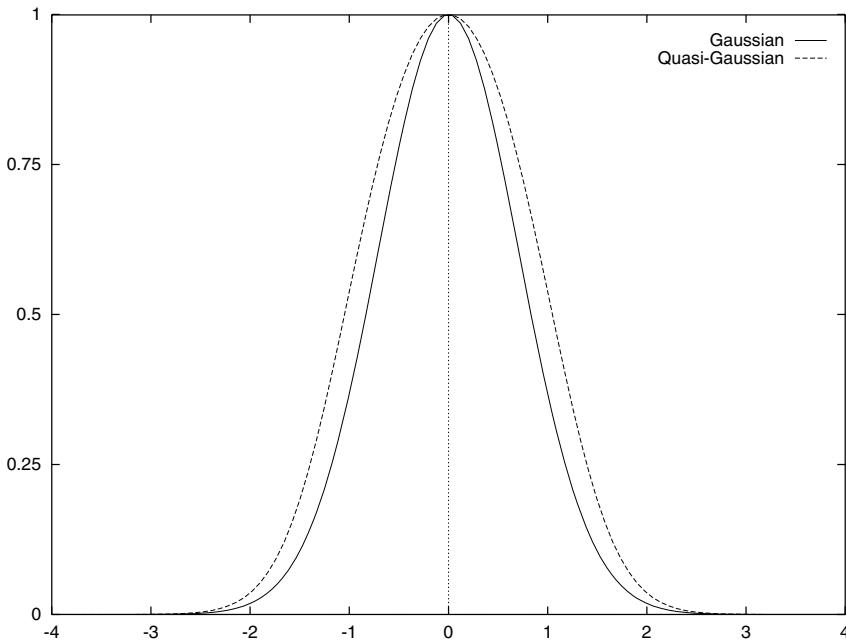


Fig. 7.5. A “quasi-Gaussian” activation function as a affine transformed sigmoidal

7.4 Theoretical Justification

Given the context of the numerical experiments from the previous section, we are now ready to see how an SMLP can be thought of as a “radially extended” version of an MLP. In this section, I will rewrite Equation 7.1 into a form that is equivalent to an RBFN; thus, we will see how it is possible for an SMLP to almost perfectly approximate an RBFN.

The first step is to more closely examine the sigmoidal activation function. Let $\text{sigmoid}(x) = 1/(1 + \exp(-x))$ and $\text{gauss}(x) = \exp(-x^2)$. We can define a quasi-Gaussian function as:

$$q(x) = 2 - 2/(\text{gauss}(x) + 1) = 2 - 2 \text{ sigmoid}(x^2). \quad (7.5)$$

This means that a local kernel function can be formed from an affine transformation of a sigmoid whose input has been squared.

Figure 7.5 shows how the quasi-Gaussian function relates to the true Gaussian function. Both functions are unimodal and exponentially decay in both directions. Moreover, a similar transformation can be applied to a hyperbolic tangent function; hence, it really doesn’t matter which of the two common sigmoid functions are used as either can be transformed into a basis function.

Since a basis function has a center and a width, we want to be able to form local features of arbitrary size at an arbitrary location. Typically, this means

that a basis function incorporates a distance measure such as Euclidean distance. With a center denoted by \mathbf{c}_i and a width proportional to σ , we can rewrite a normalized Euclidean distance function as follows:

$$\begin{aligned} \frac{1}{\sigma_i^2} \|\mathbf{x} - \mathbf{c}_i\|^2 &= \frac{1}{\sigma_i^2} (\mathbf{x} \cdot \mathbf{x} - 2\mathbf{c}_i \cdot \mathbf{x} + \mathbf{c}_i \cdot \mathbf{c}_i) \\ &= \left(-\frac{2}{\sigma_i^2} \mathbf{c}_i \right) \cdot \mathbf{x} + \left(\frac{1}{\sigma_i^2} \right) \mathbf{x} \cdot \mathbf{x} + \left(\frac{1}{\sigma_i^2} \mathbf{c}_i \cdot \mathbf{c}_i \right) \\ &= \sum_j \left(-\frac{2}{\sigma_i^2} c_{ij} \right) x_j + \sum_k \left(\frac{1}{\sigma_i^2} \right) x_k^2 + \left(\frac{1}{\sigma_i^2} \mathbf{c}_i \cdot \mathbf{c}_i \right) \end{aligned} \quad (7.6)$$

Thus, the equation

$$2 - 2 \text{ sigmoid} \left(\sum_j u_{ij} x_j + \sum_k v_{ik} x_k^2 + a_i, \right) \quad (7.7)$$

looks a lot like a radial basis function. By comparing Equation 7.6 to Equation 7.7 it is trivial to set the u_{ij} , v_{ik} , and a_i terms in such a way that a local “bump” is placed at a specific location with a specific width. This means that a single hidden node in an SMLP network can form a local feature in an input of any dimension. By way of comparison, Lapedes and Farber [10, 11] similarly constructed local features with standard MLPs. However, in a d -dimensional input space, one would need an MLP with two hidden layers, $2d$ hidden nodes in the first hidden layers, and another hidden node in the second hidden layer, just to form a single local “bump”.

This simple analysis shows that local features are exceptionally easy to form in an SMLP but are potentially very difficult to form in an MLP. As mentioned in Section 7.3.3, it is possible to exploit the similarity between SMLPs and RBFNs and “solve” the weights in an SMLP with a non-iterative procedure. The first step is to choose a set of basis centers that can be determined by sub-sampling or clustering the input-space of the training data. After the centers are chosen, the nearest neighbor of each center with respect to the other centers can be calculated. These distances can be used as the widths of the basis centers. Next, the centers \mathbf{c}_i and widths σ_i can be plugged into Equation 7.6 to determine the values of the u_{ij} , v_{ik} and a_i weights. Finally, the linear weights in the SMLP, w_i and b from Equation 7.1, can be solved for exactly by using a matrix pseudo-inverse procedure.

Thus, one can train an SMLP as one would an MLP or one could solve an SMLP as one would an RBFN. It is also possible to combine the approaches and let the solved weights be the initial weights for a training procedure. Using the procedure to solve the weights can sometimes cut the computational overhead for computing the weight by orders of magnitude compared to typical training methods. Moreover, as was found in the numerical experiments in Section 7.3.3, solutions found with this hybrid scheme may easily exceed the quality of solutions found with more traditional approaches.

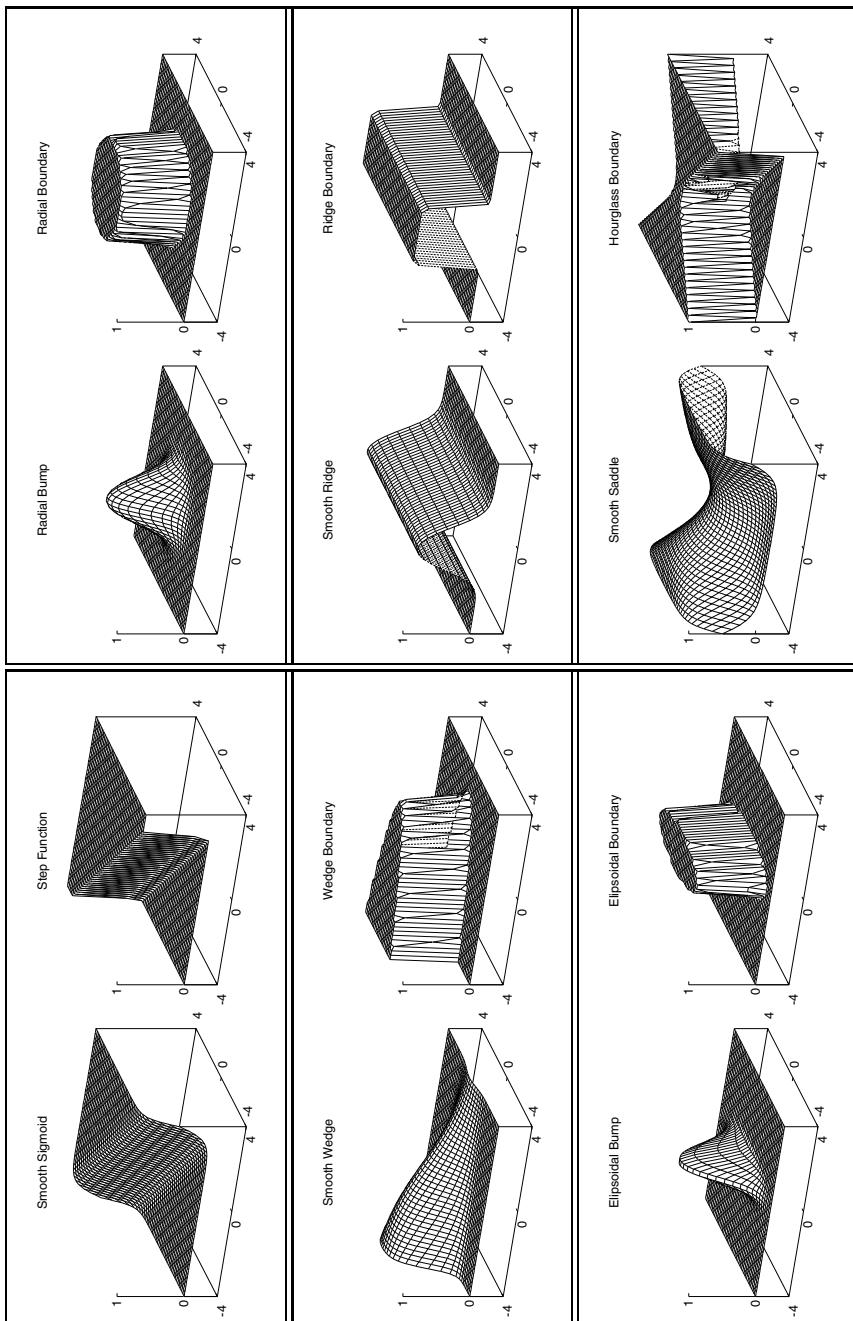


Fig. 7.6. Several difference types of surfaces and decision boundaries that can be formed by a single SMLP node

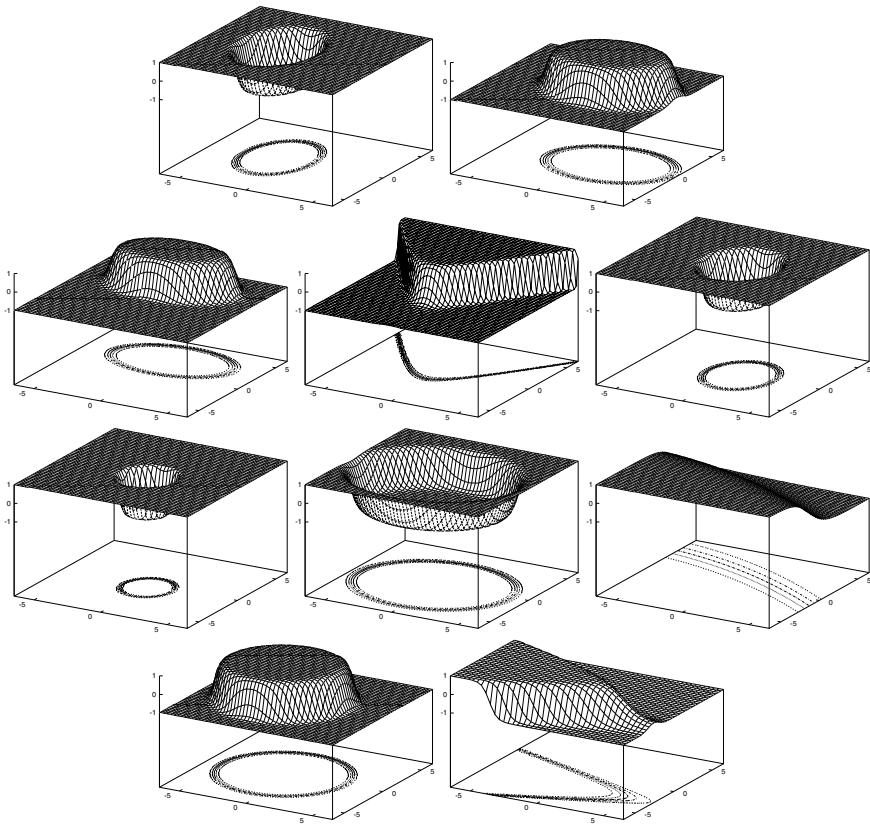


Fig. 7.7. Output response surfaces of the 10 hidden nodes in the SMLP network that solved the two spiral problem

7.5 Intuitive and Topological Justification

While the previous analysis shows that an SMLP can efficiently approximate an RBFN, the transformation from an SMLP into an RBFN examined only a single special case of the type of features that can be formed by an SMLP node. In fact, a single SMLP node can form many other types of features besides hyper-sigmoids and local bumps. To show that this is indeed the case, one only needs to compose a two-dimensional diagonal quadratic equation into a sigmoidal function to see what type of surfaces are possible.

Figure 7.6 shows some familiar surfaces and decision boundaries that can be formed with a single SMLP node. As expected, hyper-sigmoids and bumps can be formed. What is interesting, however, is that there are many types of SMLP features that are neither local nor global. For example, a ridge, wedge or saddle may look like a local or global feature if it is projected onto one lower dimension;

however, whether this projection is local or global depends on the subspace that the projection is formed.

How important is it for an architecture to be able to form these features? The very fact that these types of boundaries and surfaces have names means that they are important enough that one may need a type of model that can efficiently form them. However, if we reexamine the two-spirals problem from Section 7.3.2 it is possible to dissect the decision boundary formed by the SMLP (shown in Figure 7.3) to see how the surface was formed. What is truly interesting is that radial boundaries, wedges, and a sigmoid were all crucial to forming the entire decision surface. If the SMLP lacked the ability to form any of these features, then it is easily possible that the hidden node requirements for this problem would explode.

There is, however, one caveat with the variety of features that can be formed with a single SMLP node. The wedge, ridge, ellipse and saddle structures shown in Figure 7.6 must always be aligned with one of the input axes. In other words, it is impossible to make a ridge that would run parallel to the line defined by $x = y$. We can see that this is true by noting that $(x - y)^2$ has the term $-2xy$ in its expansion, which means that the quadratic form is non-diagonal. In general, in order to have the ability to rotate all of the features in Figure 7.6, one would need the full quadratic form, thus requiring a higher-order network instead of the SMLP. Hence, while eliminating the off-diagonal terms in a higher-order network saves a considerable number of weights, there is a cost in the types of features that can be formed.

7.6 Conclusions

We have seen that there are problems whose solutions require features of both local and global scope. MLPs excel at forming global features but have a difficult time forming local features. RBFNs are exactly the opposite. The SMLP architecture can efficiently form both types of features with only a small penalty in the number of weights. However, the increase in the number of weights is more than compensated for by improvements in the network's ability to form features. This often results in simpler networks with fewer weights that can learn much faster and approximate more accurately.

For the two main numerical studies in this work, it was found that the SMLP architecture performed as well or better than the best known techniques for the two-spirals problem and the Deterding vowel recognition data. Moreover, these results are strengthened by the fact that the average performance of the SMLP was often superior to the best known results for the other techniques.

It was also found that the dual nature of the SMLP can be exploited in the form of hybrid algorithms. SMLPs can be “solved” like an RBFN, trained like an MLP, or both. It is also noteworthy that the nonlinear weights in an SMLP are only “moderately” nonlinear. For example, gradient-based learning algorithms can be used on RBFNs but it is known that the high degree of nonlinearity found in the weights which correspond to centers and widths can often make gradient

training very slow for RBFNs when the nonlinear weight are included. Similarly, since MLPs require two hidden layers of nodes to form local features efficiently, the first layer of weights in an MLP are exceedingly nonlinear because they are eventually passed through two layers of nonlinear nodes. Counter to this, the nonlinear nodes in an SMLP are only passed through a single layer of nonlinear nodes. Although it is unproven at this time, it seems like a reasonable conjecture that SMLP networks may be intrinsically better conditioned for gradient-based learning of local “bumps” than MLPs with two hidden layers.

Acknowledgements. I thank Frans Coetzee, Chris Darken, Lee Giles, Jenny Orr, Ray Watrous, and the anonymous reviewers for many helpful comments and discussions.

References

- [1] Casdagli, M.: Nonlinear prediction of chaotic time series. *Physica D* 35, 335–356 (1989)
- [2] Deterding, D.H.: Speaker Normalisation for Automatic Speech Recognition. PhD thesis, University of Cambridge (1989)
- [3] Fahlman, S.E.: Faster-learning variations on back-propagation: An empirical study. In: *Proceedings of the 1988 Connectionist Models Summer School*. Morgan Kaufmann (1988)
- [4] Fahlman, S.E., Lebiere, C.: The cascade-correlation learning architecture. In: Touretzky, S. (ed.) *Advances in Neural Information Processing Systems*, vol. 2. Morgan Kaufmann (1990)
- [5] Finke, M., Müller, K.-R.: Estimating a-posteriori probabilities using stochastic network models. In: Mozer, M., Smolensky, P., Touretzky, D.S., Elman, J.L., Weigend, A.S. (eds.) *Proceedings of the 1993 Connectionist Models Summer School*, pp. 324–331. Erlenbaum Associates, Hillsdale (1994)
- [6] Hastie, T., Tibshirani, R.: Flexible discriminant analysis by optimal scoring. Technical report, AT&T Bell Labs, Murray Hill, New Jersey (1993)
- [7] Hastie, T., Tibshirani, R.: Discriminant adaptive nearest neighbor classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 18(6), 607–616 (1996)
- [8] Hochreiter, S., Schmidhuber, J.: Lococode. Technical Report FKI-222-97, Fakultät für Informatik, Technische Universität München (1997)
- [9] Lang, K.J., Witbrock, M.J.: Learning to tell two spirals apart. In: *Proceedings of the 1988 Connectionist Models Summer School*. Morgan Kaufmann, San Francisco (1988)
- [10] Lapedes, A., Farber, R.: Nonlinear signal processing using neural networks: Prediction and system modelling. Technical Report LA-UR-87-2662, Los Alamos National Laboratory, Los Alamos, NM (1987)
- [11] Lapedes, A., Farber, R.: How neural nets work. In: Anderson, D.Z. (ed.) *Neural Information Processing Sysytems*, pp. 442–456. American Institute of Physics, New York (1988)
- [12] Lawrence, S., Tsui, A.C., Back, A.D.: Function approximation with neural networks and local methods: Bias, variance and smoothness. In: Bartlett, P., Burkitt, A., Williamson, R. (eds.) *Australian Conference on Neural Networks*, pp. 16–21. Australian National University (1996)

- [13] Lee, S., Kil, R.M.: Multilayer feedforward potential function networks. In: IEEE international Conference on Neural Networks, pp. 1:161–1:171. SOS Printing, San Diego (1988)
- [14] Lee, Y.C., Doolen, G., Chen, H.H., Sun, G.Z., Maxwell, T., Lee, H.Y., Giles, C.L.: Machine learning using higher order correlation networks. *Physica D* 22-D, 276–306 (1986)
- [15] Moody, J., Darken, C.: Learning with localized receptive fields. In: Touretsky, D., Hinton, G., Sejnowski, T. (eds.) *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann (1988)
- [16] Moody, J., Darken, C.: Fast learning in networks of locally-tuned processing units. *Neural Computation* 1, 281–294 (1989)
- [17] Niranjan, M., Fallside, F.: Neural networks and radial basis functions in classifying static speech patterns. *Computer Speech and Language* 4, 275–289 (1990)
- [18] Pao, Y.H.: *Adaptive Pattern Recognition and Neural Networks*. Addison-Wesley Publishing Company, Inc., Reading (1989)
- [19] Robinson, A.J.: *Dynamic Error Propagation Networks*. PhD thesis, Cambridge University (1989)
- [20] Rumelhart, D.E., McClelland, J.L.: the PDP Research Group. In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 2. MIT Press (1986)
- [21] Sarle, W.: *The comp.ai.neural-nets Frequently Asked Questions List* (1997)
- [22] Schetzen, M.: *The Volterra and Wiener Theories of Nonlinear Systems*. John Wiley and Sons, New York (1980)
- [23] Schölkopf, B., Smola, A., Müller, K.-R.: Nonlinear component analysis as a kernel eigenvalue problem. Technical report, Max-Planck-Institut für biologische Kybernetik, 1996. *Neural Computation* 10(5), 1299–1319 (1998)
- [24] Volterra, V.: *Theory of Functionals and of Integro-differential Equations*. Dover (1959)
- [25] Werbos, P.: *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University (1974)

A Dozen Tricks with Multitask Learning^{*}

Rich Caruana

Just Research and Carnegie Mellon University,
 4616 Henry Street, Pittsburgh, PA 15213
 caruana@cs.cmu.edu
<http://www.cs.cmu.edu/~caruana/>

Abstract. Multitask Learning is an inductive transfer method that improves generalization accuracy on a main task by using the information contained in the training signals of other *related* tasks. It does this by learning the extra tasks in parallel with the main task while using a shared representation; what is learned for each task can help other tasks be learned better. This chapter describes a dozen opportunities for applying multitask learning in real problems. At the end of the chapter we also make several suggestions for how to get the most our of multitask learning on real-world problems.

When tackling real problems, one often encounters valuable information that is not easily incorporated in the learning process. This chapter shows a dozen ways to benefit from the information that often gets ignored. The basic trick is to create extra tasks that get trained on the same net with the main task. This *Multitask Learning* is a form of inductive transfer¹ that improves performance on the main task by using the information contained in the training signals of other *related* tasks. It does this by learning the main task in parallel with the extra tasks while using a shared representation; what is learned for each task can help other tasks be learned better.

We use the term “task” to refer to a function that will be learned from a training set. We call the important task that we wish to learn better the **main task**. Other tasks whose training signals will be used by multitask learning to learn the main task better are the **extra tasks**. Often, we do not care how well extra tasks are learned. Their sole purpose is to help the main task be learned better. We call the union of the main task and the extra tasks a **domain**. Here we restrict ourselves to domains where the tasks are defined on a common set of input features, though some of the extra tasks may be functions of only a subset of these input features.

This chapter shows that most real-world domains present a number of opportunities for multitask learning (MTL). Because these opportunities are not

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

¹ Inductive transfer is the process of transferring anything learned for one problem to help learning of other related problems.

always obvious, most of the chapter is dedicated to showing different ways useful extra tasks arise in real problems. We demonstrate several of these opportunities using real data. The chapter ends with a few suggestions that help you get the most out of multitask learning. Some of these suggestions are so important that if you don't follow them, MTL can easily hurt performance on the main task instead of helping it.

8.1 Introduction to Multitask Learning in Backprop Nets

Consider the following boolean functions defined on eight bits, $B_1 \cdots B_8$:

$$\begin{aligned} Task1 &= B_1 \vee Parity(B_2 \cdots B_6) \\ Task2 &= \neg B_1 \vee Parity(B_2 \cdots B_6) \\ Task3 &= B_1 \wedge Parity(B_2 \cdots B_6) \\ Task4 &= \neg B_1 \wedge Parity(B_2 \cdots B_6) \end{aligned}$$

where " B_i " represents the i th bit, " \neg " is logical negation, " \vee " is disjunction, " \wedge " is conjunction, and " $Parity(B_2 \cdots B_6)$ " is the parity of bits 2–6. Bits B_7 and B_8 are not used by the functions. These four tasks are related in several ways:

- they are all defined on the same inputs, bits $B_1 \cdots B_8$;
- they all ignore the same inputs, bits B_7 and B_8 ;
- each uses a common computed subfeature, $Parity(B_2 \cdots B_6)$;
- when $B_1 = 0$, Task 1 needs $Parity(B_2 \cdots B_6)$, but Task 2 does not, and vice versa;
- as with Tasks 1 and 2, when Task 3 needs $Parity(B_2 \cdots B_6)$, Task 4 does not need it, and vice versa.

We can train artificial neural nets on these tasks with backprop. Bits $B_1 \cdots B_8$ are the inputs to the net. The task values computed by the four functions are the target outputs. We create a data set by enumerating all 256 combinations of the eight input bits, and computing for each setting of the bits the task signals for Tasks 1, 2, 3, and 4 using the definitions above. This yields 256 different cases, with four different training signals for each case.

8.1.1 Single and Multitask Learning of Task 1

Consider Task 1 the main task. Tasks 2, 3, and 4 are the extra tasks. That is, we are interested only in improving the accuracy of models trained for Task 1. We've done an experiment where we train Task 1 on the three nets shown in Figure 8.1. All the nets are fully connected feed-forward nets with 8 inputs, 100 hidden units, and 1–4 outputs. Where there are multiple outputs, each output is fully connected to the hidden units. Nets were trained in batch mode using backprop with MITRE's Aspirin/MIGRAINES 6.0 with learning rate = 0.1 and momentum = 0.9.

Task 1 is trained alone on the net on the left of Figure 8.1. This is a backprop net trained on a single task. We refer to this as single task learning (STL) or single task backprop (STL-backprop). The net in the center of Figure 8.1 trains Task 1 on a net that is also trained on Task 2. The hidden layer of this net is shared by Tasks 1 and 2. This is multitask backprop (MTL-backprop) with two tasks. The net on the right side of Figure 8.1 trains Task 1 with Tasks 2, 3, and 4. The hidden layer of this net is shared by all four tasks. This is MTL-backprop with four tasks. How well will Task 1 be learned by the different nets?

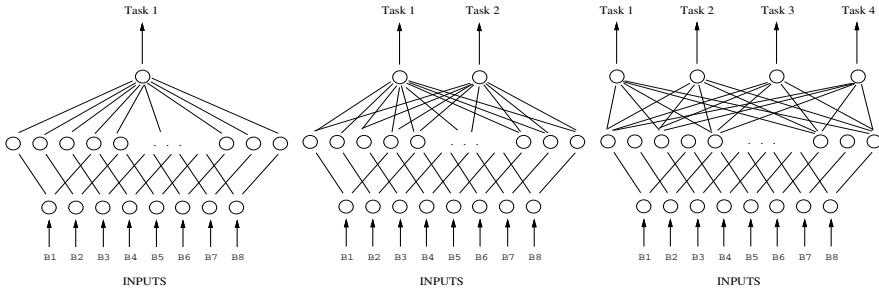


Fig. 8.1. Three Neural Net Architectures for Learning Task 1

We performed 25 independent trials by resampling training and test sets from the 256 cases. From the 256 cases, we randomly sample 128 cases for the training set, and use the remaining 128 cases as a test set. (For now we ignore the complexity of early stopping, which can be tricky with MTL nets. See section 8.3.2 for a thorough discussion of early stopping in MTL nets.)

For each trial, we trained three nets: an STL net for Task 1, an MTL net for Tasks 1 and 2, and an MTL net for Tasks 1–4. We measure performance only on the output for Task 1. When there are extra outputs for Task 2 or Tasks 2–4, these are trained with backprop, but ignored when the net is evaluated. The sole purpose of the extra outputs is to affect what is learned in the hidden layer these outputs share with Task 1.

8.1.2 Results

Every 5000 epochs we evaluated the performance of the nets on the test set. We measured the RMS error of the output with respect to the target values, the criterion being optimized by backprop. We also measured the accuracy of the output in predicting the boolean function values. If the net output is less than 0.5, it was treated as a prediction of 0, otherwise it was treated as a prediction of 1.

Figure 8.2 shows the RMSE for Task 1 on the test set during training. The three curves in the graph are each the average of 25 trials.² RMSE on the main task, Task 1, is reduced when Task 1 is trained on a net simultaneously trained on other related tasks. RMSE is reduced when Task 1 is trained with extra Task 2, and is further reduced when extra Tasks 3 and 4 are added. *Training multiple tasks on one net does not increase the number of training patterns seen by the net. Each net sees exactly the same training cases. The MTL nets do not see more training cases; they receive more training signals with each case.*

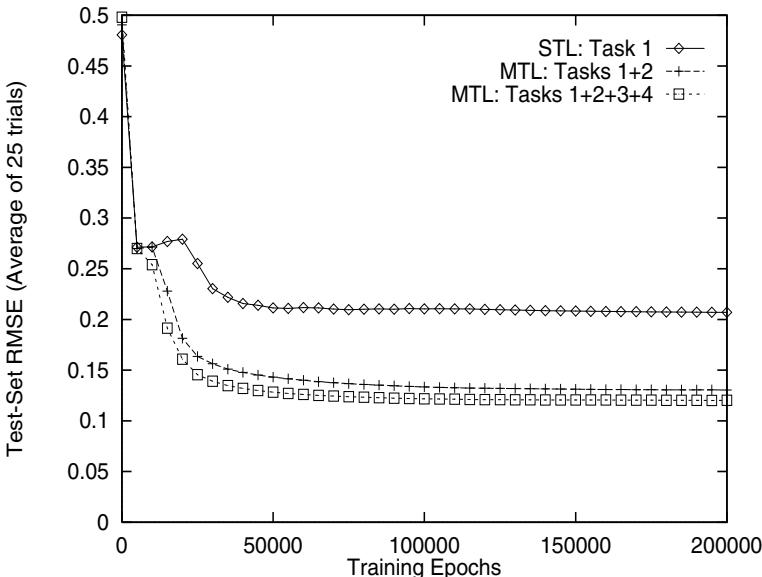


Fig. 8.2. RMSE Test-set Performance of Three Different Nets on Task 1

Figure 8.3 shows the average test-set accuracy on Task 1 for 25 trials with the three different nets. Task 1 has boolean value 1 about 75% of the time. A simple learner that learned to predict 1 all the time should achieve about 75% accuracy on Task 1. When trained alone (STL), performance on Task 1 is about 80%. When Task 1 is trained with Task 2, performance increases to about 88%. When Task 1 is trained with Tasks 2, 3, and 4, performance increases further to about 90%. Table 8.1 summarizes the results of examining the training curve from each trial.

² Average training curves can be misleading, particularly if training curves are not monotonic. For example, it is possible for method A to always achieve better error than method B, but for the average of method A to be everywhere worse than the average of method B because the regions where performance on method A is best do not align, but do align for method B. Before presenting average training curves, we always examine the individual curves to make sure the average curve is not misleading.

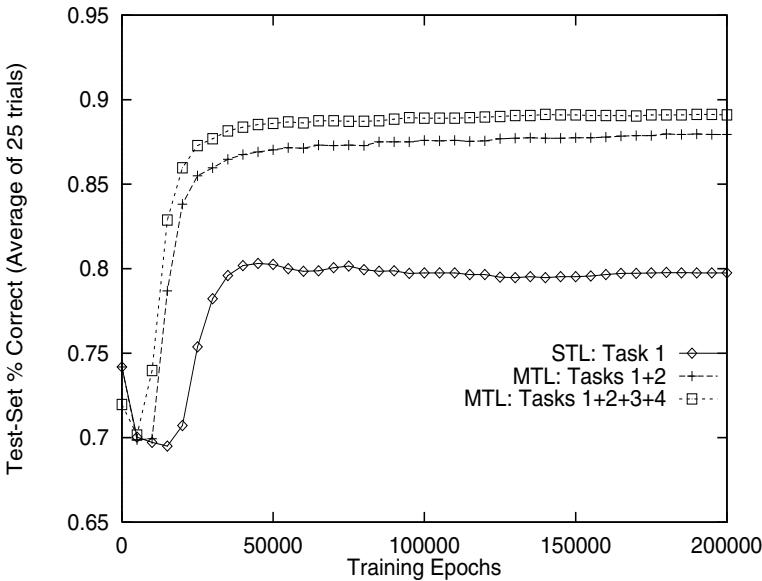


Fig. 8.3. Test-set Percent Correct of Three Different Nets on Task 1

Table 8.1. Test-set performance on Task 1 of STL of Task 1, MTL of Tasks 1 and 2, and MTL of Tasks 1, 2, 3, and 4. *** indicates performance is statistically better than STL at .001 or better, respectively.

NET	STL: 1	MTL: 1+2	MTL: 1+2+3+4
Root-Mean-Squared-Error	0.211	0.134 ***	0.122 ***
Percent Correct	79.7%	87.5% ***	88.9% ***

8.1.3 Discussion

Why is the main task learned better if it is trained on a net learning other related tasks at the same time? We ran a number of experiments to verify that the performance increase with MTL is due to the fact that the tasks are related, and not just a side effect of training multiple outputs on one net.

Adding noise to neural nets sometimes improves their generalization performance [22]. To the extent that MTL tasks are *uncorrelated*, their contribution to the aggregate gradient may appear as noise to other tasks and this might improve generalization. To see if this explains the benefits we see from MTL, in one experiment we train Task 1 on a net with three *random* tasks.

A second effect to be concerned about is that adding tasks tends to increase the effective learning rate on the input-to-hidden layer weights because the gradients from the multiple outputs add at the hidden layer, and this might favor nets with multiple outputs. To test this, we train an MTL net with four copies of Task 1. Each of the four outputs receives exactly the same training signal. This is a

degenerate form of MTL where no extra information is given to the net by the extra tasks.

A third effect that needs to be ruled out is net capacity. 100 hidden units is a lot for these tasks. Does the MTL net, which has to share the 100 hidden units among four tasks, generalize better because each task has fewer hidden units? To test for this, we train Task 1 on STL nets with 200 hidden units and with 25 hidden units. This will tell us if generalization would be better with more or less capacity.

Finally, we ran a fourth experiment based on the heuristic used in [37]. We shuffle the training signals (the target output values) for Tasks 2, 3, and 4 before training an MTL net on the four tasks. Shuffling reassigns the target values to the input vectors in the training set for Tasks 2, 3, and 4. The main task, Task 1, is not affected. The distributions of the training signals for outputs 2–4 have not changed, but the training signals are no longer related to Task 1. This is a powerful test that has the potential to rule-out many mechanisms that do not depend on relationships between the tasks.

We ran each experiment 25 times using exactly the same data sets used in the previous section. Figure 8.4 shows the generalization performance on Task 1 in the four experiments. For comparison, the performance of of STL, MTL with Tasks 1 and 2, and MTL with Tasks 1–4 from the previous section are also shown.

When Task 1 is trained with random extra tasks, performance on Task 1 drops below the performance on Task 1 when it is trained alone on an STL net. We conclude MTL of Tasks 1–4 probably does not learn Task 1 better by adding noise to the learning process through the extra outputs.

When Task 1 is trained with three additional copies of Task 1, the performance is comparable to that when Task 1 is trained alone with STL.³ We conclude that MTL does not learn Task 1 better just because backprop works better with multiple outputs.

When Task 1 is trained on an STL net with 25 hidden units, performance is comparable to the performance with 100 hidden units. Moreover, when Task 1 is trained on an STL net with 200 hidden units, it is slightly better. (The differences between STL with 25, 100, and 200 hidden units are not statistically significant.) We conclude that performance on Task 1 is relatively insensitive to net size for nets between 25 and 200 hidden units, and, if anything, Task 1 would benefit from a net with more capacity, not one with less capacity. Thus it is unlikely that MTL on Tasks 1–4 performs better on Task 1 because Tasks 2–4 are using up extra capacity that is hurting Task 1.

³ We sometimes observe that training multiple copies of a task on one net does improve performance. When we have observed this, the benefit is never large enough to explain away the benefits observed with MTL. But it is interesting and surprising, as the improvement is gained without any additional information being given to the net. The most likely explanation is that the multiple connections to the hidden layer allow different hidden layer predictions to be averaged and thus act as a weak boosting mechanism.

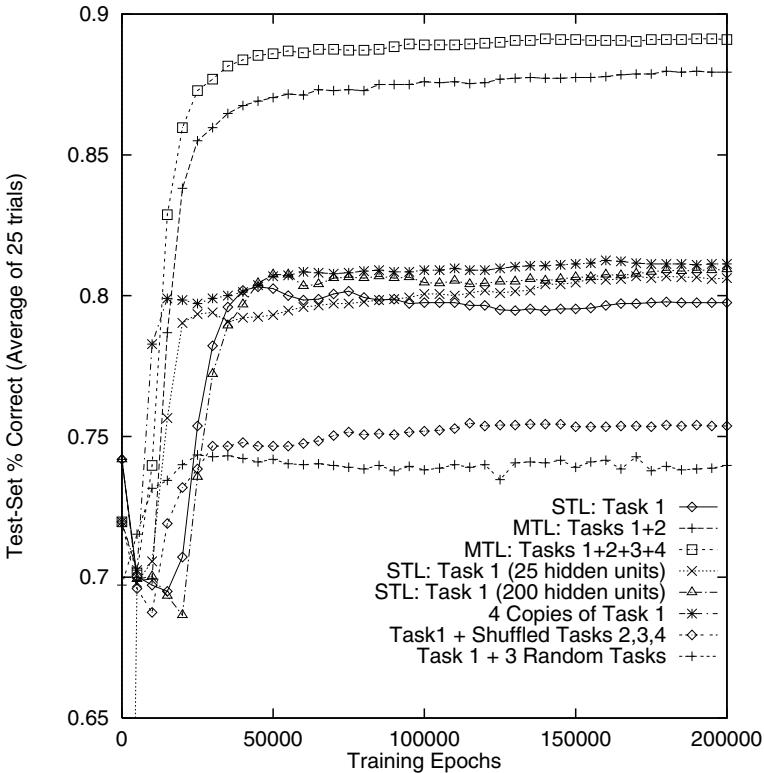


Fig. 8.4. RMSE test-set performance of Task 1 when trained with: MTL with three random tasks; MTL with three more copies of Task 1; MTL with shuffled training signals for Tasks 2–4; STL on nets with 25 or 200 hidden units.

When Task 1 is trained with training signals for Tasks 2–4 that have been shuffled, the performance of MTL drops below the performance of Task 1 trained alone on an STL net. Clearly the benefit we see with MTL on these problems is not due to some accident caused by the distribution of the extra outputs. The extra outputs must be *related* to the main task to help it.

These experiments rule out most explanations for why MTL outperforms STL on Task 1 that do not require Tasks 2–4 be related to Task 1. So why is the main task learned better when trained in parallel with Tasks 2–4?

One reason is that Task 1 needs to learn the subfeature $\text{Parity}(B_2 \cdots B_6)$ that it shares with Tasks 2–4. Tasks 2–4 give the net information about this subfeature that it would not get from Task 1 alone. For example, when $B_1 = 1$, the training signal for Task 1 contains no information about $\text{Parity}(B_2 \cdots B_6)$. We say B_1 *masks* $\text{Parity}(B_2 \cdots B_6)$ when $B_1 = 1$. But the training signals for Task 2 provide information about the Parity subfeature in exactly those cases where Task 1 is masked. Thus the hidden layer in a net trained on both Tasks 1 and 2 gets twice as much information about the Parity subfeature as a net

trained on one of these tasks, despite the fact that they see exactly the same training cases. The MTL net is getting more information with each training case.

Another reason why MTL helps Task 1 is that all the tasks are functions of the same inputs, bits $B_1 \dots B_6$, and ignore the same inputs, B_7 and B_8 . Because the tasks overlap on the features they use and don't use, the MTL net is better able select which input features to use.

A third reason why MTL helps Task 1 is that there are relationships between the way the different tasks use the inputs that promote learning good internal representations. For example, all the tasks logically combine input B_1 with a function of inputs $B_2 \dots B_6$. This similarity tends to prevent the net from learning internal representations that, for example, directly combine bits B_1 and B_2 . A net trained on all the tasks together is biased to learn more modular, in this case more correct, internal representations that support the multiple tasks. This bias towards modular internal representations reduces the net's tendency to learn spurious correlations that occur in any finite training sample: there may be a random correlation between bit B_3 and the output for Task 1 that looks fairly strong in this one training set, but if that spurious correlation does not also help the other tasks, it is less likely to be learned.

8.2 Tricks for Using Multitask Learning in the Real World

The previous section introduced multitask learning (MTL) in backprop nets using four tasks carefully designed to have relationships that make learning them in parallel work better than learning them in isolation. How often will real problems present extra tasks that allow multitask learning to improve performance on the main task?

This section shows that many real world problems yield opportunities for multitask learning. We present a dozen prototypical real-world applications where the training signals for related extra tasks are available and can be leveraged. We believe most real-world problems fall into one or more of these prototypical classes. This claim might sound surprising given that few of the test problems traditionally used in machine learning are multitask problems. We believe most of the problems used in machine learning so far have been heavily preprocessed to fit the single task learning mold. Most of the opportunities for MTL in these problems were eliminated as the problems were defined.

8.2.1 Using the Future to Predict the Present

Often valuable features become available *after* predictions must be made. If learning is done offline, these features can be collected for the training set and used for learning. These features can't be used as inputs, because they will not be available when making predictions for future test cases. They can, however, be used as extra outputs for multitask learning. The predictions the learner makes for these extra tasks will be ignored when the system is used to make

predictions for the main task. Their sole function is to provide extra information to the learner during training so that it can learn the main task better.

One source of applications of learning from the future is sequential decision making in medicine. Given the initial symptoms, decisions are made about what tests to make and what treatment to begin. New information becomes available when the tests are completed and as the patient responds (or fails to respond) to the treatment. From this new information, new decisions are made. Should more tests be made? Should the treatment be changed? Has the patient's condition changed? Is this patient now high risk, or low risk? Does the patient need to be hospitalized? Etc.

When machine learning is applied to early stages in the decision making process, only those input features that typically would be available for patients at this stage of the process are usually used. This is unfortunate. In an historical database, all of the patients may have run the full course of medical testing and treatment and their final outcome may be known. Must we ignore the results of lab tests and other valuable features in the database just because these will not be available for patients at the stage of medical decision making for which we wish to learn a model?

The Pneumonia Risk Prediction Problem. Consider pneumonia. There are 3,000,000 cases of pneumonia each year in the U.S., 900,000 of which get hospitalized. Most pneumonia patients recover given appropriate treatment, and many can be treated effectively without hospitalization. Nonetheless, pneumonia is serious: 100,000 of those hospitalized for pneumonia die from it, and many more are at elevated risk if not hospitalized.

Consider the problem of predicting a patient's risk from pneumonia before they are hospitalized. (The problem is not to diagnose if the patient has pneumonia, but to determine how much risk the pneumonia poses to the patient.) A primary goal in medical decision making is to accurately, swiftly, and economically identify patients at high risk from diseases like pneumonia so that they may be hospitalized to receive aggressive testing and treatment; patients at low risk may be more comfortably, safely, and economically treated at home.

Some of the most useful features for assessing risk are the lab tests that become available only after a patient is hospitalized. It is the *extra* lab tests made after patients are admitted to the hospital that we use as extra tasks for MTL; they cannot be used as inputs because they will not be available for most future patients when making the decision to hospitalize.⁴

The most useful decision aid for this problem would be to predict which patients will live or die. This is too difficult. In practice, the best that can be achieved is to estimate a probability of death (POD) from the observed symptoms. In fact, it is sufficient to learn to order patients by POD so lower-risk patients can be discriminated from higher risk patients; patients at least risk may then be considered for outpatient care.

⁴ Other researchers who tackled this problem ignored the the lab tests because they knew they would not be available at run time and did not see ways to use them other than as inputs.

The performance criteria used by others working with this database [15] is the accuracy with which one can select prespecified fractions of a patient population who will live. For example, given a population of 10,000 patients, find the 20% of this population at *least* risk. To do this we learn a risk model, and a threshold for this risk model, that allows 20% of the population (2000 patients) to fall below it. If 30 of the 2000 patients below this threshold die, the error rate is $30/2000 = 0.015$. We say that the error rate for FOP 0.20 is 0.015 (FOP stands for “fraction of population”). Here we consider FOPs 0.1, 0.2, 0.3, 0.4, and 0.5. Our goal is to learn models and thresholds such that the error rate at each FOP is minimized.

Multitask Learning and Pneumonia Risk Prediction. The straightforward approach to this problem is to use backprop to train an STL net to learn to predict which patients live or die, and then use the real-valued predictions of this net to sort patients by risk. This STL net has 30 inputs for the 30 basic pre-hospitalization measurements, a single hidden layer, and a single output trained with targets 0=lived, 1=died.⁵ Given a large training set, a net trained this way should learn to predict the probability of death for each patient, not which patients live or die. If the training sample is small, the net will overfit and learn a very nonlinear function that outputs values near 0/1 for cases in the training set, but which does not generalize well. It is critical to use early stopping to halt training before this happens.

We developed a method called *Rankprop* specifically for this domain. Rankprop learns to rank patients without learning to predict mortality (0=lived,1=died). Figure 8.5 compares the performance of squared error on 0/1 targets with rankprop on this problem. Rankprop outperforms traditional backprop using squared error on targets 0=lived,1=died by 10%-40% on this domain, depending on which FOP is used for comparison. See [9] for details about rankprop.⁶

There are 35 future lab values that we use as extra backprop *outputs*, as shown in Figure 8.6. The expectation is that these extra outputs will bias the shared hidden layer toward representations that better capture important features of each patient’s condition, and that this will lead to more accurate predictions of patient risk at the main task output.

The STL net has 30 inputs, 8 hidden units, and one output trained to predict risk with rankprop. The MTL net has the same 30 inputs, 64 hidden units, one output for rankprop, and 35 extra outputs trained with squared error. (Preliminary experiments suggested 8–32 hidden units was optimal for STL, and that MTL performs best with nets as large as 512 hidden units. We used 8 and 64 hidden units so that we could run many experiments.) The 35 extra outputs on

⁵ We tried both squared error and cross entropy. The difference between the two was small. Squared error performed slightly better.

⁶ We use rankprop for the rest of our experiments on this domain because it is the best performer we know of on this problem. We want to see if MTL can make the best method better.

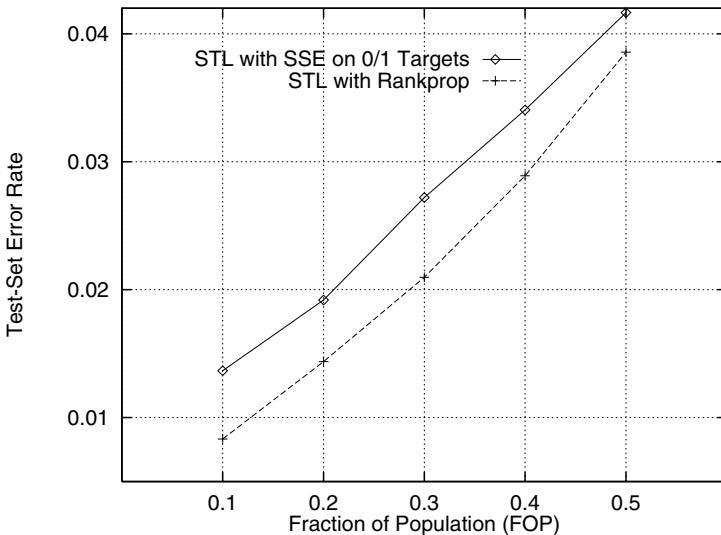


Fig. 8.5. The performance of SSE (0/1 targets) and rankprop on the 5 FOPs in the pneumonia domain. Lower error indicates better performance.

the MTL net (see Figure 8.6) are trained at the same time the net is trained to predict risk.

We train the net using training and validation sets containing 1000 patients randomly drawn from the database. Training is halted on both the STL and MTL nets when overfitting is observed on the main rankprop risk task. On the MTL net, the performance of the extra tasks is not taken into account for early stopping. Only the performance of the output for the main task is considered when deciding when to stop training. (See section 8.3.2 for more discussion of early stopping with MTL nets.) Once training is halted, the net is tested on the remaining unused patients in the database.

Results. Table 8.2 shows the mean performance of ten runs of rankprop using STL and MTL. The bottom row shows the percent improvement in performance obtained on this problem by using the future lab measurements as extra MTL outputs. Negative percentages indicate MTL reduces error. Although MTL lowers the error at each FOP compared with STL, only the differences at FOP 0.3, 0.4, and 0.5 are statistically significant with ten trials using a standard t-test.

The improvement from MTL is 5–10%. This improvement can be of considerable consequence in medical domains. To verify that the benefit from MTL is due to relationships between what is learned for the future labs and the main task, we ran the shuffle test (see section 8.1.3). We shuffled the training signals for the extra tasks in the training sets before training the nets with MTL.

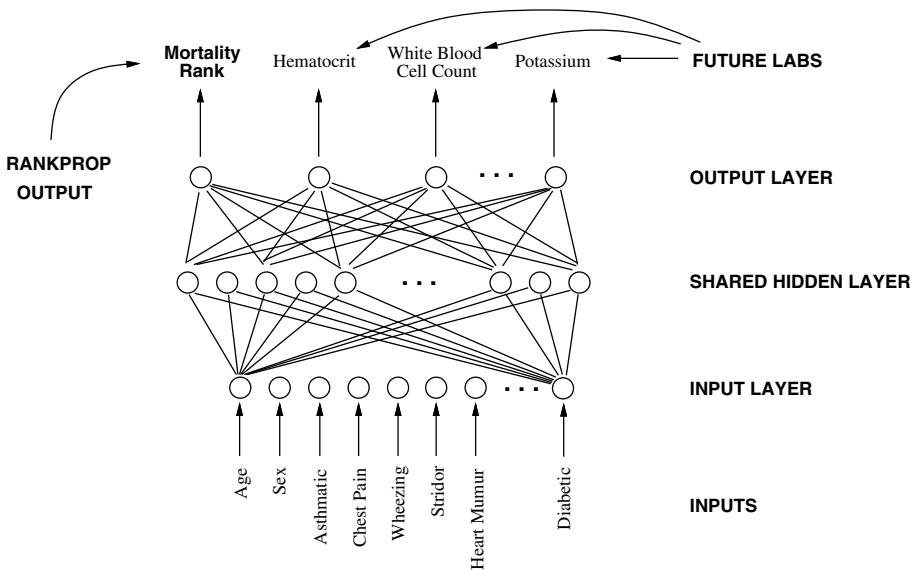


Fig. 8.6. Using future lab results as extra outputs to bias learning for the main risk prediction task. The lab tests would help most if they could be used as inputs, but will not yet have been measured when risk must be predicted, so we use them as extra outputs for MTL instead.

Table 8.2. Error Rates (fraction deaths) for STL with Rankprop and MTL with Rankprop on Fractions of the Population predicted to be at low risk (FOP) between 0.0 and 0.5. MTL makes 5–10% fewer errors than STL.

FOP	0.1	0.2	0.3	0.4	0.5
STL Rankprop	.0083	.0144	.0210	.0289	.0386
MTL Rankprop	.0074	.0127	.0197	.0269	.0364
% Change	-10.8%	-11.8%	-6.2% *	-6.9% *	-5.7% *

Figure 8.7 shows the results of MTL with shuffled training signals for the extra tasks. The results of STL and MTL with unshuffled extra tasks are also shown. Shuffling the training signals for the extra tasks reduces the performance of MTL below that of STL. We conclude that it is the relationship between the main task and the extra tasks that lets MTL perform better on the main task; the benefit disappears when these relationships are broken by shuffling the extra task signals.

We have also run experiments where we use the future lab tests as inputs to a net trained to predict risk, and impute the values for the lab tests when they are missing on future test cases. Imputing missing values for the lab tests did not yield performance comparable to MTL on this problem. Similar experiments with feature nets [17] also failed to yield improvements comparable to MTL.

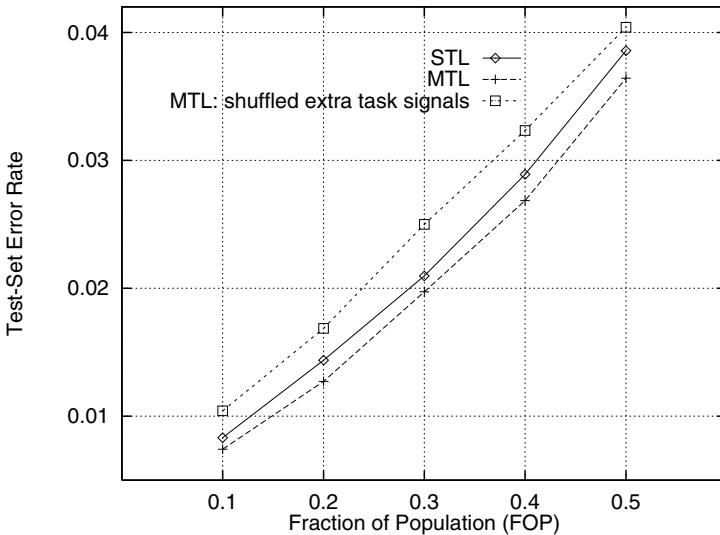


Fig. 8.7. Performance of STL, MTL, and MTL with shuffled extra task signals on pneumonia risk prediction at the five FOPs

Future measurements are available in many *offline* learning problems. As just one very different example, a robot or autonomous vehicle can more accurately measure the size, location, and identity of objects if it passes nearer them in the future. For example, road stripes and the edge of the road can be detected reliably as a vehicle passes alongside them, but detecting them far ahead of the vehicle is hard. Since driving brings future road closer to the car, stripes and road borders can be measured accurately as the car passes them. Dead reckoning allows these future measurements to be added to the training set. They can't be used as *inputs*; They won't be available in time while driving. But they can be used to augment a training set. We suspect that using future measurements as extra outputs will be a frequent source of extra tasks in real problems.

8.2.2 Multiple Metrics

Sometimes it is hard to capture everything that is important in one error metric. When alternate metrics capture different, but useful, aspects of a problem, MTL can be used to benefit from the multiple metrics. One example of this is the pneumonia problem in the previous section. Rankprop outperforms backprop using traditional squared error on this problem, but has trouble learning to rank cases at such low risk that virtually all patients survive because these cases provide little ordering information. Interestingly, squared error performs best when cases have high purity, such as in regions of feature space where most cases have low risk. *Squared error is at its best where rankprop is weakest.* Adding an extra output trained with squared error to a net learning to predict

pneumonia risk with rankprop improves the accuracy of the rankprop output an additional 5-10% for the least-risk cases. The earliest example of using multiple output representations we know of is [38] which uses both SSE and cross-entropy outputs for the same task.

8.2.3 Multiple Output Representations

Sometimes it is not apparent what output encoding is best for a problem. Distributed output representations often help *parts* of a problem be learned well because the parts have separate error gradients. But non-distributed output representations are sometimes more accurate. Consider the problem of learning to classify a face as one of twenty faces. One output representation is to have one output code for each face. Another representation is to have outputs code for features such as beard/no_beard, glasses/no_glasses, long_hair/short, eye_color(blue, brown), male/female, that are sufficient to distinguish the faces. Correct classification, however, may require that each feature be correctly predicted. The non-distributed output coding that uses one output for each individual may be more reliable. But training the net to recognize specific traits should help, too. MTL is one way to merge these conflicting requirements in one net by using both output representations, even if only one representation will be used for prediction.

A related approach to multiple output encodings is error correcting codes [18]. Here, multiple encodings for the outputs are designed so that the combined prediction is less sensitive to occasional errors in some of the outputs. It is not clear how much ECOC benefits from MTL-like mechanisms. In fact, ECOC may benefit from being trained on STL nets (instead of MTL nets) so that different outputs do not share the same hidden layer and thus are less correlated. But see [27] for ways of using MTL to *decorrelate* errors in multiple outputs to boost committee machine performance.

8.2.4 Time Series Prediction

The simplest way to use MTL for time series prediction is to use a single net with multiple outputs, each output corresponding to the same task at a different time. This net makes predictions for the same task at different times. We tested this on a robot domain where the goal is to predict what the robot will sense 1, 2, 4, and 8 meters in the future as it moves forward. Training all four of these distances on one MTL net improved the accuracy of the long range predictions about 10% (see chapter 17 where MTL is used in a time series application).

8.2.5 Using Non-operational Features

Some features are impractical to use at run time, either because they are too expensive to compute, or because they need human expertise that won't be available. We usually have more time, however, to prepare our training sets.

When it is impractical to compute some features on the fly at run time, but practical to compute them for the training set, these features can be used as extra outputs to help learning. Pattern recognition provides a good example of this. We tested MTL on a door recognition problem where the goal is to recognize doorways and doorknobs. The extra tasks were features such as the location of door edges and doorway centers that required laborious hand labelling that would not be applied to the test set. The MTL nets that were trained to predict these additional hand-labelled features were 25% more accurate at locating doors and doorknobs. Other domains where hand-labelling can be used to augment training sets this way include text domains, medical domains, acoustic domains, and speech domains.

8.2.6 Using Extra Tasks to Focus Attention

Learning often uses large, ubiquitous patterns in the inputs, while ignoring small or less common inputs that might also be useful. MTL can be used to coerce the learner to attend to patterns in the input it would otherwise ignore. This is done by forcing it to learn internal representations to support related tasks that depend on these patterns.

A good example is road following. Here, STL nets often ignore lane markings when learning to steer because lane markings are usually a small part of the image, are constantly changing, and are often difficult to see (even for humans). If a net learning to steer is also required to learn to recognize road stripes, the net will learn to attend to those parts of the image where stripes occur. To the extent that the stripe tasks are learnable, the net will develop internal representations to support them. Since the net is also learning to steer using the same hidden layer, the steering task can use the parts of the stripe hidden representation that are useful for steering.

We tested this idea using a road image simulator developed by Pomerleau to permit rapid testing of learning methods for road-following domains [28]. Figure 8.8 shows several 2-D road images.

The principal task is to predict steering direction. For the MTL experiments, we used eight additional tasks:

- whether the road is one or two lanes
- location of left edge of road
- location of road center
- intensity of region bordering road
- location of centerline (if any)
- location of right edge of road
- intensity of road surface
- intensity of centerline (if any)

These additional tasks are all computable from the internal variables in the simulator. Table 8.3 shows the average performance of ten runs of single and multitask learning on each of these tasks. The MTL net has 32 inputs, 16 hidden units, and 9 outputs. The 36 STL nets have 32 inputs, 2, 4, 8 or 16 hidden units, and 1 output each.

The last two columns compare STL and MTL. The first is the percent reduction in error of MTL over the best STL run. Negative percentages indicate MTL

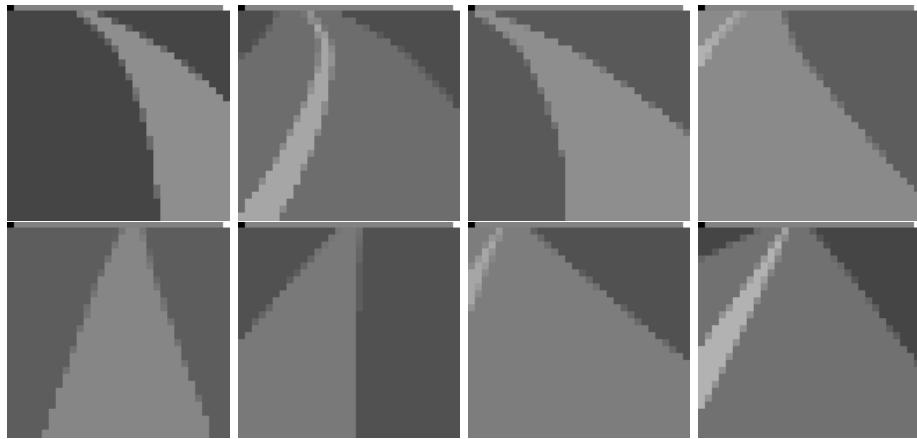


Fig. 8.8. Sample single and two lane roads generated with Pomerleau’s road simulator

performs better. The last column is the percent improvement of MTL over the average STL performance. On the important steering task, MTL outperforms STL 15–30%.

We ran a follow-up experiment to test how important centerstripes are to the STL and MTL nets. We eliminated the stripes from the images in a test set. If MTL learned more about centerstripes than STL, and uses what it learned about centerstripes for the main steering task, we expect to see steering performance degrade more for MTL than for STL when we remove the centerstripes from the images. Error increased more for the MTL nets than for the STL nets, suggesting the MTL nets are making more use of the stripes in the images.

Table 8.3. Performance of STL and MTL on the road following domain. The underlined entries in the STL columns are the STL runs that performed best. Differences statistically significant at .05 or better are marked with an *.

TASK	ROOT-MEAN SQUARED ERROR ON TEST SET							
	Single Task Backprop (STL)				MTL		Change MTL to Best STL	Change MTL to Mean STL
	2HU	4HU	8HU	16HU	16HU			
1 or 2 Lanes	.201	.209	.207	.178	.156	-12.4% *	-21.5% *	
Left Edge	<u>.069</u>	.071	.073	.073	<u>.062</u>	-10.1% *	-13.3% *	
Right Edge	.076	.062	.058	<u>.056</u>	<u>.051</u>	-8.9% *	-19.0% *	
Line Center	.153	<u>.152</u>	.152	.152	<u>.151</u>	-0.7%	-0.8%	
Road Center	.038	<u>.037</u>	.039	.042	<u>.034</u>	-8.1% *	-12.8% *	
Road Greylevel	<u>.054</u>	.055	.055	.054	<u>.038</u>	-29.6% *	-30.3% *	
Edge Greylevel	<u>.037</u>	.038	.039	.038	<u>.038</u>	2.7%	0.0%	
Line Greylevel	.054	.054	<u>.054</u>	.054	<u>.054</u>	0.0%	0.0%	
Steering	.093	<u>.069</u>	.087	.072	<u>.058</u>	-15.9% *	-27.7% *	

8.2.7 *Hints:* Tasks Hand-Crafted by a Domain Expert

Extra outputs can be used to *inject rule hints* into nets about what they should learn [32, 33]. This is MTL where the extra tasks are carefully engineered to coerce the net to learn specific internal representations. Hints can also be provided to backprop nets via extra terms in the error signal backpropagated for the main task output [1, 2]. The extra error terms constrain what is learned to satisfy desired properties of main task such as monotonicity [31], symmetry, or transitivity with respect to certain sets of inputs. MTL, which does not use extra error terms on task outputs, could be used in concert with these techniques.

8.2.8 Handling *other* Categories in Classification

In real-world applications of digit recognition, some of the images given to the classifier may be alphabetic characters or punctuation marks instead of digits. One way to prevent accidentally classifying a “t” as a one or seven is to create an “other” category that is the correct classification for non-digit images. The large variety of characters mapped to this “other” class makes learning this class potentially very difficult. MTL suggests an alternate way to do this. Split the “other” class into separate classes for the individual characters that are trained in parallel with the main digit tasks. A single output coding for the “other” class can be used, as well. Breaking the “other” category into multiple tasks gives the net more learnable error signal for these cases [26].

8.2.9 Sequential Transfer

MTL is parallel transfer. Often tasks arise serially and we can’t wait for all of them to begin learning. In these cases we can use parallel transfer to perform sequential transfer. If the training data can be stored, do MTL using whatever tasks are available when it is time to start learning, and re-train as new tasks or new data arise. If training data cannot be stored, or if we already have models for which data is not available, we can still use MTL. Use the models to generate synthetic data that is then used as extra training signals. This approach to sequential transfer avoids catastrophic interference (forgetting old tasks while learning new ones). Moreover, it is applicable where the analytical methods of evaluating domain theories required by some serial transfer methods [29, 34] are not available. For example, the domain theory need not be differentiable, it only needs to make predictions. One issue that arises when synthesizing data from prior models is what distribution to sample from. See [16] for a discussion of synthetic sampling.

8.2.10 Similar Tasks With Different Data Distributions

Sometimes there are multiple instances of the same problem, but the distribution of samples differs for each instantiation. For example, most hospitals diagnose

and treat the same diseases, but the demographics of the patients each hospital serves is different. Hospitals in Florida see older patients, urban hospitals see poorer patients, etc. Models trained separately for each hospital would perform best, but often there is insufficient data to train a separate model for each hospital. Pooling the data, however, may not lead to models that are accurate for each hospital. MTL provides one solution to this problem. Use one net to make predictions for each hospital, using a different output on the net for each hospital. Because each patient is a training case for only one hospital, error can be backpropagated only through the one output that has a target value for each input vector.

8.2.11 Learning with Hierarchical Data

In many domains, the data falls in a hierarchy of classes. Most applications of machine learning to hierarchical data make little use of the hierarchy. MTL provides one way of exploiting hierarchical information. When training a model to classify data at one level in the hierarchy, include as extra tasks the classification tasks that arise for ancestors, descendants, and siblings of the current classification task. The easiest way to accomplish this is to train one MTL net to predict all class distinctions in the hierarchy at the same time.

8.2.12 Some Inputs Work Better as Outputs

The common practice in backprop nets is to use all features that will be available for test cases as inputs, and have outputs only for tasks that need to be predicted. On real problems, however, learning often works better given a carefully selected subset of the features to use as inputs[7, 23, 24]. One way to benefit from features not used as inputs is to use them as extra outputs for MTL. We've done experiments with both synthetic and real problems where moving some features from the input side of the net to the output side of the net improves performance on the main task. We use feature selection to select those features that should be used as inputs, and then treat some of the remaining features as extra tasks.

Figure 8.9 shows the ROC Area on a pneumonia problem as the number of input features on the backprop net varies.⁷ ROC Areas closer to 1 indicate better performance. There are 192 features available for most patients. Using all 192 features as inputs (Net1) is suboptimal. Better performance is obtained by using the first 50 features selected with feature selection (Net2). The horizontal line at the top of the graph (Net3) shows the ROC Area obtained by using the first 50 features as inputs, and the *next* 100 features as extra *outputs*. Using these same 150 features all as inputs (Net4) yields worse performance.⁸

⁷ This is not the same pneumonia problem used in section 8.2.1.

⁸ Although the 95% confidence intervals for Net2 and Net3 overlap with ten trials, a paired t-test shows the results are significant at .01.

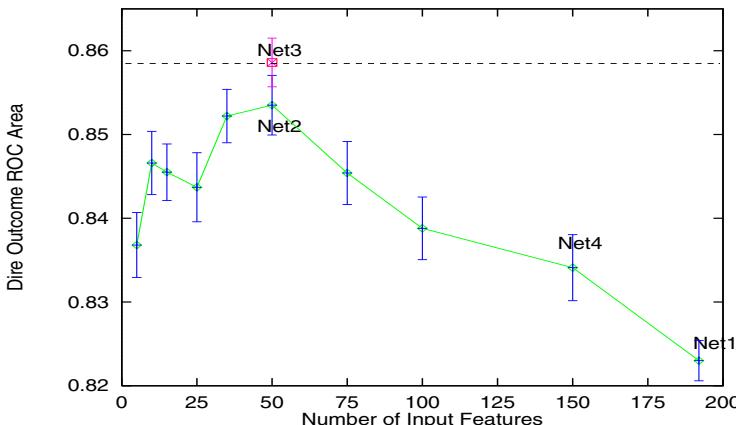


Fig. 8.9. ROC Area on the Pneumonia Risk Prediction Task vs. the number of input features used by the backprop net

8.3 Getting the Most Out of MTL

The basic machinery for doing multitask learning in neural nets is present in backprop. Backprop, however, was not designed to do MTL well. This chapter presents suggestions for how to make MTL in backprop nets work better. Some of the suggestions are counterintuitive, but if not used, can cause MTL to hurt generalization on the main task instead of helping it.

8.3.1 Use Large Hidden Layers

The basic idea behind MTL in backprop nets is that what is learned in the hidden layer for one task can be useful to other tasks. MTL works when tasks share hidden units. One might think that small hidden layers would help MTL by promoting sharing between tasks.

For the kinds of problems we've examined here, this usually does not work. Usually, tasks are different enough that much of what each task needs to learn does not transfer to many (or any) other tasks. Using a large hidden layer insures that there are enough hidden units for tasks to learn independent hidden layer representations when they need to. Sharing can still occur, but only when the overlap between the hidden layer representations for different tasks is strong. In many real-world problems, the loss in accuracy that results from forcing tasks to share by keeping the hidden layer small is larger than the benefit that arises from the sharing. Usually it is important to use large hidden layers with MTL.

8.3.2 Do Early Stopping for Each Task Separately

The classic NETtalk application [30] used one trained both phonemes and stresses on one backprop net. NETtalk is an early example of MTL. But the builders

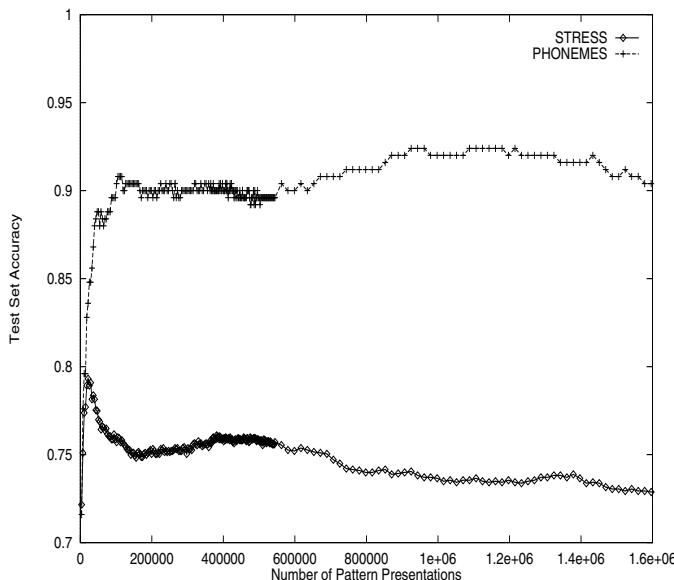


Fig. 8.10. On NETtalk, the Stress task trains very quickly and overfits long before the Phoneme task reaches peak performance

of NETtalk viewed the multiple outputs as codings for a single problem, not as independent tasks that benefited each other by being trained together.

Figure 8.10 shows the learning curves for the phoneme and stress subtasks separately. It is clear that the stress tasks begin to overfit before the phoneme tasks reach peak performance. Better performance could be obtained on NETtalk by doing early stopping on the stress and phoneme tasks individually, or by balancing their learning rates so they reach peak performance at the same time.

Early stopping prevents overfitting by halting the training of error-driven procedures like backprop before they achieve minimum error on the training set (see chapter 2). Recall the steering prediction problem from section 8.2.6. We applied MTL to this problem by training a net on eight extra tasks in addition to the main steering task. Figure 8.11 shows nine learning curves, one for each of the tasks on this MTL net. Each graph is the validation set error during training.

Table 8.4 shows the best place to halt each task. There is no one epoch where training can be stopped so as to achieve maximum performance on all tasks. If all tasks are important, and one net is used to predict all the tasks, halting training where the error summed across all outputs is minimized is the best you can do. Figure 8.12 shows the combined RMS error of the nine tasks. The best average RMSE occurs at 75,000 backprop passes.

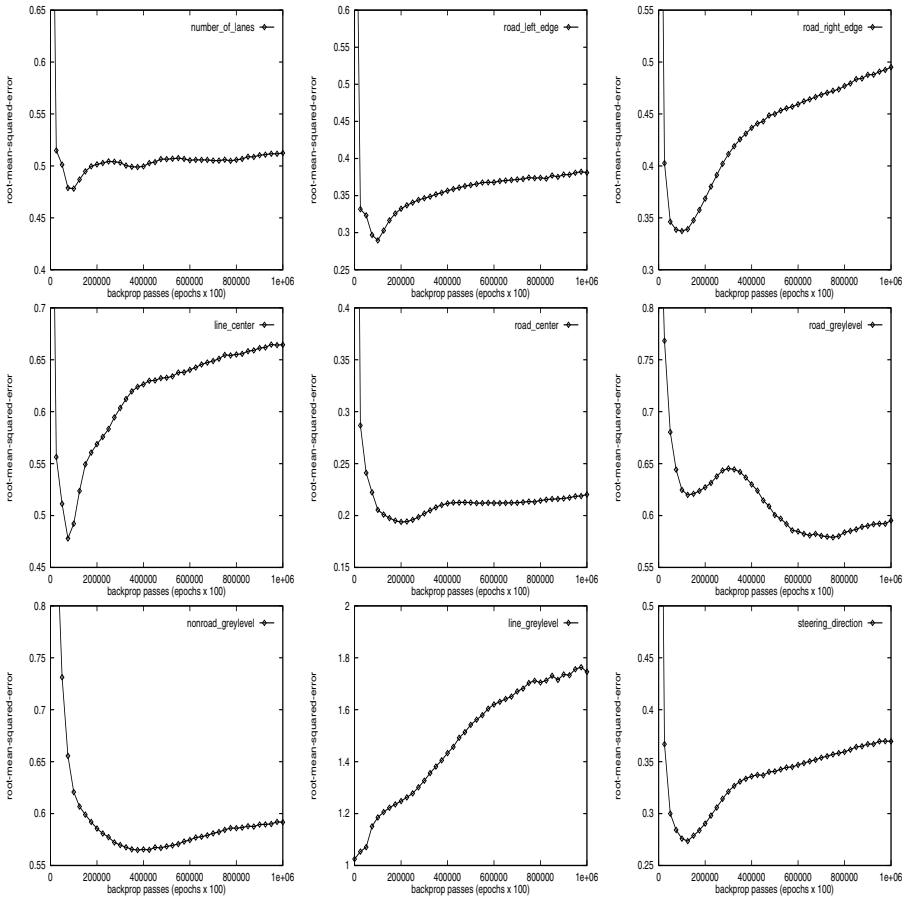


Fig. 8.11. Test-Set Performance of MTL Net Trained on Nine Tasks

But using one net to make predictions for all the tasks is suboptimal. Better performance is achieved by using the validation set to do early stopping on each output individually. The trick is to make a copy of the net at the epoch where performance on each task is best, and use this copy to make predictions for that task. After making each copy, continue training the net until the other tasks reach peak performance. Sometimes, it is best to continue training all outputs on the net, including those that have begun to overfit. Sometimes, it is better to stop training (or use a lower learning rate) for outputs that have begun to overfit. Keep in mind that once an output has begun to overfit, we no longer care how well the net performs on that task because we have a copy of the net from an earlier epoch when performance on that task was best. The only reason to continue training the task is because it may benefit other tasks that have not reached peak performance yet.

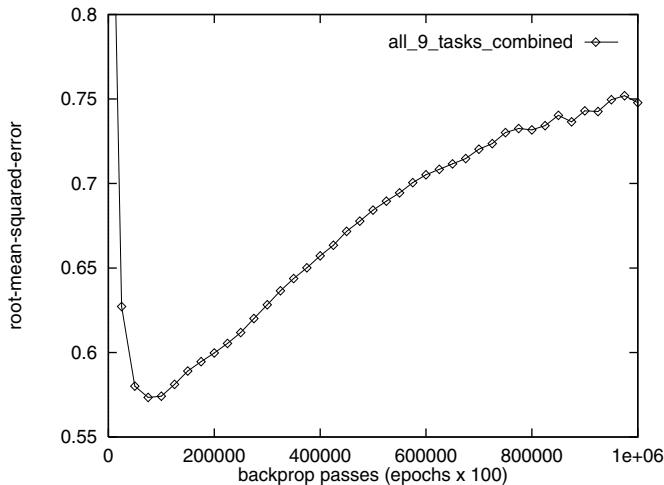


Fig. 8.12. Combined Test-Set Performance on all Nine Tasks

Table 8.4. Performance of MTL on the nine tasks in the steering domain when training is halted on each task individually compared with halting using the combined error across all tasks

TASK	Halted	Individually	Halted	Combined	Difference
	BP Pass	Performance	BP Pass	Performance	
1: 1 or 2 Lanes	100000	0.444	75000	0.456	2.7%
2: Left Edge	100000	0.309	75000	0.321	3.9%
3: Right Edge	100000	0.376	75000	0.381	1.3%
4: Line Center	75000	0.486	75000	0.486	0.0%
5: Road Center	200000	0.208	75000	0.239	14.9%
6: Road Greylevel	750000	0.552	75000	0.680	23.2%
7: Edge Greylevel	375000	0.518	75000	0.597	15.3%
8: Line Greylevel	1	1.010	75000	1.158	14.7%
9: Steering	125000	0.276	75000	0.292	5.8%

Table 8.4 compares the performance of early stopping done per task with the performance one obtains by halting training for the entire MTL net at one place using the combined error. On average, early stopping for tasks individually reduces error 9.0%. This is a large difference. For some tasks, the performance of the MTL net is worse than the performance of STL on this task if the MTL net is not halted on that task individually.

Before leaving this topic, it should be mentioned that the training curves for the individual outputs on an MTL net are not necessarily monotonic. While it is not unheard of for the test-set error of an STL net to be multimodal, the training-set error for an STL net should descend monotonically or become flat. This is not true for the errors of individual outputs on an MTL net. The training-set error summed across all outputs should never increase, but any one output may

exhibit more complex behavior. The graph for road_greylevel (graph number 6) in Figure 8.11 shows a multimodal test-set curve. The training set curve for this output is similar. This makes judging when to halt training more difficult with MTL nets. Because of this, we always do early stopping on MTL nets by training past the epoch where performance on each task appears to be best, and either retrain the net a second time (with the same random seed) to get the copies, or are careful to keep enough copies during the first run that we have whatever copies we will need.

8.3.3 Use Different Learning Rates for Different Tasks

Is it possible to control the rates at which different tasks train so they each reach their best performance at the same time? Would best performance on each task be achieved if each task reached peak performance at the same time? If not, is it better for extra tasks to learn slower or faster than the main task?

The rate at which different tasks learn using vanilla backprop is rarely optimal for MTL. Task that train slower than the main task will not have learned enough to help the main task when training on the main task is stopped. Tasks that train faster than the main task may overfit so much before the main task is learned well that either what they have learned is no longer useful to the main task, or they may drive the main task into premature overfitting.

The most direct method of controlling the rate at which different tasks learn is to use a different learning rate for each task, i.e., for each output. We have experimented with using gradient descent to find learning rates for each extra output to maximize the generalization performance on the main task. Table 8.5 shows the performance on the main steering task before and after optimizing the learning rates of the other eight extra tasks. Optimizing the learning rates for the extra MTL tasks improved performance on the main task an additional 11.5%. This improvement is over and above the original improvement of 15%–25% for MTL over STL.

Examining the training curves for all the tasks as the learning rates are optimized shows that the changes in the learning rates of the extra tasks has a significant effect on the rate at which the extra tasks are learned. Interestingly, it also has a significant effect on the rate at which the main task is learned.

Table 8.5. Performance of MTL on the main Steering Direction task before and after optimizing the learning rates of the other eight extra tasks

TRIAL	Before Optimization	After Optimization	Difference
Trial 1	0.227	0.213	-6.2%
Trial 2	0.276	0.241	-12.7%
Trial 3	0.249	0.236	-5.2%
Trial 4	0.276	0.231	-16.3%
Trial 5	0.276	0.234	-15.2%
Average	0.261	0.231	-11.5% *

This is surprising because we keep the learning rate for the main task fixed during optimization. Perhaps even more interesting is the fact that optimizing the learning rate to maximize the generalization accuracy of the main task also improved generalization on the extra tasks nearly as much as it helped the main task. What is good for the goose appears to be good for the gander.

8.3.4 Use a Private Hidden Layer for the Main Task

Sometimes the optimal number of hidden units is 100 hidden units or more per output. If there are hundreds of extra tasks this translates to thousands of hidden units. This not only creates computational difficulties, but degrades performance on the main task because most of the hidden layer representation is constructed for other tasks. The main task output unit has a massive hidden unit selection problem as it tries to use only those few hidden units that are useful to it.

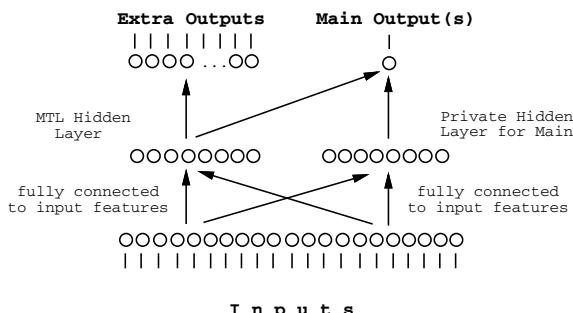


Fig. 8.13. MTL Architecture With a Private Hidden Layer for the Main Task(s), and a Shared Hidden Layer Used by the Main Task(s) and the Extra Tasks

Figure 8.13 shows a net architecture that solves this problem. Instead of one hidden layer shared equally by all tasks, there are two disjoint hidden layers. Hidden layer 1 is a private hidden layer used only by the main task(s). Hidden layer 2 is shared by the main task(s) and the extra tasks. This is the hidden layer that supports MTL transfer. Because the main task sees and affects the shared hidden layer, but the extra tasks do not affect the hidden layer reserved for the main task(s), hidden layer 2 can be kept small without hurting the main task.

8.4 Chapter Summary

We usually think of the inputs to a backprop net as the place where information is given to the net, and the outputs as the place where the net outputs predictions. Backprop, however, pushes information into the net through the *outputs* during training. The information fed into a net through its outputs is as important as the information fed into it through its inputs.

Multitask Learning is a way of using the outputs of a backprop net to push additional information into the net during training. If the net architecture allows sharing of what is learned for different outputs, this additional information can help the main task be learned better. (See [5, 6, 4, 8, 3, 9, 11, 10, 20, 35, 36, 12, 21, 13, 14] for additional discussion about multitask learning.)

MTL trains multiple tasks in parallel not because this is a more efficient way to learn multiple tasks, but because the information in the training signals for the extra tasks can help the main task be learned better. Sometimes what is optimal for the main task is not optimal for the extra tasks. It is important to optimize the technique so that performance on the *main* task is best, even if this hurts performance on the extra tasks. If the extra tasks are important too, it may be best to rerun learning for each important extra task, with the technique optimized for each task one at a time.

This chapter presented a number of opportunities for using extra outputs to leverage information that is available in real domains. The trick in most of these applications is to view the outputs of the net as inputs that are used only during learning. Any information that is available when the net is trained, but which would not be available later when the net is used for prediction, can potentially be used as extra outputs. There are many domains where useful extra tasks will be available. The list of prototypical domains provided in this chapter is not complete. More kinds of extra tasks will be identified in the future.

Acknowledgements. R. Caruana was supported by ARPA grant F33615-93-1-1330, NSF grant BES-9315428, and Agency for Health Care Policy grant HS06468. The work to find input features that worked better when used as extra outputs is joint work with Virginia de Sa, who was supported by postdoctoral fellowship from the Sloan Foundation. We thank the University of Toronto for the Xerion Simulator, and D. Koller and M. Sahami for the use of their feature selector.

References

- [1] Abu-Mostafa, Y.S.: Learning from Hints in Neural Networks. *Journal of Complexity* 6(2), 192–198 (1990)
- [2] Abu-Mostafa, Y.S.: Hints. *Neural Computation* 7, 639–671 (1995)
- [3] Baxter, J.: Learning Internal Representations. In: COLT 1995, Santa Cruz, CA (1995)
- [4] Baxter, J.: Learning Internal Representations. Ph.D. Thesis, The Flinders University of South Australia (December 1994)
- [5] Caruana, R.: Multitask Learning: A Knowledge-Based Source of Inductive Bias. In: Proceedings of the 10th International Conference on Machine Learning, ML 1993, University of Massachusetts, Amherst, pp. 41–48 (1993)
- [6] Caruana, R.: Multitask Connectionist Learning. In: Proceedings of the 1993 Connectionist Models Summer School, pp. 372–379 (1994)
- [7] Caruana, R., Freitag, D.: Greedy Attribute Selection. In: ICML 1994, Rutgers, NJ, pp. 28–36 (1994)

- [8] Caruana, R.: Learning Many Related Tasks at the Same Time with Backpropagation. In: NIPS 1994, pp. 656–664 (1995)
- [9] Caruana, R., Baluja, S., Mitchell, T.: Using the Future to “Sort Out” the Present: Rankprop and Multitask Learning for Medical Risk Prediction. In: Proceedings of Advances in Neural Information Processing Systems, NIPS 1995, pp. 959–965 (1996)
- [10] Caruana, R., de Sa, V.R.: Promoting Poor Features to Supervisors: Some Inputs Work Better As Outputs. In: NIPS 1996 (1997)
- [11] Caruana, R.: Multitask Learning. Machine Learning 28, 41–75 (1997)
- [12] Caruana, R.: Multitask Learning. Ph.D. thesis, Carnegie Mellon University, CMU-CS-97-203 (1997)
- [13] Caruana, R., O’Sullivan, J.: Multitask Pattern Recognition for Autonomous Robots. In: The Proceedings of the IEEE Intelligent Robots and Systems Conference (IROS 1998), Victoria (1998) (to appear)
- [14] Caruana, R., de Sa, V.R.: Using Feature Selection to Find Inputs that Work Better as Outputs. In: The Proceedings of the International Conference on Neural Nets (ICANN 1998), Sweden (1998) (to appear)
- [15] Cooper, G.F., Aliferis, C.F., Ambrosino, R., Aronis, J., Buchanan, B.G., Caruana, R., Fine, M.J., Glymour, C., Gordon, G., Hanusa, B.H., Janosky, J.E., Meek, C., Mitchell, T., Richardson, T., Spirtes, P.: An Evaluation of Machine Learning Methods for Predicting Pneumonia Mortality. Artificial Intelligence in Medicine 9, 107–138 (1997)
- [16] Craven, M., Shavlik, J.: Using Sampling and Queries to Extract Rules from Trained Neural Networks. In: Proceedings of the 11th International Conference on Machine Learning, ML 1994, Rutgers University, New Jersey, pp. 37–45 (1994)
- [17] Davis, I., Stentz, A.: Sensor Fusion for Autonomous Outdoor Navigation Using Neural Networks. In: Proceedings of IEEE’s Intelligent Robots and Systems Conference (1995)
- [18] Dietterich, T.G., Bakiri, G.: Solving Multiclass Learning Problems via Error-Correcting Output Codes. Journal of Artificial Intelligence Research 2, 263–286 (1995)
- [19] Fine, M.J., Singer, D., Hanusa, B.H., Lave, J., Kapoor, W.: Validation of a Pneumonia Prognostic Index Using the MedisGroups Comparative Hospital Database. American Journal of Medicine (1993)
- [20] Ghosn, J., Bengio, Y.: Multi-Task Learning for Stock Selection. In: NIPS 1996 (1997)
- [21] Heskes, T.: Solving a Huge Number of Similar Tasks: A Combination of Multi-task Learning and a Hierarchical Bayesian Approach. In: Proceedings of the 15th International Conference on Machine Learning, Madison, Wisconsin, pp. 233–241 (1998)
- [22] Holmstrom, L., Koistinen, P.: Using Additive Noise in Back-propagation Training. IEEE Transactions on Neural Networks 3(1), 24–38 (1992)
- [23] John, G., Kohavi, R., Pfleger, K.: Irrelevant Features and the Subset Selection Problem. In: ICML 1994, Rutgers, NJ, pp. 121–129 (1994)
- [24] Koller, D., Sahami, M.: Towards Optimal Feature Selection. In: ICML 1996, Bari, Italy, pp. 284–292 (1996)
- [25] Le Cun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackal, L.D.: Backpropagation Applied to Handwritten Zip-Code Recognition. Neural Computation 1, 541–551 (1989)
- [26] Le Cun, Y.: Private communication (1997)

- [27] Munro, P.W., Parmanto, B.: Competition Among Networks Improves Committee Performance. In: Proceedings of Advances in Neural Information Processing Systems, NIPS 1996, vol. 9 (1997) (to appear)
- [28] Pomerleau, D.A.: Neural Network Perception for Mobile Robot Guidance. Doctoral Thesis, Carnegie Mellon University: CMU-CS-92-115 (1992)
- [29] Pratt, L.Y., Mostow, J., Kamm, C.A.: Direct Transfer of Learned Information Among Neural Networks. In: Proceedings of AAAI 1991 (1991)
- [30] Sejnowski, T.J., Rosenberg, C.R.: NETtalk: A Parallel Network that Learns to Read Aloud. John Hopkins: JHU/EECS-86/01 (1986)
- [31] Sill, J., Abu-Mostafa, Y.: Monotonicity Hints. In: Proceedings of Neural Information Processing Systems, NIPS 1996, vol. 9 (1997) (to appear)
- [32] Suddarth, S.C., Holden, A.D.C.: Symbolic-neural Systems and the Use of Hints for Developing Complex Systems. International Journal of Man-Machine Studies 35(3), 291–311 (1991)
- [33] Suddarth, S.C., Kergosien, Y.L.: Rule-injection Hints as a Means of Improving Network Performance and Learning Time. In: Proceedings of EURASIP Workshop on Neural Nets, pp. 120–129 (1990)
- [34] Thrun, S.: Explanation-Based Neural Network Learning: A Lifelong Learning Approach. Kluwer Academic Publisher (1996)
- [35] Thrun, S., Pratt, L. (eds.): Machine Learning. Second Special Issue on Inductive Transfer (1997)
- [36] Thrun, S., Pratt, L. (eds.): Learning to Learn. Kluwer (1997)
- [37] Valdes-Perez, R., Simon, H.A.: A Powerful Heuristic for the Discovery of Complex Patterned Behavior. In: Proceedings of the 11th International Conference on Machine Learning, ML 1994, Rutgers University, New Jersey, pp. 326–334 (1994)
- [38] Weigend, A., Rumelhart, D., Huberman, B.: Generalization by Weight-Elimination with Application to Forecasting. In: Proceedings of Advances in Neural Information Processing Systems, NIPS 1990, vol. 3, pp. 875–882 (1991)

Solving the Ill-Conditioning in Neural Network Learning^{*}

Patrick van der Smagt and Gerd Hirzinger

German Aerospace Research Establishment,
 Institute of Robotics and System Dynamics,
 P.O. Box 1116, D-82230 Wessling, Germany
 smagt@dlr.de
<http://www.robotic.dlr.de/Smagt/>

Abstract. In this paper we investigate the feed-forward learning problem. The well-known ill-conditioning which is present in most feed-forward learning problems is shown to be the result of the structure of the network. Also, the well-known problem that weights between ‘higher’ layers in the network have to settle before ‘lower’ weights can converge is addressed. We present a solution to these problems by modifying the structure of the network through the addition of linear connections which carry shared weights. We call the new network structure the *linearly augmented feed-forward network*, and it is shown that the universal approximation theorems are still valid. Simulation experiments show the validity of the new method, and demonstrate that the new network is less sensitive to local minima and learns faster than the original network.

9.1 Introduction

One of the major problems with feed-forward network learning remains the accuracy and speed of the learning algorithms. Since the learning problem is a complex and highly nonlinear one [12, 4], iterative learning procedures must be used to solve the optimisation problem [2, 14]. A continuing desire to improve the behavior of the learning algorithm has lead to many excellent optimisation algorithms which are especially tailored for feed-forward network learning.

However, an important problem is the particular form of the error function that represents the learning problem. It has long been noted [10, 16] that the derivatives of the error function are usually ill-conditioned. This ill-conditioning is reflected in error landscapes which contain many saddle points and flat areas.

Although this problem can be solved by using stochastic learning methods (e.g., [9, 1, 13]), these methods require many learning iterations in order to find an optimum, and are therefore not suited for problems where fast learning is a requirement. We therefore remain focused on gradient-based learning methods.

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

Algorithms exist which attempt to find well-behaving minima [7], yet an important factor of the learning problem remains the structure of the feed-forward network.

In this chapter an explanation of the ill-conditioned learning problem is provided as well as a solution to alleviate this problem. Section 9.2 formally introduces the learning problem, and describes the problem of singularities in the learn matrices. In section 9.3 the cause of the singularities are analyzed and an adapted learning rule is introduced which alleviates this problem. Section 9.4 discusses a few applications.

9.2 The Learning Process

With a neural network $\mathcal{N} : \Re^N \times \Re^n \rightarrow \Re^M$ we create an approximation to a set of p learning samples $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \dots, (\mathbf{x}_p, \mathbf{y}_p)\}$, with $\mathbf{x}_i \in \Re^N$ and $\mathbf{y}_i \in \Re^M$, for which holds that $\forall 1 \leq i \leq p : \mathcal{F}(\mathbf{x}_i) = \mathbf{y}_i$. The function $\mathcal{F} : \Re^N \rightarrow \Re^M$ is called the **model function**.

Let n be the number of free parameters W of the network. In this particular case we are interested in approximating the learning samples rather than the underlying function \mathcal{F} , or assume that the p learning samples are representative for \mathcal{F} .

The o th output of the function that is represented by the neural network can be written as

$$\mathcal{N}(\mathbf{x}, W)_o = \sum_h w_{ho} s \left(\sum_i w_{ih} x_i + \theta_h \right) + \theta_o \quad (9.1)$$

where $s(x)$ the transfer function, o indicates an output unit, h a hidden unit, i an input unit. The symbol w_{ho} indicates an element of W corresponding with the connection from hidden unit h to output unit o ; w_{ih} is similarly used for a connection from input unit i to hidden unit h . Finally, θ is a bias weight and therefore an element of W . An exemplar feed-forward network with one input and output unit and two hidden units is depicted in figure 9.1.

The learning task consists of minimizing an **approximation error**, which is usually defined as

$$E_{\mathcal{N}}(W) = \sum_{i=1}^p \|\mathcal{N}(\mathbf{x}_i, W) - \mathbf{y}_i\| \quad (9.2)$$

where for $\|\cdot\|$ we prefer to use the L_2 norm. We will leave the subscript \mathcal{N} out when no confusion arises. $E(W)$ is (highly) nonlinear in W , such that iterative search techniques are required to find the W for which $E(W)$ is sufficiently small.

Finally we define the **residual pattern error**

$$e_{M(i-1)+j} = \|\mathcal{N}(\mathbf{x}_i, W)_j - \mathbf{y}_{ij}\|, \quad (9.3)$$

i.e., the error in the j 'th output value for the i 'th learning sample.

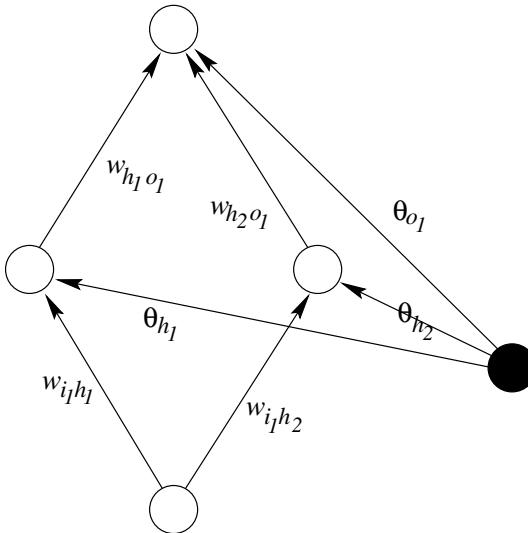


Fig. 9.1. An exemplar feed-forward neural network. The circles represent neurons; the black filled circle is a bias unit, and always carries an activation value of ‘1’.

9.2.1 Learning Methodology

Gradient based learning methods are characterized by considering low-order terms from the Taylor expansion to the approximation error,

$$E(W + W_0) = E(W_0) + \nabla E|_{W_0} W + W^T \nabla^2 E|_{W_0} W + \dots \quad (9.4)$$

In most cases more than the second order term is neglected. We define

$$\tilde{E}_1(W + W_0) = E(W_0) + \nabla E|_{W_0} W \quad (9.5)$$

and

$$\tilde{E}_2(W + W_0) = \tilde{E}_1(W) + W^T \nabla^2 E|_{W_0} W \quad (9.6)$$

being the first-order and second-order approximation to E , respectively.

By locally considering the approximation error as a first- or second-order function of W , we can use several existing approximation methodologies to minimize E . When, according to the local second-order approximation information, E is minimized, the local information is updated and a second minimization step is carried out. This process is repeated until a minimum is found.

Well-known minimization methods are steepest descent (a variant of which is known as error backpropagation), conjugate gradient optimisation, Levenberg-Marquardt optimisation, variable metric methods, and (quasi-) Newton methods. Each of these methods has its advantages and disadvantages which are discussed elsewhere [14].

The optimisation methods work in principle as follows. By considering the second-order approximation to E , an optimum can be analytically found if ∇E

and $\nabla^2 E$ are known. After the optimum has been located, ∇E and $\nabla^2 E$ are recomputed using the local information, and the minimum is relocated. This process is repeated until a minimum is found.

Naturally, the success of this approach stands or falls with the form of the error function. If the error function is not too complex but smooth, and can be reasonably approximated by a quadratic function, the discussed optimisation methods are a reliable and fast way of finding minima. In feed-forward network training, however, the error functions appear to have a large number of flat areas where minimization is a difficult task due to limited floating point accuracy.

9.2.2 Condition of the Learning Problem

The flat areas of the error function can be formalized as follows. We define the $(Mp) \times n$ Jacobian matrix as

$$J \equiv \begin{pmatrix} \nabla e_1^T \\ \nabla e_2^T \\ \vdots \\ \nabla e_{Mp}^T \end{pmatrix} \quad \text{where} \quad \nabla e_k \equiv \begin{pmatrix} \frac{\partial e_k}{\partial w_1} \\ \frac{\partial e_k}{\partial w_2} \\ \vdots \\ \frac{\partial e_k}{\partial w_n} \end{pmatrix} \quad (9.7)$$

such that we can write J as

$$J \equiv \begin{pmatrix} \frac{\partial e_1}{\partial w_1} & \frac{\partial e_1}{\partial w_2} & \cdots & \frac{\partial e_1}{\partial w_n} \\ \frac{\partial e_2}{\partial w_1} & \frac{\partial e_2}{\partial w_2} & \cdots & \frac{\partial e_2}{\partial w_n} \\ \vdots & \ddots & & \vdots \\ \frac{\partial e_{Mp}}{\partial w_1} & \frac{\partial e_{Mp}}{\partial w_2} & \cdots & \frac{\partial e_{Mp}}{\partial w_n} \end{pmatrix}. \quad (9.8)$$

In the learning process we thus encounter that $\nabla E = 2J^T e$.

We furthermore define the Hessian $H = \nabla^2 E$, which is the matrix of second order derivatives of E . We are interested in the eigenvalues and eigenvectors of H . Since H is a positive semidefinite symmetric matrix close to a local minimum, its eigenvalues are all positive real numbers. When an eigenvalue is very small, the effect of moving in the direction of the corresponding eigenvector on the approximation error is very small. This means that, in that direction, the error function is (nearly) singular. The singularity of the error function can be expressed in the condition of H , which is defined as the quotient of its largest and its smallest eigenvalues.

As mentioned above, a bad condition of H may occur at minima or flat points in the error function $E(W)$. It appears that feed-forward learning tasks are generally characterized by having a singular or near-singular Hessian matrix. Although the said learning methods are mathematically not influenced by a badly conditioned Hessian, it does lead to inaccuracies due to the limited floating point accuracy of the digital computer.

9.2.3 What Causes the Singularities

Reference [10] lists a few cases in which the Hessian may become singular or near-singular. The listed reasons are associated with the ill character of the sigmoid transfer function. Typical for this function is the fact that $\lim_{x \rightarrow \infty} s(x) = c_+$ and $\lim_{x \rightarrow -\infty} s(x) = c_-$. Also, bad conditioning can be the result of uncentered data, and can also be alleviated [11].

Assuming that some neuron is in this ‘saturated’ state for all learning patterns, its input weights will have a delta equal to 0 (according to the backpropagation rule) such that these weights will never change. For each of the incoming weights of this neuron, this leads to a 0-row in H , and therefore singularity.

However, there is another important reason for singularity, which may especially occur after network initialization. When a multi-layer feed-forward network has a small (e.g., less than 0.1) weight leaving from a hidden unit, the influence of the *weights that feed into this hidden unit* is significantly reduced. Therefore the $\partial e / \partial w_k$ will be close to 0, leading to near-null rows in J and a near-singular H .

We observe that this kind of singularity is very common and touches a characteristic problem in feed-forward network learning: The gradients in the lower-layer weights are influenced by the higher-layer weights. A related problem is the influence that the change of the weights between the hidden and output units have on the change of the input weights; when they rapidly and frequently change, as will be the case during the initial stages of learning, the lower weights will have nonsensical repeated perturbations.

9.2.4 Definition of Minimum

A minimum is said to be reached when the derivative of the error function is zero, i.e., $\partial E / \partial w_{1 \leq k \leq n} = 0$. Since the gradient of the error function equals the column-sum of the Jacobian, a minimum has been reached when

$$\forall 1 \leq k \leq n : \sum_{i=1}^{M_p} \frac{\partial e_i}{\partial w_k} = 0, \quad (9.9)$$

i.e., when each of the columns adds up to 0. Eq. (9.9) defines the minimum for a batch-learning system: the gradient, when summed over all learning samples, should be 0. This also means, however, that it may occur that elements of a column-sum cancel each other out, even when not all elements of the Jacobian are 0. Differently put, it may occur that the gradient for some patterns are non-zero, whereas the gradients sum up to zero.

The optimal case is reached when *all* elements of the Jacobian are 0. This means, of course, that the residual error of each pattern is 0.

9.3 Local Minima are Caused by BackPropagation

In this section we propose a new neural network structure which alleviates the above problems. In the standard backpropagation learning procedure, the

gradient of the error function with respect to the weights is determined by computing the following steps for each learning pattern:

1. For each output unit o , compute the delta $\delta_o = y_o - a_o$ where a_o is the activation value for that unit, when a learning sample is presented.
2. Compute the weight derivative for the weights w_{ho} from the hidden to output units:

$$\Delta w_{ho} = \delta_o a_h$$

where a_h is the activation value for the hidden unit.

3. Compute the delta for the hidden unit:

$$\delta_h = \sum_o \delta_o w_{ho} s'(a_h).$$

4. Compute the weight derivative for the weights w_{ih} from the input to hidden units:

$$\Delta w_{ih} = \delta_h a_i = \sum_o \delta_o w_{ho} s'(a_h) a_i. \quad (9.10)$$

The gradient is then computed as the summation of the Δw 's.

From (9.10) we can see that there are four cases when the gradient for a weight from an input to a hidden unit is negligible, such that the corresponding row and column in the Hessian are near-zero:

- When δ_o is small. This is correct, since that case means that the output of the network is close to its desired output.
- When w_{ho} is small. This is an undesired situation: A small weight from hidden to output unit paralyzes weights from input to hidden units. This is especially important since the weight might have to change its value later.
- When $s'(a_h)$ is small; this occurs when the weight w_{ih} is large. Again, this saturation type of paralysis is undesired.
- Finally, when a_i is small. This is desired: When the input value is insignificant, it should have no influence on the output.

9.3.1 A New Neural Network Structure

In order to alleviate these problems, we propose a change to the learning system of (9.10) as follows:

$$\Delta w_{ih} = \sum_o \delta_o (w_{ho} s'[a_h] + c) a_i = \delta_h a_i + c \sum_o \delta_o a_i.$$

By adding a constant c to the middle term, we can solve both paralysis problems. In effect, an extra connection from each input unit to each output unit is created, with a weight value coupled to the weight from the input to hidden unit.

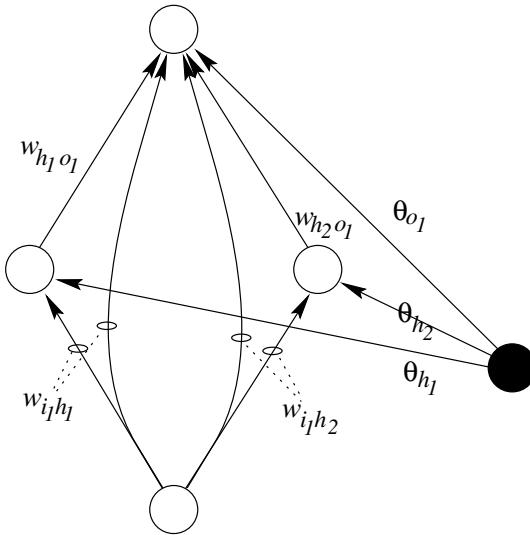


Fig. 9.2. An exemplar adapted feed-forward neural network

Although c can be made a learnable parameter, in the sequel we will assume $c = 1$. The o 'th output of the neural network is now computed as

$$\mathcal{M}(\mathbf{x}, W)_o = \sum_h \left(w_{ho} s \left[\sum_i w_{ih} x_i + \theta_h \right] + \sum_i w_{ih} x_i \right) + \theta_o. \quad (9.11)$$

We call the new network the **linearly augmented feed-forward network**. The structure of this network is depicted in figure 9.2. Note the equivalence of \mathcal{N} and \mathcal{M} , viz.,

$$\mathcal{M}(\mathbf{x}, W)_o \equiv \mathcal{N}(\mathbf{x}, W)_o + \sum_h \sum_i w_{ih} x_i. \quad (9.12)$$

9.3.2 Influence on the Approximation Error E

Although the optimal W will be different for the networks \mathcal{N} and \mathcal{M} , we can still compare the forms of the error functions $E_{\mathcal{N}}$ vs. $E_{\mathcal{M}}$. Using (9.2) and (9.11) we can compute the error in the approximation of a single learning sample (\mathbf{x}, y) with N inputs, κ hidden units, and a single output:

$$\begin{aligned} E_{\mathcal{M}}(W)^2 &= (\mathcal{M}[\mathbf{x}, W] - y)^2 \\ &= \left(\mathcal{N}(\mathbf{x}, W) - y + \sum_i \sum_h w_{ih} x_i \right)^2 \\ &= E_{\mathcal{N}}(W)^2 + 2 \sum_i \sum_h w_{ih} x_i (\mathcal{N}[\mathbf{x}, W] - y) + \left(\sum_i \sum_h w_{ih} x_i \right)^2. \end{aligned} \quad (9.13)$$

The error for the network \mathcal{M} differs from the error for \mathcal{N} in two terms. When we consider $E_{\mathcal{M}}$ at those values of W where $E_{\mathcal{N}}$ is minimal, we can see that the difference between $E_{\mathcal{N}}$ and $E_{\mathcal{M}}$ consists of a normalization term $\|\sum_h \mathbf{w}_h^T \mathbf{x}\|$; \mathbf{w}_h is the vector of weights connecting the inputs to hidden unit h . This non-negative term will only be zero when the vectors $\mathbf{w}_1^T \mathbf{x}, \mathbf{w}_2^T \mathbf{x}, \dots, \mathbf{w}_{\kappa}^T \mathbf{x}$ cancel each other out for each input vector \mathbf{x} which is in the training set. In words: $E_{\mathcal{M}}(W)^2$ introduces a penalty for hidden units doing the same thing, thus making the network use its resources more efficiently.

9.3.3 \mathcal{M} and the Universal Approximation Theorems

It has been shown in various publications [3, 6, 8] that the ordinary feed-forward neural network \mathcal{N} can represent any Borel-measurable function with a single layer of hidden units which have sigmoidal or Gaussian activation functions in the hidden layer.

Theorem 1. *The network \mathcal{M} can represent any Borel-measurable function with a single layer of hidden units which have sigmoidal or Gaussian activation functions in the hidden layer.*

Proof. We show that any network \mathcal{N} can be written as a network \mathcal{M} ; therefore, the class of networks \mathcal{M} are universal approximators.

By using (9.1) and (9.11),

$$\begin{aligned} \mathcal{N}(\mathbf{x}, W)_o &= \sum_{h=1}^{\kappa} w_{ho} s \left(\sum_i w_{ih} x_i + \theta_h \right) + \theta_o \\ &= \sum_{h=1}^{\kappa} \left(w_{ho} s \left[\sum_i w_{ih} x_i + \theta_h \right] + \sum_i w_{ih} x_i \right) + \theta_o - \sum_{h=1}^{\kappa} \sum_i w_{ih} x_i \\ &= \mathcal{M}(\mathbf{x}, W)_o + \sum_{l=\kappa+1}^{2\kappa} \left(0 \left[\sum_i -w_{i,l-\kappa} x_i + 0 \right] + \sum_i -w_{i,l-\kappa} x_i \right) + 0 \\ &= \mathcal{M}(\mathbf{x}, W)_o + \mathcal{M}(\mathbf{x}, V)_o \end{aligned}$$

where V is a weight matrix such that the elements of V corresponding to the weights from hidden to output units are 0, and the other weights equal the negation of its W counterparts. Furthermore, bias weights are set to 0.

The sum $\mathcal{M}(\mathbf{x}, W)_o + \mathcal{M}(\mathbf{x}, V)_o$ represents two \mathcal{M} -networks, which can also be written as a single $\mathcal{M}(\mathbf{x}, W')$ -network with the double amount of hidden units, where $W' = [WV]$.

Using the theorems from [3, 6, 8] the proof is complete.

Note that it is also possible to write each \mathcal{M} -network as an \mathcal{N} -network, by doubling the number of hidden units and using infinitesimal weights from the input units to these hidden units, and their multiplicative inverse for the weights from these hidden to the output units.

Figure 9.3 shows the equivalence of an \mathcal{N} and \mathcal{M} network for the two-hidden unit case.

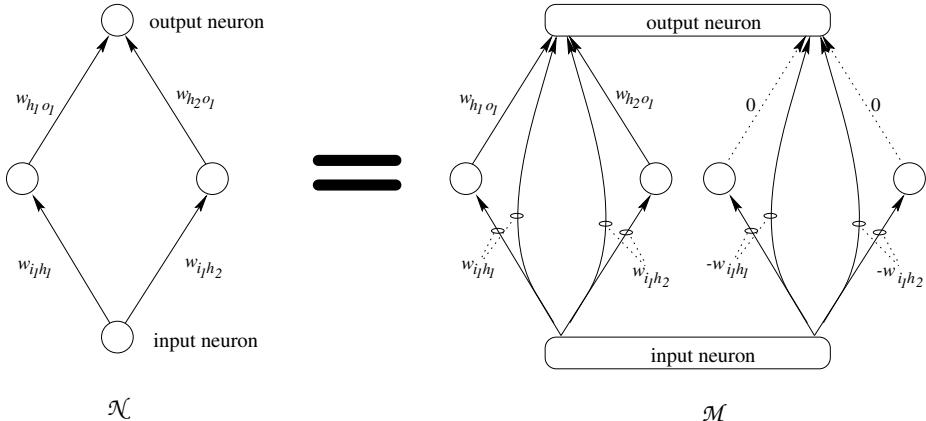


Fig. 9.3. Equivalence of an \mathcal{N} -network (left) and an \mathcal{M} -network (right). Note that bias units are left out for clarity.

9.3.4 Example

As an example, we train a network with a single hidden unit, and no bias connections, to represent the learning samples $(1, 1)$ and $(2, 2)$. The hidden unit has a sigmoid activation function. The function that is computed by the network thus is $\mathcal{N}(x, W) = w_2 s(w_1 x)$ for the original neural network, and $\mathcal{M}(x, W) = w_2 s(w_1 x) + w_1 x$ for the adapted neural network. We use the sigmoid function $s(x) = 1/(1 + e^{-x})$ as activation function, which has the following well-known properties: $\lim_{x \rightarrow \infty} s(x) = 1$, $\lim_{x \rightarrow -\infty} s(x) = 0$, and $\lim_{x \rightarrow \pm\infty} s'(x) = 0$.

Figure 9.4 shows the error function and its derivatives for this neural network. In the top row of this figure we see the original neural network. Notice in the top middle figure, depicting $\partial E / \partial w_1$, that $\partial E / \partial w_1 \approx 0$ for small values of w_2 . In other words, when w_2 is small, w_1 will hardly change its value. Similarly, when w_1 is large, then $\partial E / \partial w_1$ will be small due to the fact that the derivative of the transfer function is nearly 0. In the bottom row the modified neural network is depicted. Left, again, the error function. The middle figure clearly shows that the derivative has no areas anymore which are zero or very small. The right figure still shows that, if w_1 has a large negative value, $\partial E / \partial w_2$ is negligible: after all, the activation value of the hidden unit is near-zero.

9.4 Applications

We have applied the new learning scheme to several approximation problems. In all problems, each network has been run 3,000 times with different initial random values for the weights. In order to train the network, we used a Polak-Ribière conjugate gradient optimization technique with Powell restarts [14].

The applications with real data (problems 3 and 4) use two independent sets of data: a learn set and a cross-validation set. In all cases, the network was

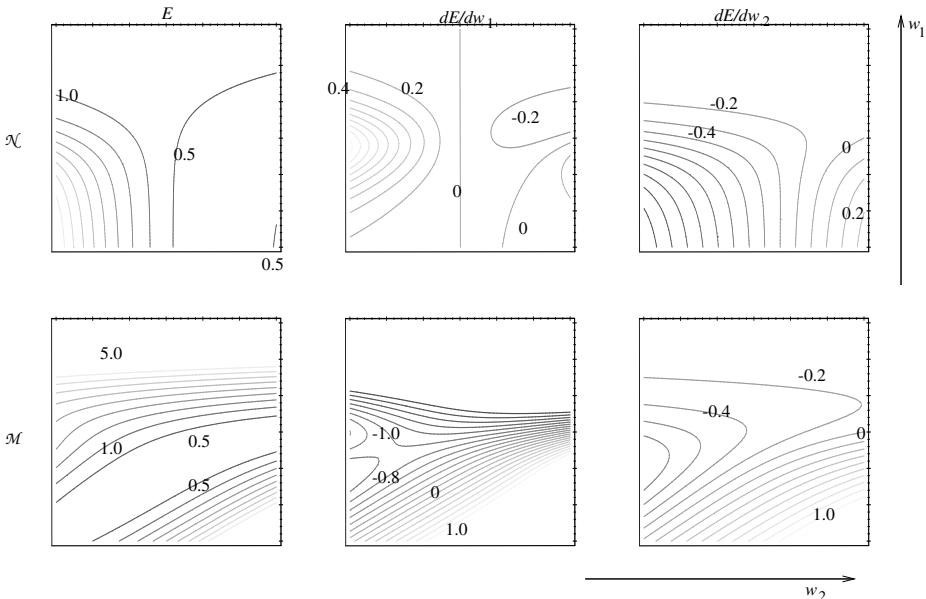


Fig. 9.4. Error function and derivatives using the original and adapted learning rule. The top row shows the error function for the original learning rule (left), as well as its derivative $\partial E/\partial w_1$ (centre) and $\partial E/\partial w_2$ (right). The bottom row shows the same graphs for the adapted learning rule. The contour lines have a distance of 0.5 (left graphs) and 0.2 (middle and right graphs).

trained until the error over the cross-validation set was minimal (i.e., up to but not beyond the point where the network started to over-fit). The approximation errors that are reported are computed using the samples in the cross-validation set.

Problem 1: (synthetic data) the XOR problem. The well-known XOR problem consists of representing four learning samples $[(0, 0), 0]$, $[(0, 1), 1]$, $[(1, 0), 1]$, and $[(1, 1), 0]$. The network has two inputs, two hidden units, and one output.

It has been noted [5] that the XOR problem is very atypical in neural network learning, because it is penalized for generalization. Nevertheless, the XOR problem is generally considered as a small standard learning benchmark problem.

Whereas the network \mathcal{N} reaches local minima in 22.4% of the runs, the linearly augmented network \mathcal{M} always reached a global minimum. For \mathcal{N} , the average number of steps to obtain an approximation error of 0.0 (within the 32-bit floating point accuracy of the computer) equals 189.1; for \mathcal{M} , 98.3 steps were required.

When using the ordinary error backpropagation learning rule (i.e., no conjugate gradient learning), the XOR learning problem has been reported to lead to 8.7% local minima [14].

Problem 2: (synthetic data) approximating $\tan(x)$. As a second test, the networks have been used for the approximation of the function $\mathcal{F}(x) = \tan(x)$. The function has been uniformly sampled in the domain $[0, \pi]$ using 20 learning samples in total. For the approximation we used a network with one input, five hidden units in a single hidden layer, and one output.

With network \mathcal{N} , 14.6% of the runs lead to a local minimum. The linearly augmented neural network is not perfect; it is stuck in a local minimum in 6.2% of the runs. In all other cases, a global minimum of very close to 0.0 was found.

Problem 3: (real data) approximating robot arm inverse kinematics. Thirdly we approximated data describing a hand-eye coordination learning problem with a Manutec R2 robot arm.

The data is organized as follows. There are five input variables, describing the current position of the robot arm in joint angles θ_2, θ_3 , as well as the visual position of an object with respect to the hand-held camera in pixel coordinates x, y, z . The output part of the data consists of the required joint rotations $\Delta\theta_1, \Delta\theta_2$, and $\Delta\theta_3$ necessary to reach the object. See [15] for further description of this hand-eye coordination problem. In this particular problem, 1103 learning samples are used; 1103 samples are used for cross-validation. The optimal of six hidden units in a single layer [15] is used. Only the cross-validating data is used for evaluating the network.

With the normal network \mathcal{N} , in 2.3% of the cases the network got stuck in a minimum with an unacceptably high error for both the learn and test sets. This never occurred in the 3,000 trials in which the linearly augmented network was used. The cross-validated approximation error after 10,000 learning steps equals $4.20 \cdot 10^{-4}$ for network \mathcal{N} , and $4.04 \cdot 10^{-4}$ for network \mathcal{M} (both values are average per learning sample). The new method shows a slight improvement here.

Problem 4: (real data) gear box deformation data. The final test consists of the following problem, which is encountered in the control of a newly developed lightweight robot arm. A gear box connects a DC motor with a robot arm segment. In order to position the robot arm segment at a desired joint angle θ_a , the DC motor has to exert a force τ . However, in the normal setup a joint angle decoder is only available at the DC motor side, which measures the angle θ_m . By mounting an extra decoder at the arm segment in a laboratory setup we can measure θ_a . The actual joint angle θ_a differs slightly from θ_m because of the gear-box elasticity. We attempt to learn θ_a from $(\theta_m, \dot{\theta}_m, \tau)$.

In order to learn these data, we use a network with 3 inputs, 6 hidden units in a single layer, and one output. The data consists of 4,000 learning samples as well as 4,000 samples used for the cross-validation. In each run, up to 10,000 learning iterations are performed but not beyond the point of overfitting.

Both network \mathcal{N} as \mathcal{M} do not get stuck in a minimum with an unacceptably high error for both the learn and test sets. A surprise, however, is encountered in the cross-validated approximation error that is computed after 10,000 learning steps. For network \mathcal{N} , this error equals $2.85 \cdot 10^{-4}$ per learning sample; for \mathcal{M} ,

however, this error is as low as $1.87 \cdot 10^{-6}$ per sample. Further data analysis has shown that the data has strong linear components, which explains the fact that the approximation error is two orders of magnitude lower.

Results are summarized in table 9.1.

Table 9.1. Results of the ordinary feed-forward network \mathcal{N} and the linearly augmented feed-forward network \mathcal{M} on the four problems

		\mathcal{N}	\mathcal{M}
XOR	% local minima	22.4	0.0
	avg. steps	189.1	98.3
tan	% local minima	14.6	6.2
robot	% local minima	2.3	0.0
3D data	avg. error	$4.20 \cdot 10^{-4}$	$4.04 \cdot 10^{-4}$
gear box	% local minima	0.0	0.0
data	avg. error	$2.85 \cdot 10^{-4}$	$1.87 \cdot 10^{-6}$

9.5 Conclusion

It has been shown that the ordinary backpropagation learning rule leads to bad conditioning in the matrix of second-order derivatives of the error function which is encountered in feed-forward neural network learning. This again leads to local minima and saddle points in the error landscape. In order to alleviate this problem, we have introduced an adaptation to the backpropagation rule, which can be implemented as an adapted feed-forward neural network structure. We call this network the the *linearly augmented feed-forward neural network*. The adaptation leads to a learning rule which obtains stable values for the weights which connect the input units with the hidden units, even while the weights from hidden to output units change.

A mathematical analysis shows the validity of the method; in particular, the universal approximation theorems are shown to remain valid with the new neural network structure. The application of the new method to two sets of synthetic data and two sets of real data shows that the new method is much less sensitive to local minima, and reaches an optimum in fewer iterations.

Acknowledgments. The authors acknowledge the help of Alin Albu-Schäffer in supplying the gear box data.

References

- [1] Aarts, E., Korst, J.: Simulated Annealing and Boltzmann Machines. John Wiley & Sons (1989)
- [2] Battiti, R.: First- and second-order methods for learning: Between steepest descent and Newton's method. Neural Computation 4, 141–166 (1992)

- [3] Cybenko, G.: Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems* 2(4), 303–314 (1989)
- [4] DasGupta, B., Siegelmann, H.T., Sontag, E.D.: On the complexity of training neural networks with continuous activation functions. *IEEE Transactions on Neural Networks* 6(6), 1490–1504 (1995)
- [5] Fahlman, S.E.: An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-0-162, Carnegie Mellon University (September 1988)
- [6] Funahashi, K.-I.: On the approximate realization of continuous mappings by neural networks. *Neural Networks* 2(3), 183–192 (1989)
- [7] Hochreiter, S., Schmidhuber, J.: Flat minima. *Neural Computation* 9(1), 1–42 (1997)
- [8] Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. *Neural Networks* 2(5), 359–366 (1989)
- [9] Robbins, H., Monro, S.: A stochastic approximation method. *Annals of Mathematical Statistics* 22(1), 400–407 (1951)
- [10] Saarinen, S., Bramley, R., Cybenko, G.: Ill-conditioning in neural network training problems. *Siam Journal of Scientific Computing* 14(3), 693–714 (1993)
- [11] Schraudolph, N.N.: On centering neural network weight updates. Technical Report IDSIA-19-97, IDSIA (April 1997)
- [12] Sontag, E.D., Sussmann, H.J.: Backpropagation can give rise to spurious local minima even for networks without hidden layers. *Complex Systems* 3(1), 91–106 (1989)
- [13] Unnikrishnan, K.P., Venugopal, K.P.: Alopex: A correlation based learning algorithm for feedforward and recurrent neural networks. *Neural Computation* 6, 469–490 (1994)
- [14] van der Smagt, P.: Minimisation methods for training feed-forward networks. *Neural Networks* 7(1), 1–11 (1994)
- [15] van der Smagt, P.: Visual Robot Arm Guidance using Neural Networks. PhD thesis, Dept of Computer Systems, University of Amsterdam (March 1995)
- [16] Zhang, Q.J., Zhang, Y.J., Ye, W.: Local-sparse connection multilayer networks. In: Proc. IEEE Conf. Neural Networks, pp. 1254–1257. IEEE (1995)

Centering Neural Network Gradient Factors*

Nicol N. Schraudolph

IDSIA, Corso Elvezia 36
 6900 Lugano, Switzerland
nic@idsia.ch
<http://www.idsia.ch/>

Abstract. It has long been known that neural networks can learn faster when their input and hidden unit activities are centered about zero; recently we have extended this approach to also encompass the centering of error signals [15]. Here we generalize this notion to *all* factors involved in the network’s gradient, leading us to propose centering the slope of hidden unit activation functions as well. Slope centering removes the linear component of backpropagated error; this improves credit assignment in networks with shortcut connections. Benchmark results show that this can speed up learning significantly without adversely affecting the trained network’s generalization ability.

10.1 Introduction

Centering is a general methodology for accelerating learning in adaptive systems of the type exemplified by neural networks — that is, systems that are typically nonlinear, continuous, and redundant; that learn incrementally from examples, generally by some form of gradient descent. Its basic tenet is:

All pattern-dependent factors entering the update equation for a neural network weight should be centered, i.e., have their average over patterns subtracted out.

Prior Work. It is well-known that the inputs to an LMS adaptive filter should be centered to permit rapid yet stable adaptation [22], and it has been argued [12] that the same applies to input and hidden unit activity in a multi-layer network. Although Sejnowski [16] proposed a variant of Hebbian learning in which both the pre- and postsynaptic factors of the weight update are centered, the idea was not taken up when backpropagation became popular. The benefits of centering error signals in multi-layer networks were thus reported only recently [15]; here we finally suggest centering as a general methodology, and present backpropagation equations in which *all* factors are centered.

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

Independence of Architecture. Although centering is introduced here in the context of feedforward networks with sigmoid activation functions, the approach itself has a far wider reach. The implementation details may vary, but in essence centering is not tied to any particular architecture: its principles are equally applicable to feedforward and recurrent networks, with sigmoid, radial, or other basis functions, with or without topological structure, time delays, multiplicative gates, *etc.* — in short, the host of architectural elements used in neural network design.

Independence of Learning Algorithm. Similarly, centering is not wedded to any particular learning algorithm either. It may be applied to deterministic (batch) or stochastic (online) gradient descent; more importantly, it may be freely combined with more sophisticated optimization techniques such as expectation maximization, conjugate gradient and quasi-Newton methods. It also leaves available the many useful tricks often employed with stochastic gradient descent, such as momentum, learning rate adaptation, gradient normalization, and so forth. Due to this flexibility, centering has the potential to further accelerate even the fastest of these methods.

Overview. Section 10.2 introduces the centering approach in terms of the modifications it mandates for ordinary backpropagation learning. We then discuss implementation details in Section 10.3 before presenting benchmark results in Section 10.4. Section 10.5 concludes our paper with a brief analysis of how centering facilitates faster learning.

10.2 Centered Backpropagation

The backpropagation learning algorithm is characterized by three equations, describing the forward propagation of activity, the backpropagation of error, and the modification of weights, respectively. Here are the implications of centering for each of these three equations:

10.2.1 Activity Propagation

Conventional. Consider a neural network with activation of node j given by

$$x_j = f_j(y_j), \quad y_j = \sum_{i \in A_j} w_{ij} \check{x}_i, \quad (10.1)$$

where f_j is a nonlinear (typically sigmoid) activation function, w_{ij} are the synaptic weights, and A_j denotes the set of *anterior* nodes feeding their activity \check{x}_i into node j . Conventionally, the anterior nodes' output is fed forward directly, *i.e.*, $(\forall i) \check{x}_i \equiv x_i$. We imply that nodes are activated via Equation 10.1 in appropriately ordered (feedforward) sequence, and that some have their values clamped so as to represent external inputs to the network. In particular, we posit a bias input $x_0 \equiv 1$ and require that all nodes are connected to it: $(\forall j > 0) 0 \in A_j$.

Centered. As suggested by LeCun et al. [12], the activity of the network’s input and hidden units should be centered to permit faster learning (see Chapter 1). We do so by setting

$$(\forall i > 0) \quad \check{x}_i = x_i - \langle x_i \rangle, \quad (10.2)$$

where $\langle \cdot \rangle$ denotes averaging over training samples (see Section 10.3 for ways to implement this operator). Note that the bias input must *not* be centered — since $x_0 = \langle x_0 \rangle = 1$, it would otherwise become inoperative.

10.2.2 Weight Modification

Conventional. The weights w_{ij} of the network given in Equation 10.1 are typically optimized by gradient descent in some objective function E . With a local step size η_{ij} for each weight, this results in the weight update equation

$$\Delta w_{ij} = \eta_{ij} \delta_j \check{x}_i, \quad \text{where } \delta_j = -\partial E / \partial y_j. \quad (10.3)$$

Centered. We have recently proposed [15] that the error signals δ_j should be centered as well to achieve even faster convergence. This is done by updating all non-bias weights via

$$(\forall i > 0) \quad \Delta w_{ij} = \eta_{ij} \check{\delta}_j \check{x}_i, \quad \text{where } \check{\delta}_j = \delta_j - \langle \delta_j \rangle. \quad (10.4)$$

As before, this centered update must *not* be used for bias weights, for otherwise they would remain forever stuck (except for stochastic fluctuations) at their present values:

$$* \quad \langle \Delta w_{0j} \rangle \propto \langle \check{\delta}_j \rangle = \langle \delta_j \rangle - \langle \delta_j \rangle = 0. \quad (10.5)$$

Instead, bias weights are updated conventionally (Equation 10.3). Since this means that the average error $\langle \delta_j \rangle$ is given exclusively to the bias weight w_{0j} , we have previously called this technique *d.c. error shunting* [15].

10.2.3 Error Backpropagation

Conventional. For output units, the error δ_j can be computed directly from the objective function; for hidden units, it must be derived through error back-propagation:

$$\delta_j = f'_j(y_j) \gamma_j, \quad \gamma_j = \sum_{k \in P_j} w_{jk} \delta_k, \quad (10.6)$$

where P_j denotes the set of *posterior* nodes fed from node j , and $f'_j(y_j)$ is the node’s current *slope* — the derivative of its nonlinearity f_j at the present level of activation.

Centered. By inserting Equation 10.6 into Equation 10.4, we can express the weight update for hidden units as a triple product of their activity, backpropagated error, and slope:

$$\Delta w_{ij} \propto f'_j(y_j) \check{\gamma}_j \check{x}_i, \quad (10.7)$$

where $\check{\gamma}_j$ denotes backpropagated *centered* errors. It is not necessary to center the $\check{\gamma}_j$ explicitly since

$$\langle \check{\gamma}_j \rangle = \left\langle \sum_{k \in P_j} w_{jk} \check{\delta}_k \right\rangle = \sum_{k \in P_j} w_{jk} \langle \check{\delta}_k \rangle = 0. \quad (10.8)$$

By centering activity and error signals we have so far addressed two of the three factors in Equation 10.7, leaving the remaining third factor — the node’s slope — to be dealt with. This is done by modifying the nonlinear part of error backpropagation (Equation 10.6) to

$$\delta_j = \check{f}'_j(y_j) \check{\gamma}_j, \quad \text{where } \check{f}'_j(y_j) = f'_j(y_j) - \langle f'_j(y_j) \rangle. \quad (10.9)$$

Decimation of Linear Errors. Note that for a *linear* node n we would have $f'_n(y_n) \equiv 1$, and Equation 10.9 would always yield $\delta_n \equiv 0$. In other words, slope centering (for any node) blocks backpropagation of the *linear component* of error signals — that component which a linear node in the same position would receive. Viewed in terms of error decimation, we have thus taken the logical next step past error centering, which removed the d.c. (constant) component of error signals.

Shortcuts. It was important then to have a parameter — the bias weight — to receive and correct the d.c. error component about to be eliminated. Likewise, we now require additional weights to implement the linear mapping from anterior to posterior nodes that the unit in question is itself no longer capable of. Formally, we demand that for each node j for which Equation 10.9 is used, we have

$$(\forall i \in A_j) \quad P_j \subseteq P_i. \quad (10.10)$$

We refer to connections that bypass a node (or layer) in this fashion as *shortcuts*. It has been noted before that neural network learning sometimes improves with the addition of shortcut weights. In our own experiments (see Section 10.4), however, we find that it is slope centering that makes shortcut weights genuinely useful.

A Complementary Approach? In Chapter 9, van der Smagt and Hirzinger also advocate shortcuts as a means for accelerating neural network learning. Note, however, that their use of shortcuts is quite different from ours: in order to improve the conditioning of a neural network, they add shortcut connections whose weights are coupled to (shared with) *existing* weights. They thus suitably modify the network’s topology without adding new weight parameters, or

deviating from a strict gradient-based optimization framework. By contrast, we deliberately decimate the linear component of the gradient for hidden units in order to focus them on their nonlinear task. We then use shortcuts with *additional* weight parameters to take care of the linear mapping that the hidden units now ignore.

While both these approaches use shortcuts to achieve their ends, from another perspective they appear almost complementary: whereas we eliminate the linear component from our gradient, van der Smagt and Hirzinger in fact *add* just such a component to theirs. It may even be possible to profitably combine the two approaches in a single — admittedly rather complicated — neural network architecture.

10.3 Implementation Techniques

We can distinguish a variety of approaches to centering a variable in a neural network in terms of how the averaging operator $\langle \cdot \rangle$ is implemented. Specifically, averaging may be performed either exactly or approximately, and applied either *a priori*, or adaptively during learning in either batch (deterministic) or online (stochastic) settings:

centering method:	approximate	exact
<i>a priori</i>	<i>by design</i>	<i>extrinsic</i>
adaptive { online batch	<i>running average</i> <i>previous batch</i>	<i>—</i> <i>two-pass, single-pass</i>

10.3.1 A Priori Methods

By Design. Some of the benefits of centering may be reaped without *any* modification of the learning algorithm, simply by setting up the system appropriately. For instance, the hyperbolic tangent (tanh) function with its symmetric range from -1 to 1 will typically produce better-centered output than the commonly used logistic sigmoid $f(y) = 1/(1 + e^{-y})$ ranging from 0 to 1, and is therefore the preferred activation function for hidden units [12]. Similarly, the input representation can (and should) be chosen such that inputs will be roughly centered. When using shortcuts, one may even choose *a priori* to subtract a constant (say, half their maximum) from hidden unit slopes to improve their centering.

We refer to these approximate methods as centering *by design*. Though inexact, they provide convenient and easily implemented tricks to speed up neural network learning. Regardless of whether further acceleration techniques will be required or not, it is generally a good idea to keep centering in mind as a design principle when setting up learning tasks for neural networks.

Extrinsic. Quantities that are extrinsic to the network — *i.e.*, not affected by its weight changes — may often be centered exactly prior to learning. In

particular, for any given training set the network's inputs can be centered in this fashion. Even in online settings where the training set is not known in advance, it is sometimes possible to perform such *extrinsic* centering based upon prior knowledge of the training data: instead of a time series $\mathbf{x}(t)$ one might for instance present the temporal difference signal $\mathbf{x}(t) - \mathbf{x}(t-1)$ as input to the network, which will be centered if $\mathbf{x}(t)$ is stationary.

10.3.2 Adaptive Methods

Online. When learning online, the immediate environment of a single weight within a multi-layer network is highly non-stationary, due to the simultaneous adaptation of other weights, if not due to the learning task itself. A uniquely defined average of some signal $\mathbf{x}(t)$ to be centered is therefore not available online, and we must make do with *running averages* — smoothed versions of the signal itself. A popular smoother is the *exponential trace*

$$\bar{\mathbf{x}}(t+1) = \alpha \bar{\mathbf{x}}(t) + (1-\alpha) \mathbf{x}(t), \quad (10.11)$$

which has the advantage of being *history-free* and *causal*, i.e., requiring neither past nor future values of \mathbf{x} for the present update. The free parameter α (with $0 \leq \alpha \leq 1$) determines the time scale of averaging. Its choice is not trivial: if it is too small, $\bar{\mathbf{x}}$ will be too noisy; if it is too large, the average will lag too far behind the (drifting) signal.

Note that the computational cost of centering by this method is proportional to the number of nodes in the network. In densely connected networks, this is dwarfed by the number of weights, so that the propagation of activities and error signals through these weights dominates the computation. The cost of online centering will therefore make itself felt in small or sparsely connected networks only.

Two-Pass Batch. A simple way to implement exact centering in a batch learning context is to perform *two* passes through the training set for each weight update: the first to calculate the required averages, the second to compute the resulting weight changes. This obviously may increase the computational cost of network training by almost a factor of two. For relatively small networks and training sets, the activity and error for each node and pattern can be stored during the first pass, so that the second pass only consists of the weight update (Equation 10.4). Where this is not possible, a feedforward-only first pass (Equation 10.1) is sufficient to compute average activities and slopes; error centering may then be implemented via one of the other methods described here.

Previous Batch. To avoid the computational overhead of a two-pass method, one can use the averages collected over the *previous* batch in the computation of weight changes for the current batch. This approximation assumes that the averages involved do not change too much from batch to batch; this may result in

stability problems in conjunction with very high learning rates. Computationally this method is quite attractive in that it is cheaper still than the online technique described above. When mini-batches are used for training, both approaches can be combined profitably by centering with an exponential trace over mini-batch averages.

Single-Pass Batch. It is possible to perform exact centering in just a *single* pass through a batch of training patterns. This is done by expanding the triple product of the fully centered batch weight update (*cf.* Equation 10.7). Using f'_j as a shorthand for $f'_j(y_j)$, we have

$$\begin{aligned}\Delta w_{ij} &\propto \langle (x_i - \langle x_i \rangle)(\gamma_j - \langle \gamma_j \rangle)(f'_j - \langle f'_j \rangle) \rangle \\ &= \langle x_i \gamma_j f'_j \rangle - \langle \langle x_i \rangle \gamma_j f'_j \rangle - \langle x_i \langle \gamma_j \rangle f'_j \rangle - \langle x_i \gamma_j \langle f'_j \rangle \rangle \\ &\quad + \langle \langle x_i \rangle \langle \gamma_j \rangle f'_j \rangle + \langle \langle x_i \rangle \gamma_j \langle f'_j \rangle \rangle + \langle x_i \langle \gamma_j \rangle \langle f'_j \rangle \rangle - \langle \langle x_i \rangle \langle \gamma_j \rangle \langle f'_j \rangle \rangle \\ &= \langle x_i \gamma_j f'_j \rangle - \langle x_i \rangle \langle \gamma_j f'_j \rangle - \langle \gamma_j \rangle \langle x_i f'_j \rangle - \langle x_i \gamma_j \rangle \langle f'_j \rangle + 2 \langle x_i \rangle \langle \gamma_j \rangle \langle f'_j \rangle \quad (10.12)\end{aligned}$$

In addition to the ordinary (uncentered) batch weight update term $\langle x_i \gamma_j f'_j \rangle$ and the individual averages $\langle x_i \rangle$, $\langle \gamma_j \rangle$, and $\langle f'_j \rangle$, the single-pass centered update (10.12) thus also requires collection of the sub-products $\langle x_i \gamma_j \rangle$, $\langle x_i f'_j \rangle$, and $\langle \gamma_j f'_j \rangle$. Due to the extra computation involved, the single-pass batch update is not necessarily more efficient than a two-pass method. It is simplified considerably, however, when not all factors are involved — for instance, when activities have already been centered *a priori* so that $\langle x_i \rangle \approx 0$.

Note that the expansion technique shown here may be used to derive an exact single-pass batch method for *any* weight update that involves the addition (or subtraction) of some quantity that must be computed from the entire batch of training patterns. This includes algorithms such as BCM learning [4, 10] and binary information gain optimization [14].

10.4 Empirical Results

While activity centering has long been part of backpropagation lore, and empirical results for error centering have been reported previously [15], slope centering is being proposed for the first time here. It is thus too early to assess its general applicability or utility; here we present a number of experiments designed to show the typical effect that centering has on speed and reliability of convergence as well as generalization performance in feedforward neural networks trained by accelerated backpropagation methods.

The next section describes the general setup and acceleration techniques used in all our experiments. Subsequent sections then present our respective results for two well-known benchmarks: the toy problem of symmetry detection in binary patterns, and a difficult vowel recognition task.

10.4.1 Setup of Experiments

Benchmark Design. For each benchmark task we performed a number of *experiments* to compare performance with *vs.* without various forms of centering. Each experiment consisted of 100 *runs* starting from different initial weights but identical in all other respects. For each run, networks were initialized with random weights from a zero-mean Gaussian distribution with standard deviation 0.3. All experiments were given the same sequence of random numbers for their 100 weight initializations; the seed for this sequence was picked only after the design of the benchmark had been finalized.

Training Modality. In order to make the results as direct an assessment of centering as possible, training was done in batch mode so as to avoid the additional free parameters (*e.g.*, smoothing time constants) required by online methods. Where not done *a priori*, centering was then implemented with the exact two-pass batch method. In addition, we always updated the hidden-to-output weights of the network *before* backpropagating error through them. This is known to sometimes improve convergence behavior [17], and we have found it to increase stability at the large step sizes we desire.

Competitive Controls. The ordinary backpropagation (plain gradient descent) algorithm has many known defects, and a large number of acceleration techniques has been proposed for it. We informally tested a number of such techniques, then picked the combination that achieved the fastest reliable convergence. This combination — *vario-* η and *bold driver* — was then used for all experiments reported here. Thus any performance advantage for centering reported thereafter has been realized *on top of* a state-of-the-art accelerated gradient method as control.

Vario- η [23, page 48]. This interesting technique — also described in Chapter 17 — sets the local learning rate for each weight inversely proportional to the standard deviation of its stochastic gradient. The weight change thus becomes

$$\Delta w_{ij} = \frac{-\eta g_{ij}}{\varrho + \sigma(g_{ij})}, \text{ where } g_{ij} \equiv \frac{\partial E}{\partial w_{ij}} \text{ and } \sigma(u) \equiv \sqrt{\langle u^2 \rangle - \langle u \rangle^2}, \quad (10.13)$$

with the small positive constant ϱ preventing division by near-zero values. Vario- η can be used in both batch and online modes, and is quite effective in that it not only performs gradient normalization, but also adapts step sizes to the level of noise in the local gradient signal.

We used vario- η for all experiments reported here, with $\varrho = 0.1$. In a batch implementation this leaves only one free parameter to be determined: the global learning rate η .

Bold Driver [11, 21, 2, 3]. This algorithm for adapting the global learning rate η is simple and effective, but only works for batch learning. Starting from some initial value, η is increased by a certain factor after each batch in which the error did not increase by more than a very small constant ε (required for numerical stability). Whenever the error rises by more than ε , however, the last weight change is undone, and η decreased sharply.

All experiments reported here were performed using bold driver with a learning rate increment of 2%, a decrement of 50%, and $\varepsilon=10^{-10}$. These values were found to provide fast, reliable convergence across all experiments. Due to the amount of recomputation they require, we do count the “failed” epochs (whose weight changes are subsequently undone) in our performance figures.

10.4.2 Symmetry Detection Problem

In our first benchmark, a fully connected feedforward network with 8 inputs, 8 hidden units and a single output is to learn the symmetry detection task: given an 8-bit binary pattern at the input, it is to signal at the output whether the pattern is symmetric about its middle axis (target = 1) or not (target = 0). This is admittedly a toy problem, although not a trivial one.

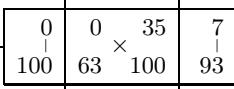
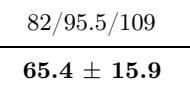
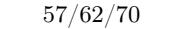
Since the target is binary, we used a logistic output unit and cross-entropy loss function. For each run the network was trained on all 256 possible patterns until the root-mean-square error of its output over the batch fell below 0.01. We recorded the number of epochs required to reach this criterion, but did not test for generalization ability on this task.

Error and Activity Centering. In our first set of experiments we examined the separate and combined effect of centering the network’s activity and/or error signals. For convenience, activity centering was performed *a priori* by using -1 and 1 as input levels, and the hyperbolic tangent (\tanh) as activation function for hidden units. The off-center control experiments were done with 0 and 1 as input levels and the logistic activation function $f(y) = 1/(1 + e^{-y})$. Note that all differences between the \tanh and logistic nonlinearities are eliminated by the vario- η algorithm, *except* for the eccentricity of their respective outputs.

Results. Table 10.1 shows that centering either activity or error signals produced an approximate 7-fold increase in convergence speed. In no instance was a run that used one (or both) of these centering methods slower than the corresponding control without centering. The similar magnitude of the speed-up suggests that it may be due to the improved conditioning of the Hessian achieved by centering either errors or activities (see Section 10.5). Note, however, that activity centering beat error centering almost 2/3 of the time in the direct comparison.

On the other hand, error centering appeared to improve the *reliability* of convergence: it cut the convergence time’s coefficient of variation (the ratio between its standard deviation and mean, henceforth: c.v.) in half while activity centering

Table 10.1. The effect of centering activities and/or error signals on the symmetry detection task without shortcuts. Reported are the empirical mean, standard deviation, and 25th/50th/75th percentile (rounded to three significant digits) of the number of epochs required to converge to criterion. Also shown is the result of directly comparing runs with identical random seeds. The number of runs in each comparison may sum to less than 100 due to ties.

error signals: activities:	conventional		centered	
	mean ± st.d. quartiles	direct comparison: # of faster runs		mean ± st.d. quartiles
off-center (0/1)	669 ± 308 453/580/852	0 – 100 	0 – 100 	97.5 ± 21.8 82/95.5/109
centered (-1/1)	93.1 ± 46.7 67.5/79.5/94	100 – 93 	14 – 84 	65.4 ± 15.9 57/62/70

left it unchanged. We speculate that this may be the beneficial effect of centering on the *backpropagated* error, which does not occur for activity centering.

Finally, a further speedup of 50% (while maintaining the lower c.v.) occurred when both activity and error signals were centered. This may be attributed to the fact that our centering of hidden unit activity *by design* (*cf.* Section 10.3) was only approximate. To assess the significance of these effects, note that since the data was collected over 100 runs, the standard error of the reported mean time to convergence is $1/\sqrt{100} = 1/10$ its reported standard deviation.

Shortcuts and Slope Centering. In the second set of experiments we left both activity and error signals centered, and examined the separate and combined effect of adding shortcuts and/or slope centering. Note that since the complement of a symmetric bit pattern is also symmetric, the symmetry detection task has *no* linear component at all — we would therefore expect shortcuts to be of minimal benefit here.

Results. Table 10.2 shows that indeed adding shortcuts alone was not beneficial — in fact it slowed down convergence in over 80% of the cases, and significantly increased the c.v. Subsequent addition of slope centering, however, brought about an almost 3-fold increase in learning speed, and restored the original c.v. of about 1/4. When used together, slope centering and shortcuts never increased convergence time, and on average cut it in half. By contrast, slope centering *without* shortcuts failed to converge at all about 1/3 of the time. This may come as a surprise, considering that the given task had no linear component. However, consider the following:

Table 10.2. The effect of centering slopes and/or adding shortcuts on the symmetry detection task with centered activity and error signals. Results are shown in the same manner as in Table 10.1.

topology: slopes:	conventional		centered	
	mean ± st.d. quartiles	direct comparison: # of faster runs	mean ± st.d. quartiles	
short- cuts? no	65.4 ± 15.9 57/62/70	52 – 48 81 17	*	51.6 ± 16.2 43/64.5/∞
	90.4 ± 31.1 69.5/80/102	0 × 61 39 99 0 – 100	4 95	33.1 ± 8.6 28/31/35

* Mean and standard deviation exclude 34 runs which did not converge.

Need for Shortcuts. Due to the monotonicity of their nonlinear transfer function, hidden units always carry some linear moment, in the sense of a positive correlation between their net input and output. In the absence of shortcuts, the hidden units must arrange themselves so that their linear moments together match the overall linear component of the task (here: zero). This adaptation process is normally driven by the linear component of the error — which slope centering removes.

The remaining nonlinear error signals can still jostle the hidden units into an overall solution, but such an indirect process is bound to be unreliable: as it literally removes slope from the error surface, slope centering creates numerous local minima. Shortcut weights turn these local minima into global ones by modeling the missing (linear) component of the gradient, thereby freeing the hidden units from any responsibility to do so.

In summary, while a network without shortcuts trained with slope centering may converge to a solution, the addition of shortcut weights is necessary to ensure that slope centering will not be detrimental to the learning process. Conversely, slope centering can prevent shortcuts from acting as redundant “detractors” that impede learning instead of assisting it. These two techniques should therefore always be used in conjunction.

10.4.3 Vowel Recognition Problem

Our positive experiences with centering on the symmetry detection task immediately raise two further questions: 1) will these results transfer to more challenging, realistic problems, and 2) is the gain in learning speed — as often happens — bought at the expense of generalization ability? In order to address these questions, we conducted further experiments with the speaker-independent vowel recognition data due to Deterding [5], a popular benchmark for which good generalization performance is rather difficult to achieve.

The Task. The network’s task is to recognize the eleven steady-state vowels of British English in a speaker-independent fashion, given 10 spectral features (specifically: LPC-derived log area ratios) of the speech signal. The data consists of 990 patterns to be classified: 6 instances for each of the 11 vowels spoken by each of 15 speakers. We follow the convention of splitting it into a training set containing the data from the first 8 (4 male, 4 female) speakers, and a test set containing those of the remaining 7 (4 male, 3 female). Note that there is no separate validation set available.

Prior Work. Robinson [13] pioneered the use of Deterding’s data as a benchmark by comparing the performance of a number of neural network architectures on it. Interestingly, none of his methods could outperform the primitive single nearest neighbor approach (which misclassifies 44% of test patterns), thus posing a challenge to the pattern recognition community. Trained on the task as formulated above, conventional backpropagation networks in fact appear to reach their limits at error rates of around 42% [6, 9], while an adaptive nearest neighbor technique can achieve 38% [7]. In Chapter 7, Flake reports comparably favorable results for RBF networks as well as his own hybrid architectures. Even better performance can be obtained by using speaker sex/identity information [19, 20], or by training a separate model for each vowel [8]. By combining these two approaches, a test set error of 23% has been reached [18], the lowest we are aware of to date.

Training and Testing. We trained fully connected feedforward networks with 10 inputs, 22 hidden units, and 11 logistic output units by minimization of cross-entropy loss. The target was 1 for the output corresponding to the correct vowel, 0 for all others. Activity centering was done *a priori* by explicitly centering the inputs (separately for training and test set), and by using the tanh nonlinearity for hidden units. The uncentered control experiments used the original input data, and logistic activation functions.

The relatively small size of our networks enabled us to run all experiments out to 2000 epochs of training. After each epoch, the network’s generalization ability was measured in terms of its misclassification rate on the test set. For the purpose of testing, a maximum likelihood approach was adopted: the network’s highest output for a given test pattern was taken to indicate its classification of that pattern.

First Results. Figure 10.1 shows how the average test set error (over 100 runs) evolved during training in each of the 8 experiments we performed for this benchmark. For all curves, error bars were at most the size of the marks shown along the curve, and have therefore been omitted for clarity. Following our experience on the symmetry detection task, shortcuts and slope centering were always used in conjunction whereas activity and error centering were examined independently. The following effects can be discerned:

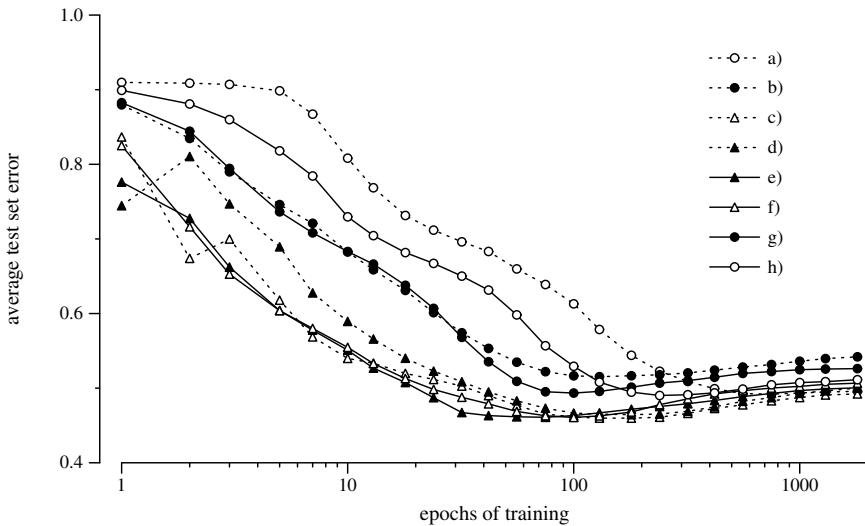


Fig. 10.1. Evolution of the average test set error while learning the vowel recognition task with activity centering (triangular marks), error centering (filled marks), and/or slope centering with shortcut weights (solid lines), *vs.* their uncentered controls. Experiments are denoted a)–h) as in Table 10.3.

1. All experiments with activity centering (triangular marks) clearly outperformed all experiments without it (circular marks) in both average convergence speed and minimum average test set error.
2. All experiments with shortcuts and slope centering (solid lines) outperformed the corresponding experiment without them (dashed lines).
3. With one notable exception (experiment d), error centering (filled marks) sped up convergence significantly. Its effect was greatest in the experiments without activity centering.
4. The best experiment in terms of both convergence speed and minimum average test set error was e), the fully centered one; the worst was a), the fully conventional one.

The qualitative picture that emerges is that centering appears to significantly speed up convergence without adversely affecting the trained network's generalization ability. We will now attempt to quantify this finding.

Quantifying the Effect. Since the curves in Figure 10.1 are in fact superpositions of 100 nonlinear curves each, they are ill-suited to quantitative analysis: value and location of the minimum average test set error do not tell us anything about the distribution of such minima across individual runs — not even their average value or location. In order to obtain such quantitative results, we need to identify an appropriate minimum in test set error for each run. This will allow us to directly compare runs with identical initial weights across experiments,

as well as to characterize the distribution of minima within each experiment by aggregate statistics (*e.g.*, mean, standard deviation, quartiles) for both the minimum test set error, and the time taken to reach it.

A fair and consistent strategy to identify minima suitable for the quantitative comparisons we have in mind is not trivial to design. Individual runs may have multiple minima in test set error, or none at all. If we were to just use the global minimum over the duration of the run (2 000 epochs), we would not be able to distinguish a fast method which makes some insignificant improvement to a long-found minimum late in the run from a slow method which takes that long to reach its first minimum. Given that we are concerned with both the quality of generalization performance and the speed with which it is achieved, a greedy strategy for picking appropriate minima is indicated.

Identification of Minima. We follow the evolution of test set error over the course of each run, noting new minima as we encounter them. If the best value found so far is not improved upon within a certain period of time, we pick it as *the* minimum of that run for the purpose of quantitative analysis. The appropriate length of waiting period before giving up on further improvement is a difficult issue — see Chapter 2 for a discussion. For a fair comparison between faster and slower optimization methods, it should be proportional to the time it took to reach the minimum in question: a slow run then has correspondingly more time to improve its solution than a fast one.

Unfortunately this approach fails if a minimum of test set error occurs during the initial transient, within the first few epochs of training: the waiting period would then be too short, causing us to give up prematurely. On the other hand, we cannot wait longer than the overall duration of the run. We therefore stop looking for further improvement in a run after $\min(2\,000, 2\epsilon + 100)$ epochs, where ϵ records when the network first achieved the lowest test set error seen so far in that run. Only 9 out of the 800 runs reported here expired at the upper limit of 2 000 epochs, so we are confident that its imposition did not significantly skew our results.

Test Set Used as Validation Set. Note that since we use the test set to determine at which point to compare performance, we have effectively appropriated it as a validation set. The minimum test set errors reported below are therefore *not* unbiased estimators for the network’s ability to generalize to novel speakers, and should not be compared to proper measurements of this ability (for which the test set must not affect the training procedure in any way). Nonetheless, let us not forget that the lowest test set error does measure the network’s generalization ability in a consistent fashion after all: even though these scores are all biased to favor a particular set of novel speakers (the test set), by no means does this render their comparison *against each other* insignificant.

Overview of Results. Table 10.3 summarizes quantitative results obtained in this fashion for the vowel recognition problem. To assess their significance, recall

Table 10.3. Minimum test set error (misclassification rate in %), and the number of epochs required to reach it, for the vowel recognition task. Except for the different layout, results are reported in the same manner as in Tables 10.1 and 10.2. Due to space limitations, only selected pairs of experiments are compared directly.

experiment	features:			performance measure:									
	centering		shortcuts	minimum test set error					epochs required				
	activ.	error		mean ± st.d.	dir. comparison:	quartiles	# of better runs	mean ± st.d.	dir. comparison:	quartiles	# of faster runs		
a)				48.0 ± 3.6 45.7/47.3/50.0		67	17	13	19	37		554 ± 321 365/486/691	3 97
b)		✓		49.1 ± 2.9 47.0/49.6/50.9		31						125 ± 82 67.5/104/163	51 90
c)	✓			43.9 ± 2.5 42.3/43.9/46.0		10	82					156 ± 110 75/137/215	47 48
d)	✓	✓		44.3 ± 2.3 42.9/44.2/45.9		89	51	84				158 ± 141 72/124/186	52 96
e)	✓	✓	✓	44.2 ± 2.5 42.3/44.4/46.3			46	65	80			72.4 ± 55.5 37.5/51.5/81	21 78
f)	✓		✓	44.2 ± 2.8 42.3/44.4/46.1			49	49	68			113 ± 64 69.5/97.5/148	75 24
g)		✓	✓	46.8 ± 3.7 44.0/47.0/48.9		27	47					126 ± 139 64/94/138	88 22
h)			✓	46.5 ± 3.1 44.5/46.8/48.5		43	51					270 ± 164 162/235/316	84 15
						56	31	29	30	61			12 15

that the standard error in the mean of a performance measure reported here is $1/\sqrt{100} = 1/10$ of its reported standard deviation. Figure 10.2 depicts the same data (except for the direct comparisons) graphically in form of cumulative histograms for the minimum test set error and the number of epochs required to reach it.

The results generally confirm the trends observed in Figure 10.1. Runs in the fully centered experiment e) clearly converged most rapidly — and to test set errors that were among the best. Compared to the conventional setup a), full centering converged almost 8 times faster on average while generalizing better 80% of the time.

Generalization Performance. Average misclassification rates on the test set ranged from 44% to 49%, which we consider a fair result given our comparatively small networks. They cluster into three groups: networks with activity centering

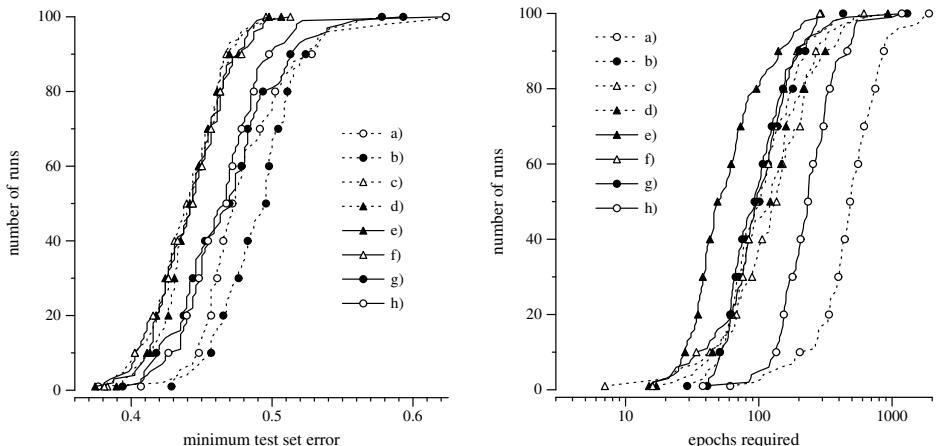


Fig. 10.2. Cumulative histograms for the minimum test set error (left), and the number of epochs required to reach it (right), for the vowel recognition task. Curves are labelled as in Figure 10.1, and marked every 10th percentile.

achieved around 44%, the two others with shortcuts and slope centering came in under 47%, while the remaining two only reached 48–49%. The cumulative histogram (Figure 10.2, left) shows that all activity-centered networks had an almost identical distribution of minimum test set errors.

Note that centering the inputs changes the task, and that the addition of shortcuts changes the network topology. It is possible that this — rather than centering *per se* — accounts for their beneficial effect on generalization. Error centering was the one feature in our experiments that changed the dynamics of learning exclusively. Its addition appeared to slightly *worsen* generalization, particularly in the absence of other forms of centering. This could be caused by a reduction (due to centering) of the effective number of parameters in what is already a rather small model. Such an effect should not overly concern us: one could easily recoup the lost degrees of freedom by slightly increasing the number of hidden units for centered networks.

Convergence Speed. All three forms of centering examined here clearly sped up convergence, both individually and in combination. A slight anomaly appeared in that the addition of error centering in going from experiment c) to d) had no significant effect on the average number of epochs required. A look at the cumulative histogram (Figure 10.2, right) reveals that while experiment d) is ahead between the 20th and 80th percentile, c) had fewer unusually slow runs than d), and a few exceptionally fast ones.

With the other forms of centering in place, the addition of error centering was unequivocally beneficial: average convergence time decreased from 113 epochs in f) to 72.4 epochs in e). The histogram shows that the fully centered e) is far ahead of the competition through almost the entire percentile range.

Finally, it is interesting to note that the addition of shortcuts and slope centering, both on their own and to a network with activity and error centering, roughly doubled the convergence speed — the same magnitude of effect as observed on the symmetry detection task.

10.5 Discussion

The preceding section has shown that centering can indeed have beneficial effects on the learning speed and generalization ability of a neural network. Why is this so? In what follows, we offer an explanation from three (partially overlapping) perspectives, considering in turn the effect of centering on the condition number of the Hessian, the level of noise in the local gradient, and the credit assignment between different parts of the network.

Conditioning the Hessian. It is well known that the minimal convergence time for first-order gradient descent on quadratic error surfaces is inversely related to the condition number of the Hessian matrix, *i.e.*, the ratio between its largest and its smallest eigenvalue. A common strategy for accelerating gradient descent is therefore to seek to improve the condition number of the Hessian.

For a single linear node $y = \mathbf{w}^T \mathbf{x}$ with squared loss function, the Hessian is simply the covariance matrix of the inputs: $H = \langle \mathbf{x} \mathbf{x}^T \rangle$. Its largest eigenvalue is typically caused by the d.c. component of \mathbf{x} [12]. Centering the inputs removes that eigenvalue, thus conditioning the Hessian and permitting larger step sizes (*cf.* Chapter 1). For batch learning, error centering has exactly the same effect on the local weight update:

$$\Delta \mathbf{w} \propto \langle (\delta - \langle \delta \rangle) \mathbf{x} \rangle = \langle \delta \mathbf{x} \rangle - \langle \delta \rangle \langle \mathbf{x} \rangle = \langle \delta (\mathbf{x} - \langle \mathbf{x} \rangle) \rangle \quad (10.14)$$

Error centering does go further than activity centering, however, in that it also affects the error backpropagated to anterior nodes. Moreover, Equation 10.14 does not hold for online learning, where the gradient is noisy.

Noise Reduction. It can be shown that centering improves the signal-to-noise ratio of the local gradient. Omitting the slope factor for the sake of simplicity, consider the noisy weight update

$$\Delta w_{ij} \propto (\delta_j + \phi)(x_i + \xi) = \delta_j x_i + \xi \delta_j + \phi x_i + \phi \xi \quad (10.15)$$

where ϕ and ξ are the noise terms, presumed to be zero-mean, and independent of activity, error, and each other. In the expansion on the right-hand side, the first term is the desired (noise-free) weight update while the others represent noise that contaminates it. While the last (pure noise) term cannot be helped, we can reduce the variance of the two mixed terms by centering δ_j and x_i so as to minimize $\langle \delta_j^2 \rangle$ and $\langle x_i^2 \rangle$, respectively.

One might of course contend that in doing so, we are also shrinking the signal $\delta_j x_i$, so that in terms of the signal-to-noise ratio we are no better — in fact,

worse — off than before. This cuts right to the heart of the matter, for centering rests upon the notion that the error signal relevant to a non-bias, non-shortcut weight *is* the fully centered weight update, and that any d.c. components in $\delta_j x_i$ should therefore also be regarded as a form of noise. This presumption can of course be maintained only because we do have bias and shortcut weights to address the error components that centering removes.

Improved Credit Assignment. From the perspective of a network that has these additional parameters, then, centering is a way to improve the assignment of responsibility for the network’s errors: constant errors are shunted to the bias weights, linear errors to the shortcut weights, and the remainder of the network is bothered only with those parts of the error signal that actually require a nonlinearity. Centering thus views hidden units as a scarce resource that should only be called upon where necessary. Given the computational complications that arise in the training of nonlinear nodes, we submit that this is an appropriate and productive viewpoint.

Future Work. While the results reported here are quite promising, more experiments are required to assess the general applicability and effectiveness of centering. For feedforward networks, we would like to explore the use of centering with multiple hidden layers, stochastic (online) gradient descent, and for function approximation (rather than classification) problems. The centering approach *per se*, however, is rather more general than that, and so further ahead we anticipate its application to a range of more sophisticated network architectures, learning algorithms, and problem domains.

Acknowledgments. I would like to thank the editors of this book as well as my colleagues Jürgen Schmidhuber, Marco Wiering, and Rafał Sałustowicz for their helpful comments. This work was supported by the Swiss National Science Foundation under grant numbers 2100–045700.95/1 and 2000–052678.97/1.

References

- [1] Anderson, J., Rosenfeld, E. (eds.): *Neurocomputing: Foundations of Research*. MIT Press, Cambridge (1988)
- [2] Battiti, R.: Accelerated back-propagation learning: Two optimization methods. *Complex Systems* 3, 331–342 (1989)
- [3] Battiti, R.: First- and second-order methods for learning: Between steepest descent and Newton’s method. *Neural Computation* 4(2), 141–166 (1992)
- [4] Bienenstock, E., Cooper, L., Munro, P.: Theory for the development of neuron selectivity: Orientation specificity and binocular interaction in visual cortex. *Journal of Neuroscience* 2 (1982); Reprinted in [1]
- [5] Deterding, D.H.: Speaker Normalisation for Automatic Speech Recognition. PhD thesis, University of Cambridge (1989)

- [6] Finke, M., Müller, K.-R.: Estimating a-posteriori probabilities using stochastic network models. In: Mozer, M.C., Smolensky, P., Touretzky, D.S., Elman, J.L., Weigend, A.S. (eds.) *Proceedings of the 1993 Connectionist Models Summer School*, Boulder, CO. Lawrence Erlbaum Associates, Hillsdale (1994)
- [7] Hastie, T.J., Tibshirani, R.J.: Discriminant adaptive nearest neighbor classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 18(6), 607–616 (1996)
- [8] Herrmann, M.: On the merits of topography in neural maps. In: Kohonen, T. (ed.) *Proceedings of the Workshop on Self-Organizing Maps*, pp. 112–117. Helsinki University of Technology (1997)
- [9] Hochreiter, S., Schmidhuber, J.: Feature extraction through lococode. *Neural Computation* (1998) (to appear)
- [10] Intrator, N.: Feature extraction using an unsupervised neural network. *Neural Computation* 4(1), 98–107 (1992)
- [11] Lapedes, A., Farber, R.: A self-optimizing, nonsymmetrical neural net for content addressable memory and pattern recognition. *Physica D* 22, 247–259 (1986)
- [12] LeCun, Y., Kanter, I., Solla, S.A.: Eigenvalues of covariance matrices: Application to neural-network learning. *Physical Review Letters* 66(18), 2396–2399 (1991)
- [13] Robinson, A.J.: Dynamic Error Propagation Networks. PhD thesis, University of Cambridge (1989)
- [14] Schraudolph, N.N., Sejnowski, T.J.: Unsupervised discrimination of clustered data via optimization of binary information gain. In: Hanson, S.J., Cowan, J.D., Giles, C.L. (eds.) *Advances in Neural Information Processing Systems*, vol. 5, pp. 499–506. Morgan Kaufmann, San Mateo (1993)
- [15] Schraudolph, N.N., Sejnowski, T.J.: Tempering backpropagation networks: Not all weights are created equal. In: Touretzky, D.S., Mozer, M.C., Hasselmo, M.E. (eds.) *Advances in Neural Information Processing Systems*, vol. 8, pp. 563–569. MIT Press, Cambridge (1996)
- [16] Sejnowski, T.J.: Storing covariance with nonlinearly interacting neurons. *Journal of Mathematical Biology* 4, 303–321 (1977)
- [17] Shah, S., Palmieri, F., Datum, M.: Optimal filtering algorithms for fast learning in feedforward neural networks. *Neural Networks* 5, 779–787 (1992)
- [18] Tenenbaum, J.B., Freeman, W.T.: Separating style and content. In: Mozer, M.C., Jordan, M.I., Petsche, T. (eds.) *Advances in Neural Information Processing Systems*, vol. 9, pp. 662–668. The MIT Press, Cambridge (1997)
- [19] Turney, P.D.: Exploiting Context When Learning to Classify. In: Brazdil, P.B. (ed.) *ECML 1993. LNCS*, vol. 667, pp. 402–407. Springer, Heidelberg (1993)
- [20] Turney, P.D.: Robust classification with context-sensitive features. In: *Proceedings of the Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pp. 268–276 (1993)
- [21] Vogl, T.P., Mangis, J.K., Rigler, A.K., Zink, W.T., Alkon, D.L.: Accelerating the convergence of the back-propagation method. *Biological Cybernetics* 59, 257–263 (1988)
- [22] Widrow, B., McCool, J.M., Larimore, M.G., Johnson Jr., C.R.: Stationary and nonstationary learning characteristics of the LMS adaptive filter. *Proceedings of the IEEE* 64(8), 1151–1162 (1976)
- [23] Zimmermann, H.G.: Neuronale Netze als Entscheidungskalkül. In: Rehkugler, H., Zimmermann, H.G. (eds.) *Neuronale Netze in der Ökonomie: Grundlagen und finanzwirtschaftliche Anwendungen*, pp. 1–87. Vahlen Verlag, Munich (1994)

Avoiding Roundoff Error in Backpropagating Derivatives*

Tony Plate

School of Mathematical and Computing Sciences,
 Victoria University, Wellington, New Zealand
`tap@mcs.vuw.ac.nz`
`http://www.mcs.vuw.ac.nz/~tap/`

Abstract. One significant source of roundoff error in backpropagation networks is the calculation of derivatives of unit outputs with respect to their total inputs. The roundoff error can lead result in high relative error in derivatives, and in particular, derivatives being calculated to be zero when in fact they are small but non-zero. This roundoff error is easily avoided with a simple programming trick which has a small memory overhead (one or two extra floating point numbers per unit) and an insignificant computational overhead.

11.1 Introduction

Backpropagating derivatives is an essential part of training multilayer networks. Accuracy of these derivatives is important to many training methods, especially ones which use second-order information, such as conjugate gradients. The standard formula for backpropagating error derivatives (eg., as given in Ripley [3], Bishop [1], and Rumelhart, Hinton, and Williams [4]) use floating point arithmetic in such a way that can result in significant roundoff error. In particular, small derivatives are rounded off to zero. These errors can cause second-order methods to become confused (because of inaccurate gradients) and can also cause the weight search to stop or be very slow because some derivatives are calculated to be zero when in fact they are small but non-zero. This chapter explains how this particular source of roundoff error can be avoided simply and cheaply. The method applies to both logistic and tanh units, and to the sum-squared and cross-entropy error functions.

In this chapter, the symbol “=” is used to denote mathematical equality, and the symbol “ \leftarrow ” is used to denote an assignment to some floating-point variable. Floating point values are denoted by an asterisk, eg., x_i^* is the floating point version of x_i .

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

11.2 Roundoff Error in Sigmoid Units

Consider a non-input unit whose output y_i is computed as the logistic function of its total input x_i :

$$y_i = \frac{1}{1 + \exp(-x_i)} \quad \text{and} \quad y_i^* \leftarrow \frac{1}{1 + \exp(-x_i^*)}.$$

In the backpropagation phase of training, we need to calculate the partial derivative of the error with respect to the total input. This is done using the chain rule:

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial x_i}.$$

The standard way of calculating $\frac{\partial y_i}{\partial x_i}$ is to use the formula relating it to y_i , which for the logistic function is

$$\left(\frac{\partial y_i}{\partial x_i} \right)^* \leftarrow y_i^*(1 - y_i^*).$$

This computation is a potential source of roundoff error: if the actual value of y_i is so close to one that y_i^* is exactly one, then $(\frac{\partial y_i}{\partial x_i})^*$ will equal zero.

Figures 11.1 show the values of this expression calculated in single and double precision floating point arithmetic. In single precision, when x_i is greater than about 17.33, $y_i^*(1 - y_i^*)$ evaluates to zero. For x_i values slightly lower than 17.33 there is significant quantization. In double precision $y_i^*(1 - y_i^*)$ evaluates to zero when x_i is greater than about 36.74.

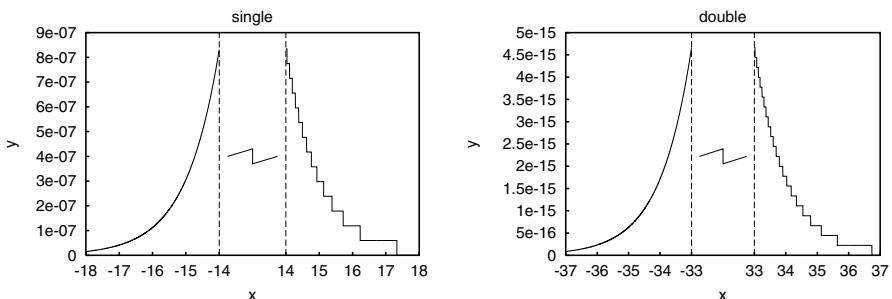


Fig. 11.1. Quantization in calculated values of unit derivatives $(\frac{\partial y}{\partial x})$ for logistic units, computed using the formula $y^*(1 - y^*)$ where $y^* \leftarrow 1/(1 + \exp(-x^*))$. Roundoff error causes quantization on the right hand side of each plot.

Note that in Figure 11.1 the roundoff error due to the calculation of $1 - y^*$ only occurs for positive x . For negative x , y^* approaches zero, and is accurately represented, and the relative roundoff error in $1 - y^*$ is insignificant. This provides a clue as to how to avoid the roundoff error for positive x : don't compute $1 - y^*$

when y^* is close to one. Indeed, these roundoff errors can be avoided entirely by storing an extra value with each unit, which is $z_i = 1 - y_i$, calculated accurately in floating point arithmetic as follows:

$$y_i^* \leftarrow \frac{1}{1 + \exp(-x_i^*)} \quad \text{and} \quad z_i^* \leftarrow \frac{\exp(-x_i^*)}{1 + \exp(-x_i^*)}.$$

Together, these two floating points numbers accurately represent the unit output at its extremes: y_i^* is an accurate representation when the output is close to zero, and z_i^* is an accurate representation of one minus the output when the output is close to one. With them, $\frac{\partial y_i}{\partial x_i}$ is simply and accurately calculated as follows:

$$\left(\frac{\partial y_i}{\partial x_i} \right)^* \leftarrow y_i^* z_i^*.$$

Implementing these calculations requires the storage of one extra floating point number per unit (i.e., z_i^*), and the computation of one extra division per unit. This extra resource usage is insignificant because the total resource requirements for a forward- and back-propagation pass are proportional to the number of weights in the network, which is usually of the order of the square of the number of units.

The value y_i is also used in calculating partial derivatives for weights on connections emerging from unit i . However, the errors in the representations of values of y_i close to one do not cause high relative errors in these weight derivatives (except in special circumstances where derivatives from different examples cancel each other, but there is no simple remedy in these rare situations). Hence, it is generally safe to use just y_i^* for these calculations.

11.2.1 Sum-Squared Error Computations

Roundoff errors can also occur in the calculation of errors and their derivatives. This is unlikely to be important if targets are rounded-off versions of true targets, e.g., targets like 0.3129871 or 0.9817523, because such targets have as much roundoff error as the values of unit outputs.

However, if sum-squared error is used, and targets are all 0 or 1 and are accurate (i.e., 1 is not a rounded-off version of 0.99999999 or 1.00000001), and unit i computes a logistic function of its total input, then the following formulas can be used (where t_i is the target for unit i , $E = \frac{1}{2}(t_i - y_i)^2$, and $z_i = 1 - y_i$, calculated as before):

$$E^* \leftarrow \begin{cases} \frac{1}{2}(t_i^* - y_i^*)^2 & \text{if } y_i^* \leq 0.5 \\ \frac{1}{2}((t_i^* - 1) + z_i^*)^2 & \text{if } y_i^* > 0.5 \end{cases}$$

$$\left(\frac{\partial E}{\partial y_i} \right)^* \leftarrow \begin{cases} y_i^* - t_i^* & \text{if } y_i^* \leq 0.5 \\ (1 - t_i^*) - z_i^* & \text{if } y_i^* > 0.5 \end{cases}$$

These formulas could be split into a greater number of simpler cases if t_i were always 0 or 1, but as they are they are correct for general values of t_i and accurate when t_i is 0 or 1.

11.2.2 Single Logistic-Output Cross-Entropy Computations

If the network has a single output unit (unit i) which represents probability in a two-class classification problem, then the cross-entropy error function [1] for one example is as follows:

$$E = t_i \log y_i + (1 - t_i) \log(1 - y_i).$$

This error function is most appropriately used with a logistic function on the output unit. In this case, errors and partial derivatives can be calculated from the following formulas, and the calculations will be accurate for $t_i = 0$ or 1 (where $z_i = 1 - y_i$, calculated as before). To get accurate results, it is necessary to use the function¹ $\text{log1p}(x)$, which computes $\log(1 + x)$ accurately for tiny x .

$$E^* \leftarrow \begin{cases} t_i^* \log y_i^* + (1 - t_i^*) \log1p(-y_i^*) & \text{if } y_i^* \leq 0.5 \\ t_i^* \log1p(-z_i^*) + (1 - t_i^*) \log z_i^* & \text{if } y_i^* > 0.5 \end{cases}$$

$$\left(\frac{\partial E}{\partial x_i} \right)^* \leftarrow \begin{cases} y_i^* - t_i^* & \text{if } y_i^* \leq 0.5 \\ 1 - t_i^* - z_i^* & \text{if } y_i^* > 0.5 \end{cases}$$

11.2.3 Other Approaches to Avoiding Zero-Derivatives with the Logistic Function

Fahlman [2] suggested adding 0.1 to $\frac{\partial y_i}{\partial x_i}$ in order to decrease learning time by eliminating flat spots in the training surface (ie., spots with small derivatives), and also to avoid zero-derivatives due to roundoff. In Fahlman's experiments this technique improved the learning time. However, it also makes the derivatives incorrect with respect to the error function. This may not matter in networks where the actual output numbers do not mean very much other than "high" or "low". However, in networks where achieving accurate estimates of the targets is important, eg., where the targets are probabilities, or targets are continuous values in a modeling task, this technique is undesirable as it causes the gradient search to not minimize the error, but to minimize some other quantity instead.

Another technique sometimes used with neural networks is to transform targets from the range $[0, 1]$ to $[0.1, 0.9]$. Again, a secondary motivation is to avoid zero-derivatives, and again, this technique should not be used where actual output values have any more significance than "high" or "low".

11.3 Softmax and Cross-Entropy Computations

In networks which must classify instances into one of k mutually exclusive classes, cross-entropy is often used as the error measure, together with the softmax output function. The softmax output function allows the outputs of the k output units to be interpreted as probabilities: it forces each output to be between zero

¹ $\text{log1p}(x)$ is available in Unix math libraries.

and one, and forces their total to be one. Assuming that outputs units are numbered 1 to k , the equations for softmax and cross-entropy (for one example) are as follows:

$$y_i = \frac{\exp(x_i)}{\sum_{j=1}^k \exp(x_j)} \quad \text{and} \quad E = \sum_{j=1}^k t_j \log y_j.$$

The derivative of the error with respect to x has a very simple form:

$$\frac{\partial E}{\partial x_i} = y_i - t_i.$$

This equation is subject to high relative error due to roundoff when t_i is exactly one and y_i is close to one. This can happen often in training neural nets, and can lead to derivatives being calculated as zero when in fact they are just small.

This roundoff error, and also possible overflow in the computation of y_i , can be avoided by using the following computations, which use extra floating point variables to store the values of $z_i = 1 - y_i$ and $x'_i = x_i - \max_j x_j$:

$$\begin{aligned} m^* &\leftarrow \max_j x_j^* & z_i^* &\leftarrow \begin{cases} 1 - y_i^* & \text{if } y_i^* \leq 0.5 \\ \frac{1}{Q^*} \sum_{j \neq i} \exp(x_j'^*) & \text{if } y_i^* > 0.5 \end{cases} \\ x'_i &\leftarrow x_i^* - m^* & \left(\frac{\partial E}{\partial x_i} \right)^* &\leftarrow \begin{cases} y_i^* - t_i^* & \text{if } y_i^* \leq 0.5 \\ (1 - t_i^*) - z_i^* & \text{if } y_i^* > 0.5 \end{cases} \\ Q^* &\leftarrow \sum_j \exp(x_j'^*) & E^* &\leftarrow \sum_j \begin{cases} t_j^* \log(y_j^*) & \text{if } y_i^* \leq 0.5 \\ t_j^* \log1p(-z_j^*) & \text{if } y_i^* > 0.5 \end{cases} \\ y_i^* &\leftarrow \frac{1}{Q^*} \exp(x_i'^*) \end{aligned}$$

The space overhead is low: at most two extra floating point variables per output unit, depending on the implementation. The time overhead is also low. The only lengthy additional computation is the second alternative for z_i^* , which can be performed for at most one i because it is impossible for more than one y_i to be greater than 0.5.

11.4 Roundoff Error in Tanh Units

The same ideas apply to the calculation of derivatives in tanh units, which are described by the following equations:

$$y_i = \frac{1 - \exp(-2x_i)}{1 + \exp(-2x_i)} \quad \text{and} \quad \frac{\partial y_i}{\partial x_i} = (1 - y_i)(1 + y_i).$$

For tanh units, the output y_i can take on values between -1 and 1. Hence, to represent outputs accurately at the extremes we need two extra floating point numbers

to store the values $z_i = 1 - y_i$ and $u_i = 1 + y_i$. Then the following expressions evaluated in floating point arithmetic will avoid unnecessary roundoff error:

$$\begin{aligned} v_i^* &\leftarrow \exp(-2x_i^*) & y_i^* &\leftarrow u_i^* - 1 \\ u_i^* &\leftarrow \frac{2}{1 + v_i^*} & \left(\frac{\partial y_i}{\partial x_i}\right)^* &\leftarrow z_i^* u_i^* \\ z_i^* &\leftarrow v_i^* u_i^* \end{aligned}$$

The tanh function is not often used on output units, but if desired, formulas for accurately calculating errors and derivatives when targets are always 1 or -1 are easily derived.

11.5 Why Bother?

Since the derivatives which are affected by roundoff error are very small, the proposal to calculate them accurately might provoke the response “why bother?” For on-line learning methods (stochastic gradient), there is probably no point, as computing small gradients to high accuracy is unlikely to make any difference. However, for nets trained in batch mode with second-order methods such as conjugate gradients, small derivatives can be quite important, for the following reasons:

1. Quantization in error or derivatives can confuse line search routines, so avoiding quantization is a good thing.
2. Many small derivatives can add up.
3. Computing small but non-zero derivatives allows training methods to continue as opposed to stopping because of zero derivatives. Some training methods can make significant progress with small derivatives, so it is possible that the weights will move out of the flat area of the error function.

Indeed, there is little reason not to compute these values accurately, as the extra storage and computations are insignificant.

References

- [1] Bishop, C.: Neural Networks for Pattern Recognition. Oxford University Press (1995)
- [2] Fahlman, S.E.: Fast-learning variations on back-propagation: An empirical study. In: Touretzky, D., Hinton, G., Sejnowski, T. (eds.) Proceedings of the 1988 Connectionist Models Summer School, pp. 38–51. Morgan Kaufmann, San Mateo (1989)
- [3] Ripley, B.D.: Pattern Recognition and Neural Networks. Cambridge University Press (1995)
- [4] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In: Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1, pp. 318–362. MIT Press, Cambridge (1986)

Representing and Incorporating Prior Knowledge in Neural Network Training*

Preface

The present section focuses on tricks for four important aspects in learning: (1) incorporation of prior knowledge, (2) choice of representation for the learning task, (3) unequal class prior distributions, and finally (4) large network training.

Patrice Simard, et al. review the famous **tangent distance** and **tangent propagation algorithms**. Included are tricks for speeding and increasing the stability that were not published in previous work. One important trick for obtaining good performance in their methods is **smoothing** (see p. 249) of the input data, which is illustrated for 2D handwritten character recognition images. To obtain the tangent we must compute the derivative, however, this obviously cannot be calculated for discrete images. To compute tangent vectors, it must be possible to interpolate between the pixel values of the images (or the features of the images). Since many interpolations are possible, a “smoothing” regularizer is used because it has the extra benefit of imposing some control on the locality of the transformation invariance. However, care must be taken not to over smooth (or useful features are washed away). Another trick is to use the so-called **elastic tangent distance** (see p. 249) which eliminates the problem of singular systems which arises when there is zero distance between two patterns, or when their tangent vectors are parallel. Finally, through a very refined **hierarchy of resolution and accuracy** (see p. 251) the tangent distance algorithm can be sped up by two or tree orders of magnitude over a straight forward implementation.

Tangentprop takes the invariance idea even further by making it possible to incorporate local or global invariances and prior knowledge directly into the loss function of backpropagation and to **backpropagate tangents** efficiently (see p. 259). The chapter concludes with a digression on the mathematical background (Lie groups) and the simplifications that follow from the theory of this very successful (record breaking OCR) algorithm.

In the next chapter Larry Yaeger, et al. present an entire collection of tricks for an application that also is of strong commercial interest, on-line handwritten character recognition. These tricks were used by the authors to develop the recognizer in the Apple Computer’s Newton MessagePad® and eMate® products. Their recognizer consists of three main components: a segmenter, a classifier, and a context driven search component. Of particular interest is the intricate **representation of stroke information** (see p. 274) that serves as input to the classifier. Prior knowledge has led the authors to include specific stroke

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

features, grayscale character images, and coarse character attributes (stroke count and character aspect ratio), all combined into an ensemble of networks. The ensemble decisions are then combined in algorithmic search through a hypothesis space of dictionaries and combinations of dictionaries comprising a broad coverage, weakly applied language model.

An extensive set of tricks for training the network are also discussed, including:

Normalizing output error: Assist output activities of secondary choices to have non-zero values (*NormOutErr*). This enhances the robustness for the subsequent integrated search procedure since the search can now also take the n best choices into account (p. 276).

Negative training: Reduce the effect of invalid character segmentation (p. 278).

Stroke warping: Generate randomly warped versions of patterns to increase the generalization ability (p. 279).

Frequency balancing: Reduce the problem of unbalanced class priors using the simple trick of repeating low frequency classes more often in order to force the network to allocate more resources to these cases (p. 280).

Error emphasis: Account for different and uncommon writing *styles* by presenting difficult or unusual patterns more often (p. 281).

Quantized weights: Enable the neural network classifier to run with only one-byte weights and train with a temporary additional two bytes (p. 282).

In chapter 14, Steve Lawrence, et al. discuss several different tricks for alleviating the problem of unbalanced class prior probabilities. Some theoretical explanations are also provided. In the first trick, **prior scaling** (p. 296), weight updates are scaled so that the total expected update for each class is equal. The second trick, **probabilistic sampling** (p. 298), slightly modifies the frequency balancing in chapter 13. Here, a class is first chosen and then from within this class a sample is drawn. The next trick is referred to as **post scaling** (p. 298). The network is trained as usual but the network outputs are rescaled after training. This method can also be used to optimize other criteria that are different from the loss function the network has been trained with. Finally the authors propose to **equalize class memberships** (p. 299) by either subsampling the class with higher frequency or by duplicating patterns from the class with lower frequency (however they report that this trick works least efficiently). The effectiveness of each of these tricks is examined and compared for an ECG classification problem.

Training problems with thousands of classes and millions of examples, as are common for speech and handwritten character recognition problems, pose a major challenge. While many of the training techniques discussed so far work well for moderate size nets, they can fail miserably for these extremely large problems. In chapter 15, Jürgen Fritsch and Michael Finke design a representation and architecture for such large scale learning problems and, like the previous two chapters, they also tackle the problem of unbalanced class priors (since not all of the 24k subphonemes are equally probable). They exemplify their approach by building a large vocabulary speech recognizer. In the first step they break down the task into a hierarchy of smaller decision problems of controllable size

(**divide and conquer**, p. 313) and estimate the conditional probabilities for each node of the decision tree with a neural network. The network training uses **mini batches** and **individual adaptive learning rates** that are increased if progress is made in weight space and decreased if the fluctuations in weight space are too high (p. 332). These estimated probabilities – modeled by every single neural network node – are combined to give an overall estimate of the class decision probabilities.

The authors either determine the **decision tree** structure manually or estimate it by their **ACID clustering** algorithm (p. 324). Interestingly, the manual structure design was outperformed by the proposed agglomerative clustering scheme. No doubt that prior knowledge helps to achieve better classification results. However, this astonishing result indicates that human prior knowledge, although helpful in general, is suboptimal for structuring such a large task, particularly since automatic clustering allows for fine-grain subdivision of the classification task and aims for uniformity of priors. This desirable goal is hardly achievable by manual construction of the classification hierarchy. Furthermore, the human prior knowledge also does not provide the best basis from which a machine learning algorithm can learn optimally, a fact that is important to keep in mind for other applications as well.

Jenny & Klaus

12

Transformation Invariance in Pattern Recognition – Tangent Distance and Tangent Propagation^{*}

Patrice Y. Simard¹, Yann A. LeCun¹, John S. Denker¹, and Bernard Victorri²

¹ Image Processing Services Research Lab, AT&T Labs - Research,
100 Schulz Drive, Red Bank, NJ 07701-7033, USA
patrice@research.att.com
<http://www.research.att.com/info/patrice>

² CNRS, ELSAP, ENS, 1 rue Maurice Arnoux, F-92120 MONTROUGE, France

Abstract. In pattern recognition, statistical modeling, or regression, the amount of data is a critical factor affecting the performance. If the amount of data and computational resources are unlimited, even trivial algorithms will converge to the optimal solution. However, in the practical case, given limited data and other resources, satisfactory performance requires sophisticated methods to regularize the problem by introducing *a priori* knowledge. Invariance of the output with respect to certain transformations of the input is a typical example of such *a priori* knowledge. In this chapter, we introduce the concept of tangent vectors, which compactly represent the essence of these transformation invariances, and two classes of algorithms, “tangent distance” and “tangent propagation”, which make use of these invariances to improve performance.

12.1 Introduction

Pattern Recognition is one of the main tasks of biological information processing systems, and a major challenge of computer science. The problem of pattern recognition is to classify objects into categories, given that objects in a particular category may have widely-varying features, while objects in different categories may have quite similar features. A typical example is handwritten digit recognition. Characters, typically represented as fixed-size images (say 16 by 16 pixels), must be classified into one of 10 categories using a *classification function*. Building such a classification function is a major technological challenge, as irrelevant variabilities among objects of the same class must be eliminated, while meaningful differences between objects of different classes must be identified. These classification functions for most real pattern recognition tasks are too complicated to be synthesized “by hand” using only what humans know about the task. Instead, we use sophisticated techniques that combine humans’ *a priori* knowledge with information automatically extracted from a set of labeled examples

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

(the training set). These techniques can be divided into two camps, according to the number of parameters they require: the “memory based” algorithms, which in effect store a sizeable subset of the entire training set, and the “learned-function” techniques, which learn by adjusting a comparatively small number of parameters. This distinction is arbitrary because the patterns stored by a memory-based algorithm can be considered the parameters of a very complex learned function. The distinction is however useful in this work, because memory based algorithms often rely on a metric which can be modified to incorporate transformation invariances, while learned-function algorithms consist of selecting a classification function, the derivatives of which can be constrained to reflect the same transformation invariances. The two methods for incorporating invariances are different enough to justify two independent sections.

12.1.1 Memory Based Algorithms

To compute the classification function, many practical pattern recognition systems, and several biological models, simply store all the examples, together with their labels, in a memory. Each incoming pattern can then be compared to all the stored prototypes, and the labels associated with the prototypes that best match the input determine the output. The above method is the simplest example of the *memory-based* models. Memory-based models require three things: a *distance measure* to compare inputs to prototypes, an *output function* to produce an output by combining the labels of the prototypes, and a *storage scheme* to build the set of prototypes.

All three aspects have been abundantly treated in the literature. Output functions range from simply voting the labels associated with the k closest prototypes (K-Nearest Neighbors), to computing a score for each class as a linear combination of the distances to all the prototypes, using fixed [21] or learned [5] coefficients. Storage schemes vary from storing the entire training set, to picking appropriate subsets of it (see [8], chapter 6, for a survey) to learned-functions such as learning vector quantization (LVQ) [17] and gradient descent. Distance measures can be as simple as the Euclidean distance, assuming the patterns and prototypes are represented as vectors, or more complex as in the generalized quadratic metric [10] or in elastic matching methods [15].

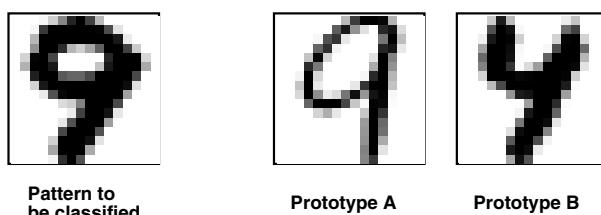


Fig. 12.1. According to the Euclidean distance the pattern to be classified is more similar to prototype B. A better distance measure would find that prototype A is closer because it differs mainly by a rotation and a thickness transformation, two transformations which should leave the classification invariant.

A simple but inefficient pattern recognition method is to use a simple distance measure, such as Euclidean distance between vectors representing the raw input, combined with a very large set of prototypes. This method is inefficient because almost all possible instances of a category must be present in the prototype set. In the case of handwritten digit recognition, this means that digits of each class in all possible positions, sizes, angles, writing styles, line thicknesses, skews, etc... must be stored. In real situations, this approach leads to impractically large prototype sets or to mediocre recognition accuracy as illustrated in Figure 12.1. An unlabeled image of a thick, slanted “9” must be classified by finding the closest prototype image out of two images representing respectively a thin, upright “9” and a thick, slanted “4”. According to the Euclidean distance (sum of the squares of the pixel to pixel differences), the “4” is closer. The result is an incorrect classification. The classical way of dealing with this problem is to use a so-called *feature extractor* whose purpose is to compute a representation of the patterns that is minimally affected by transformations of the patterns that do not modify their category. For character recognition, the representation should be invariant with respect to position, size changes, slight rotations, distortions, or changes in line thickness. The design and implementation of feature extractors is the major bottleneck of building a pattern recognition system. For example, the problem illustrated in Figure 12.1 can be solved by deslanting and thinning the images.

An alternative to this is to use an invariant *distance measure* constructed in such a way that the distance between a prototype and a pattern will not be affected by irrelevant transformations of the pattern or of the prototype. With an invariant distance measure, each prototype can match many possible instances of pattern, thereby greatly reducing the number of prototypes required.

The natural way of doing this is to use “deformable” prototypes. During the matching process, each prototype is deformed so as to best fit the incoming pattern. The quality of the fit, possibly combined with a measure of the amount of deformation, is then used as the distance measure [15]. With the example of Figure 12.1, the “9” prototype would be rotated and thickened so as to best match the incoming “9”. This approach has two shortcomings. First, a set of allowed deformations must be designed based on *a priori* knowledge. Fortunately, this is feasible for many tasks, including character recognition. Second, the search for the best-matching deformation is often enormously expensive, and/or unreliable. Consider the case of patterns that can be represented by vectors. For example, the pixel values of a 16 by 16 pixel character image can be viewed as the components of a 256-dimensional vector. One pattern, or one prototype, is a point in this 256-dimensional space. Assuming that the set of allowable transformations is continuous, the set of all the patterns that can be obtained by transforming one prototype using one or a combination of allowable transformations is a surface in the 256-D pixel space. More precisely, when a pattern P is transformed (e.g. rotated) according to a transformation $s(P, \alpha)$ which depends on one parameter α (e.g. the angle of the rotation), the set of all the transformed patterns

$$S_P = \{x \mid \exists \alpha \text{ for which } x = s(P, \alpha)\} \quad (12.1)$$

is a one-dimensional curve in the vector space of the inputs. In the remainder of this chapter, we will always assume that we have chosen s be differentiable with respect to both P and α and such that $s(P, 0) = P$.

When the set of transformations is parameterized by n parameters α_i (rotation, translation, scaling, etc.), the intrinsic dimension of the manifold S_P is at most n . For example, if the allowable transformations of character images are horizontal and vertical shifts, rotations, and scaling, the surface will be a 4-dimensional manifold.

In general, the manifold will not be linear. Even a simple image translation corresponds to a highly non-linear transformation in the high-dimensional pixel space. For example, if the image of an “8” is translated upward, some pixels oscillate from white to black and back several times. Matching a deformable prototype to an incoming pattern now amounts to finding the point on the surface that is at a minimum distance from the point representing the incoming pattern. This non-linearity makes the matching much more expensive and unreliable. Simple minimization methods such as gradient descent (or conjugate gradient) can be used to find the minimum-distance point, however, these methods only converge to a *local* minimum. In addition, running such an iterative procedure for each prototype is usually prohibitively expensive.

If the set of transformations happens to be linear in pixel space, then the manifold is a linear subspace (a hyperplane). The matching procedure is then reduced to finding the shortest distance between a point (vector) and a hyperplane, which is an easy-to-solve quadratic minimization problem. This special case has been studied in the statistical literature and is sometimes referred to as Procrustes analysis [24]. It has been applied to signature verification [12] and on-line character recognition [26].

This chapter considers the more general case of non-linear transformations such as geometric transformations of gray-level images. Remember that even a simple image translation corresponds to a highly non-linear transformation in the high-dimensional pixel space. The main idea of the chapter is to approximate the surface of possible transforms of a pattern by its tangent plane at the pattern, thereby reducing the matching to finding the shortest distance between two planes. This distance is called the *tangent distance*. The result of the approximation is shown in Figure 12.2, in the case of rotation for handwritten digits. At the top of the figure, is the theoretical curve in pixel space which represents equation (12.1), together with its linear approximation. Points of the transformation curve are depicted below for various amounts of rotation (each angle corresponds to a value of α). The bottom of Figure 12.2 depicts the linear approximation of the curve $s(P, \alpha)$ given by the Taylor expansion of s around $\alpha = 0$:

$$s(P, \alpha) = s(P, 0) + \alpha \frac{\partial s(P, \alpha)}{\partial \alpha} + O(\alpha^2) \approx P + \alpha T. \quad (12.2)$$

This linear approximation is completely characterized by the point P and the tangent vector $T = \frac{\partial s(P, \alpha)}{\partial \alpha}$. Tangent vectors, also called the Lie derivatives of

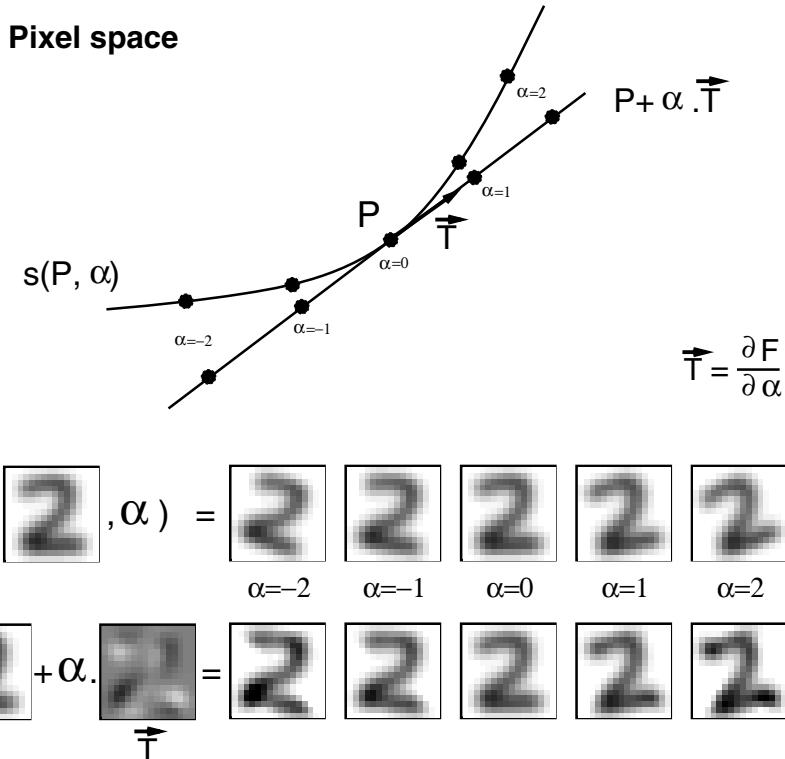


Fig. 12.2. Top: Representation of the effect of the rotation in pixel space. Middle: Small rotations of an original digitized image of the digit “2”, for different angle values of α . Bottom: Images obtained by moving along the tangent to the transformation curve for the same original digitized image P by adding various amounts (α) of the tangent vector T .

the transformation s , will be the subject of section 12.4. As can be seen from Figure 12.2, for reasonably small angles ($\|\alpha\| < 1$), the approximation is very good.

Figure 12.3 illustrates the difference between the Euclidean distance, the full invariant distance (minimum distance between manifolds) and the tangent distance. In the figure, both the prototype and the pattern are deformable (two-sided distance), but for simplicity or efficiency reasons, it is also possible to deform only the prototype or only the unknown pattern (one-sided distance).

Although in the following we will concentrate on using tangent distance to recognize images, the method can be applied to many different types of signals: temporal signals, speech, sensor data...

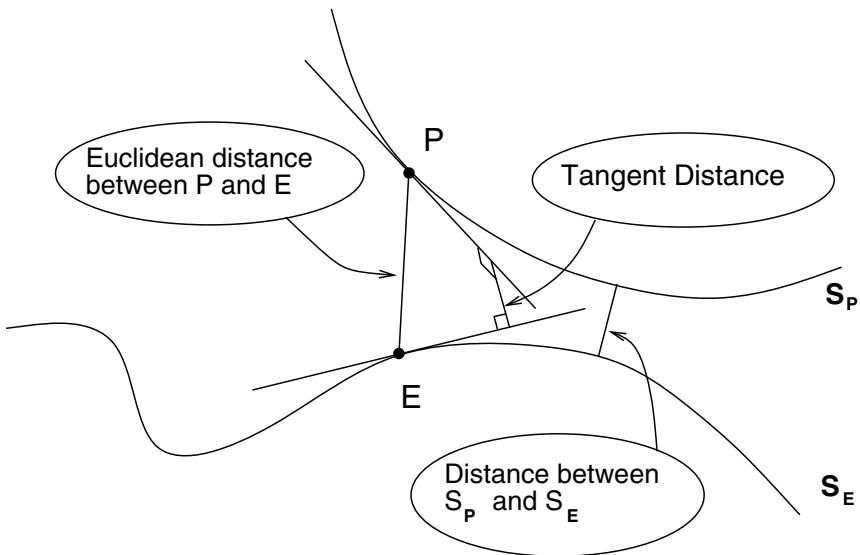


Fig. 12.3. Illustration of the Euclidean distance and the tangent distance between P and E . The curves S_p and S_e represent the sets of points obtained by applying the chosen transformations (for example translations and rotations) to P and E . The lines going through P and E represent the tangent to these curves. Assuming that working space has more dimensions than the number of chosen transformations (on the diagram, assume one transformation in a 3-D space) the tangent spaces do not intersect and the tangent distance is uniquely defined.

12.1.2 Learned-Function Algorithms

Rather than trying to keep a representation of the training set, it is also possible to choose a classification function by learning a set of parameters. This is the approach taken in neural networks, curve fitting, regression, et cetera.

We assume all data is drawn independently from a given statistical distribution \mathcal{P} , and our learning machine is characterized by the set of functions it can implement, $G_w(x)$, indexed by the vector of parameters w . We write $F(x)$ to represent the “correct” or “desired” labeling of the point x . The task is to find a value for w such that G_w best approximates F . We can use a finite set of training data to help find this vector. We assume the correct labeling $F(x)$ is known for all points in the training set. For example, G_w may be the function computed by a neural net having weights w , or G_w may be a polynomial having coefficients w . Without additional information, finding a value for w is an ill-posed problem unless the number of parameters is small and/or the size of the training set is large. This is because the training set does not provide enough information to distinguish the best solution among all the candidate ws . This problem is illustrated in Figure 12.4 (left). The desired function F (solid line) is to be approximated by a function G_w (dotted line) from four examples $\{(x_i, F(x_i))\}_{i=1,2,3,4}$. As exemplified in the picture, the fitted function G_w

largely disagrees with the desired function F between the examples, but it is not possible to infer this from the training set alone. Many values of w can generate many different functions G_w , some of which may be terrible approximations of F , even though they are in complete agreement with the training set. Because of this, it is customary to add “regularizers”, or additional constraints, to restrict the search of an acceptable w . For example, we may require the function G_w to be “smooth”, by adding the constraint that $\|w\|^2$ should be minimized. It is important that the regularizer reflects a property of F , hence regularizers depend on *a priori* knowledge about the function to be modeled.

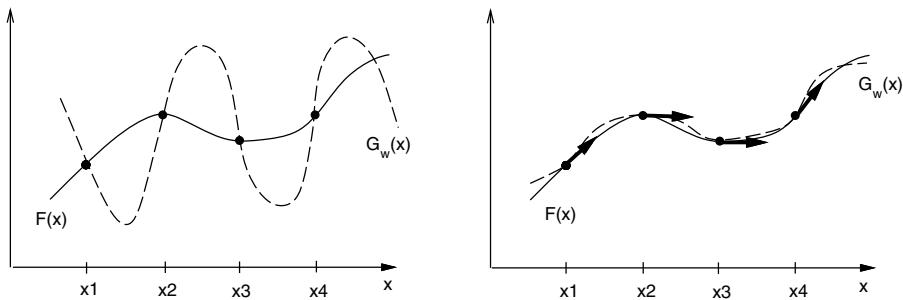


Fig. 12.4. Learning a given function (solid line) from a limited set of examples (x_1 to x_4). The fitted curves are shown by dotted line. Left: The only constraint is that the fitted curve goes through the examples. Right: The fitted curves not only go through each example but also its derivatives evaluated at the examples agree with the derivatives of the given function.

Selecting a good family $\mathcal{G} = \{G_w, w \in \Re^q\}$ of functions is a difficult task, sometimes known as “model selection” [16, 14]. If \mathcal{G} contains a large family of functions, it is more likely that it will contain a good approximation of F (the function we are trying to approximate), but it is also more likely that the selected candidate (using the training set) will generalize poorly because many functions in \mathcal{G} will agree with the training data and take outrageous values between the training samples. If, on the other hand, \mathcal{G} contains a small family of functions, it is more likely that a function G_w which fits the data will be a good approximation of F . The capacity of the family of functions \mathcal{G} is often referred to as the VC dimension [28, 27]. If a large amount of data is available, \mathcal{G} should contain a large family of functions (high VC dimension), so that more functions can be approximated, and in particular, F . If, on the other hand, the data is scarce, \mathcal{G} should be restricted to a small family of functions (low VC dimension), to control the values between the (more distant) samples¹. The VC dimension can also be

¹ Note that this point of view also applies to memory based systems. In the case where *all* the training data can be kept in memory, however, the VC dimension is infinite, and the formalism is meaningless. The VC dimension is a learning paradigm and is not useful unless learning is involved.

controlled by putting a knob on how much effect is given to some regularizers. For instance it is possible to control the capacity of a neural network by adding “weight decay” as a regularizer. Weight decay is a heuristic that favors smooth classification functions, by making a tradeoff by decreasing $\|w\|^2$ at the cost, usually, of slightly increased error on the training set. Since the optimal classification function is not necessarily smooth, for instance at a decision boundary, the weight decay regularizer can have adverse effects.

As mentioned earlier, the regularizer should reflect interesting properties (*a priori* knowledge) of the function to be learned. If the functions F and G_w are assumed to be differentiable, which is generally the case, the search for G_w can be greatly improved by requiring that G_w ’s derivatives evaluated at the points $\{x_i\}$ are more or less equal (this is the regularizer knob) to the derivatives of F at the same points (Figure 12.4 right). This result can be extended to multidimensional inputs. In this case, we can impose the equality of the derivatives of F and G_w in *certain directions*, not necessarily in all directions of the input space. Such constraints find immediate use in traditional pattern recognition problems. It is often the case that *a priori* knowledge is available on how the desired function varies with respect to some transformations of the input. It is straightforward to derive the corresponding constraint on the directional derivatives of the fitted function G_w in the directions of the transformations (previously named tangent vectors). Typical examples can be found in pattern recognition where the desired classification function is known to be invariant with respect to some transformation of the input such as translation, rotation, scaling, etc., in other words, the directional derivatives of the classification function in the directions of these transformations is zero.

This is illustrated in Figure 12.4. The right part of the figure shows how the additional constraints on G_w help generalization by constraining the values of G_w outside the training set. For every transformation which has a known effect on the classification function, a regularizer can be added in the form of a constraint on the directional derivative of G_w in the direction of the tangent vector (such as the one depicted in Figure 12.2), computed from the curve of transformation.

The next section will analyze in detail how to use a distance based on tangent vector in memory based algorithms. The subsequent section will discuss the use of tangent vectors in neural network, with the tangent propagation algorithm. The last section will compare different algorithms to compute tangent vectors.

12.2 Tangent Distance

The Euclidean distance between two patterns P and E is in general not appropriate because it is sensitive to irrelevant transformations of P and of E . In contrast, the transformed distance $D(E, P)$ is defined to be the minimal distance between the two manifolds S_P and S_E , and is therefore invariant with respect to the transformation used to generate S_P and S_E (see Figure 12.3). Unfortunately, these manifolds have no analytic expression in general, and finding the distance

between them is a difficult optimization problem with multiple local minima. Besides, true invariance is not necessarily desirable since a rotation of a “6” into a “9” does not preserve the correct classification.

Our approach consists of computing the minimum distance between the linear surfaces that best approximate the non-linear manifolds S_P and S_E . This solves three problems at once: 1) linear manifolds have simple analytical expressions which can be easily computed and stored, 2) finding the minimum distance between linear manifolds is a simple least squares problem which can be solved efficiently and, 3) this distance is locally invariant but not globally invariant. Thus the distance between a “6” and a slightly rotated “6” is small but the distance between a “6” and a “9” is large. The different distances between P and E are represented schematically in Figure 12.3.

The figure represents two patterns P and E in 3-dimensional space. The manifolds generated by s are represented by one-dimensional curves going through E and P respectively. The linear approximations to the manifolds are represented by lines tangent to the curves at E and P . These lines do not intersect in 3 dimensions and the shortest distance between them (uniquely defined) is $D(E, P)$. The distance between the two non-linear transformation curves $\mathcal{D}(E, P)$ is also shown on the figure.

An efficient implementation of the tangent distance $D(E, P)$ will be given in the next section, using image recognition as an illustration. We then compare our methods with the best known competing methods. Finally we will discuss possible variations on the tangent distance and how it can be generalized to problems other than pattern recognition.

12.2.1 Implementation

In this section we describe formally the computation of the tangent distance. Let the function s transform an image P to $s(P, \alpha)$ according to the parameter α . We require s to be differentiable with respect to α and P , and require $s(P, 0) = P$. If P is a 2 dimensional image for instance, $s(P, \alpha)$ could be a rotation of P by the angle α . If we are interested in all transformations of images which conserve distances (isometry), $s(P, \alpha)$ would be a rotation by α_θ followed by a translation by α_x, α_y of the image P . In this case $\alpha = (\alpha_\theta, \alpha_x, \alpha_y)$ is a vector of parameters of dimension 3. In general, $\alpha = (\alpha_1, \dots, \alpha_m)$ is of dimension m .

Since s is differentiable, the set $S_P = \{x \mid \exists \alpha \text{ for which } x = s(P, \alpha)\}$ is a differentiable manifold which can be approximated to the first order by a hyperplane T_P . This hyperplane is tangent to S_P at P and is generated by the columns of matrix

$$L_P = \left. \frac{\partial s(P, \alpha)}{\partial \alpha} \right|_{\alpha=0} = \left[\frac{\partial s(P, \alpha)}{\partial \alpha_1}, \dots, \frac{\partial s(P, \alpha)}{\partial \alpha_m} \right]_{\alpha=0} \quad (12.3)$$

which are vectors tangent to the manifold. If E and P are two patterns to be compared, the respective tangent planes T_E and T_P can be used to define a new distance D between these two patterns. The tangent distance $D(E, P)$ between

E and P is defined by

$$D(E, P) = \min_{x \in T_E, y \in T_P} \|x - y\|^2. \quad (12.4)$$

The equation of the tangent planes T_E and T_P is given by:

$$E'(\alpha_E) = E + L_E \alpha_E \quad (12.5)$$

$$P'(\alpha_P) = P + L_P \alpha_P \quad (12.6)$$

where L_E and L_P are the matrices containing the tangent vectors (see equation (12.3)) and the vectors α_E and α_P are the coordinates of E' and P' (using bases L_E and L_P) in the corresponding tangent planes. Note that E' , E , L_E and α_E denote vectors and matrices in linear equations (12.5). For example, if the pixel space was of dimension 5, and there were two tangent vectors, we could rewrite equation (12.5) as

$$\begin{bmatrix} E'_1 \\ E'_2 \\ E'_3 \\ E'_4 \\ E'_5 \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ E_3 \\ E_4 \\ E_5 \end{bmatrix} + \begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \\ L_{31} & L_{32} \\ L_{41} & L_{42} \\ L_{51} & L_{52} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix}. \quad (12.7)$$

The quantities L_E and L_P are attributes of the patterns so in many cases they can be precomputed and stored.

Computing the tangent distance

$$D(E, P) = \min_{\alpha_E, \alpha_P} \|E'(\alpha_E) - P'(\alpha_P)\|^2 \quad (12.8)$$

amounts to solving a linear least squares problem. The optimality condition is that the partial derivatives of $D(E, P)$ with respect to α_P and α_E should be zero:

$$\frac{\partial D(E, P)}{\partial \alpha_E} = 2(E'(\alpha_E) - P'(\alpha_P))^\top L_E = 0 \quad (12.9)$$

$$\frac{\partial D(E, P)}{\partial \alpha_P} = 2(P'(\alpha_P) - E'(\alpha_E))^\top L_P = 0. \quad (12.10)$$

Substituting E' and P' by their expressions yields to the following linear system of equations, which we must solve for α_P and α_E :

$$L_P^\top (E - P - L_P \alpha_P + L_E \alpha_E) = 0 \quad (12.11)$$

$$L_E^\top (E - P - L_P \alpha_P + L_E \alpha_E) = 0. \quad (12.12)$$

The solution of this system is

$$(L_{PE} L_{EE}^{-1} L_E^\top - L_P^\top)(E - P) = (L_{PE} L_{EE}^{-1} L_{EP} - L_{PP}) \alpha_P \quad (12.13)$$

$$(L_{EP} L_{PP}^{-1} L_P^\top - L_E^\top)(E - P) = (L_{EE} - L_{EP} L_{PP}^{-1} L_{PE}) \alpha_E \quad (12.14)$$

where $L_{EE} = L_E^\top L_E$, $L_{PE} = L_P^\top L_E$, $L_{EP} = L_E^\top L_P$ and $L_{PP} = L_P^\top L_P$. LU decompositions of L_{EE} and L_{PP} can be precomputed. The most expensive part

in solving this system is evaluating L_{EP} (L_{PE} can be obtained by transposing L_{EP}). It requires $m_E \times m_P$ dot products, where m_E is the number of tangent vectors for E and m_P is the number of tangent vectors for P . Once L_{EP} has been computed, α_P and α_E can be computed by solving two (small) linear systems of respectively m_E and m_P equations. The tangent distance is obtained by computing $\|E'(\alpha_E) - P'(\alpha_P)\|$ using the value of α_P and α_E in equations (12.5) and (12.6). If n is the dimension of the input space (i.e. the length of vector E and P), the algorithm described above requires roughly $n(m_E+1)(m_P+1)+3(m_E^3+m_P^3)$ multiply-adds. Approximations to the tangent distance can however be computed more efficiently.

12.2.2 Some Illustrative Results

Local Invariance. The “local² invariance” of tangent distance can be illustrated by transforming a reference image by various amounts and measuring its distance to a set of prototypes.

The bottom of Figure 12.5 shows 10 typical handwritten digit images. One of them – the digit “3” – is chosen to be the reference. The reference is translated horizontally by the amount indicated in the abscissa. There are ten curves for Euclidean distance and ten more curves for tangent distance, measuring the distance between the translated reference and one of the 10 digits.

Since the reference was chosen from the 10 digits, it is not surprising that the curve corresponding to the digit “3” goes to 0 when the reference is not translated (0 pixel translation). It is clear from the figure that if the reference (the image “3”) is translated by more than 2 pixels, the Euclidean distance will confuse it with other digits, namely “8” or “5”. In contrast, there is no possible confusion when tangent distance is used. As a matter of fact, in this example, the tangent distance correctly identifies the reference up to a translation of 5 pixels! Similar curves were obtained with all the other transformations (rotation, scaling, etc...).

The “local” invariance of tangent distance with respect to small transformations generally implies more accurate classification for much larger transformations. This is the single most important feature of tangent distance.

The locality of the invariance has another important benefit: Local invariance can be enforced with *very few* tangent vectors. The reason is that for infinitesimal (local) transformations, there is a direct correspondence³ between the tangent vectors of the tangent plane and the various compositions of transformations. For example, the three tangent vectors for X-translation, Y-translation and

² Local invariance refers to invariance with respect to small transformations (i.e. a rotation of a very small angle). In contrast, global invariance refers to invariance with respect to arbitrarily large transformations (i.e. a rotation of 180 degrees). Global invariance is not desirable in digit recognition, since we need to distinguish “6” from a “9”.

³ An isomorphism actually, see “Lie algebra” in [6].

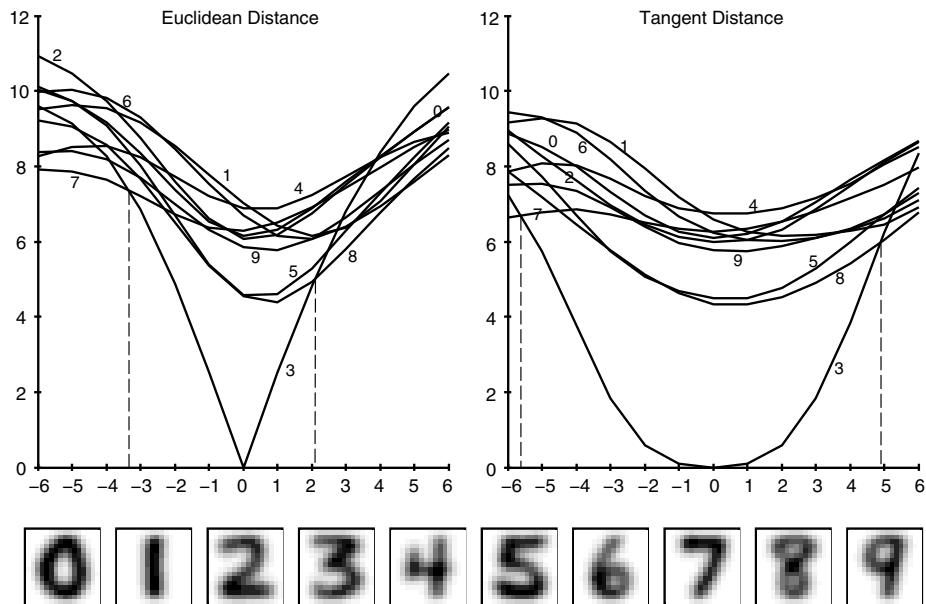


Fig. 12.5. Euclidean and tangent distances between 10 typical images of handwritten digits and a translated image of the digit “3”. The abscissa represents the amount of horizontal translation (measured in pixels).

rotations around the origin, generate a tangent plane corresponding to all the possible compositions of horizontal translations, vertical translations and rotations. The resulting tangent distance is then locally invariant to *all* the translations and *all* the rotations (around any center). Figure 12.6 further illustrates this phenomenon by displaying points in the tangent plane generated from only 5 tangent vectors. Each of these images looks like it has been obtained by applying various combinations of scaling, rotation, horizontal and vertical skewing, and thickening. Yet, the tangent distance between any of these points and the original image is 0.

Handwritten Digit Recognition. Experiments were conducted to evaluate the performance of tangent distance for handwritten digit recognition. An interesting characteristic of digit images is that we can readily identify a set of local transformations which do not affect the identity of the character, while covering a large portion of the set of possible *instances* of the character. Seven such image transformations were identified: X and Y translations, rotation, scaling, two hyperbolic transformations (which can generate shearing and squeezing), and line thickening or thinning. The first six transformations were chosen to span the set of all possible linear coordinate transforms in the image plane. (Nevertheless, they correspond to highly non-linear transforms in pixel space.) Additional

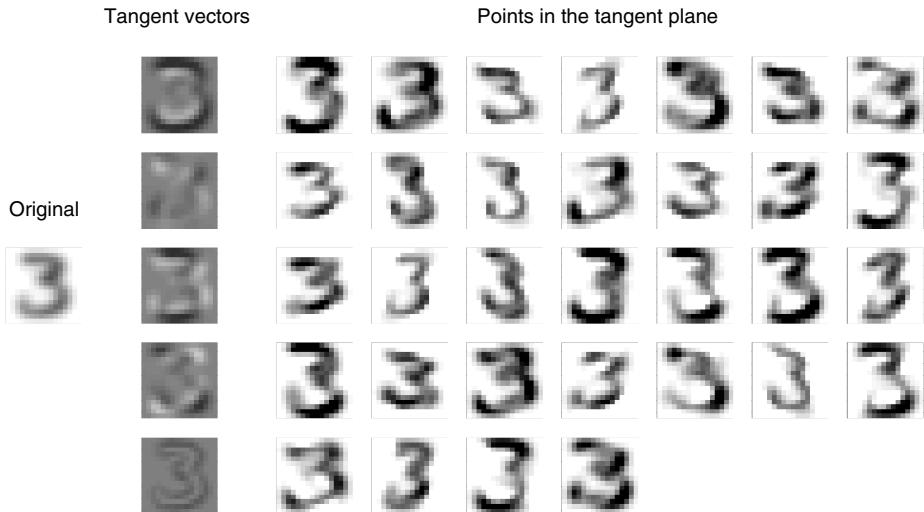


Fig. 12.6. Left: Original image. Middle: 5 tangent vectors corresponding respectively to the 5 transformations: scaling, rotation, expansion of the X axis while compressing the Y axis, expansion of the first diagonal while compressing the second diagonal and thickening. Right: 32 points in the tangent space generated by adding or subtracting each of the 5 tangent vectors.

transformations have been tried with less success. Three databases were used to test our algorithm:

US Postal Service Database: The database consisted of 16×16 pixel size-normalized images of handwritten digits, coming from US mail envelopes. The training and testing set had respectively 9709 and 2007 examples.

NIST1 Database: The second experiment was a competition organized by the National Institute of Standards and Technology (NIST) in Spring 1992. The object of the competition was to classify a test set of 59,000 handwritten digits, given a training set of 223,000 patterns.

NIST2 Database: The third experiment was performed on a database made out of the training and testing database provided by NIST (see above). NIST had divided the data into two sets which unfortunately had different distributions. The training set (223,000 patterns) was easier than the testing set (59,000 patterns). In our experiments we combined these two sets 50/50 to make a training set of 60,000 patterns and testing/validation sets of 10,000 patterns each, all having the same characteristics.

For each of these three databases we tried to evaluate human performance to benchmark the difficulty of the database. For USPS, two members of our group went through the test set and both obtained a 2.5% raw error performance. The human performance on NIST1 was provided by the National Institute of Standard and Technology. The human performance on NIST2 was measured on

a small subsample of the database and must therefore be taken with caution. Several of the leading algorithms were tested on each of these databases.

The first experiment used the K-Nearest Neighbor algorithm, using the ordinary Euclidean distance. The prototype set consisted of all available training examples. A 1-Nearest Neighbor rule gave optimal performance in USPS while a 3-Nearest Neighbors rule performed better in NIST2.

The second experiment was similar to the first, but the distance function was changed to tangent distance with 7 transformations. For the USPS and NIST2 databases, the prototype set was constructed as before, but for NIST1 it was constructed by cycling through the training set. Any patterns which were misclassified were added to the prototype set. After a few cycles, no more prototypes are added (the training error was 0). This resulted in 10,000 prototypes. A 3-Nearest Neighbors rule gave optimal performance on this set.

Other algorithms such as neural nets [18, 20], optimal margin classifier [7], local learning [3] and boosting [9] were also used on these databases. A case study can be found in [20].

Table 12.1. Results: Performances in % of errors for (in order) human, K-nearest neighbor, tangent distance, Lenet1 (simple neural network), Lenet4 (large neural network), optimal margin classifier (OMC), local learning (LL) and boosting (Boost)

	Human	K-NN	T.D.	Lenet1	Lenet4	OMC	LL	Boost
USPS	2.5	5.7	2.5	4.2		4.3	3.3	2.6
NIST1	1.6		3.2		3.7			4.1
NIST2	0.2	2.4	1.1	1.7	1.1	1.1	1.1	0.7

The results are summarized in Table 12.1. As illustrated in the table, the tangent distance algorithm equals or outperforms all other algorithms we tested, in all cases except one: Boosted LeNet 4 was the winner on the NIST2 database. This is not surprising. The K-nearest neighbor algorithm (with no preprocessing) is very unsophisticated in comparison to local learning, optimal margin classifier, and boosting. The advantage of tangent distance is the *a priori* knowledge of transformation invariance embedded into the distance. When the training data is sufficiently large, as is the case in NIST2, some of this knowledge can be picked up from the data by the more sophisticated algorithms. In other words, the value of *a priori* knowledge decreases as the size of the training set increases.

12.2.3 How to Make Tangent Distance Work

This section is dedicated to the technological “know how” which is necessary to make tangent distance work with various applications. “Tricks” of this sort are usually not published for various reasons (they are not always theoretically sound, page area is too valuable, the tricks are specific to one particular application, commercial competitive considerations discourage telling everyone how

to reproduce the result, etc.), but they are often a determining factor in making the technology a success. Several of these techniques will be discussed here.

Smoothing the Input Space: This is the single most important factor in obtaining good performance with tangent distance. By definition, the tangent vectors are the Lie derivatives of the transformation function $s(P, \alpha)$ with respect to α . They can be written as:

$$L_P = \frac{\partial s(P, \alpha)}{\partial \alpha} \Big| = \lim_{\epsilon \rightarrow 0} \frac{s(P, \epsilon) - s(P, 0)}{\epsilon}. \quad (12.15)$$

It is therefore very important that s be differentiable (and well behaved) with respect to α . In particular, it is clear from equation (12.15) that $s(P, \epsilon)$ must be computed for ϵ arbitrarily small. Fortunately, even when P can only take discrete values, it is easy to make s differentiable. The trick is to use a smoothing interpolating function C_σ as a preprocessing for P , such that $s(C_\sigma(P), \alpha)$ is differentiable (with respect to $C_\sigma(P)$ and α , not with respect to P). For instance, if the input space for P is binary images, $C_\sigma(P)$ can be a convolution of P with a Gaussian function of standard deviation σ . If $s(C_\sigma(P), \alpha)$ is a translation of α pixels, the derivative of $s(C_\sigma(P), \alpha)$ can easily be computed since $s(C_\sigma(P), \epsilon)$ can be obtained by translating Gaussian functions. This preprocessing will be discussed in more details in section 12.4.

The smoothing factor σ controls the locality of the invariance. The smoother the transformation curve defined by s is, the longer the linear approximation will be valid. In general the best smoothing is the maximum smoothing which does not blur the features. For example, in handwritten character recognition with 16x16 pixel images, a Gaussian function with a standard deviation of 1 pixel yielded the best results. Increased smoothing led to confusion (such as a “5” mistaken for “6” because the lower loop had been closed by the smoothing) and decreased smoothing didn’t make full use of the invariance properties.

If the available computation time allows it, the best strategy is to extract features first, smooth shamelessly, and then compute the tangent distance on the smoothed features.

Controlled Deformation: The linear system given in equation (12.8) is singular if some of the tangent vectors for E or P are parallel. Although the probability of this happening is zero when the data is taken from a real-valued continuous distribution (as is the case in handwritten character recognition), it is possible that a pattern may be duplicated in both the training and the test set, resulting in a division by zero error. The fix is quite simple and elegant. Equation (12.8) can be replaced by equation:

$$D(E, P) = \min_{\alpha_E, \alpha_P} \|E + L_E \alpha_E - P - L_P \alpha_P\|^2 + k \|L_E \alpha_E\|^2 + k \|L_P \alpha_P\|^2. \quad (12.16)$$

The physical interpretation of this equation, depicted in Figure 12.7, is that the point $E'(\alpha_E)$ on the tangent plane T_E is attached to E with a spring with spring constant k and to $P'(\alpha_p)$ (on the tangent plane T_P) with spring constant 1, and $P'(\alpha_p)$ is also attached to P with spring constant k . (All three springs

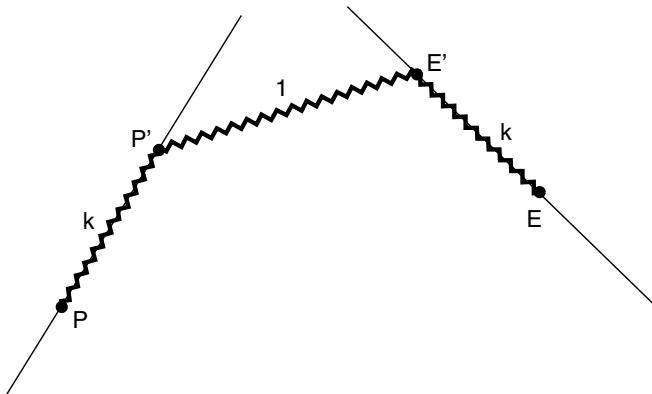


Fig. 12.7. The tangent distance between E and P is the elastic energy stored in each of the three springs connecting P , P' , E' and E . P' and E' can move without friction along the tangent planes. The spring constants are indicated on the figure.

have zero natural length.) The new tangent distance is the total potential elastic energy stored of all three springs at equilibrium. As for the standard tangent distance, the solution can easily be obtained by differentiating equation (12.16) with respect to α_E and α_P . The differentiation yields:

$$L_P^\top(E - P - L_P(1 + k)\alpha_P + L_E\alpha_E) = 0 \quad (12.17)$$

$$L_E^\top(E - P - L_P\alpha_P + L_E(1 + k)\alpha_E) = 0. \quad (12.18)$$

The solution of this system is

$$(L_{PE}L_{EE}^{-1}L_E^\top - (1 + k)L_P^\top)(E - P) = (L_{PE}L_{EE}^{-1}L_{EP} - (1 + k)^2L_{PP})\alpha_P \quad (12.19)$$

$$(L_{EP}L_{PP}^{-1}L_P^\top - (1 + k)L_E^\top)(E - P) = ((1 + k)^2L_{EE} - L_{EP}L_{PP}^{-1}L_{PE})\alpha_E \quad (12.20)$$

where $L_{EE} = L_E^\top L_E$, $L_{PE} = L_P^\top L_E$, $L_{EP} = L_E^\top L_P$ and $L_{PP} = L_P^\top L_P$. The system has the same complexity as the vanilla tangent distance except that, it always has a solution for $k \geq 0$, and is more numerically stable. Note that for $k = 0$, it is equivalent to the standard tangent distance, while for $k = \infty$, we have the Euclidean distance. This approach is also very useful when the number of tangent vectors is greater or equal than the number of dimensions of the space. The standard tangent distance would most likely be zero (when the tangent spaces intersect), but the “spring” tangent distance still expresses valuable information about the invariances.

If the number of dimension of the input space is large compared to the number of tangent vectors, keeping k as small as possible is better because it doesn’t interfere with the “sliding” along the tangent plane (E' and P' are less constrained).

Contrary to intuition, there is no danger of sliding too far in high dimensional space because tangent vectors are always roughly orthogonal and they could only slide far if they were parallel.

Hierarchy of Distances: If several invariances are used, classification using tangent distance alone would be quite expensive. Fortunately, if a typical memory based algorithm is used, for example, K-nearest neighbors, it is quite unnecessary to compute the full tangent distance between the unclassified pattern and all the labeled samples. In particular, if a crude estimate of the tangent distance indicates with a sufficient confidence that a sample is very far from the pattern to be classified, no more computation is needed to know that this sample is not one of the K-nearest neighbors. Based on this observation one can build a hierarchy of distances which can greatly reduce the computation of each classification. Let's assume, for instance, that we have m approximations D_i of the tangent distance, ordered such that D_1 is the crudest approximation of the tangent distance and D_m is exactly tangent distance (for instance D_1 to D_5 could be the Euclidean distance with increasing resolution, and D_6 to D_{10} each add a tangent vector at full resolution).

The basic idea is to keep a pool of all the prototypes which could potentially be the K-nearest neighbors of the unclassified pattern. Initially the pool contains all the samples. Each of the distances D_i corresponds to a stage of the classification process. The classification algorithm has 3 steps at each stage, and proceeds from stage 1 to stage m or until the classification is complete: Step 1: the distance D_i between all the samples in the pool and the unclassified pattern is computed. Step 2: A classification and a confidence score is computed with these distances. If the confidence is good enough, let's say better than C_i (for instance, if all the samples left in the pool are in the same class) the classification is complete, otherwise proceed to step 3. Step 3: The K_i closest samples, according to distance D_i are kept in the pool, while the remaining samples are discarded.

Finding the K_i closest samples can be done in $O(p)$ (where p is the number of samples in the pool) since these elements need not to be sorted [22, 2]. The reduced pool is then passed to stage $i + 1$.

The two constants C_i and K_i must be determined in advance using a validation set. This can easily be done graphically by plotting the error as a function of K_i and C_i at each stage (starting with all K_i equal to the number of labeled samples and $C_i = 1$ for all stages). At each stage there is a minimum K_i and minimum C_i which give optimal performance on the validation set. By taking larger values, we can decrease the probability of making errors on the test sets. The slightly worse performance of using a hierarchy of distances is often well worth the speed-up. The computational cost of a pattern classification is then equal to:

$$\text{computational cost} \approx \sum_i \frac{\text{number of prototypes at stage } i}{\text{complexity at stage } i} \times \frac{\text{distance to reach stage } i}{\text{probability}} \quad (12.21)$$

All this is better illustrated with an example as in Figure 12.8. This system was used for the USPS experiment described in a previous section. In classification

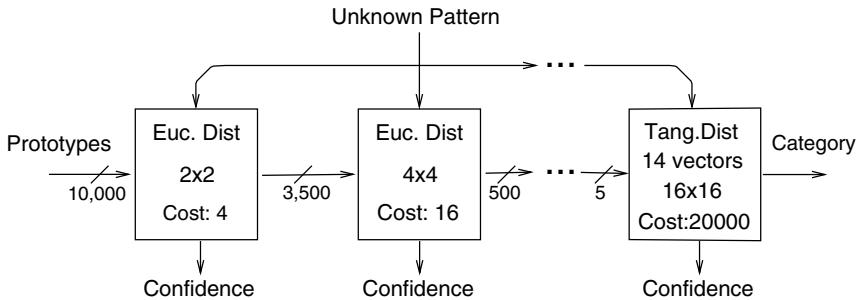


Fig. 12.8. Pattern recognition using a hierarchy of distances. The filter proceeds from left (starting with the whole database) to right (where only a few prototypes remain). At each stage distances between prototypes and the unknown pattern are computed and sorted; then the best candidate prototypes are selected for the next stage. As the complexity of the distance increases, the number of prototypes decreases, making computation feasible. At each stage a classification is attempted and a confidence score is computed. If the confidence score is high enough, the remaining stages are skipped.

of handwritten digits (16x16 pixel images), D_1 , D_2 , and D_3 , were the Euclidean distances at resolution 2×2 , 4×4 and 8×8 respectively. D_4 was the one sided tangent distance with X-translation, on the sample side only, at resolution 8×8 . D_5 was the double sided tangent distance with X-translation at resolution 16×16 . Each of the subsequent distances added one tangent vector on each side (Y-translation, scaling, rotation, hyperbolic deformation1, hyperbolic deformation2 and thickness) until the full tangent distance was computed (D_{11}).

Table 12.2 shows the expected number of multiply-adds at each of the stages. It should be noted that the full tangent distance need only be computed for 1 in 20 unknown patterns (probability 0.05), and only with 5 samples out of the original 10,000. The net speed up was in the order of 500, compared with computing the full tangent distance between every unknown pattern and every sample (this is 6 times faster than computing the Euclidean distance at full resolution).

Multiple Iterations: Tangent distance can be viewed as one iteration of a Newton-type algorithm which finds the points of minimum distance on the true transformation manifolds. The vectors α_E and α_P are the coordinates of the two closest points in the respective tangent spaces, but they can also be interpreted as the value for the real (non-linear) transformations. In other words, we can use α_E and α_P to compute the points $s(E, \alpha_E)$ and $s(P, \alpha_P)$, the real non-linear transformation of E and P . From these new points, we can recompute the tangent vectors, and the tangent distance and reiterate the process. If the appropriate conditions are met, this process can converge to a local minimum in the distance between the two transformation manifolds of P and E .

This process did not improve handwritten character recognition, but it yielded impressive results in face recognition [29]. In that case, each successive iteration was done at increasing resolution (hence combining hierarchical distances and multiple iterations), making the whole process computationally efficient.

Table 12.2. Summary computation for the classification of 1 pattern: The first column is the distance index, the second column indicates the number of tangent vectors (0 for the Euclidean distance), and the third column indicates the resolution in pixels, the fourth is K_i or the number of prototypes on which the distance D_i must be computed, the fifth column indicates the number of additional dot products which must be computed to evaluate distance D_i , the sixth column indicates the probability of not skipping that stage after the confidence score has been used, and the last column indicates the total average number of multiply-adds which must be performed (product of column 3 to 6) at each stage.

i	# of T.V.	Reso	# of proto (K_i)	# of prod	Probab	# of mul/add
1	0	4	9709	1	1.00	40,000
2	0	16	3500	1	1.00	56,000
3	0	64	500	1	1.00	32,000
4	1	64	125	2	0.90	14,000
4	2	256	50	5	0.60	40,000
6	4	256	45	7	0.40	32,000
7	6	256	25	9	0.20	11,000
8	8	256	15	11	0.10	4,000
9	10	256	10	13	0.10	3,000
10	12	256	5	15	0.05	1,000
11	14	256	5	17	0.05	1,000

12.3 Tangent Propagation

The previous section dealt with memory-based techniques. We now apply tangent-distance principles to learned-function techniques.

The key idea is to incorporate the invariance directly into the learned classification function. In this section, we present an algorithm, called “tangent propagation”, in which gradient descent is used to propagate information about the invariances of the training data. The process is a generalization of the widely-used “back propagation” method, which propagates information about the training data itself.

We again assume all data is drawn independently from a given statistical distribution \mathcal{P} , and our learning machine is characterized by the set of functions it can implement, $G_w(x)$, indexed by the vector of parameters w . Ideally, we would like to find w which minimizes the energy function

$$\mathcal{E} = \int \|G_w(x) - F(x)\|^2 d\mathcal{P}(x) \quad (12.22)$$

where $F(x)$ represents the “correct” or “desired” labeling of the point x . In the real world we must estimate this integral using only a finite set of training points B drawn from the distribution \mathcal{P} . That is, we try to minimize

$$E_p = \sum_{i=1}^p \|G_w(x_i) - F(x_i)\| \quad (12.23)$$

where the sum runs over the training set B . An estimate of w can be computed by following a gradient descent using the weight-update rule:

$$\Delta w = -\eta \frac{\partial E_p}{\partial w}. \quad (12.24)$$

Let’s consider an input transformation $s(x, \alpha)$ controlled by a parameter α . As always, we require that s is differentiable and that $s(x, 0) = x$. Now, in addition to the known labels of the training data, we assume that $\frac{\partial F(s(x_i, \alpha))}{\partial \alpha}$ is known at $\alpha = 0$ for each point x in the training set. To incorporate the invariance property into $G_w(x)$, we add that the following constraint on the derivative:

$$E_r = \sum_{i=1}^p \left| \frac{\partial G_w(s(x_i, \alpha))}{\partial \alpha} - \frac{\partial F(s(x_i, \alpha))}{\partial \alpha} \right|_{\alpha=0}^2 \quad (12.25)$$

should be small at $\alpha = 0$. In many pattern classification problems, we are interested in the local classification invariance property for $F(x)$ with respect to the transformation s (the classification does not change when the input is slightly transformed), so we can simplify equation (12.25) to:

$$E_r = \sum_{i=1}^p \left| \frac{\partial G_w(s(x_i, \alpha))}{\partial \alpha} \right|_{\alpha=0}^2 \quad (12.26)$$

since $\frac{\partial F(s(x_i, \alpha))}{\partial \alpha} = 0$. To minimize this term we can modify the gradient descent rule to use the energy function

$$E = \eta E_p + \mu E_r \quad (12.27)$$

with the weight update rule:

$$\Delta w = -\frac{\partial E}{\partial w}. \quad (12.28)$$

The learning rates (or regularization parameters) η and μ are tremendously important, because they determine the tradeoff between learning the invariances (based on the chosen directional derivatives) versus learning the label itself (i.e. the zeroth derivative) at each point in the training set.

The local variation of the classification function, which appears in equation (12.26) can be written as:

$$\frac{\partial G_w(s(x, \alpha))}{\partial \alpha} \Big|_{\alpha=0} = \frac{\partial G_w(s(x, \alpha))}{\partial s(x, \alpha)} \frac{\partial s(x, \alpha)}{\partial \alpha} \Big|_{\alpha=0} = \nabla_x G_w(x) \cdot \frac{\partial s(x, \alpha)}{\partial \alpha} \Big|_{\alpha=0} \quad (12.29)$$

since $s(x, \alpha) = x$ if $\alpha = 0$ and where $\nabla_x G_w(x)$ is the Jacobian of $G_w(x)$ for pattern x , and $\partial s(\alpha, x)/\partial \alpha$ is the *tangent vector* associated with transformation s as described in the previous section. Multiplying the tangent vector by the Jacobian involves one forward propagation through a “linearized” version of the network. If α is multi-dimensional, the forward propagation must be repeated for each tangent vector.

The theory of Lie algebras [11] ensures that compositions of local (small) transformations correspond to linear combinations of the corresponding tangent vectors (this result will be discussed further in section 12.4). Consequently, if $E_r(x) = 0$ is verified, the network derivative in the direction of a linear combination of the tangent vectors is equal to the same linear combination of the desired derivatives. In other words, if the network is successfully trained to be locally invariant with respect to, say, horizontal translations and vertical translations, it will be invariant with respect to compositions thereof.

It is possible to devise an efficient algorithm, “tangent prop”, for performing the weight update (equation (12.28)). It is analogous to ordinary backpropagation, but in addition to propagating neuron activations, it also propagates the tangent vectors. The equations can be easily derived from Figure 12.9.

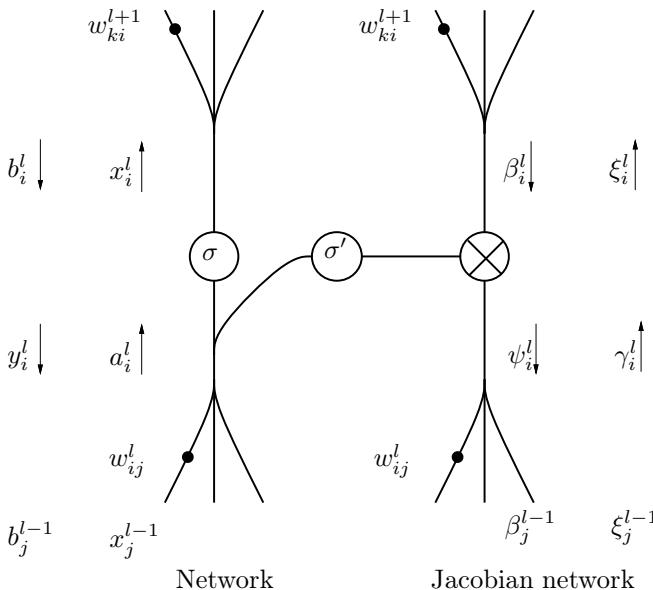


Fig. 12.9. Forward propagated variables (a, x, γ, ξ), and backward propagated variables (b, y, β, ψ) in the regular network (roman symbols) and the Jacobian (linearized) network (greek symbols). Converging forks (in the direction in which the signal is traveling) are sums, diverging forks just duplicate the values.

12.3.1 Local Rule

The forward propagation equation is:

$$a_i^l = \sum_j w_{ij}^l x_j^{l-1} \quad x_i^l = \sigma(a_i^l) \quad (12.30)$$

where σ is a non linear differentiable function (typically a sigmoid). The forward propagation starts at the first layer ($l = 1$), with x^0 being the input layer, and ends at the output layer ($l = L$). Similarly, The tangent forward propagation (tangent prop) is defined by:

$$\gamma_i^l = \sum_j w_{ij}^l \xi_j^{l-1} \quad \xi_i^l = \sigma'(a_i^l) \gamma_i^l. \quad (12.31)$$

The tangent forward propagation starts at the first layer ($l = 1$), with ξ^0 being the tangent vector $\frac{\partial s(x, \alpha)}{\partial \alpha}$, and ends at the output layer ($l = L$). The tangent gradient backpropagation can be computed using the chain rule:

$$\frac{\partial E}{\partial \xi_i^l} = \sum_k \frac{\partial E}{\partial \gamma_k^{l+1}} \frac{\partial \gamma_k^{l+1}}{\partial \xi_i^l} \quad \frac{\partial E}{\partial \gamma_i^l} = \frac{\partial E}{\partial \xi_i^l} \frac{\partial \xi_i^l}{\partial \gamma_i^l} \quad (12.32)$$

$$\beta_i^l = \sum_k \psi_k^{l+1} w_{ki}^{l+1} \quad \psi_i^l = \beta_i^l \sigma'(a_i^l). \quad (12.33)$$

The tangent backward propagation starts at the output layer ($l = L$), with ξ^L being the network variation $\frac{\partial G_w(s(x, \alpha))}{\partial \alpha}$, and ends at the input layer. Similarly, the gradient backpropagation equation is:

$$\frac{\partial E}{\partial x_i^l} = \sum_k \frac{\partial E}{\partial a_k^{l+1}} \frac{\partial a_k^{l+1}}{\partial x_i^l} \quad \frac{\partial E}{\partial a_i^l} = \frac{\partial E}{\partial x_i^l} \frac{\partial x_i^l}{\partial a_i^l} + \frac{\partial E}{\partial \xi_i^l} \frac{\partial \xi_i^l}{\partial a_i^l} \quad (12.34)$$

$$b_i^l = \sum_k y_k^{l+1} w_{ki}^{l+1} \quad y_i^l = b_i^l \sigma'(a_i^l) + \beta_i \sigma''(a_i^l) \gamma_i^l. \quad (12.35)$$

The standard backward propagation starts at the output layer ($l = L$), with $x^L = G_w(x^0)$ being the network output, and ends at the input layer. Finally, the weight update is:

$$\Delta w_{ij}^l = -\frac{\partial E}{\partial a_i^l} \frac{\partial a_i^l}{\partial w_{ij}^l} - \frac{\partial E}{\partial \gamma_i^l} \frac{\partial \gamma_i^l}{\partial w_{ij}^l} \quad (12.36)$$

$$\Delta w_{ij}^l = -y_i^l x_j^{l-1} - \psi_i^l \xi_j^{l-1}. \quad (12.37)$$

The computation requires one forward propagation and one backward propagation per pattern and per tangent vector during training. After the network is trained, it is approximately locally invariant with respect to the chosen transformation. After training, the evaluation of the learned function is in all ways identical to a network which is not trained for invariance (except that the weights have different values).

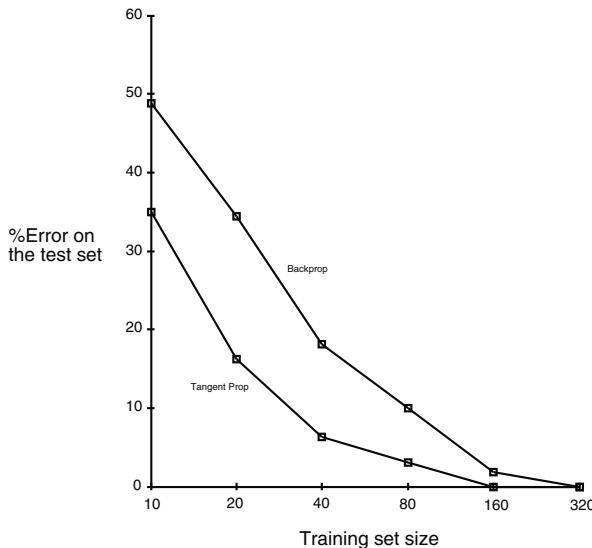


Fig. 12.10. Generalization performance curve as a function of the training set size for the tangent prop and the backprop algorithms

12.3.2 Results

Two experiments illustrate the advantages of tangent prop. The first experiment is a classification task, using a small (linearly separable) set of 480 binary images of handwritten digits. The training sets consist of 10, 20, 40, 80, 160 or 320 patterns, and the test set contains 160 patterns. The patterns are smoothed using a Gaussian kernel with standard deviation of one half pixel. For each of the training set patterns, the tangent vectors for horizontal and vertical translation are computed. The network has two hidden layers with locally connected shared weights, and one output layer with 10 units (5194 connections, 1060 free parameters) [19]. The generalization performance as a function of the training set size for traditional backprop and tangent prop are compared in Figure 12.10. We have conducted additional experiments in which we implemented not only translations but also rotations, expansions and hyperbolic deformations. This set of 6 generators is a basis for all linear transformations of coordinates for two dimensional images. It is straightforward to implement other generators including gray-level-shifting, “smooth” segmentation, local continuous coordinate transformations and independent image segment transformations.

The next experiment is designed to show that in applications where data is highly correlated, tangent prop yields a large speed advantage. Since the distortion model implies adding lots of highly correlated data, the advantage of tangent prop over the distortion model becomes clear.

The task is to approximate a function that has plateaus at three locations. We want to enforce local invariance near each of the training points (Figure 12.11, bottom). The network has one input unit, 20 hidden units and one output unit.

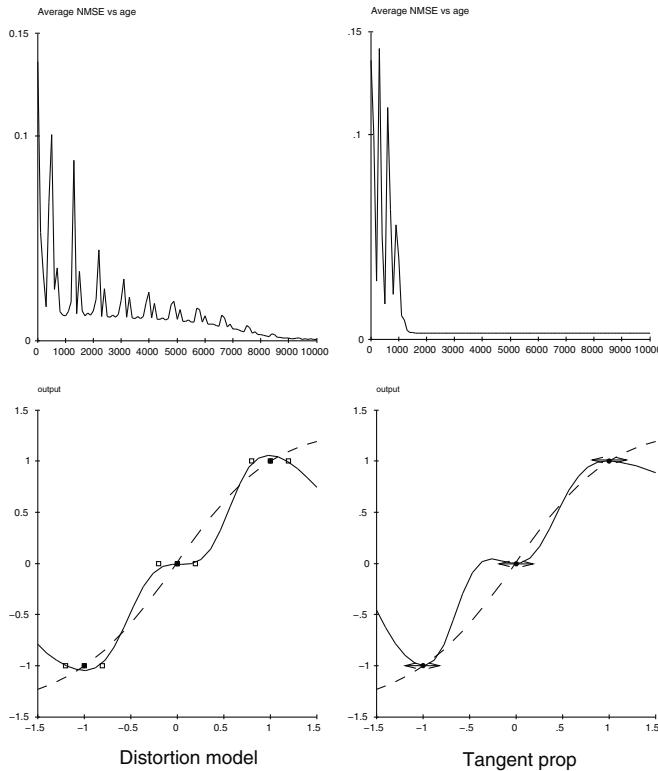


Fig. 12.11. Comparison of the distortion model (left column) and tangent prop (right column). The top row gives the learning curves (error versus number of sweeps through the training set). The bottom row gives the final input-output function of the network; the dashed line is the result for unadorned back prop.

Two strategies are possible: either generate a small set of training points covering each of the plateaus (open squares on Figure 12.11 bottom), or generate one training point for each plateau (closed squares), and enforce local invariance around them (by setting the desired derivative to 0). The training set of the former method is used as a measure of performance for both methods. All parameters were adjusted for approximately optimal performance in all cases. The learning curves for both models are shown in Figure 12.11 (top). Each sweep through the training set for tangent prop is a little faster since it requires only 6 forward propagations, while it requires 9 in the distortion model. As can be seen, stable performance is achieved after 1300 sweeps for the tangent prop, versus 8000 for the distortion model. The overall speedup is therefore about 10.

Tangent prop in this example can take advantage of a very large regularization term. The distortion model is at a disadvantage because the only parameter that effectively controls the amount of regularization is the magnitude of the distortions, and this cannot be increased to large values because the right answer is only invariant under *small* distortions.

12.3.3 How to Make Tangent Prop Work

Large Network Capacity: Relatively few experiments have been done with tangent propagation. It is clear, however, that the invariance constraint can be extremely beneficial. If the network does not have enough capacity, it will not benefit from the extra knowledge introduced by the invariance.

Interleaving of the Tangent Vectors: Since the tangent vectors introduce even more correlation inside the training set, a substantial speed up can be obtained by alternating a regular forward and backward propagation with a tangent forward and backward propagation (even if there are several tangent vectors, only one is used at each pattern). For instance, if there were 3 tangent vectors, the training sequence could be:

$$x_1, t_1(x_1), x_2, t_2(x_2), x_3, t_3(x_3), x_4, t_1(x_4), x_5, t_2(x_5), \dots \quad (12.38)$$

where x_i means a forward and backward propagation for pattern i and $t_j(x_i)$ means a tangent forward and backward propagation of tangent vector j of pattern i . With such interleaving, the learning converges faster than grouping all the tangent vectors together. Of course, this only makes sense with on-line updates as opposed to batch updates.

12.4 Tangent Vectors

In this section, we consider the general paradigm for transformation invariance and for the tangent vectors which have been used in the two previous sections. Before we introduce each transformation and their corresponding tangent vectors, a brief explanation is given of the theory behind the practice. There are two aspects to the problem. First it is possible to establish a formal connection between groups of transformations of the input space (such as translation, rotation, etc. of \Re^2) and their effect on a functional of that space (such as a mapping of \Re^2 to \Re , which may represent an image, in continuous form). The theory of Lie groups and Lie algebra [6] allows us to do this. The second problem has to do with coding. Computer images are finite vectors of discrete variables. How can a theory which was developed for differentiable functionals of \Re^2 to \Re be applied to these vectors? We first give a brief explanation of the theorems of Lie groups and Lie algebras which are applicable to pattern recognition. Next, we explore solutions to the coding problem. Finally some examples of transformation and coding are given for particular applications.

12.4.1 Lie Groups and Lie Algebras

Consider an input space \mathcal{I} (the plane \Re^2 for example) and a differentiable function f which maps points of \mathcal{I} to \Re .

$$f : X \in I \longmapsto f(X) \in \Re. \quad (12.39)$$

The function $f(X) = f(x, y)$ can be interpreted as the continuous (defined for all points of \Re^2) equivalent of the discrete computer image $P[i, j]$.

Next, consider a family of transformations t_α , parameterized by α , which maps bijectively a point of \mathcal{I} to a point of \mathcal{I}

$$t_\alpha : X \in \mathcal{I} \longmapsto t_\alpha(X) \in \mathcal{I}. \quad (12.40)$$

We assume that t_α is differentiable with respect to α and X , and that t_0 is the identity. For example t_α could be the group of affine transformations of \Re^2 :

$$t_\alpha : \begin{pmatrix} x \\ y \end{pmatrix} \longmapsto \begin{pmatrix} x + \alpha_1 x + \alpha_2 y + \alpha_5 \\ \alpha_3 x + y + \alpha_4 y + \alpha_6 \end{pmatrix} \quad \text{with} \quad \begin{vmatrix} 1 + \alpha_1 & \alpha_2 \\ \alpha_3 & 1 + \alpha_4 \end{vmatrix} \neq 0. \quad (12.41)$$

This is a Lie group⁴ with 6 parameters. Another example is the group of direct isometry:

$$t_\alpha : \begin{pmatrix} x \\ y \end{pmatrix} \longmapsto \begin{pmatrix} x \cos \theta - y \sin \theta + a \\ x \sin \theta + y \cos \theta + b \end{pmatrix} \quad (12.42)$$

which is a Lie group with 3 parameters.

We now consider the functional $s(f, \alpha)$, defined by

$$s(f, \alpha) = f \circ t_\alpha^{-1}. \quad (12.43)$$

This functional s , which takes another functional f as an argument, should remind the reader of Figure 12.2 where P , the discrete equivalent of f , is the argument of s .

The Lie algebra associated with the action of t_α on f is the space generated by the m local transformations L_{α_i} of f defined by:

$$L_{\alpha_i}(f) = \left. \frac{\partial s(f, \alpha)}{\partial \alpha_i} \right|_{\alpha=0}. \quad (12.44)$$

We can now write the local approximation of s as:

$$s(f, \alpha) = f + \alpha_1 L_{\alpha_1}(f) + \alpha_2 L_{\alpha_2}(f) + \cdots + \alpha_m L_{\alpha_m}(f) + o(\|\alpha\|^2)(f). \quad (12.45)$$

This equation is the continuous equivalent of equation (12.2) used in the introduction.

The following example illustrates how L_{α_i} can be computed from t_α . Let's consider the group of direct isometry defined in equation (12.42) (with parameter $\alpha = (\theta, a, b)$ as before, and $X = (x, y)$)

$$s(f, \alpha)(X) = f((x - a) \cos \theta + (y - b) \sin \theta, -(x - a) \sin \theta + (y - b) \cos \theta). \quad (12.46)$$

If we differentiate around $\alpha = (0, 0, 0)$ with respect to θ , we obtain

$$\frac{\partial s(f, \alpha)}{\partial \theta}(X) = y \frac{\partial f}{\partial x}(x, y) + (-x) \frac{\partial f}{\partial y}(x, y) \quad (12.47)$$

⁴ A Lie group is a group that is also a differentiable manifold such that the differentiable structure is compatible with the group structure.

that is

$$L_\theta = y \frac{\partial}{\partial x} + (-x) \frac{\partial}{\partial y}. \quad (12.48)$$

The transformation $L_a = -\frac{\partial}{\partial x}$ and $L_b = -\frac{\partial}{\partial y}$ can be obtained in a similar fashion. All local transformations of the group can be written as

$$s(f, \alpha) = f + \theta(y \frac{\partial f}{\partial x} + (-x) \frac{\partial f}{\partial y}) - a \frac{\partial f}{\partial x} - b \frac{\partial f}{\partial y} + o(\|\alpha\|^2)(f) \quad (12.49)$$

which corresponds to a linear combination of the 3 basic operators L_θ , L_a and L_b ⁵. The property which is most important to us is that the 3 operators generate the whole space of local transformations. The result of applying the operators to a function f , such as a 2D image for example, is the set of vectors which we have been calling “tangent vector” in the previous sections. Each point in the tangent space correspond to a unique transformation and conversely any transformation of the Lie group (in the example all rotations of any angle and center together with all translations) corresponds to a point in the tangent plane.

12.4.2 Tangent Vectors

The last problem which remains to be solved is the problem of coding. Computer images, for instance, are coded as a finite set of discrete (even binary) values. These are hardly the differentiable mappings of \mathcal{I} to \mathfrak{R} which we have been assuming in the previous subsection.

To solve this problem we introduce a smooth interpolating function C which maps the discrete vectors to a continuous mapping of \mathcal{I} to \mathfrak{R} . For example, if P is a image of n pixels, it can be mapped to a continuously valued function f over \mathfrak{R}^2 by convolving it with a two dimensional Gaussian function g_σ of standard deviation σ . This is because g_σ is a differentiable mapping of \mathfrak{R}^2 to \mathfrak{R} , and P can be interpreted as a sum of impulse functions. In the two dimensional case we can write the new interpretation of P as:

$$P'(x, y) = \sum_{i,j} P[i][j] \delta(x - i)\delta(y - j) \quad (12.50)$$

where $P[i][j]$ denotes the finite vector of discrete values, as stored in a computer. The result of the convolution is of course differentiable because it is a sum of Gaussian functions. The Gaussian mapping is given by:

$$C_\sigma : P \longmapsto f = P' * g_\sigma. \quad (12.51)$$

In the two dimensional case, the function f can be written as:

$$f(x, y) = \sum_{i,j} P[i][j] g_\sigma(x - i, y - j). \quad (12.52)$$

⁵ These operators are said to generate a Lie algebra, because on top of the addition and multiplication by a scalar, there is a special multiplication called “Lie bracket” defined by $[L_1, L_2] = L_1 \circ L_2 - L_2 \circ L_1$. In the above example we have $[L_\theta, L_a] = L_b$, $[L_a, L_b] = 0$, and $[L_b, L_\theta] = L_a$.

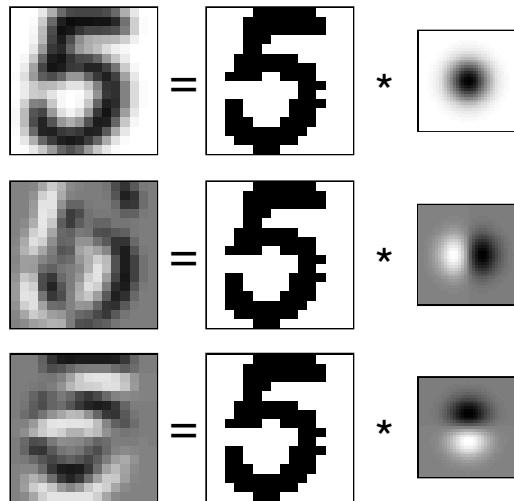


Fig. 12.12. Graphic illustration of the computation of f and two tangent vectors corresponding to $L_x = \partial/\partial x$ (X-translation) and $L_x = \partial/\partial y$ (Y-translation), from a binary image I . The Gaussian function $g(x, y) = \exp(-\frac{x^2+y^2}{2\sigma^2})$ has a standard deviation of $\sigma = 0.9$ in this example although its graphic representation (small images on the right) have been rescaled for clarity.

Other coding functions C can be used, such as cubic spline or even bilinear interpolation. Bilinear interpolation between the pixels yields a function f which is differentiable almost everywhere. The fact that the derivatives have two values at the integer locations (because the bilinear interpolation is different on both side of each pixels) is not a problem in practice – just choose one of the two values.

The Gaussian mapping is preferred for two reasons: First, the smoothing parameter σ can be used to control the locality of the invariance. This is because when f is smoother, the local approximation of equation (12.45) is valid for larger transformations. And second, when combined with the transformation operator L , the derivative can be applied on the closed form of the Gaussian function. For instance, if the X-translation operator $L = \frac{\partial}{\partial x}$ is applied to $f = P' * g_\sigma$, the actual computation becomes:

$$L_X(f) = \frac{\partial}{\partial x}(P' * g_\sigma) = P' * \frac{\partial g_\sigma}{\partial x}. \quad (12.53)$$

because of the differentiation properties of convolution when the support is compact. This is easily done by convolving the original image with the X-derivative of the Gaussian function g_σ . This operation is illustrated in Figure 12.12. Similarly, the tangent vector for scaling can be computed with

$$L_S(f) = \left(x \frac{\partial}{\partial x} + y \frac{\partial}{\partial y} \right) (I * g_\sigma) = x(I * \frac{\partial g_\sigma}{\partial x}) + y(I * \frac{\partial g_\sigma}{\partial y}). \quad (12.54)$$

This operation is illustrated in Figure 12.13.

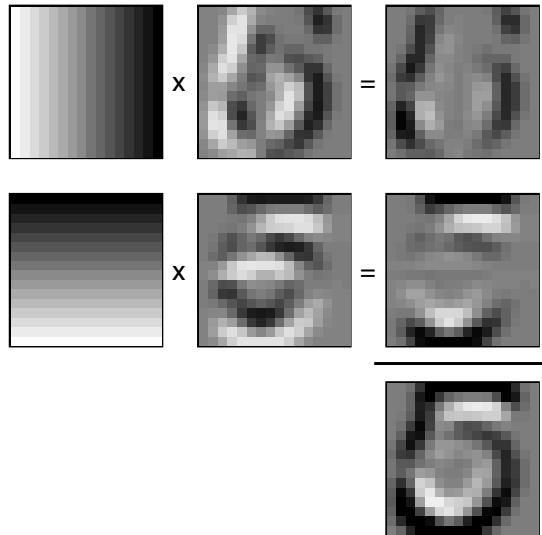


Fig. 12.13. Graphic illustration of the computation of the tangent vector $T_u = D_x S_x + D_y S_y$ (bottom image). In this example the displacement for each pixel is proportional to the distance of the pixel to the center of the image ($D_x(x, y) = x - x_0$ and $D_y(x, y) = y - y_0$). The two multiplications (horizontal lines) as well as the addition (vertical right column) are done pixel by pixel.

12.4.3 Important Transformations in Image Processing

This section summarizes how to compute the tangent vectors for image processing (in 2D). Each discrete image I_i is convolved with a Gaussian of standard deviation g_σ to obtain a representation of the continuous image f_i , according to equation:

$$f_i = I_i * g_\sigma. \quad (12.55)$$

The resulting image f_i will be used in all the computations requiring I_i (except for computing the tangent vector). For each image I_i , the tangent vectors are computed by applying the operators corresponding to the transformations of interest to the expression $I_i * g_\sigma$. The result, which can be precomputed, is an image which is the tangent vector. The following list contains some of the most useful tangent vectors:

X-translation: This transformation is useful when the classification function is known to be invariant with respect to the input transformation:

$$t_\alpha : \begin{pmatrix} x \\ y \end{pmatrix} \longmapsto \begin{pmatrix} x + \alpha \\ y \end{pmatrix}. \quad (12.56)$$

The Lie operator is defined by:

$$L_X = \frac{\partial}{\partial x}. \quad (12.57)$$

Y-translation: This transformation is useful when the classification function is known to be invariant with respect to the input transformation:

$$t_\alpha : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x \\ y + \alpha \end{pmatrix}. \quad (12.58)$$

The Lie operator is defined by:

$$L_Y = \frac{\partial}{\partial y}. \quad (12.59)$$

Rotation: This transformation is useful when the classification function is known to be invariant with respect to the input transformation:

$$t_\alpha : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x \cos \alpha - y \sin \alpha \\ x \sin \alpha + y \cos \alpha \end{pmatrix}. \quad (12.60)$$

The Lie operator is defined by:

$$L_R = y \frac{\partial}{\partial x} + (-x) \frac{\partial}{\partial y}. \quad (12.61)$$

Scaling: This transformation is useful when the classification function is known to be invariant with respect to the input transformation:

$$t_\alpha : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x + \alpha x \\ y + \alpha y \end{pmatrix}. \quad (12.62)$$

The Lie operator is defined by:

$$L_S = x \frac{\partial}{\partial x} + y \frac{\partial}{\partial y}. \quad (12.63)$$

Parallel hyperbolic transformation: This transformation is useful when the classification function is known to be invariant with respect to the input transformation:

$$t_\alpha : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x + \alpha x \\ y - \alpha y \end{pmatrix}. \quad (12.64)$$

The Lie operator is defined by:

$$L_S = x \frac{\partial}{\partial x} - y \frac{\partial}{\partial y}. \quad (12.65)$$

Diagonal hyperbolic transformation: This transformation is useful when the classification function is known to be invariant with respect to the input transformation:

$$t_\alpha : \begin{pmatrix} x \\ y \end{pmatrix} \longmapsto \begin{pmatrix} x + \alpha y \\ y + \alpha x \end{pmatrix}. \quad (12.66)$$

The Lie operator is defined by:

$$L_S = y \frac{\partial}{\partial x} + x \frac{\partial}{\partial y}. \quad (12.67)$$

The resulting tangent vector is the norm of the gradient of the image, which is very easy to compute.

Thickening: This transformation is useful when the classification function is known to be invariant with respect to variation of thickness. This is known in morphology as dilation and its inverse, erosion. It is very useful in certain domains (such as handwritten character recognition because) thickening and thinning are natural variations which correspond to the pressure applied on a pen, or to different absorption properties of the ink on the paper. A dilation (resp. erosion) can be defined as the operation of replacing each value $f(x,y)$ by the largest (resp. smallest) value of $f(x',y')$ found within a neighborhood of a certain shape, centered at (x,y) . The region is called the structural element. We will assume that the structural element is a sphere of radius α . We define the thickening transformation as the function which takes the function f and generates the function f'_α defined by:

$$f'_\alpha(X) = \max_{\|r\| \leq \alpha} f(X+r) \quad \text{for } \alpha \geq 0 \quad (12.68)$$

$$f'_\alpha(X) = \min_{\|r\| \leq -\alpha} f(X+r) \quad \text{for } \alpha \leq 0. \quad (12.69)$$

The derivative of the thickening for $\alpha \geq 0$ can be written as:

$$\lim_{\alpha \rightarrow 0} \frac{f'(X) - f(X)}{\alpha} = \lim_{\alpha \rightarrow 0} \frac{\max_{\|r\| \leq \alpha} f(X+r) - f(X)}{\alpha}. \quad (12.70)$$

$f(X)$ can be put within the max expression because it does not depend on $\|r\|$. Since $\|\alpha\|$ tends toward 0, we can write:

$$f(X+r) - f(X) = r \cdot \nabla f(X) + O(\|r\|^2) \approx r \cdot \nabla f(X). \quad (12.71)$$

The maximum of

$$\max_{\|r\| \leq \alpha} f(X+r) - f(X) = \max_{\|r\| \leq \alpha} r \cdot \nabla f(X) \quad (12.72)$$

is attained when r and $\nabla f(X)$ are co-linear, that is when

$$r = \alpha \frac{\nabla f(X)}{\|\nabla f(X)\|} \quad (12.73)$$

assuming $\alpha \geq 0$. It can easily be shown that this equation holds when α is negative, because we then try to minimize equation (12.69). We therefore have:

$$\lim_{\alpha \rightarrow 0} \frac{f'_\alpha(X) - f(X)}{\alpha} = \|\nabla f(X)\| \quad (12.74)$$

which is the tangent vector of interest. Note that this is true for α positive or negative. The same tangent vector describes both thickening and thinning. Alternatively, we can use our computation of the displacement r and define the following transformation of the input:

$$t_\alpha(f) : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x + \alpha r_x \\ y + \alpha r_y \end{pmatrix} \quad (12.75)$$

where

$$(r_x, r_y) = r = \alpha \frac{\nabla f(X)}{\|\nabla f(X)\|}. \quad (12.76)$$

This transformation of the input space is different for each pattern f (we do not have a Lie group of transformations, but the field structure generated by the (pseudo Lie) operator is still useful. The operator used to find the tangent vector is defined by:

$$L_T = \|\nabla\| \quad (12.77)$$

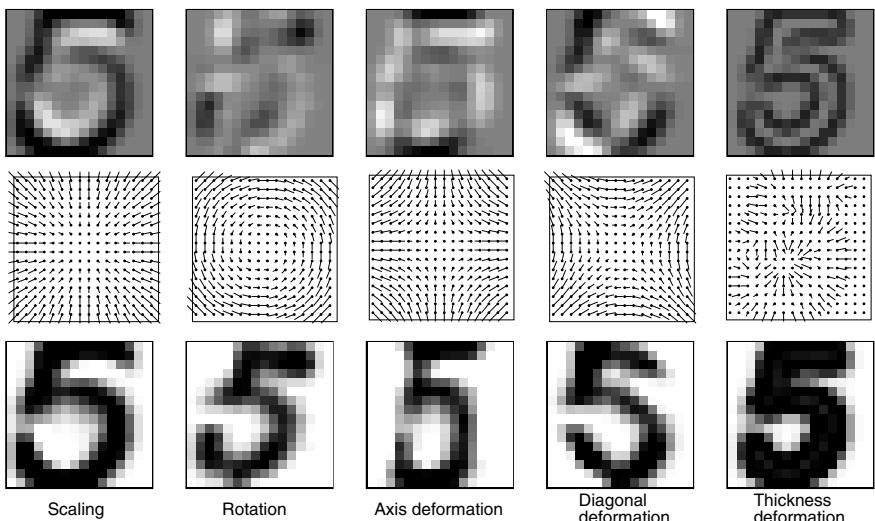


Fig. 12.14. Illustration of 5 tangent vectors (top), with corresponding displacements (middle) and transformation effects (bottom). The displacement D_x and D_y are represented in the form of vector field. It can be noted that the tangent vector for the thickness deformation (right column) correspond to the norm of the gradient of the gray level image.

which means that the tangent vector image is obtained by computing the normalized gray level gradient of the image at each point (the gradient at each point is normalized).

The last 5 transformations are depicted in Figure 12.14 with the tangent vector. The last operator corresponds to a thickening or thinning of the image. This unusual transformation is extremely useful for handwritten character recognition.

12.5 Conclusion

The basic tangent distance algorithm is quite easy to understand and implement. Even though hardly any preprocessing or learning is required, the performance is surprisingly good and compares well to the best competing algorithms. We believe that the main reason for this success is its ability to incorporate *a priori* knowledge into the distance measure. The only algorithm which performed better than tangent distance on one of the three databases was boosting, which has similar *a priori* knowledge about transformations built into it.

Many improvements are of course possible. For instance, smart preprocessing can allow us to measure the tangent distance in a more appropriate “feature” space, instead of the original pixel space. In image classification, for example, the features could be horizontal and vertical edges. This would most likely further improve the performance⁶ The only requirement is that the preprocessing must be differentiable, so that the tangent vectors can be computed (propagated) into the feature space.

It is also straightforward to modify more complex algorithms such as LVQ (learning vector quantization) to use a tangent distance. In this case even the tangent vectors can be trained. The derivation has been done for batch training [13] and for on-line training [23] of the tangent vectors. When such training is performed, the *a priori* knowledge comes from other constraints imposed on the tangent vectors (for instance how many tangent vectors are allowed, which classes of transformation do they represent, etc).

Finally, many optimizations which are commonly used in distance based algorithms can be used successfully with tangent distance to speed up computation. The multi-resolution approach have already been tried successfully [25]. Other methods like “multi-edit-condensing” [1, 30] and K-d tree [4] are also possible.

The main advantage of tangent distance is that it is a modification of a standard distance measure to allow it to incorporate *a priori* knowledge that is specific to the problem at hand. Any algorithms based on a common distance measure (as it is often the case in classification, vector quantization, predictions, etc...) can potentially benefit from a more problem-specific distance. Many of these “distance based” algorithms do not require any learning, which means that they can be adapted instantly by just adding new patterns in the database. These additions are leveraged by the *a priori* knowledge put in the tangent distance.

⁶ There may be an additional cost for computing the tangent vectors in the feature space if the feature space is very complex.

The two drawbacks of tangent distance are its memory and computational requirements. The most computationally and memory efficient algorithms generally involve learning [20]. Fortunately, the concept of tangent vectors can also be used in learning. This is the basis for the tangent propagation algorithm. The concept is quite simple: instead of learning a classification function from examples of its values, one can also use information about its derivatives. This information is provided by the tangent vectors. Unfortunately, not many experiments have been done in this direction. The two main problems with tangent propagation are that the capacity of the learning machine has to be adjusted to incorporate the additional information pertinent to the tangent vectors, and that training time must be increased. After training, the classification time and complexity are unchanged, but the classifier's performance is improved.

To a first approximation, using tangent distance or tangent propagation is like having a much larger database. If the database was plenty large to begin with, tangent distance or tangent propagation would not improve the performance. To a better approximation, tangent vectors are like using a distortion model to magnify the size of the training set. In many cases, using tangent vectors will be preferable to collecting (and labeling!) vastly more training data, and preferable (especially for memory-based classifiers) to dealing with all the data generated by the distortion model. Tangent vectors provide a compact and powerful representation of *a priori* knowledge which can easily be integrated in the most popular algorithms.

Acknowledgement. P.S. and Y.L. gratefully acknowledge NSF grant No. INT-9726745.

References

- [1] Devijver, P.A., Kittler, J.: Pattern Recognition, A Statistical Approach. Prentice-Hall, Englewood Cliffs (1982)
- [2] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: Data Structures and Algorithms. Addison-Wesley (1983)
- [3] Bottou, L., Vapnik, V.N.: Local learning algorithms. Neural Computation 4(6), 888–900 (1992)
- [4] Broder, A.J.: Strategies for efficient incremental nearest neighbor search. Pattern Recognition 23, 171–178 (1990)
- [5] Broomhead, D.S., Lowe, D.: Multivariable functional interpolation and adaptive networks. Complex Systems 2, 321–355 (1988)
- [6] Choquet-Bruhat, Y., DeWitt-Morette, C., Dillard-Bleick, M.: Analysis, Manifolds and Physics. North-Holland, Amsterdam (1982)
- [7] Cortes, C., Vapnik, V.: Support vector networks. Machine Learning 20, 273–297 (1995)
- [8] Dasarathy, B.V.: Nearest Neighbor (NN) Norms: NN Pattern classification Techniques. IEEE Computer Society Press, Los Alamitos (1991)
- [9] Drucker, H., Schapire, R., Simard, P.Y.: Boosting performance in neural networks. International Journal of Pattern Recognition and Artificial Intelligence 7(4), 705–719 (1993)

- [10] Fukunaga, K., Flick, T.E.: An optimal global nearest neighbor metric. *IEEE transactions on Pattern analysis and Machine Intelligence* 6(3), 314–318 (1984)
- [11] Gilmore, R.: Lie Groups, Lie Algebras and some of their Applications. Wiley, New York (1974)
- [12] Hastie, T., Kishon, E., Clark, M., Fan, J.: A model for signature verification. Technical Report 11214-910715-07TM, AT&T Bell Laboratories (July 1991)
- [13] Hastie, T., Simard, P.Y.: Metrics and models for handwritten character recognition. *Statistical Science* 13 (1998)
- [14] Hastie, T.J., Tibshirani, R.J.: Generalized Linear Models. Chapman and Hall, London (1990)
- [15] Hinton, G.E., Williams, C.K.I., Revow, M.D.: Adaptive elastic models for hand-printed character recognition. In: *Advances in Neural Information Processing Systems*, pp. 512–519. Morgan Kaufmann Publishers (1992)
- [16] Hoerl, A.E., Kennard, R.W.: Ridge regression: Biased estimation for non-orthogonal problems. *Technometrics* 12, 55–67 (1970)
- [17] Kohonen, T.: Self-organization and associative memory. Springer Series in Information Sciences, vol. 8. Springer (1984)
- [18] Le Cun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Handwritten digit recognition with a back-propagation network. In: Touretzky, D. (ed.) *Advances in Neural Information Processing Systems*, vol. 2, Morgan Kaufmann, Denver (1989)
- [19] LeCun, Y.: Generalization and network design strategies. In: Pfeifer, R., Schreter, Z., Fogelman, F., Steels, L. (eds.) *Connectionism in Perspective*, Zurich, Switzerland (1989); Elsevier, An extended version was published as a technical report of the University of Toronto
- [20] LeCun, Y., Jackel, L.D., Bottou, L., Cortes, C., Denker, J.S., Drucker, H., Guyon, I., Muller, U.A., Sackinger, E., Simard, P., Vapnik, V.: Learning algorithms for classification: A comparison on handwritten digit recognition. In: Oh, J.H., Kwon, C., Cho, S. (eds.) *Neural Networks: The Statistical Mechanics Perspective*, pp. 261–276. World Scientific (1995)
- [21] Parzen, E.: On estimation of a probability density function and mode. *Ann. Math. Stat.* 33, 1065–1076 (1962)
- [22] Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: *Numerical Recipes*. Cambridge University Press, Cambridge (1988)
- [23] Schwenk, H.: The diabolo classifier. *Neural Computation* (1998) (in press)
- [24] Sibson, R.: Studies in the robustness of multidimensional scaling: Procrustes statistics. *J. R. Statist. Soc.* 40, 234–238 (1978)
- [25] Simard, P.Y.: Efficient computation of complex distance metrics using hierarchical filtering. In: *Advances in Neural Information Processing Systems*. Morgan Kaufmann Publishers (1994)
- [26] Sinden, F., Wilfong, G.: On-line recognition of handwritten symbols. Technical Report 11228-910930-02IM, AT&T Bell Laboratories (June 1992)
- [27] Vapnik, V.N.: Estimation of dependences based on empirical data. Springer (1982)
- [28] Vapnik, V.N., Chervonenkis, A.Y.: On the uniform convergence of relative frequencies of events to their probabilities. *Th. Prob. and its Applications* 17(2), 264–280 (1971)
- [29] Vasconcelos, N., Lippman, A.: Multiresolution tangent distance for affine-invariant classification. In: *Advances in Neural Information Processing Systems*, vol. 10, pp. 843–849. Morgan Kaufmann Publishers (1998)
- [30] Voisin, J., Devijver, P.: An application of the multiedit-condensing technique to the reference selection problem in a print recognition system. *Pattern Recognition* 20(5), 465–474 (1987)

Combining Neural Networks and Context-Driven Search for On-line, Printed Handwriting Recognition in the Newton*

Larry S. Yaeger¹, Brandyn J. Webb², and Richard F. Lyon³

¹ Apple Computer 5540 Bittersweet Rd. Beanblossom, IN 46160

² The Future 4578 Fieldgate Rd. Oceanside, CA 92056

³ Foveonics, Inc. 10131-B Bubb Rd. Cupertino, CA 95014

larryy@pobox.com

<http://www.beanblossom.in.us/larryy/>

Abstract. While on-line handwriting recognition is an area of long-standing and ongoing research, the recent emergence of portable, pen-based computers has focused urgent attention on usable, practical solutions. We discuss a combination and improvement of classical methods to produce robust recognition of hand-printed English text, for a recognizer shipping in new models of Apple Computer's Newton MessagePad® and eMate®. Combining an artificial neural network (ANN), as a character classifier, with a context-driven search over segmentation and word recognition hypotheses provides an effective recognition system. Long-standing issues relative to training, generalization, segmentation, models of context, probabilistic formalisms, etc., need to be resolved, however, to get excellent performance. We present a number of recent innovations in the application of ANNs as character classifiers for word recognition, including integrated multiple representations, normalized output error, negative training, stroke warping, frequency balancing, error emphasis, and quantized weights. User-adaptation and extension to cursive recognition pose continuing challenges.

13.1 Introduction

Pen-based hand-held computers are heavily dependent upon fast and accurate handwriting recognition, since the pen serves as the primary means for inputting data to such devices. Some earlier attempts at handwriting recognition have utilized strong, limited language models to maximize accuracy. However, this approach has proven to be unacceptable in real-world applications, generating disturbing and seemingly random word substitutions – known colloquially within Apple and Newton as “The Doonesbury Effect”, due to Gary Trudeau’s satirical look at first-generation Newton recognition performance. But the original handwriting recognition technology in the Newton, and the current, much-improved

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

“Cursive Recognizer” technology, both of which were licensed from ParaGraph International, Inc., are not the subject of this article.

In Apple’s Advanced Technology Group (aka Apple Research Labs), we pursued a different approach, using bottom-up classification techniques based on trainable artificial neural networks (ANNs), in combination with comprehensive but weakly-applied language models. To focus our work on a subproblem that was tractable enough to lead to usable products in a reasonable time, we initially restricted the domain to hand-printing, so that strokes are clearly delineated by pen lifts. By simultaneously providing accurate character-level recognition, dictionaries exhibiting very wide coverage of the language, and the ability to write entirely outside those dictionaries, we have produced a hand-print recognizer that some have called the “first usable” handwriting recognition system. The ANN character classifier required some innovative training techniques to perform its task well. The dictionaries required large word lists, a regular expression grammar (to describe special constructs such as date, time, phone numbers, etc.), and a means of combining all these dictionaries into a comprehensive language model. And well balanced prior probabilities had to be determined for in-dictionary and out-of-dictionary writing. Together with a maximum-likelihood search engine, these elements form the basis of the so-called “Print Recognizer”, that was first shipped in Newton OS 2.0 based MessagePad 120 units in December, 1995, and has shipped in all subsequent Newton devices. In the MessagePad 2000 and 2100, despite retaining its label as a “Print Recognizer”, it has been extended to handle connected characters (as well as a full Western European character set).

There is ample prior work in combining low-level classifiers with dynamic time warping, hidden Markov models, Viterbi algorithms, and other search strategies to provide integrated segmentation and recognition for writing [15] and speech [11]. And there is a rich background in the use of ANNs as classifiers, including their use as low-level character classifiers in a higher-level word recognition system [2]. But these approaches leave a large number of open-ended questions about how to achieve acceptable (to a real user) levels of performance. In this paper, we survey some of our experiences in exploring refinements and improvements to these techniques.

13.2 System Overview

Apple’s print recognizer (APR) consists of three conceptual stages – Tentative Segmentation, Classification, and Context-Driven Search – as indicated in Figure 13.1. The primary data upon which we operate are simple sequences of (x,y) coordinate pairs, plus pen-up/down information, thus defining stroke primitives. The Segmentation stage decides which strokes will be combined to produce *segments* - the tentative groupings of strokes that will be treated as possible characters - and produces a sequence of these segments together with legal transitions between them. This process builds an implicit graph which is then labeled in the Classification stage and examined for a maximum likelihood interpretation in the Search stage. The Classification stage evaluates each segment using the

ANN classifier, and produces a vector of output activations that are used as letter-class probabilities. The Search stage then uses these class probabilities together with models of lexical and geometric context to find the N most likely word or sentence hypotheses.

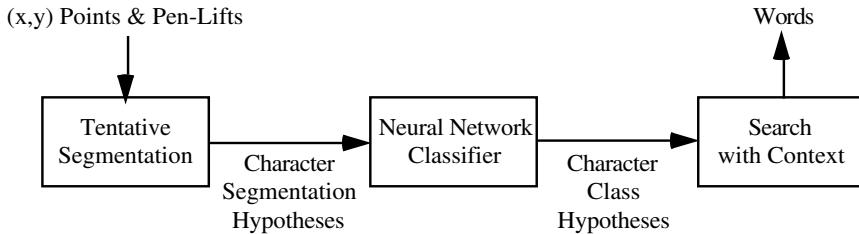


Fig. 13.1. A simplified block diagram of our hand-print recognizer

13.3 Tentative Segmentation

Character segmentation – the process of deciding which strokes comprise which characters – is inherently ambiguous. Ultimately this decision must be made, but, short of writing in boxes, it is impossible to do so (with any accuracy) in advance, external to the recognition process. Hence the initial segmentation stage in APR produces multiple, tentative groupings of strokes, and defers the final segmentation decisions until the search stage, thus integrating those segmentation decisions with the overall recognition process.

APR uses a potentially exhaustive, sequential enumeration of stroke combinations to generate a sequence of viable character-segmentation hypotheses. These *segments* are subjected to some obvious constraints (such as “all strokes must be used” and “no strokes may be used twice”), and some less obvious filters (to cull “impossible” segmentations for the sake of efficiency). The resulting algorithm produces the actual segments that will be processed as possible characters, along with the legal transitions between these segments.

The legal transitions are defined by *forward* and *reverse delays*. The forward delay indicates the next possible segment in the sequence (though later segments may also be legal), pointing just past the last segment that shares the trailing stroke of the current segment. The reverse delay indicates the start of the current batch of segments, all of which share the same leading stroke. Due to the enumeration scheme, a segment’s reverse delay is the same as its stroke count minus one, unless preceding segments (sharing the same leading stroke) were eliminated by the filters mentioned previously. These two simple delay parameters (per segment) suffice to define an implicit graph of all legal segment transitions. For a transition from segment number i to segment number j to be legal, the sum of segment i ’s forward delay plus segment j ’s reverse delay must be equal to $j - i$. Figure 13.2 provides an example of some ambiguous ink and the segments that might be generated from its strokes, supporting interpretations of “dog”, “clog”, “cbg”, or even “%g”.

Ink	Segment Number	Segment	Stroke Count	Forward Delay	Reverse Delay
clog	1	C	1	3	0
	2	c	2	4	1
	3	cl	3	4	2
	4		1	2	0
	5	l	2	2	1
	6	o	1	1	0
	7	g	1	0	0

Fig. 13.2. Segmentation of strokes into tentative characters or segments

13.4 Character Classification

The output of the segmentation stage is a stream of segments that are then passed to an ANN for classification as characters. Except for the architecture and training specifics detailed below, a fairly standard multi-layer perceptron trained with error back-propagation (BP) provides the ANN character classifier at the heart of APR. A large body of prior work exists to indicate the general applicability of ANN technology as a classifier providing good estimates of *a posteriori* probabilities of each class given the input ([5, 12, 11], and others cited herein). Compelling arguments have been made for why ANNs providing posterior probabilities in a probabilistic recognition formulation should be expected to outperform other recognition approaches [8], and ANNs have performed well as the core of speech recognition systems [10].

13.4.1 Representation

A recurring theme in ANN research is the extreme importance of the representation of the data that is given as input to the network. We experimented with a variety of input representations, including stroke features both anti-aliased (gray-scale) and not (binary), and images both anti-aliased and not, and with various schemes for positioning and scaling the ink within the image input window. In every case, anti-aliasing was a significant win. This is consistent with others' findings, that ANNs perform better when presented with smoothly varying, distributed inputs than they do when presented with binary, localized inputs. Almost the simplest image representation possible, a non-aspect-ratio-preserving, expand-to-fill-the-window image (limited only by a maximum scale factor to keep from blowing dots up to the full window size), together with either a single unit or a thermometer code (some number of units turned on in sequence to represent larger values) for the aspect ratio, proved to be the most effective single-classifier solution. However, the best overall classifier accuracy was ultimately obtained

by combining multiple distinct representations into nearly independent, parallel classifiers, joined at a final output layer. Hence representation proved not only to be as important as architecture, but, ultimately, to help define the architecture of our nets. For our final, hand-optimized system, we utilize four distinct inputs, as indicated in Figure 13.3. The stroke count representation was dithered (changed randomly at a small probability), to expand the effective training set, prevent the network from fixating on this simple input, and thereby improve the network’s ability to generalize. A schematic of the various input representations can be seen as part of the architecture drawing in Figure 13.4 in the next section.

Input Feature	Resolution	Description
Image	14x14	anti-aliased, scale-to-window, scale-limited
Stroke	20x9	anti-aliased, limited resolution tangent slope, resampled to fixed number of points
Aspect Ratio	1x1	normalized and capped to [0,1]
Stroke Count	5x1	dithered thermometer code

Fig. 13.3. Input representations used in APR

13.4.2 Architecture

As with representations, we experimented with a variety of architectures, including simple fully-connected layers, receptive fields, shared weights, multiple hidden layers, and, ultimately, multiple nearly independent classifiers tied to a common output layer. The final choice of architecture includes multiple input representations, a first hidden layer (separate for each input representation) using receptive fields, fully connected second hidden layers (again distinct for each representation), and a final, shared, fully-connected output layer. Simple scalar features – aspect ratio and stroke count – connect to both second hidden layers. The final network architecture, for our original English-language system, is shown in Figure 13.4.

Layers are fully connected, except for the inputs to the first hidden layer on the image side. This first hidden layer on the image side consists of 8 separate grids, each of which accepts inputs from the image input grid with its own receptive field sizes and strides, shown parenthetically in Figure 13.4 as (x-size x y-size; x-stride, y-stride). A stride is the number of units (pixels) in the input image space between sequential positionings of the receptive fields, in a given direction. The 7x2 and 2x7 side panels (surrounding the central 7x7 grid) pay special attention to the edges of the image. The 9x1 and 1x9 side panels specifically examine full-size vertical and horizontal features, respectively. The 5x5 grid observes features at a different spatial scale than the 7x7 grid.

Combining the two classifiers at the output layer, rather than, say, averaging the outputs of completely independent classifiers, allows generic BP to learn the best way to combine them, which is both convenient and powerful. But our *integrated multiple-representations* architecture is conceptually related to and motivated by prior experiments at combining nets such as Steve Nowlan’s “mixture of experts” [7].

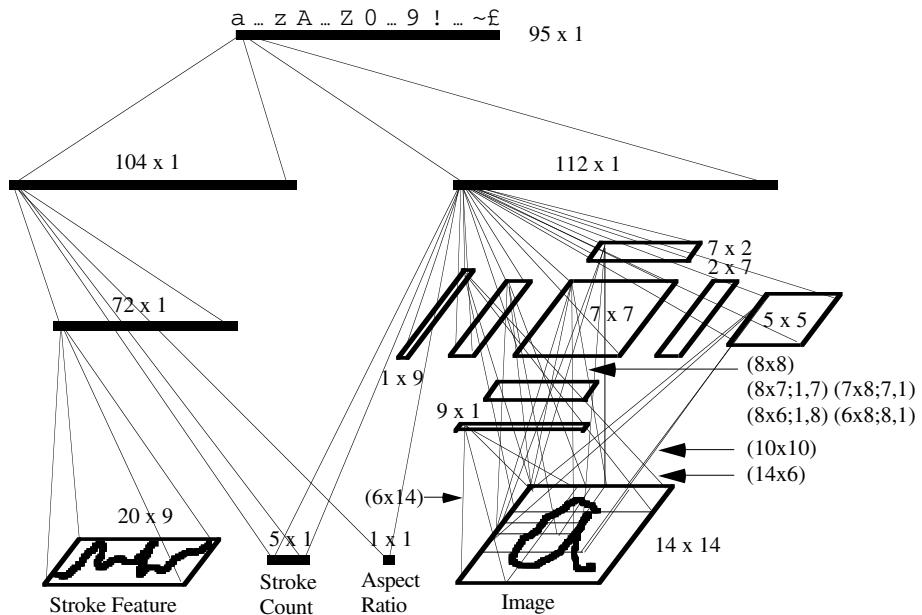


Fig. 13.4. Final English-language net architecture. (See the text for an explanation of the notation.)

13.4.3 Normalizing Output Error

Analyzing a class of errors involving words that were misrecognized due to perhaps a single misclassified character, we realized that the net was doing a poor job of representing second and third choice probabilities. Essentially, the net was being forced to attempt unambiguous classification of intrinsically ambiguous patterns due to the nature of the mean squared error minimization in BP, coupled with the typical training vector which consists of all 0's except for the single 1 of the target. Lacking any viable means of encoding legitimate probabilistic ambiguity into the training vectors, we decided to try "normalizing" the "pressure towards 0" vs. the "pressure towards 1" introduced by the output error during training. We refer to this technique as *NormOutErr*, due to its normalizing effect on target versus non-target output error.

We reduce the BP error for non-target classes relative to the target class by a factor that normalizes the total non-target error seen at a given output unit relative to the total target error seen at that unit. Assuming a training set with equal representation of classes, this normalization should then be based on the number of non-target versus target classes in a typical training vector, or, simply, the number of output units (minus one). Hence for non-target output units, we scale the error at each unit by a constant:

$$e' = Ae$$

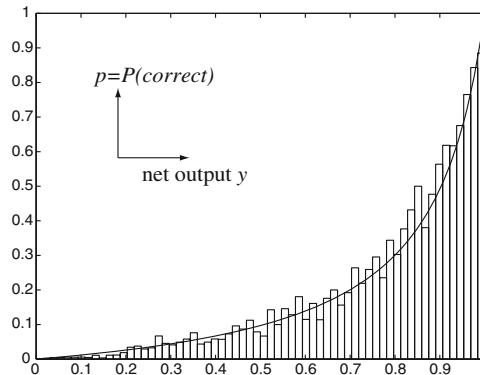


Fig. 13.5. Empirical p vs. y histogram for a net trained with $A = 0.11$ ($d = 0.1$), with the corresponding theoretical curve

where e is the error at an output unit, and A is defined to be:

$$A = \frac{1}{d(N_{outputs} - 1)}$$

where $N_{outputs}$ is the number of output units, and d is our tuning parameter, typically ranging from 0.1 to 0.2. Error at the target output unit is unchanged. Overall, this raises the activation values at the output units, due to the reduced pressure towards zero, particularly for low-probability samples. Thus the learning algorithm no longer converges to a least mean-squared error (LMSE) estimate of $p(class|input)$, but to an LMSE estimate of a nonlinear function $f(p(class|input), A)$ depending on the factor A by which we reduced the error pressure toward zero.

Using a simple version of the technique of [3], we worked out what that resulting nonlinear function is. The net will attempt to converge to minimize the modified quadratic error function

$$\langle \hat{E}^2 \rangle = p(1 - y)^2 + A(1 - p)y^2$$

by setting its output y for a particular class to

$$y = \frac{p}{A - Ap + p}$$

where $p = p(class|input)$, and A is as defined above. For small values of p , the activation y is increased by a factor of nearly $1/A$ relative to the conventional case of $y = p$, and for high values of p the activation is closer to 1 by nearly a factor of A . The inverse function, useful for converting back to a probability, is

$$p = \frac{yA}{yA + 1 - y}$$

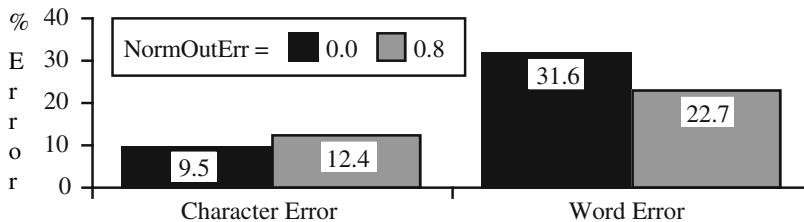


Fig. 13.6. Character and word error rates for two different values of $\text{NormOutErr}(d)$. A value of 0.0 disables NormOutErr , yielding normal BP. The unusually high value of 0.8 ($A = 0.013$) produces nearly equal pressures towards 0 and 1.

We verified the fit of this function by looking at histograms of character-level empirical percentage-correct versus y , as in Figure 13.5.

Even for this moderate amount of output error normalization, it is clear that the lower-probability samples have their output activations raised significantly, relative to the 45° line that $A = 1$ yields.

The primary benefit derived from this technique is that the net does a much better job of representing second and third choice probabilities, and low probabilities in general. Despite a small drop in top choice character accuracy when using NormOutErr , we obtain a very significant increase in word accuracy by this technique. Figure 13.6 shows an exaggerated example of this effect, for an atypically large value of d (0.8), which overly penalizes character accuracy; however, the 30% decrease in word error rate is normal for this technique. (Note: These data are from a multi-year-old experiment, and are not necessarily representative of current levels of performance on any absolute scale.)

13.4.4 Negative Training

The previously discussed inherent ambiguities in character segmentation necessarily result in the generation and testing of a large number of invalid segments. During recognition, the network must classify these invalid segments just as it would any valid segment, with no knowledge of which are valid or invalid. A significant increase in word-level recognition accuracy was obtained by performing *negative training* with these invalid segments. This consists of presenting invalid segments to the net during training, with all-zero target vectors. We retain control over the degree of negative training in two ways. First is a *negative-training factor* (ranging from 0.2 to 0.5) that modulates the learning rate (equivalently by modulating the error at the output layer) for these negative patterns. This reduces the impact of negative training on positive training, thus modulating the impact on characters that specifically look like elements of multi-stroke characters (e.g., I, 1, l, o, O, 0). Secondly, we control a *negative-training probability* (ranging between 0.05 and 0.3), which determines the probability that a particular negative sample will actually be trained on (for a given presentation). This

both reduces the overall impact of negative training, and significantly reduces training time, since invalid segments are more numerous than valid segments. As with *NormOutErr*, this modification hurts character-level accuracy a little bit, but helps word-level accuracy a lot.

13.4.5 Stroke Warping

During training (but not during recognition), we produce random variations in stroke data, consisting of small changes in skew, rotation, and x and y linear and quadratic scalings. This produces alternate character forms that are consistent with stylistic variations within and between writers, and induces an explicit aspect ratio and rotation invariance within the framework of standard back-propagation. The amounts of each distortion to apply were chosen through cross-validation experiments, as just the amount needed to yield optimum generalization. (*Cross-validation* is a standard technique for *early stopping* of ANN training, to prevent over-learning of the training set, and thus reduced accuracy on new data outside that training set. The technique consists of keeping aside some subset of the available data, the cross-validation set, and testing on it at some interval, but never training on it, and then stopping the training when accuracy ceases to improve on this cross-validation set, despite the fact that accuracy might continue to improve on the training set.) We chose relative amounts of the various transformations by testing for optimal final, converged accuracy on a cross-validation set. We then increased the amount of all stroke warping being applied to the training set, just to the point at which accuracy on the training set ceased to diverge from accuracy on the cross-validation set.

We also examined a number of such samples by eye to verify that they represent a natural range of variation. A small set of such variations is shown in Figure 13.7.



Fig. 13.7. A few random stroke warpings of the same original “m” data

Our stroke warping scheme is somewhat related to the ideas of Tangent Dist and Tangent Prop [14, 13] (see chapter 12), in terms of the use of predetermined families of transformations, but we believe it is much easier to implement. It is also somewhat distinct in applying transformations on the original coordinate data, as opposed to using distortions of images. The voice transformation scheme of [4] is also related, but they use a static replication of the training set through a small number of transformations, rather than dynamic random transformations of an essentially infinite variety.

13.4.6 Frequency Balancing

Training data from natural English words and phrases exhibit very non-uniform priors for the various character classes, and ANNs readily model these priors. However, as with *NormOutErr*, we find that reducing the effect of these priors on the net, in a controlled way, and thus forcing the net to allocate more of its resources to low-frequency, low-probability classes is of significant benefit to the overall word recognition process. To this end, we explicitly (partially) balance the frequencies of the classes during training. We do this by probabilistically skipping and repeating patterns, based on a precomputed repetition factor. Each presentation of a repeated pattern is “warped” uniquely, as discussed previously.

To compute the repetition factor for a class i , we first compute a normalized frequency of that class:

$$F_i = \frac{S_i}{\bar{S}}$$

where S_i is the number of samples in class i , and \bar{S} is the average number of samples over all classes, computed in the obvious way:

$$\bar{S} = \frac{1}{C} \sum_{i=1}^C S_i$$

with C being the number of classes. Our repetition factor is then defined to be:

$$R_i = \left(\frac{a}{F_i} \right)^b$$

with a and b being adjustable controls over the amount of skipping vs. repeating and the degree of prior normalization, respectively. Typical values of a range from 0.2 to 0.8, while b ranges from 0.5 to 0.9. The factor $a < 1$ lets us do more skipping than repeating; e.g. for $a = 0.5$, classes with relative frequency equal to half the average will neither skip nor repeat; more frequent classes will skip, and less frequent classes will repeat. A value of 0.0 for b would do nothing, giving $R_i = 1.0$ for all classes, while a value of 1.0 would provide “full” normalization. A value of b somewhat less than one seems to be the best choice, letting the net keep some bias in favor of classes with higher prior probabilities.

This explicit prior-bias reduction is conceptually related to Lippmann’s [8] and Morgan and Bourlard’s [10] recommended method for converting from the net’s estimate of posterior probability, $p(\text{class}|\text{input})$, to the value needed in an HMM or Viterbi search, $p(\text{input}|\text{class})$, which is to divide by $p(\text{class})$ priors. Using that technique, however, should produce noisier estimates for low frequency classes, due to the divisions by low frequencies, resulting in a set of estimates that are not really optimized in a LMSE sense (as the net outputs are). In addition, output activations that are naturally bounded between 0 and 1, due to the sigmoid, convert to potentially very large probability estimates, requiring a re-normalization step. Our method of frequency balancing during training eliminates both of these concerns. Perhaps more significantly, frequency balancing

also allows the standard BP training process to dedicate more network resources to the classification of the lower-frequency classes, though we have no current method for characterizing or quantifying this benefit.

13.4.7 Error Emphasis

While frequency balancing corrects for under-represented classes, it cannot account for under-represented writing styles. We utilize a conceptually related probabilistic skipping of patterns, but this time for just those patterns that the net correctly classifies in its forward/recognition pass, as a form of “error emphasis”, to address this problem. We define a *correct-train probability* (ranging from 0.1 to 1.0) that is used as a biased coin to determine whether a particular pattern, having been correctly classified, will also be used for the backward/training pass or not. This only applies to correctly segmented, or “positive” patterns, and misclassified patterns are never skipped.

Especially during early stages of training, we set this parameter fairly low (around 0.1), thus concentrating most of the training time and the net’s learning capability on patterns that are more difficult to correctly classify. This is the only way we were able to get the net to learn to correctly classify unusual character variants, such as a 3-stroke “5” as written by only one training writer.

Variants of this scheme are possible in which misclassified patterns would be repeated, or different learning rates would apply to correctly and incorrectly classified patterns. It is also related to techniques that use a training subset, from which easily-classified patterns are replaced by randomly selected patterns from the full training set [6].

13.4.8 Annealing

Though some discussions of back-propagation espouse explicit formulae for modulating the learning rate over time, many seem to assume the use of a single, fixed learning rate. We view the stochastic back-propagation process as a kind of simulated annealing, with a learning rate starting very high and decreasing only slowly to a very low value. But rather than using any prespecified formula to decelerate learning, the rate at which the learning rate decreases is determined by the dynamics of the learning process itself. We typically start with a rate near 1.0 and reduce the rate by a multiplicative *decay factor* of 0.9 until it gets down to about 0.001. The rate decay factor is applied following any epoch in which the total squared error increased on the training set, relative to the previous epoch. This “total squared error” is summed over all output units and over all patterns in one full epoch, and normalized by those counts. So even though we are using “online” or stochastic gradient descent, we have a measure of performance over whole epochs that can be used to guide the “annealing” of the learning rate. Repeated tests indicate that this approach yields better results than low (or even moderate) initial learning rates, which we speculate to be related to a better ability to escape local minima.

In addition, we find that we obtain best overall results when we also allow some of our many training parameters to change over the course of a training run. In particular, the correct train probability needs to start out very low to give the net a chance to learn unusual character styles, but it should finish up near 1.0 in order to not introduce a general posterior probability bias in favor of classes with lots of ambiguous examples. We typically train a net in four “phases” according to parameters such as in Figure 13.8.

Phase	Epochs	Learning Rate	Correct Train Prob	Negative Train Prob
1	25	1.0 - 0.5	0.1	0.05
2	25	0.5 - 0.1	0.25	0.1
3	50	0.1 - 0.01	0.5	0.18
4	30	0.01 - 0.001	1.0	0.3

Fig. 13.8. Typical multi-phase schedule of learning rates and other parameters for training a character-classifier net

13.4.9 Quantized Weights

The work of Asanovic and Morgan [1] shows that two-byte (16-bit) weights are about the smallest that can be tolerated in training large ANNs via back-propagation. But memory is expensive in small devices, and RISC processors, such as the ARM-610 in the first devices in which this technology was deployed, are much more efficient doing one-byte loads and multiplies than two-byte loads and multiplies, so we were motivated to make one-byte weights work.

Running the net for recognition demands significantly less precision than does training the net. It turns out that one-byte weights provide adequate precision for recognition, if the weights are trained appropriately. In particular, a dynamic range should be fixed, and weights limited to that legal range during training, and then rounded to the requisite precision after training. For example, we find that a range of weight values from (almost) -8 to +8 in steps of 1/16 does a good job. Figure 13.9 shows a typical resulting distribution of weight values. If the weight limit is enforced during high-precision training, the resources of the net will be adapted to make up for the limit. Since bias weights are few in number, however, and very important, we allow them to use two bytes with essentially unlimited range. Performing our forward/recognition pass with low-precision, one-byte weights (a ± 3.4 fixed-point representation), we find no noticeable degradation relative to floating-point, four-byte, or two-byte weights using this scheme.

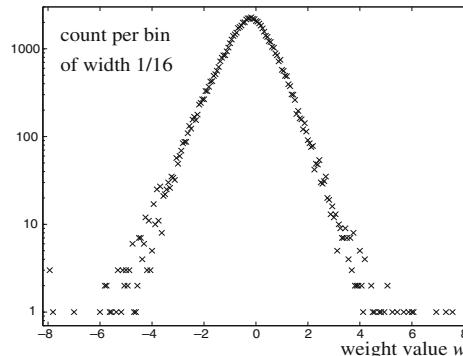


Fig. 13.9. Distribution of weight values in a net with one-byte weights, on a log count scale. Weights with magnitudes greater than 4 are sparse, but important.

We have also developed a scheme for training with augmented one-byte weights. It uses a temporary augmentation of the weight values with two additional low-order bytes to achieve precision in training, but runs the forward pass of the net using only the one-byte high-order part. Thus any cumulative effect of the one-byte rounded weights in the forward pass can be compensated through further training. Small weight changes accumulate in the low-order bytes, and only occasionally carry into a change in the one-byte weights used by the net. In a personal product, this scheme could be used for adaptation to the user, after which the low-order residuals could be discarded and the temporary memory reclaimed.

13.5 Context-Driven Search

The output of the ANN classifier is a stream of probability vectors, one vector for each segmentation hypothesis, with as many potentially nonzero probability elements in each vector as there are characters (that the system is capable of recognizing). In practice, we typically only pass the top ten (or fewer) scored character-class hypotheses, per segment, to the search engine, for the sake of efficiency. The search engine then looks for a minimum-cost path through this vector stream, abiding by the legal transitions between segments, as defined in the tentative-segmentation step discussed previously. This minimum-cost path is the APR system's best interpretation of the ink input by the user, and is returned to the system in which APR is embedded as the recognition result for whole words or sentences of the user's input.

The search is driven by a somewhat *ad hoc*, generative language model, which consists of a set of graphs that are searched in parallel. We use a simple beam search in a negative-log-probability (or *penalty*) space for the best N hypotheses. The beam is based on a fixed maximum number of hypotheses, rather than a

particular value. Each possible transition token (character) emitted by one of the graphs is scored not only by the ANN, but by the language model itself, by a simple letter-case model, and by geometric-context models discussed below. The fully integrated search process takes place over a space of character- and word-segmentation hypotheses, as well as character-class hypotheses.

13.5.1 Lexical Context

Context is essential to accurate recognition, even if that context takes the form of a very broad language model. Humans achieve just 90% accuracy on isolated characters from our database. Lacking any context this would translate to a word accuracy of not much more than 60% (0.9^5), assuming an average word length of 5 characters. We obviously need to do much better, with even lower isolated-character accuracy, and we accomplish this by the application of our context models.

A simple model of letter case and adjacency – penalizing case transitions except between the first and second characters, penalizing alphabetic-to-numeric transitions, and so on – together with the geometric-context models discussed later, is sufficient to raise word-level accuracy up to around 77%.

The next large gain in accuracy requires a genuine language model. We provide this model by means of dictionary graphs, and assemblages of those graphs combined into what we refer to as *BiGrammars*. BiGrammars are essentially scored lists of dictionaries, together with specified legal (scored) transitions between those dictionaries. This scheme allows us to use word lists, prefix and suffix lists, and punctuation models, and to enable appropriate transitions between them. Some dictionary graphs are derived from a regular-expression grammar that permits us to easily model phone numbers, dates, times, etc., as shown in Figure 13.10.

```

dig      = [0123456789]
digm01 = [23456789]

acodenums = (digm01 [01] dig)

acode   = { ("1-"?    acodenums "-") : 40 ,
            ("1"? "(" acodenums ")") : 60 }

phone = (acode? digm01 dig dig "--" dig dig dig dig)

```

Fig. 13.10. Sample of the regular-expression language used to define a simple telephone-number grammar. Symbols are defined by the equal operator; square brackets enclose multiple, alternative characters; parentheses enclose sequences of symbols; curly braces enclose multiple, alternative symbols; an appended colon followed by numbers designates a prior probability of that alternative; an appended question mark means “zero or one occurrence”; and the final symbol definition represents the graph or grammar expressed by this dictionary.

All of these dictionaries can be searched in parallel by combining them into a general-purpose BiGrammar that is suitable for most applications. It is also possible to combine subsets of these dictionaries, or special-purpose dictionaries, into special BiGrammars targeted at more limited contexts. A very simple

```
BiGrammar Phone
[Phone.lang 1. 1. 1.]
```

Fig. 13.11. Sample of a simple BiGrammar describing a telephone-only context. The BiGrammar is first named (Phone), and then specified as a list of dictionaries (Phone.lang), together with the probability of starting with this dictionary, ending with this dictionary, and cycling within this dictionary (the three numerical values).

```
BiGrammar FairlyGeneral
(.8
  (.6
    [WordList.dict .5 .8 1. EndPunct.lang .2]
    [User.dict      .5 .8 1. EndPunct.lang .2]
  )
  (.4
    [Phone.lang     .5 .8 1. EndPunct.lang .2]
    [Date.lang      .5 .8 1. EndPunct.lang .2]
  )
)

(.2
  [OpenPunct.lang 1. 0. .5
  (.6
    WordList.dict .5
    User.dict     .5
  )
  (.4
    Phone.lang    .5
    Date.lang     .5
  )
]
)
```

[EndPunct.lang 0. .9 .5 EndPunct.lang .1]

Fig. 13.12. Sample of a slightly more complex BiGrammar describing a fairly general context. The BiGrammar is first named (FairlyGeneral), and then specified as a list of dictionaries (the *.dict and *.lang entries), together with the probability of starting with this dictionary, ending with this dictionary, and cycling within this dictionary (the first three numerical values following each dictionary name), plus any dictionaries to which this dictionary may legally transition, along with the probability of taking that transition. The parentheses permit easy specification of multiplicative prior probabilities for all dictionaries contained within them. Note that in this simple example, it is not possible (starting probability = 0) to start a string with the EndPunct (end punctuation) dictionary, just as it is not possible to end a string with the OpenPunct dictionary.

BiGrammar, which might be useful to specify context for a field that only accepts telephone numbers, is shown in Figure 13.11. A more complex BiGrammar (though still far short of the complexity of our final general-input context) is shown in Figure 13.12.

We refer to our language model as being “weakly applied” because in parallel with all of the wordlist-based dictionaries and regular-expression grammars, we simultaneously search both an alphabetic-characters grammar (“wordlike”) and a completely general, any-character-anywhere grammar (“symbols”). These more flexible models, though given fairly low *a priori* probabilities, permit users to write any unusual character string they might desire. When the prior probabilities for the various dictionaries are properly balanced, the recognizer is able to benefit from the language model, and deliver the desired level of accuracy for common in-dictionary words (and special constructs like phone numbers, etc.), yet can also recognize arbitrary, non-dictionary character strings, especially if they are written neatly enough that the character classifier can be confident of its classifications.

We have also experimented with bi-grams, tri-grams, N-grams, and we are continuing experiments with other, more data-driven language models; so far, however, our generative approach has yielded the best results.

13.5.2 Geometric Context

We have never found a way to reliably estimate a baseline or topline for characters, independent of classifying those characters in a word. Non-recognition-integrated estimates of these line positions, based on strictly geometric features, have too many pathological failure modes, which produce erratic recognition failures. Yet the geometric positioning of characters most certainly bears information important to the recognition process. Our system factors the problem by letting the ANN classify representations that are independent of baseline and size, and then using separate modules to score both the absolute size of individual characters, and the relative size and position of adjacent characters.

The scoring based on absolute size is derived from a set of simple Gaussian models of individual character heights, relative to some running scale parameters computed both during learning and during recognition. This *CharHeight* score directly multiplies the scores emitted by the ANN classifier, and helps significantly in case disambiguation.

We also employ a *GeoContext* module that scores adjacent characters, based on the classification hypotheses for those characters and on their relative size and placement. *GeoContext* scores each tentative character based on its class and the class of the immediately preceding letter (for the current search hypothesis). The character classes are used to look up expected character sizes and positions in a standardized space ($\text{baseline}=0.0$, $\text{topline}=1.0$). The ink being evaluated provides actual sizes and positions that can be compared directly to the expected values, subject only to a scale factor and offset, which are chosen so as to minimize the estimated error of fit between data and model. This same quadratic error term, computed from the inverse covariance matrix of a

full multivariate Gaussian model of these sizes and positions, is used directly as GeoContext's score (or penalty, since it is applied in the -log probability space of the search engine). Figure 13.13 illustrates the bounding boxes derived from the user's ink vs. the table-driven model, with the associated error measures for our GeoContext module.

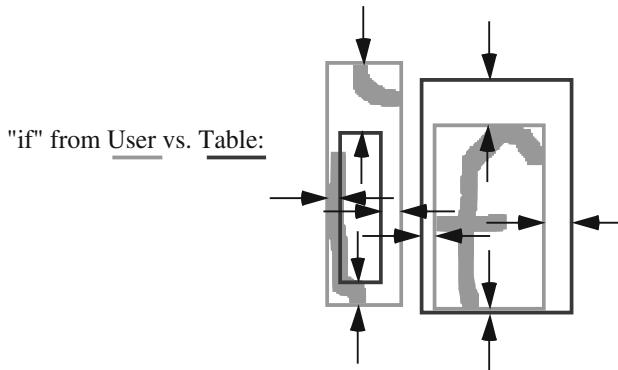


Fig. 13.13. The eight measurements that contribute to the GeoContext error vector and corresponding score for each letter pair

GeoContext's multivariate Gaussian model is learned directly from data. The problem in doing so was to find a good way to train per-character parameters of top, bottom, width, space, etc., in our standardized space, from data that had no labeled baselines, or other absolute referent points. Since we had a technique for generating an error vector from the table of parameters, we decided to use a back-propagation variant to train the table of parameters to minimize the squared error terms in the error vectors, given all the pairs of adjacent characters and correct class labels from the training set.

GeoContext plays a major role in properly recognizing punctuation, in disambiguating case, and in recognition in general. A more extended discussion of GeoContext has been provided by Lyon and Yaeger [9].

13.5.3 Integration with Word Segmentation

Just as it is necessary to integrate character segmentation with recognition via the search process, so is it essential to integrate word segmentation with recognition and search, in order to obtain accurate estimates of word boundaries, and to reduce the large class of errors associated with missegmented words. To perform this integration, we first need a means of estimating the probability of a word break between each pair of tentative characters. We use a simple statistical model of gap sizes and stroke-centroid spacing to compute this probability (*spaceProb*). Gaussian density distributions, based on means and standard deviations computed from a large training corpus, together with a prior probability scale factor, provide the basis for the word-gap and stroke-gap (non-word-gap)

models, as illustrated in Figure 13.14. Since any given gap is, by definition, either a word gap or a non-word gap, the simple ratio defined in Figure 13.14 provides a convenient, self-normalizing estimate of the word-gap probability. In practice, that equation further reduces to a simple sigmoid form, thus allowing us to take advantage of a lookup-table-based sigmoid derived for use in the ANN. In a thresholding, non-integrated word-segmentation model, word breaks would be introduced when spaceProb exceeds 0.5; i.e., when a particular gap is more likely to be a word-gap than a non-word-gap. For our integrated system, both word-break and non-word-break hypotheses are generated at each segment transition, and weighted by spaceProb and (1-spaceProb), respectively. The search process then proceeds over this larger hypothesis space to produce best estimates of whole phrases or sentences, thus integrating word segmentation as well as character segmentation.

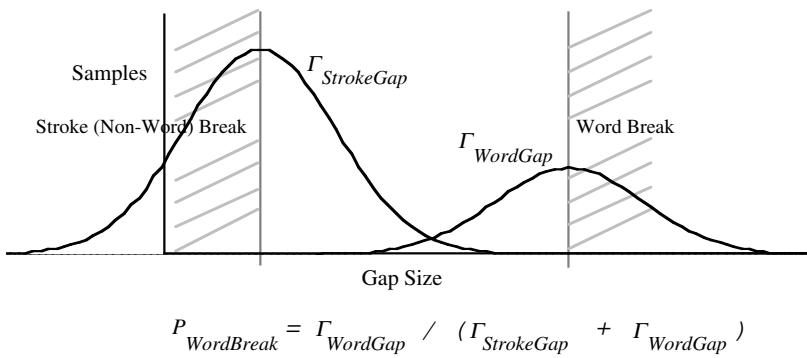


Fig. 13.14. Gaussian density distributions yield a simple statistical model of word-break probability, which is applied in the region between the peaks of the $\Gamma_{StrokeGap}$ and $\Gamma_{WordGap}$ distributions. Hashed areas indicate regions of clear cut decisions, where $P_{WordBreak}$ is set to either 0.0 or 1.0, to avoid problems dealing with tails of these simple distributions.

13.6 Discussion

The combination of elements described in the preceding sections produces a powerful, integrated approach to character segmentation, word segmentation, and recognition. Users' experiences with APR are almost uniformly positive, unlike experiences with previous handwriting recognition systems. Writing within the dictionary is remarkably accurate, yet the ease with which people can write outside the dictionary has fooled many people into thinking that the Newton's "Print Recognizer" does not use dictionaries. As discussed previously, our recognizer certainly does use dictionaries. Indeed, the broad-coverage language model, though weakly applied, is essential for high accuracy recognition. Curiously, there seems to be little problem with dictionary *perplexity* – little difficulty as a result of using very large, very complex language models. We attribute this fortunate behavior to the excellent performance of the neural network character classifier

at the heart of the system. One of the side benefits of the weak application of the language model is that even when recognition fails and produces the wrong result, the answer that is returned to the user is typically understandable by the user – perhaps involving substitution of a single character. Two useful phenomena ensue as a result. First, the user learns what works and what doesn't, especially when she refers back to the ink that produced the misrecognition, so the system trains the user gracefully over time. Second, the meaning is not lost the way it can be, all too easily, with whole word substitutions – with that “Doonesbury Effect” found in first-generation, strong-language-model recognizers.

Though we have provided legitimate accuracy statistics for certain comparative tests of some of our algorithms, we have deliberately shied away from claiming specific levels of accuracy in general. Neat printers, who are familiar with the system, can achieve 100% accuracy if they are careful. Testing on data from complete novices, writing for the first time using a metal pen on a glass surface, without any feedback from the recognition system, and with ambiguous instructions about writing with “disconnected characters” (intended to mean printing, but often interpreted to mean writing with otherwise cursive characters but separated by large spaces in a wholly unnatural style), can yield word-level accuracies as low as 80%. Of course, the entire interesting range of recognition accuracies lies between these two extremes. Perhaps a slightly more meaningful statistic comes from common reports on usenet newsgroups, and some personal testing, that suggest accuracies of 97% to 98% in regular use. But for scientific purposes, none of these numbers have any real meaning, since our testing datasets are proprietary, and the only valid tests between different recognizers would have to be based on results obtained by processing the exact same bits, or by analyzing large numbers of experienced users of the systems in the field – a difficult project which has not been undertaken.

One of the key reasons for the success of APR is the suite of innovative neural network training techniques that help the network encode better class probabilities, especially for under-represented classes and writing styles. Many of these techniques – stroke count dithering, normalization of output error, frequency balancing, error emphasis – share a unifying theme: Reducing the effect of *a priori* biases in the training data on network learning significantly improves the network's performance in an integrated recognition system, despite a modest reduction in the network's accuracy for individual characters. Normalization of output error prevents over-represented non-target classes from biasing the net against under-represented target classes. Frequency balancing prevents over-represented classes from biasing the net against under-represented classes. And stroke-count dithering and error emphasis prevent over-represented writing styles from biasing the net against under-represented writing styles. One could even argue that negative training eliminates an absolute bias towards properly segmented characters, and that stroke warping reduces the bias towards those writing styles found in the training data, although these techniques also provide wholly new information to the system.

Though we've offered arguments for why each of these techniques, individually, helps the overall recognition process, it is unclear why prior-bias reduction, in general, should be so consistently valuable. The general effect may be related to the technique of dividing out priors, as is sometimes done to convert from $p(class|input)$ to $p(input|class)$. But we also believe that forcing the net, during learning, to allocate resources to represent less frequent sample types may be directly beneficial. In any event, it is clear that paying attention to such biases and taking steps to modulate them is a vital component of effective training of a neural network serving as a classifier in a maximum-likelihood recognition system.

The majority of this paper describes a sort of snapshot of the system and its architecture as it was deployed in its first commercial release, when it was, indeed, purely a "Print Recognizer". Letters had to be fully "disconnected"; i.e., the pen had to be lifted between each pair of characters. The characters could overlap to some extent, but the ink could not be continuous. Connected characters proved to be the largest remaining class of errors for most of our users, since even a person who normally prints (as opposed to writing in cursive script) may occasionally connect a pair of characters – the cross-bar of a "t" with the "h" in "the", the "o" and "n" in any word ending in "ion", and so on. To address this issue, we experimented with some fairly straightforward modifications to our recognizer, involving the *fragmenting* of user-strokes into multiple system-strokes, or *fragments*. Once the ink representing the connected characters is broken up into fragments, we then allow our standard integrated segmentation and recognition process to stitch them back together into the most likely character and word hypotheses, as always. This technique has proven itself to work quite well, and the version of the "Print Recognizer" in the MessagePad 2000 and 2100 supports recognition of printing with connected characters. This capability was added without significant modification of the main recognition algorithms as presented in this paper. Due to certain assumptions and constraints in the current release of the software, APR is not yet a full cursive recognizer, though that is an obvious next direction to explore.

The net architecture discussed in section 13.4.2 and shown in Figure 13.4 also corresponds to the true printing-only recognizer. The final output layer has 95 elements corresponding to the full printable ASCII character set plus the British Pound sign. Initially for the German market, and now even in English units, we have extended APR to handle diacritical marks and the special symbols needed for most European languages (although there is only very limited coverage of foreign languages in the dictionaries of English units). The main innovation that permitted this extended character set was an explicit handling of any compound character as a *base* plus an *accent*. This way only a few nodes needed to be added to the neural network output layer, representing just the bases and accents, rather than all combinations and permutations of same. And training data for all compound characters sharing a common base or a common accent contributed to the network's ability to learn that base or accent, as opposed to contributing only to the explicit base+accent combination. Here again, however, the fundamental

recognizer technology has not changed significantly from that presented in this paper.

13.7 Future Extensions

We are optimistic that our algorithms, having proven themselves to work essentially as well for connected characters as for disconnected characters, may extend gracefully to full cursive.

On a more speculative note, we believe that the technique may extend well to ideographic languages, substituting radicals for characters, and ideographic characters for words.

Finally, a note about learning and user adaptation: For a learning technology such as ANNs, user adaptation is an obvious and natural fit, and was planned as part of the system from its inception. However, due to RAM constraints in the initial shipping product, and the subsequent prioritization of European character sets and connected characters, we have not yet deployed a learning system. We have, however, done some testing of user adaptation, and believe it to be of considerable value. Figure 13.15 shows a comparison of the average performance on an old user-independent net trained on data from 45 writers, and the performance for three individuals using (A) the user-independent net, (B) a net trained on data exclusively from that individual, and (C) a copy of the user-independent net adapted to the specific user by some incremental training. (Note: These data are from a multi-year-old experiment, and are not necessarily representative of current levels of performance on any absolute scale.)

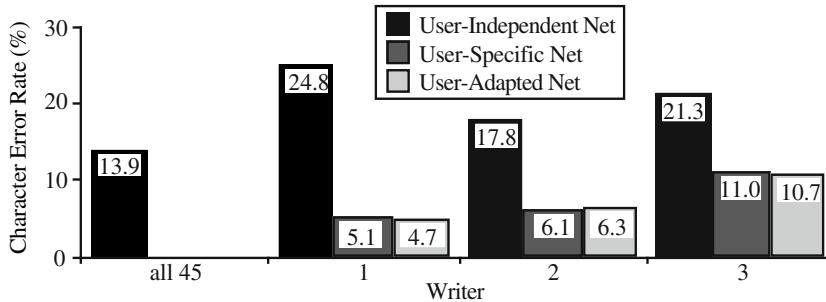


Fig. 13.15. User-adaptation test results for three individual writers with three different nets each, plus the overall results for 45 writers tested on a user-independent net trained on all 45 writers

An important distinction is being made here between “user-adapted” and “user-specific” nets. “User-specific” nets have been trained with a relatively large corpus of data exclusively from that specific user. “User-adapted” nets were based on the user-independent net, with some additional training using limited data from the user in question. All testing was performed with data held out from all training sets.

One obvious thing to note is the reduction in error rate ranging from a factor of 2 to a factor of 5 that both user-specific and user-adapted nets provide. An equally important thing to note is that the user-adapted net performs essentially as well as a user-specific net – in fact, slightly *better* for two of the three writers. Given ANNs’ penchant for local minima, we were concerned that this might not be the case. But it appears that the features learned during the user-independent net training served the user-adapted net well. We believe that a very small amount of training data from an individual will allow us to adapt the user-independent net to that user, and improve the overall accuracy for that user significantly, especially for individuals with more stylized writing, or whose writing style is underrepresented in our user-independent training corpus. And even for writers with common and/or neat writing styles, there is inherently less ambiguity in a single writer’s style than in a corpus of data necessarily doing its best to represent essentially all possible writing styles.

These results may be exaggerated somewhat by the limited data in the user-independent training corpus at the time these tests were performed (just 45 writers), and at least two of the three writers in question had particularly problematic writing styles. We have also made significant advances in our user-independent recognition accuracies since these tests were performed. Nonetheless, we believe these results are suggestive of the significant value of user adaptation, even in preference to a user-specific solution.

Acknowledgements. This work was done in collaboration with Bill Stafford, Apple Computer, and Les Vogel, Angel Island Technologies. We would also like to acknowledge the contributions of Michael Kaplan, Rus Maxham, Kara Hayes, Gene Ciccarelli, and Stuart Crawford. We also received support and assistance from the Newton Recognition, Software Testing, and Hardware Support groups. We are also indebted to our many colleagues in the connectionist community for their advice, help, and encouragement over the years, as well as our many colleagues at Apple who pitched in to help throughout the life of this project.

Some of the techniques described in this paper are the subject of pending U.S. and foreign patent applications.

References

- [1] Asanovic, K., Morgan, N.: Experimental determination of precision requirements for back-propagation training of artificial neural networks. Technical Report TR-91-036, International Computer Science Institute, Berkeley, CA (June 1991)
- [2] Bengio, Y., LeCun, Y., Nohl, C., Burges, C.: LeRec: A NN/HMM hybrid for on-line handwriting recognition. *Neural Computation* 7(6), 1289–1303 (1995)
- [3] Bourlard, H., Wellekens, C.J.: Links between markov models and multilayer perceptrons. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI* 12(12), 1167–1178 (1990)
- [4] Chang, E.I., Lippmann, R.P.: Using voice transformations to create additional training talkers for word spotting. In: Tesauro, G., Touretzky, D., Leen, T. (eds.) *Advances in Neural Information Processing Systems*, vol. 7, pp. 875–882. The MIT Press (1995)

- [5] Gish, H.: A probabilistic approach to the understanding and training of neural network classifiers. In: Proceedings of the IEEE Conference on Acoustics, Speech and Signal Processing, pp. 1361–1364. IEEE Press (1990)
- [6] Guyon, I., Henderson, D., Albrecht, P., LeCun, Y., Denker, P.: Writer independent and writer adaptive neural network for on-line character recognition. In: Impedovo, S. (ed.) From Pixels to Features III, pp. 493–506. Elsevier, Amsterdam (1992)
- [7] Jacobs, R.A., Jordan, M.I., Nowlan, S.J., Hinton, G.E.: Adaptive mixtures of local experts. *Neural Computation* 3(1), 79–87 (1991)
- [8] Lippmann, R.P.: Neural networks, bayesian *a posteriori* probabilities, and pattern classification. In: Cherkassky, V., Friedman, J.H., Wechsler, H. (eds.) From Statistics to Neural Networks – Theory and Pattern Recognition Applications, pp. 83–104. Springer, Berlin (1994)
- [9] Lyon, R., Yaeger, L.: On-line hand-printing recognition with neural networks. In: Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems, Lausanne, Switzerland. IEEE Computer Society Press (1996)
- [10] Morgan, N., Bourlard, H.: Continuous speech recognition – an introduction to the hybrid HMM/connectionist approach. *IEEE Signal Processing Mag* 13(3), 24–42 (1995)
- [11] Renals, S., Morgan, N., Cohen, M., Franco, H.: Connectionist probability estimation in the Decipher speech recognition system. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. I, pp. 601–604 (1992)
- [12] Richard, M.D., Lippmann, R.P.: Neural Network Classifiers Estimate Bayesian *a posteriori* probabilities. *Neural Computation* 3(4), 461–483 (1991)
- [13] Simard, P., LeCun, Y., Denker, J.: Efficient pattern recognition using a new transformation distance. In: Hanson, S.J., Cowan, J.D., Giles, C.L. (eds.) Proceedings of the 1992 Conference on Advances in Neural Information Processing Systems, vol. 5, pp. 50–58. Morgan Kaufmann, San Mateo (1992)
- [14] Simard, P., Victorri, B., LeCun, Y., Denker, J.: Tangent prop—A formalism for specifying selected invariances in an adaptive network. In: Moody, J.E., Hanson, S.J., Lippmann, R.P. (eds.) Advances in Neural Information Processing Systems, vol. 4, pp. 895–903. Morgan Kaufmann Publishers, Inc. (1992)
- [15] Tappert, C.C., Suen, C.Y., Wakahara, T.: The state of the art in on-line handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI* 12(8), 787–808 (1990)

Neural Network Classification and Prior Class Probabilities^{*}

Steve Lawrence¹, Ian Burns², Andrew Back³, Ah Chung Tsoi⁴,
and C. Lee Giles^{1, **}

¹ NEC Research Institute,

4 Independence Way, Princeton, NJ 08540

<http://www.neci.nj.nec.com>

² Open Access Pty Ltd, Level 2, 7–9 Albany St, St. Leonards, NSW 2065, Australia

³ RIKEN Brain Science Institute, 2-1 Hirosawa, Wako-shi, Saitama, 351-0198, Japan

⁴ Faculty of Informatics, University of Wollongong, Northfields Ave, Wollongong,
NSW 2522, Australia

{lawrence,giles}@research.nj.nec.com, ian.burns@oa.com.au,

back@brain.riken.go.jp, Ah_Chung_Tsoi@uow.edu.au

<http://www.neci.nj.nec.com/homepages/lawrence/>

Abstract. A commonly encountered problem in MLP (multi-layer perceptron) classification problems is related to the prior probabilities of the individual classes – if the number of training examples that correspond to each class varies significantly between the classes, then it may be harder for the network to learn the rarer classes in some cases. Such practical experience does not match theoretical results which show that MLPs approximate Bayesian *a posteriori* probabilities (independent of the prior class probabilities). Our investigation of the problem shows that the difference between the theoretical and practical results lies with the assumptions made in the theory (accurate estimation of Bayesian *a posteriori* probabilities requires the network to be large enough, training to converge to a global minimum, infinite training data, and the *a priori* class probabilities of the test set to be correctly represented in the training set). Specifically, the problem can often be traced to the fact that efficient MLP training mechanisms lead to sub-optimal solutions for most practical problems. In this chapter, we demonstrate the problem, discuss possible methods for alleviating it, and introduce new heuristics which are shown to perform well on a sample ECG classification problem. The heuristics may also be used as a simple means of adjusting for unequal misclassification costs.

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

** Lee Giles is also with the Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742.

14.1 Introduction

It has been shown theoretically that MLPs approximate Bayesian *a posteriori* probabilities when the desired network outputs are *1 of M* and squared-error or cross-entropy cost functions are used [6, 11, 12, 15, 23, 25, 26, 28, 29, 32]. This result relies on a number of assumptions for accurate estimation: the network must be large enough and training must find a global minimum, infinite training data is required, and the *a priori* class probabilities of the test set must be correctly represented in the training set.

In practice, MLPs have also been shown to accurately estimate Bayesian *a posteriori* probabilities for certain experiments [10]. However, a commonly encountered problem in MLP classification is related to the case when the frequency of the classes in the training set varies significantly¹. If the number of training examples for each class varies significantly between classes then there may be a bias towards predicting the more common classes [3, 4], leading to worse classification performance for the rarer classes. In [5] it was observed that classes with low *a priori* probability in a speech application were “ignored” (no samples were classified as these classes after training). Such problems indicate that either the estimation of Bayesian *a posteriori* probabilities is inaccurate, or that such estimation may not be desired (e.g. due to varying misclassification costs (this is explained further in section 14.4)). Bourlard and Morgan [7] have demonstrated inaccurate estimation of Bayesian *a posteriori* probabilities in speech recognition. This chapter discusses how the problem may occur along with methods of dealing with the problem.

14.2 The Trick

This section describes the tricks for alleviating the aforementioned problem. Motivation for their use and experimental results are provided in the following sections. The methods all consider some kind of scaling which is performed on a class by class basis².

14.2.1 Prior Scaling

A method of scaling weight updates on a class by class basis according to the prior class probabilities is proposed in this section. Consider gradient descent weight updates for each pattern: $w_{ki}^l(\text{new}) = w_{ki}^l(\text{old}) + \Delta w_{ki}^l(p)$ where $\Delta w_{ki}^l(p) = -\eta \frac{\partial E(p)}{\partial w_{ki}^l}$, p is the pattern index, and w_{ki} is the weight between neuron k in layer l and neuron i in layer $l-1$. Scaling the weight updates on a pattern by pattern

¹ For the data in general. Others have considered the case of different class probabilities between the training and test sets, e.g. [23].

² Anand et al. [2] have also presented an algorithm related to unequal prior class probabilities. However, their algorithm aims only to improve convergence speed. Additionally, their algorithm is only for two class problems and batch update.

basis is considered such that the total expected update for patterns belonging to each class is equal (i.e. independent of the number of patterns in the class):

$$\left\langle \sum_{p=1}^{N_p} |s_x \Delta w_{ki}^l(p)|_{p_c=x} \right\rangle = c_1, \forall x \in X \quad (14.1)$$

where p_c is the target classification of pattern p , c_1 is a constant, s_x is a scaling factor, x ranges over all classes X , $\langle \rangle$ denotes expectation, and the $p_c = x$ subscript indicates that the sum is only over the patterns in a particular class x . This effectively scales the updates for lower frequency classes so that they are higher – the aim is to account for the fact that lower frequency classes tend to be “ignored” in certain situations. We assume that the expected weight update for individual patterns in each class is equal:

$$\langle |\Delta w_{ki}^l(p)|_{p_c=x} \rangle = c_2, \forall x \in X \quad (14.2)$$

where c_2 is a constant not related to c_1 . The scaling factor required is therefore:

$$s_x = \frac{1}{p_x N_c} \quad (14.3)$$

where s_x is the scaling factor for all weight updates associated with a pattern belonging to class x , N_c is the number of classes, and p_x is the prior probability of class x .

Scaling as defined above invalidates the Bayesian *a posteriori* probability proofs (for example, scaling a class by two can be compared with duplicating every pattern in the data for that class – causing changes in probability distributions), i.e. there is no reason to expect that the scaling strategy will be optimal. This, and the empirical result that the scaling may improve performance, leads to the hypothesis that there may be a point between no prior scaling and prior scaling as defined above which produces performance better than either of the two extremes. The following scaling rule can be used to select a degree of scaling between the two extremes:

$$s'_x = 1 - c_s + \frac{c_s}{p_x N_c} \quad (14.4)$$

where $0 \leq c_s \leq 1$ is a constant specifying the amount of prior scaling to use. $c_s = 0$ corresponds to no scaling according to prior probabilities, and $c_s = 1$ corresponds to scaling as above. Prior scaling in this form can be expressed as training with the following alternative cost function³

³ A cost function with similar motivation, the “classification figure-of-merit” (CFM) proposed by Hampshire and Waibel [13], has been suggested as a possible improvement when prior class probabilities vary [3]. In [13], the CFM cost function leads to networks which make different errors to those trained with the MSE criterion, and can therefore be useful for improving performance by combining classifiers trained with the CFM and the MSE. However, networks trained with the CFM criterion do not result in higher classification performance than networks trained with the MSE criterion for the experiments reported in [13].

Definition 1

$$E = \frac{1}{2} \sum_{k=1}^{N_p} \sum_{j=1}^{N_c} s'_x(d_{kj} - y_{kj})^2 \quad (14.5)$$

where the network has one output for each of the N_c classes, N_p is the number of patterns, d is the desired or target output, y is the predicted output, and x is the class of pattern k .

When using prior scaling as defined in this section, the individual s'_x values can be large for classes with low prior probability. This may lead to the requirement of decreasing the learning rate in order to prevent the relatively large weight updates interfering with the gradient descent process. Comparing the use of prior scaling and not using prior scaling then becomes problematic because the optimal learning rate is different for each case. An alternative is to normalize the s'_x values so that the maximum is 1. Another possibility is to present patterns repeatedly to the network instead of scaling weight updates, i.e. for a class with a scaling factor of 2 each pattern would be presented twice. This would have the advantage of reducing the range of weight updates in terms of magnitude, e.g. an update of magnitude x might be repeated twice rather than using a single update of magnitude $2x$. This may allow the use of a higher learning rate, and therefore reduce the number of epochs required. However, a disadvantage of repeating patterns is that the effective training set would be larger, resulting in longer training times for the same number of epochs. Such a technique could be done probabilistically, and this is the subject of the next technique.

14.2.2 Probabilistic Sampling

Yaeger et al. [33] (chapter 13) have proposed a method called *frequency balancing* which is similar to the prior scaling method above. In frequency balancing, Yaeger et al. use all training samples in random order for each training epoch and allow each sample to be presented to the network a random number of times, which may be zero or more and is computed probabilistically. A balancing factor is included, which is analogous to the scaling factor above (c_s).

We introduce a very similar method here called *probabilistic sampling* whereby training patterns are chosen randomly in the following manner: the class is chosen randomly with the probability of choosing each class x , being $(1 - c_s)p_x + \frac{c_s}{N_c}$. A training sample is then chosen randomly from among all training samples for the chosen class.

14.2.3 Post Scaling

Instead of scaling weight updates or altering the effective class frequencies, it is possible to train the network as usual and then scale the outputs of the network after training. For example, the network could be trained as usual and then the outputs scaled according to the prior probabilities in a similar fashion to the prior scaling method (using equation 14.3 or 14.4). Experiments with this

technique alone show that it is not always as successful as prior scaling of the weight updates. This may be because the estimation of the lower frequency classes can be less accurate than that of the higher frequency classes [24] (the deviations of the network outputs from the true values in regions with a higher number of data points influence the squared error cost function more than the deviations in regions with a lower number of points [23]).

The post scaling technique introduced here can also be used to optimize a given criterion, e.g. the outputs may be scaled so that the probability of predicting each class matches the prior probabilities in the training set as closely as possible. Post scaling to minimize a different criterion is demonstrated in the results section. For the results in this chapter, the minimization is performed using a simple hill-climbing algorithm which adjusts a scaling factor associated with each of the outputs of the network.

14.2.4 Equalizing Class Membership

A simple method for alleviating difficulty with unequal prior class probabilities is to adjust (e.g. equalize) the number of patterns in each class, either by subsampling [24] (removing patterns from higher frequency classes), or by duplication (of patterns in lower frequency classes)⁴. For subsampling, patterns can be removed randomly, or heuristics may be used to remove patterns in regions of low ambiguity. Subsampling involves a loss of information which can be detrimental. Duplication involves a larger dataset and longer training times for the same number of training epochs (the convergence time may be longer or shorter).

14.3 Experimental Results

Results on an ECG classification problem are reported in this section after discussing the use of alternative performance measures. Results on a simple artificial problem are also included in the explanation section.

14.3.1 Performance Measures

When the interclass prior probabilities of the classes vary significantly, then the overall classification error may not be the most appropriate performance criterion. For example, a model may always predict the most common class and still provide relatively high performance. Statistics such as the Sensitivity, Positive Predictivity, and False Positive Rate can provide more meaningful results [1]. These are defined on a class by class basis as follows:

The **Sensitivity** of a class is the proportion of events labeled as that class which are correctly detected. For the two class confusion matrix shown in table 14.1 the sensitivity of class 1 is $\frac{c_{11}}{c_{11}+c_{12}}$.

⁴ The heuristic of adding noise during training [22] could be useful here as with the other techniques in this chapter.

The **Positive Predictivity** of a class is the proportion of events which were predicted to be the class and were labeled as that class. For the two class confusion matrix shown in table 14.1 the positive predictivity of class 1 is $\frac{c_{11}}{c_{11}+c_{21}}$.

The **False Positive Rate** of a class is the proportion of all patterns for other classes which were incorrectly classified as that class. For the two class confusion matrix shown in table 14.1 the false positive rate of class 1 is $\frac{c_{21}}{c_{11}+c_{21}}$.

Table 14.1. A sample confusion matrix which is used to illustrate sensitivity, positive predictivity, and false positive rate. Rows correspond to the desired classes and columns correspond to the predicted classes.

Class	1	2
1	c_{11}	c_{12}
2	c_{21}	c_{22}

No single performance criterion can be labeled as the best for comparing algorithms or models because the best criterion to use is problem dependent. Here, we take the sensitivity as defined above, and create a single performance measure, the mean squared sensitivity error (MSSE). We define the MSSE as follows:

Definition 2

$$MSSE = \frac{1}{N_c} \sum_{i=1}^{N_c} (1 - S_i)^2 \quad (14.6)$$

where N_c = the number of classes and S_i = sensitivity of class i as defined earlier.

Sensitivities range from 0 (no examples of the class correctly classified) to 1 (all examples correctly classified). Thus, a lower MSSE corresponds to better performance. We choose this criterion because each class is given equal importance and the square causes lower individual sensitivities to be penalized more (e.g. for a two class problem, class sensitivities of 100% and 0% produce a higher MSSE than sensitivities of 50% and 50%). Note that this is only one possible criterion, and other criterion could be used in order to reflect different requirements, e.g. specific misclassification costs for each class. The post scaling heuristic can be used with any criterion (and doing so may be simpler than reformulating the neural network training algorithm for the new criterion).

14.3.2 ECG Classification Problem

This section presents results using the beforementioned techniques on an ECG classification problem. The database used is the MIT-BIH Arrhythmia database [21] – a common publicly available ECG database which contains a large number of ECG records that have been carefully annotated by experts. Detection of

the following four beat types is considered: Normal (N), Premature Ventricular Contraction (PVC), Supraventricular Contraction (S), and Fusion (F) [21], i.e. there are four output classes. The four classes are denoted 1 (N), 2 (PVC), 3 (S), and 4 (F). An autoregressive model is calculated for a window of 200 samples centered over the peak of the *R*-wave of each beat. The inputs are the polar coordinates of each pole in the *z*-plane, i.e. frequency changes are reflected in the angular variation of the poles and damping is reflected in the magnitude variations. The model order was four corresponding to eight input variables. The prior probability of the classes (according to the training data) is (0.737, 0.191, 0.0529, 0.0196) corresponding to beat types (N, PVC, S, F).

MLPs with 20 hidden units were trained with stochastic backpropagation (update after each pattern) using an initial learning rate of 0.02 which was linearly reduced to zero over the training period of 500,000 updates. We used 5,000 points in each of the training, validation and test sets. The validation set was used for early stopping. The following algorithms were used – a) prior scaling with the degree of scaling, c_s , varied from 0 to 1, b) probabilistic sampling with the degree of scaling, c_s , varied from 0 to 1, c) as per a) and b) with the addition of post scaling, and d) equalizing the number of cases in each class by removing cases in more common classes. The post scaling attempted to minimize the MSSE on the training set⁵. 10 trials were performed for each case.

The median test set MSSE for d) was 0.195. The results for probabilistic sampling and probabilistic sampling plus post scaling are shown with box-whiskers plots⁶ in figure 14.1. For probabilistic sampling, the best scaling results correspond to a degree of scaling in between no scaling and scaling according to the prior probabilities ($c_s \approx 0.8$). When c_s is larger, the sensitivity of class 1 drops significantly and results in higher false positive rates for the other classes. When c_s is lower, the sensitivity of classes 3 and 4 drops significantly. It can be seen that the addition of post scaling appears to almost always improve performance for this problem. The optimal degree of scaling, $c_s \approx 0.8$, is difficult to determine *a priori*. However, it can be seen that the addition of post scaling makes the selection of c_s far less critical ($c_s = 0.3$ to $c_s = 1.0$ result in similar performance). Figure 14.2 shows confusion matrices (in graphical form). Without

⁵ 400 steps were used for the hill climbing algorithm where each step corresponded to either multiplying or dividing an individual output scale factor by a constant which was reduced linearly over time from 1.5 to 1. The time taken was short compared to the overall training time.

⁶ The distribution of results is often not Gaussian and alternative means of presenting results other than the mean and standard deviation can be more informative. Box-whiskers plots show the interquartile range (IQR) with a box and the median as a bar across the box. The whiskers extend from the ends of the box to the minimum and maximum values. The median and the IQR are simple statistics which are not as sensitive to outliers as the mean and the standard deviation [31]. The median is the value in the middle when arranging the distribution in order from the smallest to the largest value. If the data is divided into two equal groups about the median, then the IQR is the difference between the medians of these groups. The IQR contains 50% of the points.

scaling ($c_s = 0$), it can be seen that classes 3 & 4 have low sensitivity. With scaling using $c_s = 1$ all classes are now recognized, however the sensitivity of class 1 is worse and the false positive rate of classes 3 & 4 is significantly worse.

The results for prior scaling and prior scaling combined with post scaling were very similar but slightly worse than the results with probabilistic sampling. The prior scaling results are not plotted in order to make the graph easier to follow, however the qualitative results are as follows: for low c_s , prior scaling and probabilistic sampling perform very similarly. However, for high c_s , probabilistic sampling has a clear advantage for this problem. This is perhaps just as expected – the relatively high variation in prior class probabilities leads to a high variation in weight update magnitudes across the classes when using high c_s . Results for all methods can be seen in table 14.2.

Table 14.2. Results for the various methods. We show the average results for the best selection of c_s and also an average across all selections of c_s . Note that selection of the optimal value of c_s is less critical when using post scaling in addition to either the prior scaling or probabilistic sampling methods.

Method	Prior Scaling	Prior Scaling + Post Scaling	Probabilistic Sampling	Probabilistic Sampling + Post Scaling	Equalizing Membership
Average MSSE (for best c_s)	0.10 ($c_s = 0.8$)	0.096 ($c_s = 0.6$)	0.099 ($c_s = 0.8$)	0.089 ($c_s = 0.3$)	0.195
Average MSSE (over all c_s)	0.19	0.10	0.18	0.099	0.195

14.4 Explanation

This section discusses why the techniques presented can be useful, limitations of the techniques, and how they relate to the theoretical result that MLPs approximate Bayesian *a posteriori* probabilities under certain conditions.

14.4.1 Convergence and Representation Issues

We first list four possible situations:

1. The proofs regarding estimation of Bayesian *a posteriori* probabilities assume networks with an infinite number of hidden nodes in order to obtain accurate approximation. For a given problem, it can be seen that a network which is too small will be unable to estimate the probabilities accurately due to limited resources.

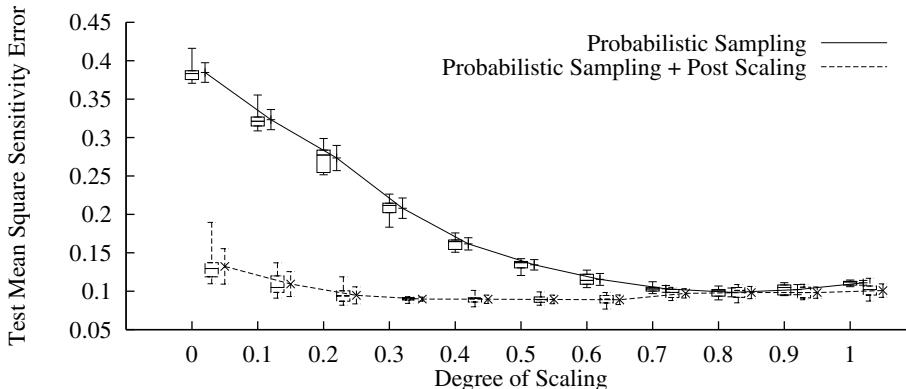


Fig. 14.1. Box-whiskers plots (on the left in each case) along with the usual mean plus and minus one standard deviation plots (on the right in each case) showing the test set MSSE for probabilistic sampling and for probabilistic sampling plus post scaling. Each result is derived from 10 trials with different starting conditions. The probabilistic sampling plus post scaling case is offset by 0.03 to aid viewing. It can be seen that the selection of the scaling degree for the best performance is not as critical when using the combination of probabilistic sampling and post scaling.

2. Training an MLP is NP-complete in general and it is well known that practical training algorithms used for MLPs often result in sub-optimal solutions (e.g. due to local minima). Often, a result of attaining a sub-optimal solution is that not all of the network resources are efficiently used. Experiments with a controlled task have indicated that the sub-optimal solutions often have smaller weights on average [17].
3. Weight decay [16] or weight elimination [30] are often used in MLP training and aim to minimize a cost function which penalizes large weights. These techniques tend to result in networks with smaller weights.
4. A commonly recommended technique with MLP classification is to set the training targets away from the bounds of the activation function (e.g. (-0.8, 0.8) instead of (-1, 1) for the tanh activation function) [14].

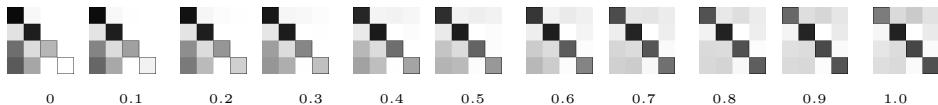


Fig. 14.2. Confusion matrices for the test set as the degree of prior scaling, c_s , is varied from 0 (left) to 1 (right). The columns correspond to the predicted classes and the rows correspond to the desired classes. The classes are (left to right and top to bottom) N, PVC, S, F. For each desired class, the predicted classes are shaded in proportion to the number of examples which are labeled as the desired class. White indicates no predictions. A general trend can be observed where classes S & F are recognized as normal when $c_s = 0$, and progressively more of the normal class examples are recognized as classes PVC, S, & F as c_s approaches 1.

These four situations can all lead to a bias towards smaller weights, or “smoother” models⁷. The possibility of such a bias is not taken into account by the proofs regarding posterior probabilities, i.e. the difference between theory and practice may, in part, be explained by violation of the assumption that sufficient convergence is obtained.

When a network is biased towards a “smoother” solution, and accurate fitting of the optimal function is not possible, the result may be a tendency to “ignore” lower frequency classes⁸, e.g. if a network has the choice of fitting either a high frequency class or a low frequency class then it can provide a lower MSE by fitting the high frequency class⁹. We demonstrate by example.

We generated artificial training data using the following distributions: class 1: $N(-5, 1, 2) + N(0, 1, 2) + N(5, 1, 2)$, class 2: $N(-2.5, 0.25, 0.5) + N(2.5, 0.25, 0.5)$, where $N(\mu, \sigma, x)$ is a normal distribution with mean μ , standard deviation σ , and is truncated to lie within $(\mu - x, \mu + x)$. We generated 500 training and test examples from these distributions with the probability of selection for classes (1,2) being (0.9,0.1), i.e. the training and test sets have nine times as many samples of class 1 as they do of class 2. Note that there is no overlap between the classes. Figure 14.3 shows typical output probability plots for training an MLP with 10 hidden nodes¹⁰ with and without probabilistic sampling. 10 trials were performed in each case with very similar results (see table 14.3). It can be seen that the network “ignores” class two without the use of probabilistic sampling.

It should be noted that using conjugate gradient training for this simple problem results in relatively accurate estimation of both classes with standard training (alternate parameters with backpropagation may also be successful). Rather than arguing for either backpropagation or conjugate gradient here (neither training algorithm is expected to always find a global minimum in general), we simply note that our experience and the experience of others [7, 18, 19, 27] suggests that conjugate gradient is not superior for many problems – i.e. backpropagation works better on one class of problems and conjugate gradient works better on another class. Conjugate gradient resulted in significantly worse performance when tested on the ECG problem. It should be noted that there are

⁷ In general, smaller weights correspond to smoother functions, however this is not always true. For example, this is not the case when fitting the function $\text{sech}(x)$ using two tanh sigmoids [8] (because $\text{sech}(x) = \lim_{d \rightarrow 0} (\tanh(x+d) - \tanh(x))/d$, i.e. the weights become indefinitely large as the approximation improves).

⁸ In relation to the representational capacity (size of the network), Barnard and Botha [3] have observed that MLP networks have a tendency to guess higher probability classes when a network is too small to approximate the decision boundaries reasonably well.

⁹ Lyon and Yaeger [20] find that their frequency balancing technique reduces the effect of the prior class probabilities on the network and effectively forces the network to allocate more resources to the lower frequency classes.

¹⁰ 500,000 stochastic training updates with backpropagation, initial learning rate 0.02 reduced linearly to zero.

many options when implementing a conjugate gradient training algorithm and that poor performance may be attributed to the implementation used. We have used a modified implementation of the algorithm from Fletcher [9].

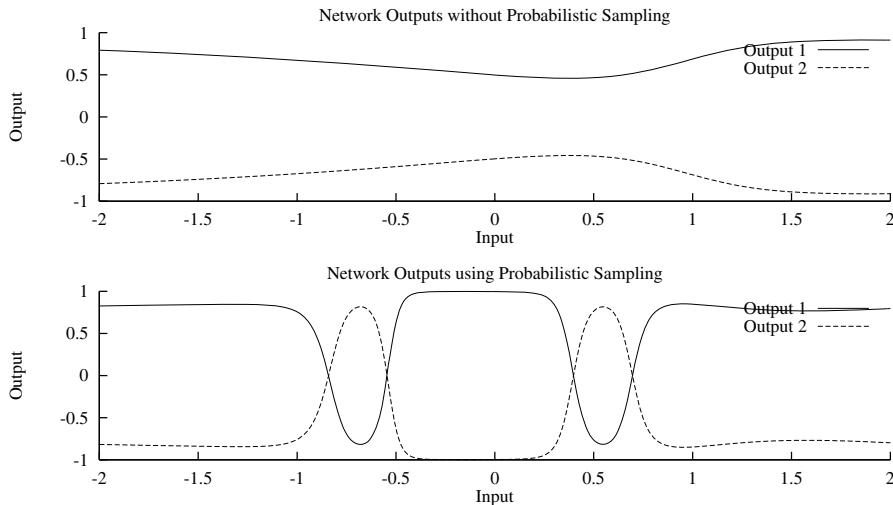


Fig. 14.3. Network outputs for the artificial problem with (below) and without (above) probabilistic sampling. It can be seen that the network “ignores” the lower frequency class without the use of probabilistic sampling. Note that the input has been normalized.

Table 14.3. Mean and standard deviation of the classification error for the artificial problem both with and without the use of probabilistic sampling

Classification Error	Mean	Standard Deviation
Standard Training	11.4	0.02
With Probabilistic Sampling	0.8	0.004

14.4.2 Overlapping Distributions

Consider figure 14.4. If classes 1 and 2 have distributions differing only by translation (c_1 and c'_2) then the decision threshold between these classes should be chosen at x_1 . Equal percentages of each of these classes will be classified as the other class. Now, if the distribution of class 2 is as shown (c_2) then the decision threshold between the classes should be chosen at x_2 . In this case, a higher percentage of class 2 will be classified as class 1 than the reverse. If it is desirable to maximize the class by class sensitivity then scaling such that the effective distribution of c_2 is c'_2 might be appropriate. Similarly, class 3 (c_3) will be “ignored” without any scaling.

Scaling on a class by class basis may be desired when i) the distribution of samples in the training set does not match the true distribution (e.g. it may be more expensive to collect samples of a particular class)¹¹, or ii) the distribution of the classes does not represent their relative importance, e.g. in a medical classification problem the cost of misclassifying a diseased case as normal may be much higher than the cost of classifying a normal case as a (possibly) diseased case [24]. The importance of each class may be independent of the class prior probabilities. Note that scaling such that lower frequency classes are made to be artificially more important can be useful when considering a higher level problem. For example, the training data from natural English words and phrases exhibit very non-uniform priors for different characters. Yaeger et al. [33] find that reducing the effect of these priors on the network using frequency balancing improves the performance of the higher level word recognition training.

Observations. a) There is no intrinsic problem if the distributions do not overlap. b) When distributions overlap, it is desirable to preprocess the data in a manner that results in reduced overlap. However, it is often not possible to obtain zero overlap (due to noise, for example).

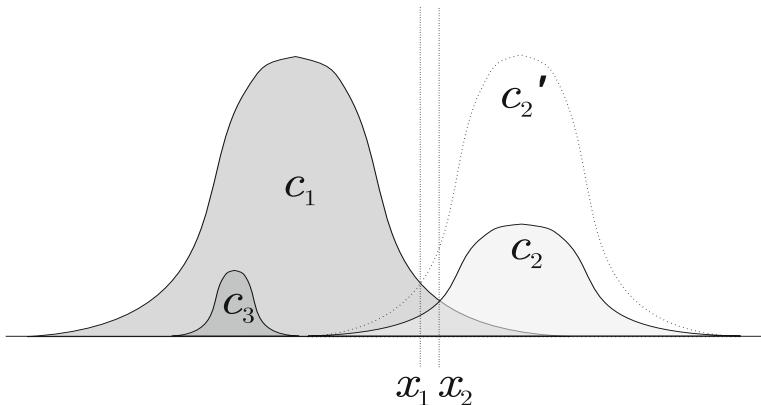


Fig. 14.4. Overlapping distributions

14.4.3 Limitations

We note a couple of limitations with the heuristics considered herein:

1. *Local issues.* The heuristics presented counteract biases in the network, training algorithm and/or training data. There is no reason for these biases to be constant throughout the input space, e.g. scaling may be helpful in one region but detrimental in another.

¹¹ It may be possible to obtain more accurate estimates of class probabilities using data that has class labels without input information. For example, word frequency information can be obtained from text databases and the frequency of various diseases can be obtained from health statistics [23].

2. *Nonlinear calibration.* There is no reason for the linear scaling heuristics used here to be optimal (in the sense that they best counteract the biases).

14.4.4 *A Posteriori* Proofs

Theoretically it is possible to show that the scaling techniques invalidate the *a posteriori* proofs – when performing scaling on a class by class basis the decision thresholds which are used to determine the winning class should be altered accordingly. This indicates another possible use of the prior scaling and probabilistic sampling techniques when the conditions given above do not exist. This use is related to the problem whereby lower frequency classes may be estimated less accurately than higher frequency classes (see section 14.2.3) – training may be performed with the heuristically altered problem (e.g. so that the class frequencies are effectively equal) and the outputs or decision thresholds can be altered accordingly.

14.5 Conclusions

In practice, training issues or characteristics of a given classification problem can mean that scaling the predicted class probabilities may improve performance in terms of overall classification error and/or in terms of an alternative criterion. We introduced algorithms which a) scale weight updates on a class by class basis according to the prior class probabilities, b) alter class frequencies probabilistically (very similar to the frequency balancing technique of Yaeger et al. [33]), and c) scale outputs after training in order to maximize a given performance criterion. For an electrocardiogram (ECG) classification problem, we found that the prior scaling, probabilistic sampling, and post scaling techniques provided better performance in comparison to a) no heuristics, and b) subsampling in order to equalize the number of cases in each class. The best performance for prior scaling and probabilistic sampling was obtained with a degree of scaling in between no scaling and scaling according to the prior probabilities. The optimal degree was difficult to determine *a priori*. However, it was found that using prior scaling or probabilistic sampling in combination with post scaling made the selection of the optimal degree far less critical.

References

- [1] AAMI. Testing and reporting performance results of ventricular Arrhythmia detection algorithms. In: Association for the Advancement of Medical Instrumentation, ECAR 1987, Arlington, VA (1987)
- [2] Anand, R., Mehrotra, K.G., Mohan, C.K., Ranka, S.: An improved algorithm for neural network classification of imbalanced training sets. IEEE Transactions on Neural Networks 4(6), 962–969 (1993)
- [3] Barnard, E., Botha, E.C.: Back-propagation uses prior information efficiently. IEEE Transactions on Neural Networks 4(5), 794–802 (1993)

- [4] Barnard, E., Casasent, D.: A comparison between criterion functions for linear classifiers, with an application to neural nets. *IEEE Transactions on Systems, Man, and Cybernetics* 19(5), 1030–1041 (1989)
- [5] Barnard, E., Cole, R.A., Hou, L.: Location and classification of plosive constants using expert knowledge and neural-net classifiers. *Journal of the Acoustical Society of America* 84(suppl. 1), S60 (1988)
- [6] Bourlard, H.A., Morgan, N.: Links between Markov models and multilayer perceptrons. In: Touretzky, D.S. (ed.) *Advances in Neural Information Processing Systems*, vol. 1, pp. 502–510. Morgan Kaufmann, San Mateo (1989)
- [7] Bourlard, H.A., Morgan, N.: *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer Academic Publishers, Boston (1994)
- [8] Scott Cardell, N., Joerding, W., Li, Y.: Why some feedforward networks cannot learn some polynomials. *Neural Computation* 6(4), 761–766 (1994)
- [9] Fletcher, R.: *Practical Methods of Optimization*, Second Edition, 2nd edn. John Wiley & Sons (1987)
- [10] Geman, S., Bienenstock, E., Doursat, R.: Neural networks and the bias/variance dilemma. *Neural Computation* 4(1), 1–58 (1992)
- [11] Gish, H.: A probabilistic approach to the understanding and training of neural network classifiers. In: *Proceedings of the IEEE Conference on Acoustics, Speech and Signal Processing*, pp. 1361–1364. IEEE Press (1990)
- [12] Hampshire, J.B., Pearlmutter, B.: Equivalence proofs for multilayer perceptron classifiers and the Bayesian discriminant function. In: Touretzky, D.S., Elman, J.L., Sejnowski, T.J., Hinton, G.E. (eds.) *Proceedings of the 1990 Connectionist Models Summer School*, Morgan Kaufmann, San Mateo (1990)
- [13] Hampshire, J.B., Waibel, A.H.: A novel objective function for improved phoneme recognition using time delay neural networks. In: *International Joint Conference on Neural Networks*, Washington, DC, pp. 235–241 (June 1989)
- [14] Haykin, S.: *Neural Networks, A Comprehensive Foundation*. Macmillan, New York (1994)
- [15] Kanaya, F., Miyake, S.: Bayes statistical behavior and valid generalization of pattern classifying neural networks. *IEEE Transactions on Neural Networks* 2(1), 471 (1991)
- [16] Krogh, A., Hertz, J.A.: A simple weight decay can improve generalization. In: Moody, J.E., Hanson, S.J., Lippmann, R.P. (eds.) *Advances in Neural Information Processing Systems*, vol. 4, pp. 950–957. Morgan Kaufmann, San Mateo (1992)
- [17] Lawrence, S., Lee Giles, C., Tsui, A.C.: Lessons in neural network training: Overfitting be harder than expected. In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI 1997*, pp. 540–545. AAAI Press, Menlo Park (1997)
- [18] LeCun, Y.: Efficient learning and second order methods. In: *Tutorial Presented at Neural Information Processing Systems*, vol. 5 (1993)
- [19] LeCun, Y., Bengio, Y.: Pattern recognition. In: Arbib, M.A. (ed.) *The Handbook of Brain Theory and Neural Networks*, pp. 711–715. MIT Press (1995)
- [20] Lyon, R., Yaeger, L.: On-line hand-printing recognition with neural networks. In: *Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, Lausanne, Switzerland. IEEE Computer Society Press (1996)
- [21] MIT-BIH. MIT-BIH Arrhythmia database directory. Technical Report BMEC TR010 (Revised), Massachusetts Institute of Technology and Beth Israel Hospital (1988)
- [22] Murray, A.F., Edwards, P.J.: Enhanced MLP performance and fault tolerance resulting from synaptic weight noise during training. *IEEE Transactions on Neural Networks* 5(5), 792–802 (1994)

- [23] Richard, M.D., Lippmann, R.P.: Neural network classifiers estimate Bayesian *a posteriori* probabilities. *Neural Computation* 3(4), 461–483 (1991)
- [24] Ripley, B.D.: Pattern Recognition and Neural Networks. Cambridge University Press, Cambridge (1996)
- [25] Rojas, R.: A short proof of the posterior probability property of classifier neural networks. *Neural Computation* 8, 41–43 (1996)
- [26] Ruck, D.W., Rogers, S.K., Kabrisky, K., Oxley, M.E., Suter, B.W.: The multilayer perceptron as an approximation to an optimal Bayes estimator. *IEEE Transactions on Neural Networks* 1(4), 296–298 (1990)
- [27] Schiffman, W., Joost, M., Werner, R.: Optimization of the backpropagation algorithm for training multilayer perceptrons. Technical report, University of Koblenz (1994)
- [28] Shoemaker, P.A.: A note on least-squares learning procedures and classification by neural network models. *IEEE Transactions on Neural Networks* 2(1), 158–160 (1991)
- [29] Wan, E.: Neural network classification: A Bayesian interpretation. *IEEE Transactions on Neural Networks* 1(4), 303–305 (1990)
- [30] Weigend, A.S., Rumelhart, D.E., Huberman, B.A.: Generalization by weight-elimination with application to forecasting. In: Lippmann, R.P., Moody, J.E., Touretzky, D.S. (eds.) Advances in Neural Information Processing Systems, vol. 3, pp. 875–882. Morgan Kaufmann, San Mateo (1991)
- [31] Weiss, N.A., Hassett, M.J.: Introductory Statistics, 2nd edn. Addison-Wesley, Reading (1987)
- [32] White, H.: Learning in artificial neural networks: A statistical perspective. *Neural Computation* 1(4), 425–464 (1989)
- [33] Yaeger, L., Lyon, R., Webb, B.: Effective training of a neural network character classifier for word recognition. In: Mozer, M.C., Jordan, M.I., Petsche, T. (eds.) Advances in Neural Information Processing Systems, vol. 9. MIT Press, Cambridge (1997)

Applying Divide and Conquer to Large Scale Pattern Recognition Tasks^{*}

Jürgen Fritsch¹ and Michael Finke²

¹ Interactive Systems Laboratories
 University of Karlsruhe
 Am Fasanengarten 5
 76128 Karlsruhe, Germany
 fritsch@ira.uka.de

² Interactive Systems Laboratories
 Carnegie Mellon University
 5000 Forbes Avenue
 Pittsburgh, PA 15213, USA
 finkem@cs.cmu.edu
<http://www.cs.cmu.edu/~finkem/>

Abstract. Rather than presenting a specific trick, this paper aims at providing a methodology for large scale, real-world classification tasks involving thousands of classes and millions of training patterns. Such problems arise in speech recognition, handwriting recognition and speaker or writer identification, just to name a few. Given the typically very large number of classes to be distinguished, many approaches focus on parametric methods to independently estimate class conditional likelihoods. In contrast, we demonstrate how the principles of modularity and hierarchy can be applied to directly estimate posterior class probabilities in a connectionist framework. Apart from offering better discrimination capability, we argue that a hierarchical classification scheme is crucial in tackling the above mentioned problems. Furthermore, we discuss training issues that have to be addressed when an almost infinite amount of training data is available.

15.1 Introduction

The majority of contributions in the field of neural computation deal with relatively small datasets and, in case of classification tasks, with a relatively small number of classes to be distinguished. Representatives of such problems include the UCI machine learning database [16] and the Proben [20] benchmark set for learning algorithms. Research concentrates on aspects such as missing data, model selection, regularization, overfitting vs. generalization and the bias/variance trade-off. Over the years, many methods and 'tricks' have been developed to optimally learn and generalize when only a limited amount of data is available.

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

On the other hand, many problems in human computer interaction (HCI) such as speech and handwriting recognition, lipreading and speaker and writer identification require comparably large training databases and also often exhibit a large number of classes to be discriminated, such as (context-dependent) phones, letters and individual speakers or writers. For example, in state-of-the-art large vocabulary continuous speech recognition, we are typically faced with an inventory of several thousand basic acoustic units and training databases consisting of several millions of preprocessed speech patterns. There is only a limited amount of publications available on the sometimes very different problems concerning the choice of learning machines and training algorithms for such tasks and datasets.

This article addresses exactly the latter kind of learning tasks and provides a principled approach to large scale classification problems, exemplifying it on the problem of connectionist speech recognition. Our approach is grounded on the powerful *divide and conquer* paradigm that traditionally has always been applied to problems of rather large size. We argue that a hierarchical approach that modularizes classification tasks is crucial in applying statistical estimators such as artificial neural networks. In that respect, this paper presents not just a single 'trick of the trade', it offers a methodology for large scale classification tasks. Such tasks have traditionally been addressed by building generative models rather than focusing on the prediction of posteriors without making strong assumptions on the distribution of the input.

The remainder of the paper is organized as follows. Section 2 presents the general approach to soft hierarchical classification. Section 3 then discusses methods to design the topology of hierarchical classifiers - a task that is of increasing importance when dealing with large numbers of classes. Finally, section 4 demonstrates in detail the application of hierarchical classification to connectionist statistical speech recognition. Section 5 concludes this paper with a summary.

15.2 Hierarchical Classification

Consider the task of classifying patterns \mathbf{x} as belonging to one of N classes ω_k . Given that we have access to the class conditional probability densities $p(\mathbf{x}|\omega_k)$, Bayes theory states that the optimal decision should be based on the a-posteriori probabilities

$$p(\omega_k|\mathbf{x}) = \frac{p(\mathbf{x}|\omega_k)p(\omega_k)}{\sum_i p(\mathbf{x}|\omega_i)p(\omega_i)}.$$

Given that equal risks are associated with all possible misclassifications, the optimal decision is to choose the class with maximum a-posteriori probability given a specific pattern \mathbf{x} . Two distinct approaches have to be considered when applying Bayes theory to a learning from examples task with generally unknown distributions. In the first approach, one tries to estimate class-conditional likelihoods $p(\mathbf{x}|\omega_k)$ and prior probabilities $p(\omega_k)$ from a labeled dataset which are then used to calculate posterior probabilities according to Bayes rule. In principle, this approach can be applied to tasks with an arbitrary large number of classes

since the class-conditional likelihoods can be estimated independently. However, such an approach focuses on the modeling of the class-conditional densities. For classification accuracy however, it is more important to model class boundaries.

The second approach accommodates this perspective by directly estimating posterior class probabilities from datasets. It was shown (e. g. [6]) that a large class of artificial neural networks such as multi-layer perceptrons and recurrent neural networks can be trained to approximate posterior class probabilities. The degree of accuracy of the approximation however depends on many factors, among them the plasticity of the network. Comparing the two approaches, the discriminative power of methods that estimate posterior probabilities directly is generally higher, resulting in better classification accuracy especially when the class-conditional distributions are very complex. This fact (among others) explains the success and popularity of neural network classifiers on many learning from examples tasks.

However, when the number of classes to be distinguished increases to say several thousand, neural network estimators of posterior probabilities fail to provide good approximations mainly because of two reasons: First, real-world problems involving such a large number of classes often exhibit an extremely non-uniform distribution of priors, see chapter 14. Many learning algorithms for neural networks (especially stochastic on-line gradient descent) have difficulties with non-uniformly distributed classes. Particularly the distribution of posteriors of infrequent classes tend to be approximated poorly. Second, and more important, one of the prerequisites for training neural networks to estimate posteriors, the 1-out-of- N coding of training targets, implies that the number of output neurons matches the number of classes. It is unfeasible to train a neural network with thousands of output neurons. Also, with increasing number of classes, the complexity of the optimum discriminant functions also increases and the potential for conflicts between classes grows. Thus, from our point of view, typical monolithic neural network classifiers are not applicable because of their limitation to tasks with relatively few classes.

15.2.1 Decomposition of Posterior Probabilities

Applying the principle of *divide and conquer*, we can break down the task of discriminating between thousands of classes into a hierarchical structure of many smaller classification tasks of controlled size. This idea underlies the approaches to decision tree architectures [5, 21, 23]. Decision trees classify input patterns by asking categorical questions at each internal node. Depending on the answer to these questions a single path is followed to one of the child nodes and the process repeats until a leaf node is reached and a (winner) class label is emitted. Therefore, decision tree classifiers can only supply us with hard decisions. No information about the confusability of a specific input pattern is given to us. Rather, we are often interested in the posterior class probabilities because we wish to have a measure of the ambiguity of a decision. Furthermore, we are sometimes required to feed a measure of the degree of membership for all

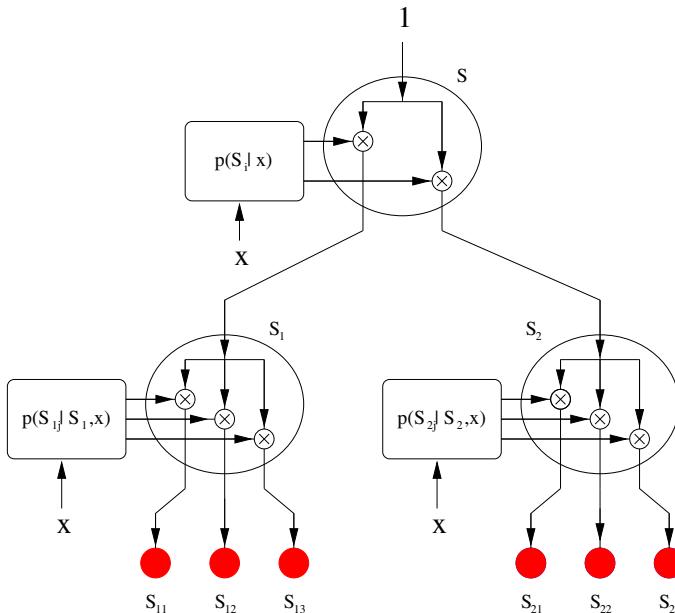


Fig. 15.1. Hierarchical decomposition of posteriors

potential classes into a superior decision making process. As we will see in section 4, statistical speech recognition is a typical example for the latter scenario.

Adhering to the *divide and conquer* approach but generalizing the decision tree framework, the statistical method of factoring posteriors can be applied to design *soft* classification trees [24, 25]. For now, we assume, that optimal posterior probabilities are available. Let S be a (possibly large) set of classes ω_k to be discriminated. Consider we have a method at our disposition which gives us a partitioning of S into M disjoint and non-empty subsets S_i such that members of S_i are almost never confused with members of S_j ($\forall j \neq i$). A particular class ω_k will now be a member of S and exactly one of the subsets S_i . Therefore, we can rewrite the posterior probability of class ω_k as a joint probability of the class and the corresponding subset S_i and factor it according to

$$\begin{aligned} p(\omega_k | \mathbf{x}) &= p(\omega_k, S_i | \mathbf{x}) \quad \text{with} \quad \omega_k \in S_i \\ &= p(S_i | \mathbf{x}) p(\omega_k | S_i, \mathbf{x}). \end{aligned}$$

Thus, the global task of discriminating between all the classes in S has been converted into (1) discriminating between subsets S_i and (2) independently discriminating between the classes ω_k remaining within each of the subsets S_i . Recursively repeating this process yields a hierarchical tree-organized structure (Fig. 15.1).

Note, that the number of subclasses S_i of each node does not need to be constant throughout the classifier tree and might be subject to optimization during the tree design phase. In order to compute the posterior probability for a specific

class, we have to follow the path from root node to the leaf corresponding to the class in question, multiplying all the conditional posteriors along the way. Both the design of the tree structure (*divide*) and the estimation and multiplication (*conquer*) of conditional posteriors at each node are important aspects in this architecture, that have to be considered thoroughly because in practice, only approximations to the conditional posteriors are available.

15.2.2 Hierarchical Interpretation

The presented architecture can be interpreted as a probability mass distribution device. At the root node, an initial probability mass of 1 is fed into the architecture. At each node, the incoming probability mass is multiplied by the respective conditional posterior probabilities and fed into the child nodes. Eventually, the probability mass is distributed among all the leaves (classes) rendering their posterior probabilities. In contrast, classifier trees are mostly used as hard-switching devices, where only a single path from root node to one of the leaves is taken.

A hierarchical decomposition of posterior probabilities through a soft classification tree offers several advantages. If one of the nodes in the tree, for example the root node fails to provide good estimates of conditional posteriors, a hard decision tree will produce many classification errors. In a soft classification tree, such shortcomings will influence the decision process less dramatically. Also, recovery from errors is often possible through a superior decision process.

Another aspect of soft classification trees that can be exploited for various purposes is the sum-to-unity property observable in any horizontal cross-section at any level of the tree. The tree can be cut off at a certain level and still be used as a soft classification tree that computes posterior class probabilities. This is equivalent to creating a new (smaller) set of classes by clustering and merging the original classes according to the tree topology. In general, the resulting classification task will be easier to solve than the original one.

Related to the sum-to-unity property of cross-sections is that the partial posteriors computed on a path from the root node to a leaf are decreasing monotonically. This in turn allows to close paths whenever a suitable threshold is reached, pruning whole subtrees with classes that would otherwise receive posteriors smaller than the threshold. This property yields the possibility to smoothly trade off classification accuracy against computational complexity. In the limit, when only a single path with highest conditional posterior is followed, the soft classification tree transmutes into a hard decision tree.

15.2.3 Estimation of Conditional Node Posteriors

Given a hierarchical decomposition of posterior class probabilities, it remains to instantiate the tree nodes with estimators for the required conditional posteriors. Conditioning a posterior on a subset of classes S_i can be accomplished by restricting the training set of the corresponding learning device to the patterns with a class label from S_i . According to this setting, the available training data in each node is distributed among all its child nodes according to the class partitioning.

While the root node receives all available training data, nodes further down the tree receive less data than their predecessors. On the other hand, specialization increases from root node to leaves. This fact has important consequences on learning speed and model selection when training whole hierarchies.

One of the important issues in hierarchical decompositions of posterior probabilities are the unavoidable inaccuracies of practical estimators for the conditional posteriors that have to be provided in each tree node. Neural networks can only be trained to *approximate* the true distribution of posterior class probabilities and the degree of accuracy depends on both the inherent difficulty of the task as given by the training set and the network structure and training schedule being used. Inaccurate approximations to the true distribution of posteriors hurt most in the upper layers of a classification tree - a fact that has to be taken into account by tree design procedures, which we will discuss next.

15.3 Classifier Tree Design

When it comes to the design of soft classifier trees, or equivalently to the design of hierarchical decompositions of class posteriors, the choice of algorithm depends mostly on the number of initial classes. We will first discuss optimal tree structures before we will turn to heuristic design algorithms necessary when dealing with the large number of classes that we have to deal with.

15.3.1 Optimality

The optimal soft classification tree for a given task and given type and structure of estimators for the conditional node posteriors is the one which results in minimum classification error in the Bayes setting. If all the node classifiers would compute the true conditional posteriors, the tree structure would have no influence on the classifier performance because any kind of factoring (through any kind of tree structure) yields an *exact* decomposition of the class posteriors. However, in practice, approximation errors of node classifiers render the choice of tree structure an important issue. For small numbers of classes, the optimal tree can in principle be found by exhaustively training and testing all possible partitionings for a particular node (starting with the root node) and choosing the one that gives the highest recognition accuracy. However, even if restricting the tree structure to binary branching nodes and balanced partitionings, the number K of partitionings that have to be examined at the root node

$$K = \frac{N!}{\left(\frac{N}{2}!\right)^2}$$

quickly brings this algorithm to its limits, even for a moderate number of classes N . Therefore, we have to consider heuristics to derive near optimal tree structures. For example, one valid possibility is to assume that the accuracy of achievable approximations to the true posteriors is related to the separability of the corresponding sets of classes.

15.3.2 Prior Knowledge

Following the above mentioned guideline, prior knowledge about the task in question can often be applied to hierarchically partition the global set of classes into reasonable subsets. The goal is to partition the remaining set of classes in a way that intuitively maximizes the separability of the subsets. For example, given a set of phones in a speech recognizer, a reasonable first partitioning would be to build subsets consisting of voiced and unvoiced phones. In larger speech recognition systems where we have to distinguish among multi-state context-dependent phones, prior knowledge such as state and context identity can be used as splitting criterion. In tasks such as speaker or writer identification, features such as gender or age are potential candidates for splitting criteria.

The advantage of such knowledge driven decompositions is a fast tree design phase which is a clear superiority of this approach when dealing with large numbers of classes. However, this method for the design of hierarchical classifiers is subjective and error prone. Two experts in a specific field might disagree strongly on what constitutes a reasonable hierarchy. Furthermore, it is not always the case that *reasonable* partitionings yield good separability of subsets. Expert knowledge can be misleading.

15.3.3 Confusion Matrices

In case the number of classes is small enough to allow the training of a single classifier, the design of a soft classifier tree can be based on the confusion matrix of the trained monolithic classifier. Indicating the confusability of each pair of classes, the confusion matrix yields relatively good measures of the separability of pairs of classes. This information can be exploited for designing a tree structure using a clustering algorithm. For instance, we can define the following (symmetric) distance measure between two disjunct sets of classes S_k and S_l

$$d(S_k, S_l) = - \sum_{\omega_i \in S_k} \sum_{\omega_j \in S_l} C(\omega_i, \omega_j | \mathcal{T}) + C(\omega_j, \omega_i | \mathcal{T})$$

where $C(\omega_i, \omega_j | \mathcal{T})$ denotes the number of times class ω_i is confused with class ω_j as measured on a set of labeled patterns \mathcal{T} . The distance $d(S_k, S_l)$ can now be used as a replacement for the usual Euclidean distance measure in a standard bottom-up clustering algorithm. Unfortunately, once the number of classes increases to several thousand, training of a monolithic classifier becomes increasingly difficult.

15.3.4 Agglomerative Clustering

Assuming that separability of classes correlates with approximation accuracy of estimators for the posterior class probabilities, we can go further and assume that separability of classes can be measured by a suitable distance between the class conditional distributions in feature space. We already introduced such a distance

measure in form of the elements of a class confusion matrix. Other, computationally less expensive distance measures would be the Euclidean distance between class means or the Mahalanobis distance between the classes second order statistics. Irrespective of the chosen distance measure, the goal always is to group the set of classes in a way that results in maximum inter- and minimum intra-group distances. Solutions to this problem are known as agglomerative clustering algorithms and a large pool of variations of the basic algorithm is available in the literature [7].

15.4 Application to Speech Recognition

In this section, we will demonstrate the main ideas and benefits of the hierarchical classifier approach on the task of large vocabulary continuous speech recognition (LVCSR). More specifically, we will focus on acoustic modeling for statistical speech recognition using hidden Markov models (HMM) [27]. To give an impression of the complexity of such a task: training databases typically consist of tens of millions of speech patterns, the number of acoustic classes being distinguished ranges from ca. 50 (monophones) to over 20000 (context-dependent polyphones).

15.4.1 Statistical Speech Recognition

The basic statistical entity in HMM based speech recognition is the posterior probability of word sequences W_1, \dots, W_N given a sequence of acoustic observations $\mathbf{X}_1, \dots, \mathbf{X}_M$ and a set of model parameters Θ

$$P(W_1, \dots, W_N | \mathbf{X}_1, \dots, \mathbf{X}_M, \Theta)$$

During training, we are seeking parameters Θ that maximize this probability on the training data

$$\hat{\Theta} = \arg \max_{\Theta} \prod_{t=1}^T P(W_1, \dots, W_{N(t)} | \mathbf{X}_1, \dots, \mathbf{X}_{M(t)}, \Theta)$$

and during recognition, we want to find the sequence of words that maximizes this probability for a given acoustic observation and fixed model parameters Θ

$$\hat{W}_1, \dots, \hat{W}_N = \arg \max_{W_1, \dots, W_N} P(W_1, \dots, W_N | \mathbf{X}_1, \dots, \mathbf{X}_M, \Theta)$$

In order to simplify the process of maximizing the posterior probability of word sequences, Bayes rule is usually applied

$$P(W_1, \dots, W_N | \mathbf{X}_1, \dots, \mathbf{X}_M) = \frac{P(\mathbf{X}_1, \dots, \mathbf{X}_M | W_1, \dots, W_N) P(W_1, \dots, W_N)}{P(\mathbf{X}_1, \dots, \mathbf{X}_M)}$$

This rule separates the estimation process into the so called *acoustic model (AM)* consisting of terms that depend on the acoustic observations $\mathbf{X}_1, \dots, \mathbf{X}_M$ and

the *language model (LM)* consisting of terms that depend only on the sequence of words W_1, \dots, W_N . In this paper we will focus on acoustic modeling using connectionist estimators as a typical example of a task involving the discrimination of thousands of classes. For a review on other important aspects of LVCSR such as pronunciation modeling, language modeling and decoding algorithms we refer the reader to [27].

The task of acoustic modeling (ignoring the denominator) is to estimate parameters Θ^{AM} which maximize

$$P(\mathbf{X}_1, \dots, \mathbf{X}_M | W_1, \dots, W_N, \Theta^{\text{AM}}).$$

Words W_i are modeled as sequences (or graphs) of phone models. The mapping from words to phone models is usually accomplished by means of a pronunciation dictionary. Phone models in turn are usually modeled as m -state left-to-right hidden Markov models (HMM) to capture the temporal and acoustic variability of speech signals. The following figure shows the process of converting a sequence of words into (1) a pronunciation graph (possibly with pronunciation variants) and (2) an HMM state graph.

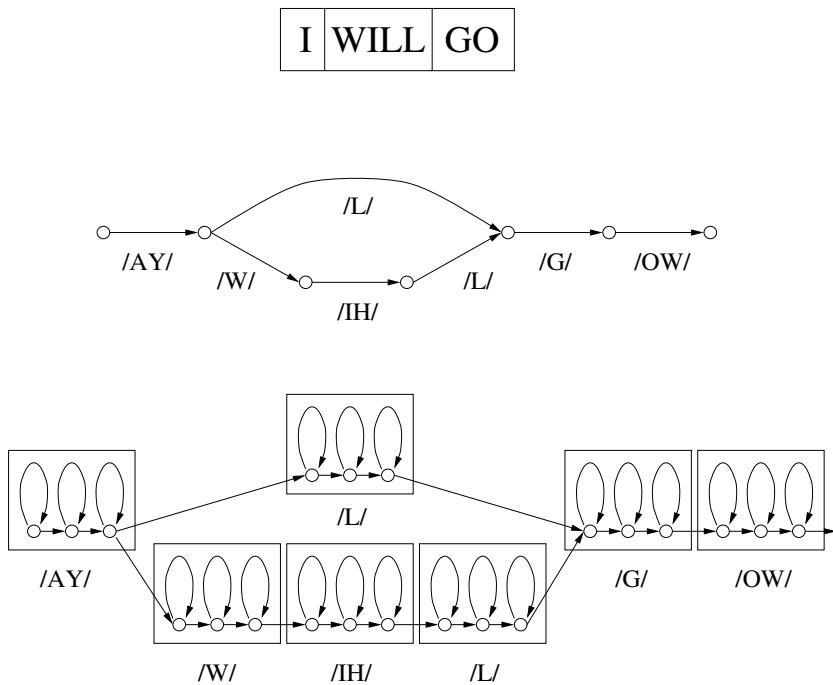


Fig. 15.2. Typical hidden Markov model in speech recognition

In this framework, where word sequences are represented as directed acyclic graphs of HMM states, the likelihood of an acoustic observation can be rewritten as

$$P(\mathbf{X}_1, \dots, \mathbf{X}_M | W_1, \dots, W_N) = \sum_{s_1, \dots, s_M} P(\mathbf{X}_1, \dots, \mathbf{X}_M | s_1, \dots, s_M) p(s_1, \dots, s_M)$$

where the summation extends over all possible state sequences s_1, \dots, s_M in the HMM model for the word sequence W_1, \dots, W_N . In the Viterbi approximation, the above likelihood is approximated by the probability of the most likely state sequence

$$P(\mathbf{X}_1, \dots, \mathbf{X}_M | W_1, \dots, W_N) \approx \max_{s_1, \dots, s_M} P(\mathbf{X}_1, \dots, \mathbf{X}_M | s_1, \dots, s_M) p(s_1, \dots, s_M).$$

Given a specific state sequence, the likelihood of the acoustic observations given that sequence can be factored as follows

$$P(\mathbf{X}_1, \dots, \mathbf{X}_M | s_1, \dots, s_M) \approx \prod_{i=1}^M p(\mathbf{X}_i | X_1, \dots, X_{i-1}, s_1, \dots, s_M) p(s_1, \dots, s_M).$$

In the application of first-order hidden Markov models to the estimation of such likelihoods one usually makes two simplifying assumptions:

- Independence of Observations:

$$P(\mathbf{X}_1, \dots, \mathbf{X}_M | s_1, \dots, s_M) \approx \prod_{i=1}^M p(\mathbf{X}_i | s_1, \dots, s_M) p(s_1, \dots, s_M)$$

- First-order Assumption:

$$P(\mathbf{X}_1, \dots, \mathbf{X}_M | s_1, \dots, s_M) \approx \prod_{i=1}^M p(\mathbf{X}_i | s_i) p(s_i | s_{i-1})$$

15.4.2 Emission and Transition Modeling

Mainstream LVCSR systems follow the above approach by modeling emission probability distributions $p(\mathbf{X}_i | s_i)$ and transition probabilities $p(s_i | s_{i-1})$ separately and independently. Emission probability distributions are usually modeled using mixture densities from the exponential family, such as the mixture of Gaussians

$$p(\mathbf{X}_i | s_i) = \sum_{k=1}^n \gamma_k N_k(\mathbf{X}_i | s_i)$$

where the γ_k denote *mixture coefficients* and the N_k *mixture component densities* (here: normal distributions). Transition probabilities on the other hand

are modeled by simple multinomial probabilities since they are conditioned on a discrete variable only (not on the input vector).

The advantage of this approach is a decoupled estimation process that separates temporal and acoustic modeling. As such, it allows to easily vary HMM state topologies after training in order to modify temporal behavior. For instance, state duplication is a popular technique to increase the minimum duration constraint in phone models. Having separated emission and transition probability estimation, state duplication consists of simply sharing acoustic models among multiple states and adapting the transition probabilities.

However, the disadvantage of the above approach is a mismatch in the dynamic range of emission and transition probabilities. The reason is that transition probabilities are modeled separately as multinomial probabilities, constrained by the requirement to sum to one. This leads to a dominant role of emission probabilities with transition probabilities hardly influencing overall system performance.

15.4.3 Phonetic Context Modeling

So far we have assumed that only one HMM is required per modeled monophone (see Fig. 15.2). Since the English language can be modeled by approximately 45 monophones, one might get the impression that only that number of HMM models need to be trained. In practice however, one observes an effect called coarticulation that causes large variations in the way specific monophones actually sound, depending on their phonetic context.

Usually, explicit modeling of phones in phonetic context yields great gains in recognition accuracy. However, it is not immediately clear how to achieve robust context-dependent modeling. Consider, for example, so called *triphone* models. A triphone essentially represents the realization of a specific monophone in a specific context spanning one phone to the left and right. Assuming an inventory of 45 monophones, the number of (theoretically) possible triphones is $45^3 = 91125$. Many of these triphones will occur rarely or never in actual speech due to the linguistic constraints in the language. Using triphones therefore results in a system which has too many parameters to train. To avoid this problem, one has to introduce a mechanism for sharing parameters across different triphone models.

Typically, a CART like decision tree is adopted to cluster triphones into *generalized triphones* based on both their a-priori probability and their acoustic similarity. Such a top-down clustering requires the specification of viable attributes to be used as questions on phonetic context in order to split tree nodes. Mostly, linguistic classes such as vowel, consonant, fricative, plosive, etc. are being employed. Furthermore, one can generalize triphones to *polyphones* by allowing dependence on a wider context (and not just the immediate left and right neighboring phones). Fig. 15.3 shows a typical decision tree for the clustering of polyphonic variations of a particular monophone state.

The collection of all leaf nodes of decision trees for each monophone state in a given system represents a robust and general set of context-dependent subphonetic units. Since each of these units corresponds to several triphone HMM

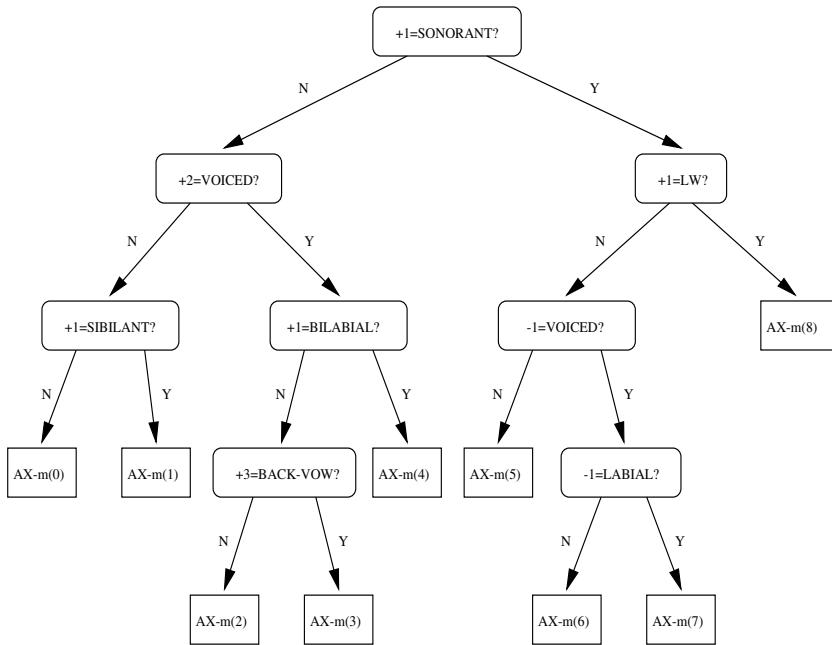


Fig. 15.3. Phonetic Context Modeling using Decision Trees. Shown is a decision tree modeling phonetic contexts of middle state (3-state HMM) of monophone /AX/.

states, they are often called *tied states*. Typically, a large vocabulary continuous speech recognizer models between 3000 and 24000 such tied states. Mainstream LVCSR systems scale to any number of context-dependent modeling units since emission and transition models are independently estimated for each tied state.

15.4.4 Connectionist Acoustic Modeling

Locally discriminant connectionist acoustic modeling is the most popular approach to integrate neural networks into an HMM framework [3, 4, 18]. It is based on converting estimates of local posterior class probabilities to scaled likelihoods using Bayes rule. These scaled likelihoods can then be used as observation probability estimates in standard HMMs. For a moderately small number N of HMM states, a neural network can be trained to jointly estimate posterior probabilities $p(s_i|\mathbf{X}_i)$ for each state s_i given an input vector \mathbf{X}_i . Bayes rule yields the corresponding scaled¹ class conditional likelihoods

$$\hat{p}(\mathbf{X}_i|s_i) = \frac{p(s_i|\mathbf{X}_i)}{p(s_i)}.$$

¹ The missing additional term consisting of the probability of the input vector $p(\mathbf{X}_i)$ is usually omitted because it is independent of the class/state identity and therefore does not influence a Viterbi style search for the most likely state sequence.

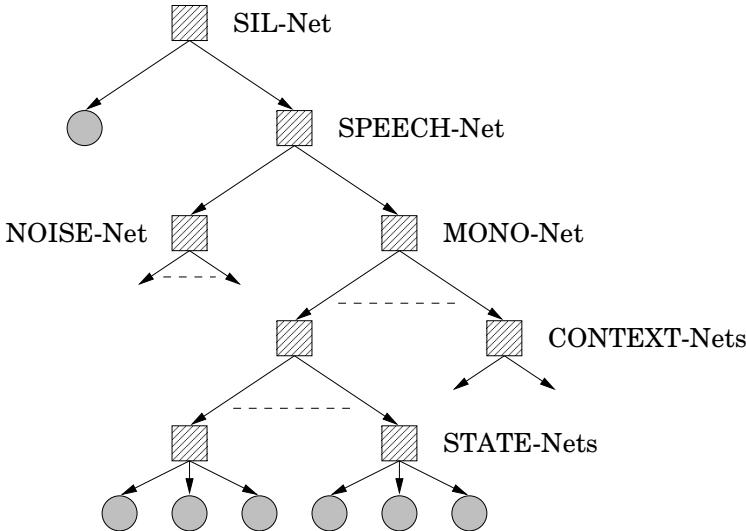


Fig. 15.4. Topology of a Hierarchy of Neural Networks (HNN) to estimate context-dependent posteriors, factored based on a-priori phonetic knowledge

While $p(s_i|\mathbf{X}_i)$ is estimated using a neural network, prior probabilities $p(s_i)$ can be estimated by relative frequencies as observed in the training data. Several researchers (e.g. [3, 14]) have reported improvements with connectionist acoustic modeling when the technique for the estimation of emission probabilities was the only difference in comparison. Since mainstream HMMs for speech recognizers are mostly trained in a maximum likelihood framework using the Expectation-Maximization (EM) algorithm, incorporation of discriminatively trained neural networks that focus on modeling of class boundaries instead of class distributions is often observed to be beneficial. Also, compared to mixtures of Gaussians based acoustic models, connectionist acoustic models are often reported to achieve the same accuracy with far less parameters.

However, when the number of HMM states is increased to model context-dependent polyphones (triphones, quintphones), a single neural network can no longer be applied to estimate posteriors. It becomes necessary to factor the posterior state probabilities [17] and modularize the process of estimating those posteriors. In most approaches, the posteriors are factored on phonetic context or monophone identity (e.g. [4, 9, 15]). Viewing factoring as a hierarchical decomposition of posteriors, we generalized the approaches to context-dependent connectionist acoustic modeling by introducing a tree structured hierarchy of neural networks (HNN) [12, 13] corresponding to a multi-level factoring of posteriors based on a-priori knowledge such as broad sound classes (silence, noises,

phones), phonetic context and HMM state identity. Fig. 15.4 shows the topology of such a structure.

At the top of this hierarchy, we discriminate silence, noise and speech sounds by means of two networks (SIL-Net, SPEECH-Net). The motivation for this specific partitioning comes from the observation that these three classes are easy to distinguish acoustically. The remainder of the tree structure decomposes the posterior of speech, conditioning on monophone, context and state identity as these are convenient sound classes modeled by any phone based HMM speech recognizer. The hierarchy of Fig. 15.4 can be decomposed even further, for instance by factoring conditional monophone posteriors (estimated by the MONO-Net) based on linguistic features (e.g. voiced/unvoiced, vowel/consonantal, fricative etc.). The motivation behind such a decomposition is twofold. First, it reduces the number of local classes in each node, improving approximation accuracy and second, it yields a decoupled and specialized set of expert networks having to handle a smaller amount of phonetic variation.

However, as mentioned in section 3, the use of prior knowledge for the design of a hierarchy of neural networks does not take into account dissimilarity of the observed classes in feature space. We therefore developed an agglomerative clustering algorithm to automatically design such hierarchies for the estimation of posteriors for a large number of classes. We termed this framework ACID/HNN [11].

15.4.5 ACID Clustering

ACID (**A**ggglomerative **C**lustering based on **I**nformation **D**ivergence) is a bottom-up clustering algorithm for the design of tree-structured soft classifiers such as a hierarchy of neural networks (HNN) [10, 11]. Although developed for connectionist acoustic modeling, the algorithm can in principle be used for any kind of classification task. Starting from a typically very large set of initial classes, for example the set of decision tree clustered HMM states in a speech recognizer², the ACID algorithm constructs a binary hierarchy. The nodes in the resulting tree are then instantiated with estimators for the respective conditional posterior probabilities, for instance in form of an HNN. The clustering metric in the ACID algorithm is the symmetric information divergence [26]

$$d(s_i, s_j) = \int_{\mathbf{x}} (p(\mathbf{x}|s_i) - p(\mathbf{x}|s_j)) \log \frac{p(\mathbf{x}|s_i)}{p(\mathbf{x}|s_j)} d\mathbf{x}$$

between class conditional densities of clusters. In contrast to standard agglomerative clustering algorithms which mostly represent clusters by their means and employ the Euclidean distance metric, we chose to represent clusters by parametric mixture densities (mixtures of Gaussians) in the ACID algorithm.

² In our case, we experimented with up to 24000 initial classes.

Modeling clusters with mixture densities is much more adequate than just using the mean and it still allows to cluster large amounts of classes in a reasonable time. The symmetric information divergence (also called Kullback-Leibler distance) measures the dissimilarity of two distributions and was therefore chosen as the clustering metric. Typically, each initial class (state) is modeled using a single full covariance multivariate Gaussian density

$$p(\mathbf{x}|s_i) = \frac{1}{\sqrt{(2\pi)^n |\Sigma_i|}} \exp\left\{-\frac{1}{2}(\mathbf{x} - \mu_i)^t \Sigma_i^{-1} (\mathbf{x} - \mu_i)\right\}$$

with mean vector μ_i and covariance matrix Σ_i . Clustering then continuously merges initial classes which corresponds to building mixture densities based on the Gaussians. The symmetric information divergence between two states s_i and s_j with Gaussian distributions amounts to

$$\begin{aligned} d(s_i, s_j) &= \frac{1}{2} \text{tr}\{(\Sigma_i - \Sigma_j)(\Sigma_j^{-1} - \Sigma_i^{-1})\} \\ &\quad + \frac{1}{2} \text{tr}\{(\Sigma_i^{-1} + \Sigma_j^{-1})(\mu_i - \mu_j)(\mu_i - \mu_j)^t\} \end{aligned}$$

The computation of this distance measure requires $O(n^2)$ multiplications and additions (assuming pre-computed inverse covariance matrices), where n is the dimensionality of the input feature space. To reduce the computational load of the ACID clustering algorithm, one can model the class conditional likelihoods with diagonal covariance matrices only. Feature space transformations such as principal component analysis and linear discriminant analysis can be used to approximate such distributions. When using diagonal covariance Gaussians, the symmetric information divergence simplifies to

$$d(s_i, s_j) = \frac{1}{2} \sum_{k=1}^n \frac{(\sigma_{jk}^2 - \sigma_{ik}^2)^2 + (\sigma_{ik}^2 + \sigma_{jk}^2)(\mu_{ik} - \mu_{jk})^2}{\sigma_{ik}^2 \sigma_{jk}^2}$$

where σ_{ik}^2 and μ_{ik} denote the k -th coefficient of the variance and mean vectors of state s_i , respectively. The evaluation of the latter distance measure requires only $O(n)$ multiplications and additions.

Making the simplifying assumption of linearity of information divergence, we can define the following distance measure between clusters of Gaussians S_k and S_l

$$D(S_k, S_l) = \sum_{s_i \in S_k} p(s_i|S_k) \sum_{s_j \in S_l} p(s_j|S_l) d(s_i, s_j)$$

This distance measure is used in the **ACID** clustering algorithm:

1. Initialize algorithm with N clusters S_i , each containing
 - (1) a parametric model of the class-conditional likelihood and
 - (2) a count C_i , indicating the frequency of class s_i in the training set.
2. Compute within cluster priors $p(s_i|S_k)$ for each cluster S_k , using the counts C_i
3. Compute the symmetric divergence measure $D(S_k, S_l)$ between all pairs of clusters S_k and S_l .
4. Find the pair of clusters with minimum divergence, S_k^* and S_l^*
5. Create a new cluster $S = S_k^* \cup S_l^*$ containing all Gaussians of S_k^* and S_l^* plus their respective class counts. The resulting parametric model is a mixture of Gaussians where the mixture coefficients are the class priors
6. Delete clusters S_k^* and S_l^*
7. While there are at least 2 clusters remaining, continue with 2.

ACID Initialization. Initialization requires to estimate class conditional likelihoods for all (tied) states modeled by the recognizer. The number N of initial classes therefore is determined by other parts of the speech recognizer, namely by the phonetic decision tree that is typically applied to cluster phonetic contexts, or equivalently to tie HMM states [27]. Initial class conditional densities for these classes can be computed from state alignments using either the Viterbi or the Forward-Backward algorithm on training data and corresponding HMM state graphs generated from training transcriptions. Estimation of initial parametric models for the ACID algorithm therefore requires a single pass through the training data. After initial models have been estimated, the actual ACID clustering does not require any further passes through the training data. Furthermore, note that this algorithm clusters HMM states without knowledge of their phonetic identity solemnly based on acoustic dissimilarity.

ACID Dendograms. For illustration purposes, Fig. 15.5 shows a dendrogram of a typical ACID clustering run on a relatively small set of only 56 initial classes corresponding to the set of single-state monophone HMMs in a context-independent speech recognizer. The set of classes consists of 44 standard English phones along with 7 noise sounds (marked with a plus), 4 phones modeling interjections (marked with an ampersand) and silence (SIL).

Already the top level split separates silence, breathing and noise sounds (lower subtree) almost perfectly from phonetic sounds (upper subtree). Furthermore, clusters of acoustically similar phones can be observed in the ACID tree, for instance

- IX,IH,IY,Y
- JH,CH,SH,ZH
- Z,S,F
- ER,AXR,R

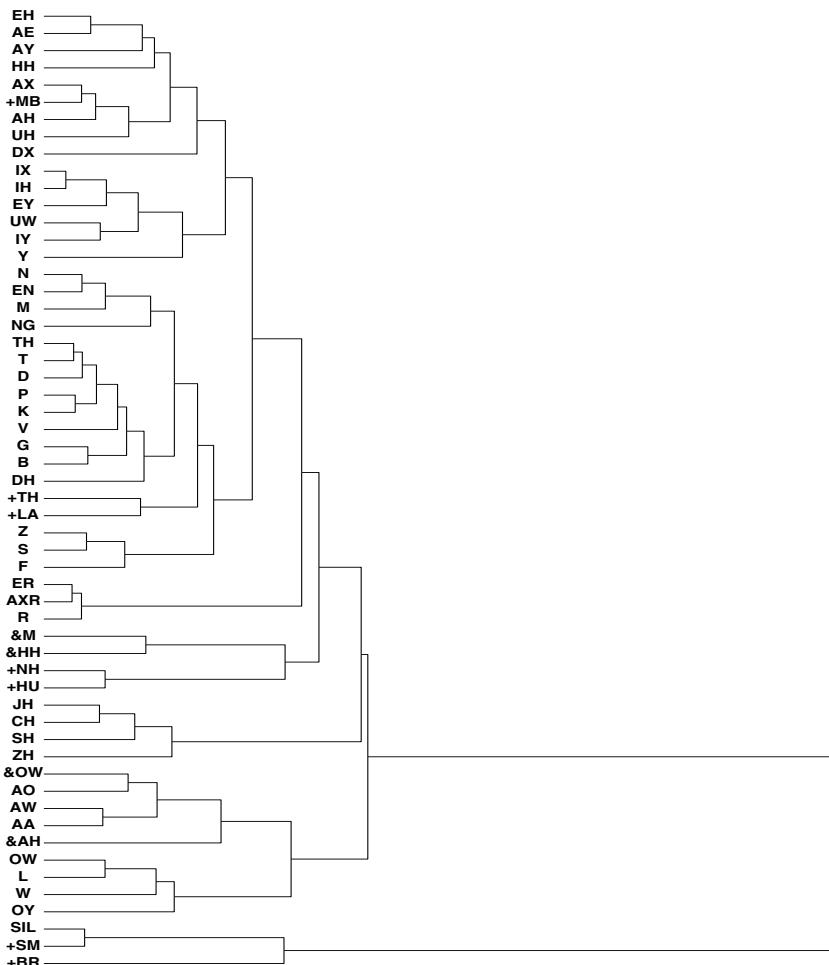


Fig. 15.5. Typical dendrogram of ACID clustering

ACID clustering was found to be quite effective in generating a hierarchical decomposition of a classification task into subtasks of increasing difficulty (when traversing the tree from root node to leaves). In the case of connectionist acoustic modeling for speech recognition, we observed that nodes in the upper layers of an ACID clustered HNN tree distinguish between broad phonetic classes, whereas nodes further down the tree begin to distinguish the particular phones within a broad phonetic class. Thus, ACID clustering constitutes an effective algorithm for discovering inherent hierarchical structure and exploiting it for modular classification.

Model Selection. The choice of model size and topology becomes very important in the application of hierarchical soft classifiers to tasks such as connectionist

speech recognition. While the global tree topology is determined by the outcome of the ACID clustering (or any other tree design procedure), it remains to decide on local (node-internal) classifier topology. The task of a local classifier is to estimate conditional posterior probabilities based on the available training data. Since a particular local estimator is conditioned on all predecessor nodes in the tree, it only receives training data from all the classes (leaves) that can be reached from the respective node. This amounts to a gradually diminishing training set when going from root node to nodes further down the tree. Fig. 15.6 shows this property of HNNs with a plot of the amount of available training patterns vs. node depth for a binary hierarchy with 6000 leaves. Note the logscale on the ordinate.

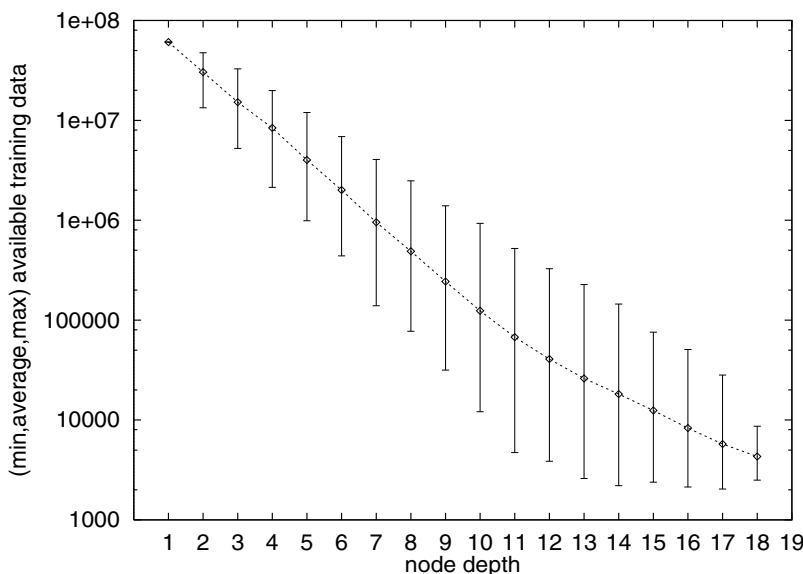


Fig. 15.6. Available Training Data in Different Depths of HNN Tree

When deciding on the local model complexity, we consider tree nodes as lying in a continuum between the following two extrema:

Top of the Hierarchy

- large amounts of training data available
- allows for large node classifiers
- relatively easy, general classification tasks

Bottom of the Hierarchy

- only small amounts of training data available
- requires relatively small node classifiers
- comparably hard classification tasks
- high degree of specialization

Ideally, the complexity of local node classifiers should be selected so as to maximize generalization ability of the complete hierarchy. Generalization, on the other hand, is influenced by three factors: (1) size and distribution of the training set, (2) model complexity and (3) classification complexity of the specific task at hand. Obviously, we can not alter the latter of these factors. Furthermore, in the context of our architecture, we assume that the size of the training set for each node is fixed by the tree topology, once the hierarchy has been designed. Therefore, we have to choose model complexity based on available training data and difficulty of classification task.

In our experiments in connectionist acoustic modeling, we typically use multi layer perceptron (MLP) nodes with a single hidden layer and control model complexity by varying the number of hidden units. We use standard projective kernels with tanh activations for the hidden units and a task dependent non-linearity for the output units (sigmoid for binary and softmax for multiway classification). The overall number of weights in such a network depends linearly on the number of hidden units. According to [1] and with some approximations, a rule of thumb is to choose the number of hidden units M to satisfy

$$N > \frac{M}{\epsilon}$$

where N is the size of the training set and ϵ is the expected error rate on the test set. In our case, the variation in the number of training patterns in the different nodes dominates the above formula. Therefore, we set the number of hidden units proportional to b^{-n} , where b is the branching factor of the classification tree and n is the node depth. As long as the tree is approximately balanced in terms of the prior distribution of child nodes, this strategy leads to hidden layers with size proportional to the number of available training patterns. A more fundamental treatment of model complexity using multiple training runs and cross validation is desirable. However, in case of large-scale applications such as speech recognition, such a strategy is not realizable because of the high computational cost resulting from very large training databases. Less heuristic approaches to select model complexity still have to be explored.

15.4.6 Training Hierarchies of Neural Networks on Large Datasets

For the demonstration of various aspects of training large and complex structures such as hierarchies of neural networks on typical datasets, we report on experiments on the Switchboard [19] speech recognition database. Switchboard is a large corpus of conversational American English dialogs, recorded in telephone quality all over the US. It consists of about 170 hours of speech which typically corresponds to about 60 million training samples. The corpus currently serves as a benchmark for the official evaluation of state-of-the-art speech recognition systems. Switchboard is a comparably hard task, current best systems achieve word error rates in the vicinity of 30-40%. Fig. 15.7 shows the structure of an HNN based connectionist acoustic model for an HMM based recognizer, in our case the Janus recognition toolkit (JanusRTk) [8].

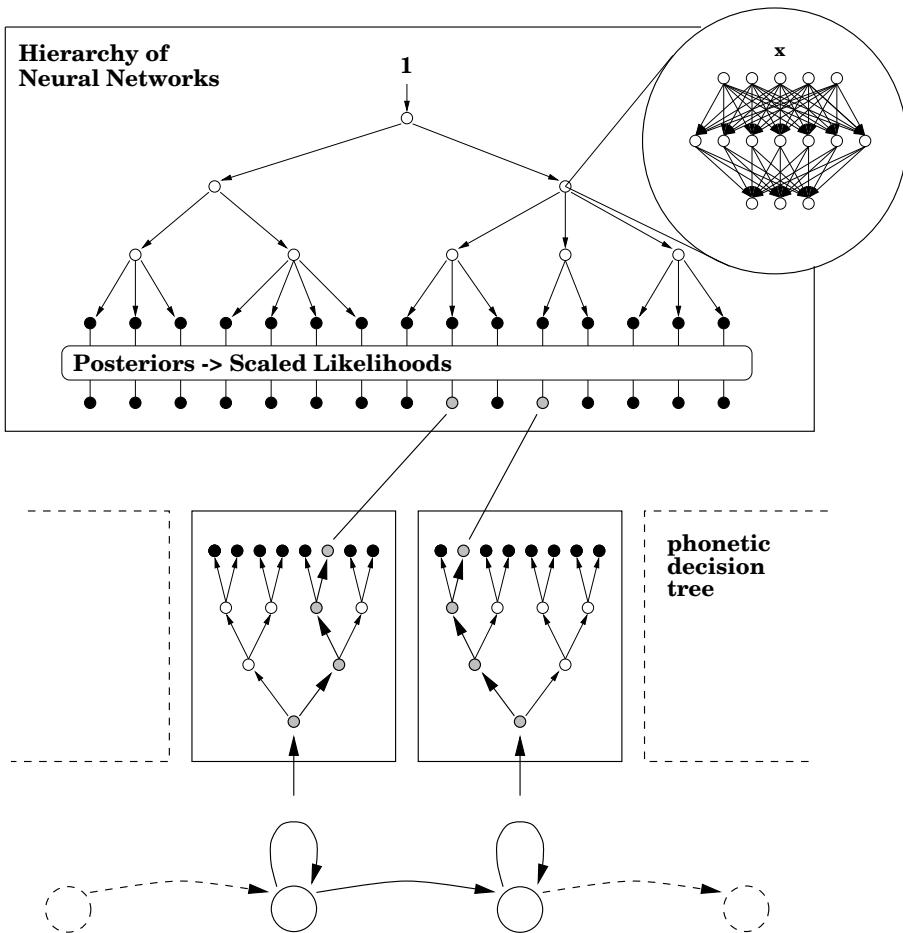


Fig. 15.7. Hierarchy of Neural Networks for Connectionist Acoustic Modeling: The upper part shows an ACID clustered HNN after node merging. This architecture computes posterior probabilities for a set of generalized polyphones. To allow for integration into the HMM framework, these posteriors are converted to scaled likelihoods. The correspondence to actual HMM states is accomplished by means of phonetic decision trees.

Due to the inherent variability and complexity of the task and the large amount of training data, typical speech recognition systems model several thousand distinct subphonetic units (HMM states) as base classes. This requires to train an estimator for posterior probabilities of thousands of distinct acoustic classes based on millions of training samples, in order to take advantage of the full modeling granularity of the speech recognizer.

In the following, we will discuss several aspects of training a hierarchical soft classifier on large datasets such as Switchboard. Due to the modular structure of the classifier, the size of the model inventory and the training database, the following discussion leads to rather unique problems and solutions. However, it is important to emphasize that these properties stem from the structure of the classifier and the size of the task - not from the specific task of acoustic modeling for speech recognition. Thus, they are transferable to comparably large tasks, e. g. handwriting, speaker or face recognition.

Classifier Tree Topology. Depending on the number of classes to be modeled, tree design algorithm, branching factor and size and structure of local node classifiers have to be chosen. For Switchboard, we were experimenting with three systems consisting of 6000, 10000 and 24000 distinct classes, respectively. We used the ACID clustering algorithm to design an initial tree structure from the set of base classes for the 6k and 24k systems. As a second step of the tree design phase, we applied a greedy node merging algorithm on the ACID clustered hierarchy. Node merging decreases the number of internal nodes while increasing the average branching factor (arity) of the tree. Training of such hierarchies is less problematic than training of the original binary tree structure since the difference among nodes (in terms of the number of available training patterns) is somewhat extenuated and the overall number of networks is reduced. However, local classification tasks change from 2-way (binomial) to more complex multi-way (multinomial) problems which might have an impact on the accuracy of estimating conditional posteriors. Therefore, we constrain the node merging algorithm to produce nodes with a maximum branching factor of 8-12. This value was found to improve training speed while not affecting overall classifier accuracy. Considerably larger branching factors are not reasonable in our case as we would gradually lose the advantage of the hierarchical structure by flattening the tree.

For the 10k system, we were using the architecture of Fig. 15.4 that was designed by prior knowledge, not taking into account any measure of class similarity. This structure exhibits a larger average branching factor and less depth than the ACID clustered trees. Although we could decrease the branching factor at the MONO node by introducing linguistic classes as mentioned earlier, we still have large branching factors at the context nodes which are much harder to resolve with prior knowledge only.

The resulting tree nodes were instantiated with MLPs of varying size of the (single) hidden layer. The local MLPs output layer were parameterized with the softmax non-linearity for two reasons. First, it complies to the property of the modeled probability distribution to sum up to one, and second, the softmax function implements the expected value of the multinomial probability density. Fig. 15.8 gives an overview of the structure of the ACID/HNN systems. Tree compactification reduced the number of internal nodes of the 24k system from 24k to about 4k by increasing the average number of local classes (average branching

level	# nodes = # networks	# hidden units/network
1	1	256
2	1	256
3	1	256
4	3	192
5	19	128
6	121	64
7	816	32
total	962	

level	# nodes = # networks	# hidden units/network
1	1	128
2	10	128
3	77	64
4	524	32
5	3434	16
total	4046	

Fig. 15.8. Overview of ACID clustered HNNs for 6k (left) and 24k (right) classes

factor) from 2 to about 8. Especially when dealing with large numbers of classes, we found that moderate tree compactification improved classifier performance. The overall numbers of parameters of the tree classifiers were 2M for the 6k system, 2.4M for the 10k system and 3.1M for the 24k system.

Training Algorithm and Parameters. Training a distributed, hierarchically organized collection of neural networks on different amounts of training data is a challenging task. Our training criterion is maximum likelihood, assuming a multinomial probability model (1-out-of- N) over all base classes. A target class label is associated with each training pattern, indicating the correct base class. All networks in nodes along the path from root node to the current target class' leaf receive the current pattern for training. Because of the large amount of training data, we use on-line (stochastic) gradient ascent in log-likelihood with small batches (10-100 patterns) to train the individual networks. More elaborate training algorithms which apply second order methods in optimizing the objective function are too costly in our scenario - a single epoch of training, processing all 60 million patterns in the training database takes 3-5 days on a Sparc Ultra workstation. A practical training algorithm therefore must not take longer than 1-4 epochs to converge. Furthermore, because of the large number of networks that have to be trained, a potential training algorithm can not be allowed to use large amounts of memory - which could be the case with second order methods. Of course, training of individual node classifiers is independent and can therefore easily be parallelized for shared memory multi-processors which alleviates the latter requirement.

Since we are relying on *stochastic* gradient ascent in our training method, we additionally use a simple momentum term to smooth gradients. Also, we use local learning rates for each network that are initialized with a global learning rate and adapted individually to the specific learning task. The global learning rate is annealed in an exponentially decaying scheme:

$$\eta_G^{n+1} = \eta_G^n * \gamma_G.$$

Typically, we use an initial global learning rate η_G between 0.001 and 0.01, a momentum constant of 0.5 and a global annealing factor γ_G of 0.999...0.9999 applied after each batch update.

In order to accommodate the different learning speeds of the node classifiers due to the different amount of available training data, we control individual learning rates using the following measure of correlation between successive gradient vectors g_{n-1} and g_n :

$$\alpha_n = \arccos\left(\frac{g_n^T g_{n-1}}{\|g_n\| \|g_{n-1}\|}\right)$$

α_n measures the angle between the gradients g_{n-1} and g_n . Small angles indicate high correlation and therefore steady movement in weight space. Therefore, we increase the learning rate linearly up to the current maximum (as determined by initial learning rate, annealing factor and number of updates performed) whenever $\alpha_n < 90^\circ$ for several batch updates M . Large angles, on the other hand, indicate random jumps in weight space. We therefore decrease the learning rate exponentially whenever $\alpha_n > 90^\circ$ for several batch updates M . In summary, we obtain the following update rule for local learning rate η_i of network i :

$$\eta_i^{n+1} = \min \left\{ \eta_G^{n+1}, \begin{cases} \eta_i^n + \delta \\ \eta_i^n * \gamma \\ \eta_i^n \end{cases} \right\} \quad \text{if } \begin{cases} \frac{1}{M} \left(\sum_{k=0}^M \alpha_{n-k} \right) < 90^\circ - \epsilon \\ \frac{1}{M} \left(\sum_{k=0}^M \alpha_{n-k} \right) > 90^\circ + \epsilon \\ \text{else} \end{cases}$$

with linear increase $\delta = 0.001 \dots 0.01$ and exponential annealing factor $\gamma = 0.5 \dots 0.9$. The number of batch updates M controls smoothing of α whereas ϵ controls the influence of the global learning rate. For $\epsilon \rightarrow 90^\circ$, local learning rates are forced to follow the global learning rate, whereas low values of ϵ allow local learning rates to develop individually. Typical values that have been used in our experiments are $M = 10$ and $\epsilon = 20^\circ$.

Adapting individual learning rates to the training speed is a critical issue in hierarchical classifiers. Networks at the top of the tree have to be trained on very large amounts of training data. Therefore, learning rates must be allowed to become relatively small in order to benefit from all the data and not reach the point of saturation too early. On the other hand, networks at the bottom of the tree have to be trained with comparably small amounts of data. In order to train these networks within the same small number of passes through the overall data, we have to apply comparably large learning rates to reach a maximum in local likelihood as fast as possible. However, unconstrained adaptation of learning rates with aggressive optimization of learning speed may result in failure to converge. In our experiments with global initialization of all networks using the

same maximum learning rate, global annealing of the maximum learning rate and local adaptation of individual learning rates that are constrained to never become larger than the global learning rate gives best results.

Generalization/Overfitting. Simply speaking, we did not observe any overfitting in our experiments. Taking a look at the training of a large hierarchy in terms of performance on an independent cross-validation set (Fig. 15.9), we can see that the likelihood on this data levels off, but never starts to decrease again, as is often observed on smaller tasks. In the plots of Fig. 15.9, the vertical lines indicate multiple epochs (passes) through the training data (consisting of 87000 utterances). Obviously, the large amount of available training data allows for excellent generalization, early stopping was not necessary. This behavior is surprising at first sight, because we did not use any kind of explicit regularization of the local MLPs. At second sight, however, we can identify several reasons for the good generalization of HNNs on this task:

- Training data can be considered very noisy in our case, since samples come from a large variety of different speakers and recording conditions. Training with noisy data is similar to regularization and therefore improves generalization [2].
- Consider the hierarchy for the 6k classes system (Fig. 15.8). Some of the 816 networks at the bottom of the tree probably have not seen enough training patterns to generalize well to new data. Although all of these networks together constitute 85% of the total number of networks, they contribute just as one of 7 networks to any particular posterior probability. The networks in the upper part of the hierarchy have the largest influence on the evaluation of posterior probabilities. For those networks, the amount of available training data can be considered so abundant that test set error approaches training set error rate. In other words, optimal generalization can be achieved.

Results. We evaluate the proposed hierarchical classifiers as connectionist acoustic models in a speech recognition system. Performance of speech recognizers is usually measured in terms of the word error rate on a reasonably large set of test utterances. In our case, we test the different acoustic classifiers with the Janus [8] recognizer on the first 30 seconds of each speaker in the official 1996 Switchboard evaluation test set, consisting of 366 utterances not present in the training set.

acoustic classifier	# classes	# parameters	word error rate
HNN	10000	2.0 M	37.3 %
ACID/HNN	6000	2.4 M	36.7 %
ACID/HNN	24000	3.1 M	33.3 %

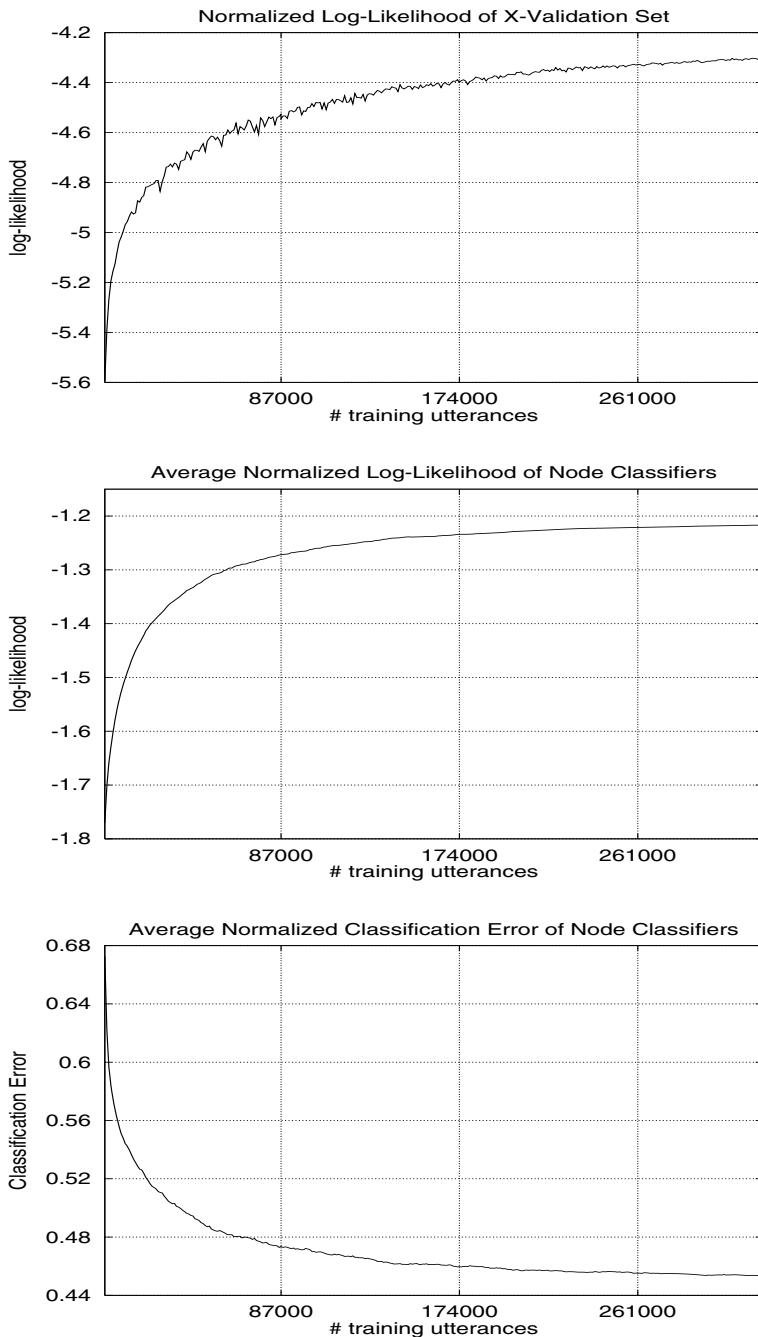


Fig. 15.9. Cross-Validation during training of 24k ACID/HNN architecture

The above results are competitive with those of state-of-the-art systems and indicate a clear advantage of the ACID clustered over the pre-determined hierarchical classifiers. We suspect, that the reason for the better performance of automatically clustered hierarchies of neural networks is the difference in tree topology. Automatically clustered HNNs such as the presented ACID/HNN trees exhibit small and comparably uniform average branching factors that allow to robustly train estimators of conditional posterior probabilities. In contrast, hand-crafted hierarchies such as the 10k HNN tree contain nodes with rather large branching factors. Fig. 15.10 shows the branching factors for all the networks in the 10k tree structure.

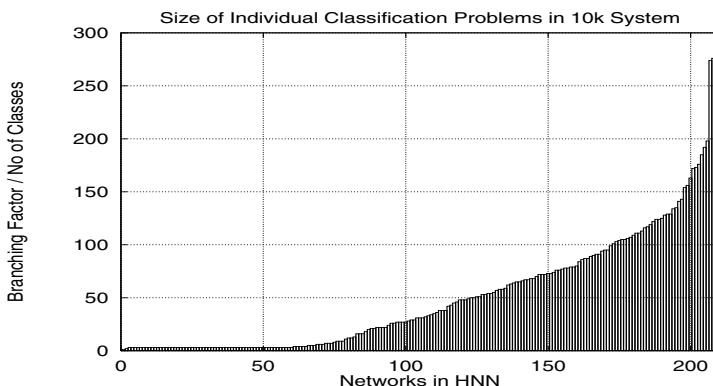


Fig. 15.10. Branching Factors of Individual Nodes in 10k HNN

The largest observed branching factor in this tree was 276. This requires the joint estimation of conditional posterior probabilities for as many as 276 classes which may result in rather poor approximations to the true posterior probabilities for some of the networks in the tree.

Furthermore, the superior performance of both ACID/HNN classifiers over the hand-crafted 10k tree, demonstrates the full scalability of the hierarchical approach and justifies the increase in the number of parameters. Earlier attempts to train hand-crafted hierarchies for 24k classes failed to provide classifiers that could be used as acoustic models in a speech recognizer. Poor approximations to the real posterior probabilities led to instabilities in decoding when dividing by priors in this case. Apart from that, we do not know of any other non-parametric approach capable of directly and discriminatively estimating posterior probabilities for such a large amount of classes.

15.5 Conclusions

We have presented and discussed a methodology for the estimation of posterior probabilities for large numbers of classes using a hierarchical connectionist

framework. The aim of the paper is to demonstrate the necessity of hierarchical approaches to modularize classification tasks in large-scale application domains such as speech recognition, where thousands of classes have to be considered and millions of training samples are available. The *divide and conquer* approach proves to be a versatile tool in breaking down the complexity of the original problem into many smaller tasks. Furthermore, agglomerative clustering techniques can be applied to automatically impose a suitable hierarchical structure on a given set of classes, even in the case this set contains tens of thousands of classes. In contrast to the relatively small standard benchmarks for learning machines, aspects such as choice of training method, model selection and generalization ability appear in different light when tackling large-scale probability estimation problems.

References

- [1] Baum, E.B., Haussler, D.: What Size Net Gives Valid Generalization? *Neural Computation* 1, 151–160 (1989)
- [2] Bishop Training, C.M.: with Noise is Equivalent to Tikhonov Regularization. *Neural Computation* 7(1), 108–116 (1995)
- [3] Bourlard, H., Morgan, N.: *Connectionist Speech Recognition – A Hybrid Approach*. Kluwer Academic Press (1994)
- [4] Bourlard, H., Morgan, N.: A Context Dependent Neural Network for Continuous Speech Recognition. In: *IEEE Proc. Intl. Conf. on Acoustics, Speech and Signal Processing*, San Francisco, CA, vol. 2, pp. 349–352 (1992)
- [5] Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: *Classification and Regression Trees*. Wadsworth International Group, Belmont (1984)
- [6] Bridle, J.: Probabilistic Interpretation of Feed Forward Classification Network Outputs, with Relationships to Statistical Pattern Recognition. In: Fogelman-Soulie, F., Héault, J. (eds.) *Neurocomputing: Algorithms, Architectures, and Applications*. Springer, New York (1990)
- [7] Duda, R., Hart, P.: *Pattern Classification and Scene Analysis*. John Wiley & Sons, Inc. (1973)
- [8] Finke, M., Fritsch, J., Geutner, P., Ries, K., Zeppenfeld, T.: The JanusRTk Switchboard/Callhome 1997 Evaluation System. In: *Proceedings of LVCSR Hub5-e Workshop*, Baltimore, Maryland, May 13-15 (1997)
- [9] Franco, H., Cohen, M., Morgan, N., Rumelhart, D., Abrash, V.: Context-Dependent Connectionist Probability Estimation in a Hybrid Hidden Markov Model – Neural Net Speech Recognition System. *Computer Speech and Language* 8(3), 211–222 (1994)
- [10] Fritsch, J., Finke, M.: ACID/HNN: Clustering Hierarchies of Neural Networks for Context-Dependent Connectionist Acoustic Modeling. In: *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, Seattle, Wa (May 1998)
- [11] Fritsch, J.: ACID/HNN: A Framework for Hierarchical Connectionist Acoustic Modeling. In: *Proceedings of IEEE Workshop on Automatic Speech Recognition and Understanding*, Santa Barbara, Ca (December 1997)
- [12] Fritsch, J., Finke, M., Waibel, A.: Context-Dependent Hybrid HME/HMM Speech Recognition using Polyphone Clustering Decision Trees. In: *Intl. Conf. on Acoustics, Speech and Signal Processing*, Munich, Germany, vol. 3, p. 1759 (1997)

- [13] Fritsch, J.: Modular Neural Networks for Speech Recognition, Tech. Report CMU-CS-96-203, Carnegie Mellon University, Pittsburgh, PA (1996)
- [14] Hochberg, M.M., Cook, G.D., Renals, S.J., Robinson, A.J., Schechtman, R.S.: The 1994 ABBOT Hybrid Connectionist-HMM Large-Vocabulary Recognition System. In: Spoken Language Systems Technology Workshop, pp. 170–176. ARPA (January 1995)
- [15] Kershaw, D.J., Hochberg, M.M., Robinson, A.J.: Context-Dependent Classes in a Hybrid Recurrent Network-HMM Speech Recognition System, Tech. Rep. CUED/F-INFENG/TR217, Cambridge University Engineering Department, Cambridge, England (1995)
- [16] Merz, C.J., Murphy, P.M.: UCI Repository of Machine Learning Databases, University of California, Department of Information and Computer Science (1996), <http://www.ics.uci.edu/~mlearn/MLRepository.html>
- [17] Morgan, N., Bourlard, H.: Factoring Networks by a Statistical Method. *Neural Computation* 4(6), 835–838 (1992)
- [18] Morgan, N., Bourlard, H.: An Introduction to Hybrid HMM/Connectionist Continuous Speech Recognition Signal Processing Magazine, 25–42 (May 1995)
- [19] NIST, Conversational Speech Recognition Workshop, DARPA Hub-5E Evaluation, May 13–15, Baltimore, Maryland (1997)
- [20] Prechelt, L.: Proben1 – A Set of Neural Network Benchmark Problems and Benchmarking Rules. Technical Report 21/94, University of Karlsruhe, Germany (1994)
- [21] Quinlan, J.R.: Induction of Decision Trees. *Machine Learn.* 1, 81–106 (1986)
- [22] Rabiner, L.R.: A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE* 77, 257–285 (1989)
- [23] Safavian, S.R., Landgrebe, D.: A Survey of Decision Tree Classifier Methodology. *IEEE Transactions on Systems, Man and Cybernetics* 21(3), 660–674 (1991)
- [24] Schürmann, J., Doster, W.: A Decision Theoretic Approach to Hierarchical Classifier Design. *Pattern Recognition* 17(3), 359–369 (1984)
- [25] Schürmann, J.: *Pattern Classification: A Unified View of Statistical and Neural Approaches*. John Wiley & Sons, Inc., New York (1996)
- [26] Tou, J.T., Gonzales, R.C.: *Pattern Recognition Principles*. Addison Wesley, Reading (1974)
- [27] Young, S.: Large Vocabulary Continuous Speech Recognition: a Review. CUED Technical Report, Cambridge University (1996)

Tricks for Time Series*

Preface

In the last section we focus on tricks related to time series analysis and economic forecasting. In chapter 16, John Moody opens with a survey of the challenges of macroeconomic forecasting including problems such as noise, nonstationarities, nonlinearities, and the lack of good *a priori* models. Lest one be discouraged, descriptions of many possible neural network solutions are next presented including hyperparameter selection (e.g. for regularization, training window length), input variable selection, model selection (size and topology of network), better regularizers, committee forecasts, and model visualization.

The survey is followed by a more detailed description of **smoothing regularizers**, **model selection methods** (e.g. prediction risk, nonlinear cross-validation (NCV) (p. 357)) and **sensitivity-based pruning (SBP)** (p. 359) for input selection. The goal of using regularizers is to introduce bias into the model. But what is the “right bias”? Weight decay may be too *ad hoc* in that it does not consider the nature of the function being learned. As an alternative, the author presents several new smoothing regularizers for both feedforward and recurrent networks that empirically are found to work better.

In model selection, **prediction risk** is used as the criterion for determining “best fits”. Several methods for estimating prediction risk are discussed and compared: generalized cross-validation (GCV), Akaike’s final prediction error (FPE), predicted squared error (PSE), and generalized prediction error (GPE) which can be expressed in terms of the effective number of parameters.

In cross validation, separate networks are trained on different subsets of the data to obtain estimates of the prediction risk. With nonlinear loss functions, however, each network may converge to distinct local minima making comparison difficult. NCV alleviates this problem by initializing each network to be trained on a CV subset in the same minimum w_0 (obtained on the full training set). This way the CV errors computed on the different subsets estimate the prediction risk locally around this minimum w_0 and not by using some remote local minima. SBP is used to select the “best subset” of input variables to use. Here a **sensitivity measure** (e.g. delta error, average gradient, average absolute gradient, RMS gradient) (p. 360) is used to measure the change in the training error that would result if a given input is removed. Input variables are ranked based on importance and, beginning with the least important, are pruned one at a time, retraining in between. Finally, John Moody shows how sensitivity measures can be examined visually over time to better understand the role of each input (p. 361). Throughout, empirical results are presented for forecasting the U.S. Index of Industrial Production.

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

In chapter 17, Ralph Neuneier and Hans Georg Zimmermann describe in detail their impressive integrated system – with a lot of different tricks – for neural network training in the context of economic forecasting. The authors discuss all design steps of their system, e.g. input preprocessing, cost functions, handling of outliers, architecture, regularization techniques, learning techniques, robust estimation, estimation of error bars, pruning techniques. As in chapter 1, the tricks here are also highly interleaved, i.e. many tricks will not retain their full efficiency if they are used individually and not en bloc. Let us start with the preprocessing for which the authors use: **squared inputs** (see chapter 7), **scaled relative differences**, **scaled forces** in form of scaled curvatures or mean reverting that can characterize turning points in time series (p. 370). To limit the influence of **outliers** the previously mentioned inputs are transformed by

$$x' = \tanh(wx),$$

where the parameters w are learned as one layer in the training process (p. 371). Subsequently, a **bottleneck network** reduces the number of inputs to the relevant ones (p. 372). Since networks for time series prediction are in general bottom heavy, i.e. the input dimension is large while the output dimension is very small, it is important to increase the number of output targets, so that more useful error signals can be backpropagated (see also [3, 2]). For this the authors introduce two output layers: (1) a **point prediction layer**, where not only the value to be predicted, i.e. y_{t+n} , but also a number of neighboring values in time are used and (2) an **interaction layer**, where differences between these values, i.e. $y_{t+n+1} - y_{t+n}$, corresponding to curvature are employed (p. 375). Following this interesting trick, several point predictions are **averaged** to reduce the variance of the prediction, similar to bagging [1] (p. 377). This overall design gives rise to an 11-layer neural network architecture, where the available prior knowledge from financial forecasting is coded.

For training this large – but rather constrained – architecture, **cost functions** are defined (p. 383) and a method based on the **CDEN approach** (p. 384) is proposed for estimating the error bars of a forecast.

To train the network the so-called **vario- η learning rule** is introduced, which is essentially a stochastic approximation of a Quasi-Newton algorithm with individual learning rates for each weight (p. 392). The authors discuss how the simple pattern-by-pattern rule has structural consequences that improve generalization behavior; or to put it differently: the stochasticity of learning implicitly includes a curvature penalty on unreliable network parts.

Then the authors raise the provoking question of the Observer-Observer dilemma: to create a model based on observed data while, at the same time, using this model to judge the correctness of new incoming data (p. 391). This leads to (1) clearing and (2) training with noise on input data. The rationale behind **clearing** is that a very noisy environment, such as financial data, will spoil a good prediction if the data is taken too seriously. That is, we are allowed to move the data point a little bit in input space in order to get smoother predictions (p. 395). Similarly, different additive noise levels for each input are chosen by

an algorithm in order to **adapt the noise level** to the (estimated) importance of the input: a small noise level is used for perfectly described or unimportant inputs whereas a large noise level should be chosen for a poorly described (likely noise corrupted) input (p. 397).

In the next step, pruning methods for optimizing the architecture are described. They are: (1) **node-pruning** (p. 401) and (2) several types of **weight-pruning**: (a) **stochastic** pruning (p. 401), (b) **early-brain-damage** pruning (p. 402), (c) **inverse-kurtosis** pruning (p. 403), and (d) **instability** pruning (p. 405). The authors use a combination of instability pruning and early-brain-damage in their application; the first gives stable models and the latter generates very sparse networks.

Finally, the whole set of tricks is combined into an integrated training process and monitored on a validation set: early/late stopping, pruning and weight decay regularization are alternated (p. 414) to obtain an excellent estimate of the German bond rate from June 1994 to May 1996 (p. 415).

Jenny & Klaus

References

- [1] Breiman, L.: Bagging predictors. *Machine Learning* 26(2), 123–140 (1996)
- [2] Caruana, R., de Sa, V.R.: Promoting poor features to supervisors: Some inputs work better as outputs. In: Mozer, M.C., Jordan, M.I., Petsche, T. (eds.) *Advances in Neural Information Processing Systems*, vol. 9, p. 389. The MIT Press (1997)
- [3] Caruana, R., Pratt, L., Thrun, S.: Multitask learning. *Machine Learning* 28, 41 (1997)

Forecasting the Economy with Neural Nets: A Survey of Challenges and Solutions^{*}

John Moody

Department of Computer Science & Engineering
 Oregon Graduate Institute of Science & Technology
 P.O. Box 91000, Portland, OR 97291, USA
moody@cse.ogi.edu
<http://www.cse.ogi.edu/~moody/>

Abstract. Macroeconomic forecasting is a very difficult task due to the lack of an accurate, convincing model of the economy. The most accurate models for economic forecasting, “black box” time series models, assume little about the structure of the economy. Constructing reliable time series models is challenging due to short data series, high noise levels, nonstationarities, and nonlinear effects. This chapter describes these challenges and presents some neural network solutions to them. Important issues include balancing the *bias/variance tradeoff* and the *noise/nonstationarity tradeoff*. A brief survey of methods includes hyperparameter selection (regularization parameter and training window length), input variable selection and pruning, network architecture selection and pruning, new smoothing regularizers, committee forecasts and model visualization. Separate sections present more in-depth descriptions of smoothing regularizers, architecture selection via the *generalized prediction error (GPE)* and *nonlinear cross-validation (NCV)*, input selection via *sensitivity based pruning (SBP)*, and model interpretation and visualization. Throughout, empirical results are presented for forecasting the U.S. Index of Industrial Production. These demonstrate that, relative to conventional linear time series and regression methods, superior performance can be obtained using state-of-the-art neural network models.

16.1 Challenges of Macroeconomic Forecasting

Of great interest to forecasters of the economy is predicting the “business cycle”, or the overall level of economic activity. The business cycle affects society as a whole by its fluctuations in economic quantities such as the unemployment rate (the misery index), corporate profits (which affect stock market prices), the demand for manufactured goods and new housing units, bankruptcy rates, investment in research and development, investment in capital equipment, savings

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

rates, and so on. The business cycle also affects important socio-political factors such as the general mood of the people and the outcomes of elections.

The standard measures of economic activity used by economists to track the business cycle include the Gross Domestic Product (GDP) and the Index of Industrial Production (IP). GDP is a broader measure of economic activity than is IP. However, GDP is computed by the U.S. Department of Commerce on only a quarterly basis, while Industrial Production is more timely, as it is computed and published monthly. IP exhibits stronger cycles than GDP, and is therefore more interesting and challenging to forecast. (See figure 16.1.) In this chapter, all empirical results presented are for forecasting the U.S. Index of Industrial Production.

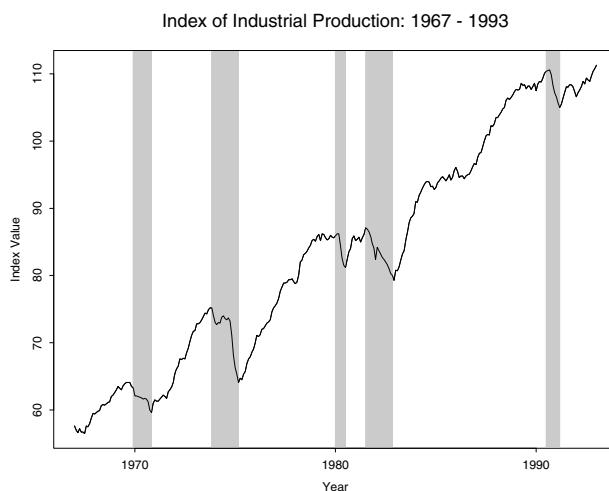


Fig. 16.1. U.S. Index of Industrial Production (IP) for the period 1967 to 1993. Shaded regions denote official recessions, while unshaded regions denote official expansions. The boundaries for recessions and expansions are determined by the National Bureau of Economic Research based on several macroeconomic series. As is evident for IP, business cycles are irregular in magnitude, duration, and structure.

Macroeconomic modeling and forecasting is challenging for several reasons:

Non-experimental Science: Like evolutionary biology and cosmology, macroeconomics is largely a non-experimental science. There is only one instance of the world economy, and the economy of each country is not a closed system. Observing the state of an economy in aggregate is difficult, and it is generally not possible to do controlled experiments.

No *a priori* Models: A convincing and accurate scientific model of business cycle dynamics is not yet available due to the complexities of the economic system, the impossibility of doing controlled experiments on the economy, and non-quantifiable factors such as mass psychology and sociology that influence

economic activity. There are two main approaches that economists have used to model the macroeconomy, econometric models and linear time series models:

Econometric Models: These models attempt to model the macroeconomy at a relatively fine scale and typically contain hundreds or thousands of equations and variables. The model structures are chosen by hand, but model parameters are estimated from the data. While econometric models are of some use in understanding the workings of the economy qualitatively, they are notoriously bad at making quantitative predictions.

Linear Time Series Models: Given the poor forecasting performance of econometric models, many economists have resorted to analyzing and forecasting economic activity by using the empirical “black box” techniques of standard linear time series analysis. Such time series models typically have perhaps half a dozen to a dozen input series. The most reliable and popular of these models during the past decade or so have been bayesian vector autoregressive (BVAR) models [22]. As we have found in our own work, however, neural networks can often outperform standard linear time series models. The lack of an *a priori* model of the economy makes input variable selection, the selection of lag structures, and network model selection critical issues.

Noise: Macroeconomic time series are intrinsically very noisy and generally have poor signal to noise ratios. (See figures 16.2 and 16.3.) The noise is due both to the many unobserved variables in the economy and to the survey techniques used to collect data for those variables that are measured. The noise distributions are typically heavy tailed and include outliers. The combination of short data series and significant noise levels makes controlling model variance, model complexity, and the *bias / variance tradeoff* important issues [9]. One measure of complexity for nonlinear models is P_{eff} , the *effective number of parameters* [24, 25]. P_{eff} can be controlled to balance bias and variance by using regularization and model selection techniques.

Nonstationarity: Due to the evolution of the world’s economies over time, macroeconomic series are intrinsically nonstationary. To confound matters, the definitions of many macroeconomic series are changed periodically as are the techniques employed in measuring them. Moreover, estimates of key series are periodically revised retroactively as better data are collected or definitions are changed. Not only do the underlying dynamics of the economy change with time, but the noise distributions for the measured series vary with time also. In many cases, such nonstationarity shortens the usable length of the data series, since training on older data will induce biases in predictions. The combination of noise and nonstationarity gives rise to a *noise / nonstationarity tradeoff* [23], where using a short training window results in too much model variance or *estimation error* due to noise in limited training data, while using a long training window results in too much model bias or *approximation error* due to nonstationarity.

Nonlinearity: Traditional macroeconomic time series models are linear [12, 14]. However, recent work by several investigators have suggested that nonlinearities can improve macroeconomic forecasting models in some cases [13, 27, 39, 35, 40].

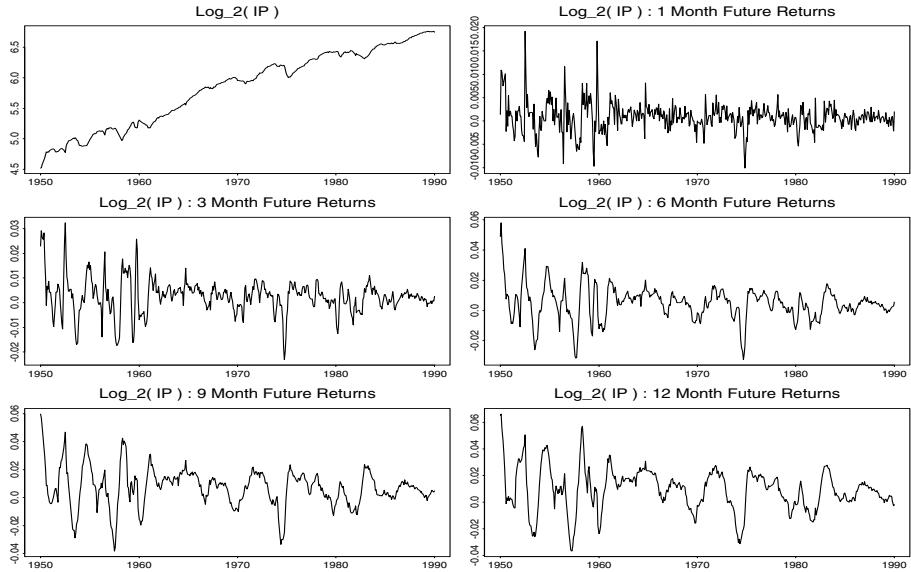


Fig. 16.2. The U.S. Index of Industrial Production and five return series (rates of change measured as log differences) for time scales of 1, 3, 6, 9, and 12 months. These return series served as the prediction targets for the standard Jan 1950 - Dec 1979 / Jan 1980 - Jan 1990 benchmark results reported in [27]. The difficulty of the prediction task is evidenced by the poor signal to noise ratios and erratic behavior of the target series. For the one month returns, the performance of our neural network predictor in table 1 suggests that the SNR is around 0.2. For all returns series, significant nonstationarities and deviations from normality of the noise distributions are present.

(See table 16.1 and figures 16.2 and 16.3.) Based upon our own experience, the degree of nonlinearity captured by neural network models of macroeconomic series tends to be mild [27, 20, 38, 42, 28, 45]. Due to the high noise levels and limited data, simpler models are favored. This makes reliable estimation of nonlinearities more difficult.

16.2 A Survey of Neural Network Solutions

We have been investigating a variety of algorithms for neural network model selection that go beyond the *vanilla* neural network approach.¹ The goal of this

¹ We define a *vanilla* neural network to be a fully connected, two-layer sigmoidal network with a full set of input variables and a fixed number of hidden units that is trained on a data window of fixed length with backprop and early stopping using a validation set. No variable selection, pruning, regularization, or committee techniques are used.

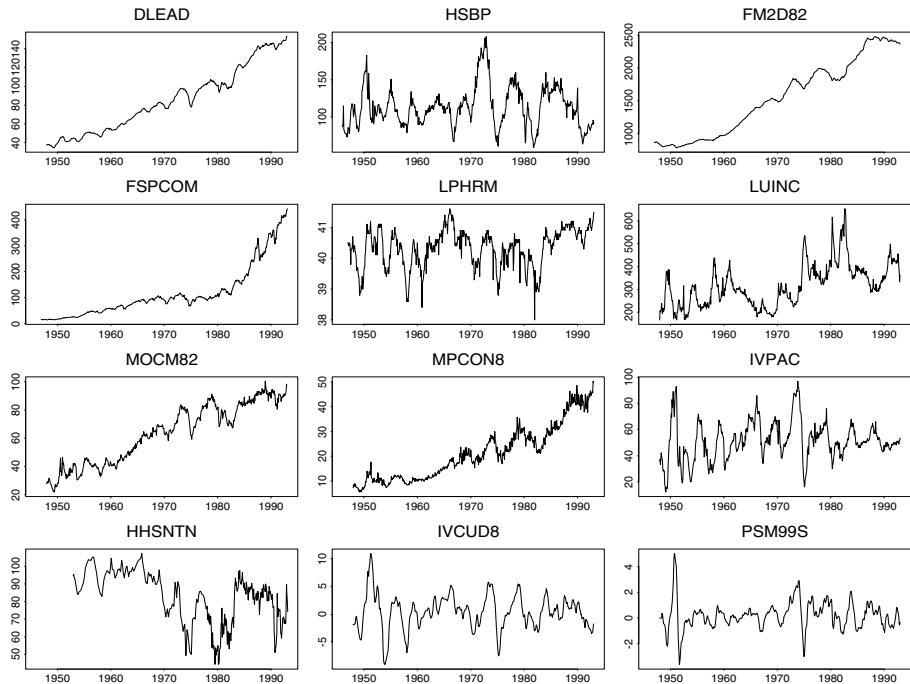


Fig. 16.3. The U.S. Index of Leading Indicators (DLEAD) and its 11 component series as currently defined. The Leading Index is a key tool for forecasting business cycles. The input variables for the IP forecasting models included transformed versions of DLEAD and several of its components [27]. The difficulty of macroeconomic forecasting is again evident, due to the high noise levels and erratic behaviors of DLEAD and its components. (Note that the component series included in DLEAD have been changed several times during the past 47 years. The labels for the various series are those defined in Citibase: HSBP denotes housing starts, FM2D82 is M2 money supply, FSPCOM is the Standard & Poors 500 stock index, and so on.)

work is to construct models with minimal prediction risk (expected test set error). The techniques that we are developing and testing are described below. Given the brief nature of this survey, I have not attempted to provide an exhaustive list of the many relevant references in the literature.

Hyperparameter Selection: Hyperparameters are parameters that appear in the training objective function, but not in the network itself. Examples include the regularization parameter, the training window length, and robust scale parameters. Examples of varying the regularization parameter and the training window length for a 12 month IP forecasting model are shown in figures 16.4 and 16.5. Varying the regularization parameter trades off bias and variance, while varying the training window length trades off noise and nonstationarity.

Table 16.1. Comparative summary of normalized prediction errors for rates of return on Industrial Production for the period January 1980 to January 1990 as presented in [27]. The four model types were trained on data from January 1950 to December 1979. The neural network models significantly outperform the trivial predictors and linear models. For each forecast horizon, the normalization factor is the variance of the target variable for the training period. Nonstationarity in the IP series makes the test errors for the trivial predictors larger than 1.0. In subsequent work, we have obtained substantially better results for the IP problem [20, 38, 42, 28, 45].

Prediction Horizon (Months)	Trivial (Average of Training Set)	Univariate AR(14) Model Iterated Pred.	Multivariate Linear Reg. Direct Pred.	Sigmoidal Nets w/ PC Pruning Direct Pred.
1	1.04	0.90	0.87	0.81
2	1.07	0.97	0.85	0.77
3	1.09	1.07	0.96	0.75
6	1.10	1.07	1.38	0.73
9	1.10	0.96	1.38	0.67
12	1.12	1.23	1.20	0.64

Input Variable Selection and Pruning: Selecting an informative set of input variables and an appropriate representation for them (“features”) is critical to the solution of any forecasting problem. The variable selection and representation problem is part of the overall model selection problem. Variable selection procedures can be either model-independent or model-dependent. The Delta Test, a model independent procedure, is a nonparametric statistical algorithm that selects meaningful predictor variables by direct examination of the data set [36]. Other model-independent techniques make use of the mutual information [4, 5, 46] or joint mutual information [46]. Sensitivity-based pruning (SBP) techniques are model-dependent algorithms that prune unnecessary or harmful input variables from a trained network [33, 30, 25, 42, 21]. Sensitivity based pruning methods are described in greater detail in section 16.4.6.

Model Selection and Pruning: A key technique for controlling the bias / variance tradeoff for noisy problems is to select the size and architecture of the network. For two-layer networks, this includes selecting the number of internal units, choosing a connectivity structure, and pruning unneeded nodes, weights, or weight matrix eigennodes. A constructive algorithm for selecting the number of internal units is sequential network construction (SNC) [2, 30, 25]. Techniques for pruning weights and internal nodes include sensitivity-based pruning methods like optimal brain damage (OBD) [19] and optimal brain surgeon (OBS) [15]. Our recently-proposed supervised principal components pruning (PCP) method [20] prunes weight matrix eigennodes, rather than weights. Since PCP does not require training to a local minimum, it can be used with early stopping. It has computational advantages over OBS, and can outperform OBD when input variables or hidden node activities are noisy and correlated. Figure 16.6 shows reductions in prediction

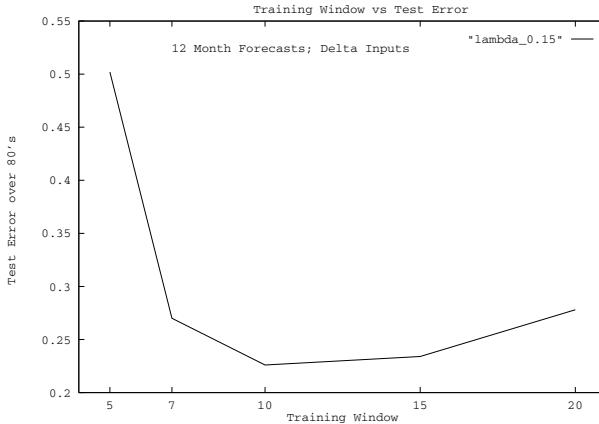


Fig. 16.4. Example of the *Noise / Nonstationary Tradeoff* and selection of the best training window, in this case 10 years [38, 28]. The longer training windows of 15 and 20 years yield higher test set error due to the model bias induced by nonstationarity. The shorter training windows of 5 and 7 years have significantly higher errors due to model variance resulting from noise in the data series and smaller data sets. The test errors correspond to models trained with the best regularization parameter 0.15 indicated in figure 16.5.

errors obtained by using PCP on a set of IP forecasting models. Section 16.4 describes the model selection problem and the use of estimates of prediction risk such as *nonlinear cross-validation* (NCV) to guide the selection process in greater detail.

Better Regularizers: Introducing biases in a model via regularization or pruning reduces model variance and can thus reduce prediction risk. Prediction risk can be best minimized by choosing appropriate biases. One such set of biases are smoothing constraints. We have proposed new classes of smoothing regularizers for both feedforward and recurrent networks [29, 45] that often yield better performance than the standard weight decay approach. These are described in greater detail in section 16.3.

Committee Forecasts: Due to the extremely noisy nature of economic time series, the control of forecast variance is a critical issue. One approach for reducing forecast variance is to average the forecasts of a committee of models. Researchers in economics have studied and used combined estimators for a long time, and generally find that they outperform their component estimators and that unweighted averages tend to outperform weighted averages, for a variety of weighting methods [12, 44, 6]. Reductions of prediction error variances obtained by unweighted committee averaging for a selection of different IP forecasting models are shown in figure 16.7.

Model Interpretation and Visualization: It is important not only to be able to make accurate forecasts, but to also understand what factors influence the forecasts that are made. This can be accomplished via the sensitivity analyses described in sections 16.4.6 and 16.4.8 and the visualization tool presented in section 16.4.8.

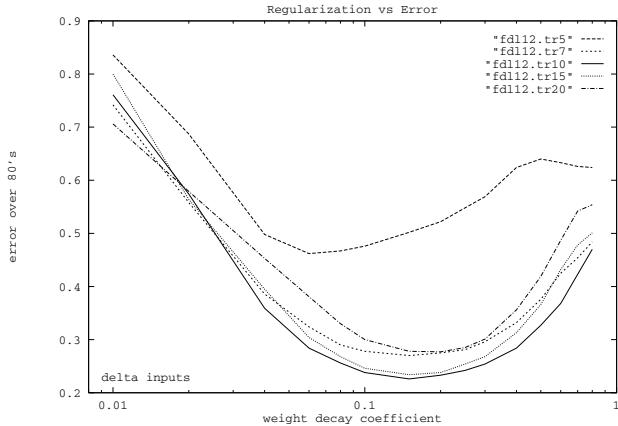


Fig. 16.5. Example of the effect of regularization (weight decay) parameter on test error [38, 28]. The five curves are for training windows of length 5, 7, 10, 15, and 20 years. The *Bias / Variance Tradeoff* is clearly evident in all the curves; the minimum test set errors occur for weight decay parameters of order 0.1. Larger errors due to bias occur for larger weight decay coefficients, while larger errors due to model variance occur for smaller values of the coefficient.

16.3 Smoothing Regularizers for Better Generalization

Introducing biases in a model via regularization or pruning reduces model variance and can thus reduce prediction risk (see also chapters 2-6). Prediction risk can be best minimized by choosing appropriate biases. Quadratic weight decay [37, 18, 17], the standard approach to regularization used in the neural nets community, is an *ad hoc* function of the network weights. Weight decay is *ad hoc* in the sense that it imposes direct constraints on the weights independent of the nature of the function being learned or the parametrization of the network model. A more principled approach is to require that the function $f(W, \mathbf{x})$ learned by the network be smooth. This can be accomplished by penalizing the m^{th} order curvature of $f(W, \mathbf{x})$. The regularization or penalty functional is then the smoothing integral

$$S(W, m) = \int d^d \mathbf{x} \Omega(\mathbf{x}) \left\| \frac{d^m f(W, \mathbf{x})}{d \mathbf{x}^m} \right\|^2, \quad (16.1)$$

where $\Omega(\mathbf{x})$ is a weighting function and $\| \cdot \|$ denotes the Euclidean tensor norm.² Since numerical computation of (16.1) generally requires expensive Monte Carlo integrations and is therefore impractical during training, we have derived algebraically simple approximations and bounds to $S(W, m)$ for feedforward

² The relation of this type of smoothing functional to radial basis functions has been studied by [10]. However, the approach developed in that work does not extend to standard feedforward sigmoidal networks, which are a special case of projective basis function networks (PBF's).

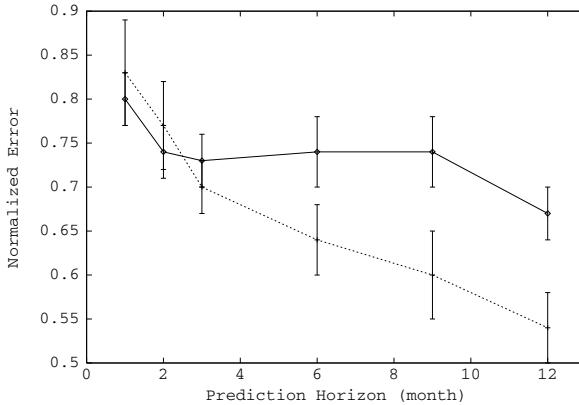


Fig. 16.6. Prediction errors for two sets of neural network models for 12 month returns for IP, with (dotted line) and without (solid line) Supervised Principal Components Pruning (PCP) [20]. Each data point is the mean error for 11 nets, while the error bars represent one standard deviation. Statistically significant improvements in prediction performance are obtained for the 6, 9, and 12 month prediction horizons by using the PCP algorithm to reduce the network complexities. While techniques like optimal brain damage and optimal brain surgeon prune weights from the network, PCP reduces network complexity and hence model variance by pruning eigenvalues of the weight matrices. Unlike the unsupervised use of principal components, PCP removes those eigenvalues that yield the greatest reduction in estimated prediction error.

networks that can be easily evaluated at each training step [29]. For these new classes of algebraically simple m^{th} -order smoothing regularizers for networks of projective basis functions (PBF's) $f(W, \mathbf{x}) = \sum_{j=1}^N u_j g[\mathbf{x}^T \mathbf{v}_j + v_{j0}] + u_0$, $W = (u, v)$ with general transfer functions $g[\cdot]$, the regularizers are:

$$R_G(W, m) = \sum_{j=1}^N u_j^2 \|\mathbf{v}_j\|^{2m-1} \quad \text{Global Form}$$

$$R_L(W, m) = \sum_{j=1}^N u_j^2 \|\mathbf{v}_j\|^{2m} \quad \text{Local Form.}$$

Our empirical experience shows that these new smoothing regularizers typically yield better prediction accuracies than standard weight decay.

In related work, we have derived an algebraically-simple regularizer for recurrent nets [45]. This regularizer can be viewed as a generalization of the first order Tikhonov stabilizer (the $m = 1$ local form above) to dynamic models. For two layer networks with recurrent connections described by

$$Y(t) = \mathbf{g}(AY(t - \tau) + VX(t)) , \quad \hat{Z}(t) = UY(t) ,$$

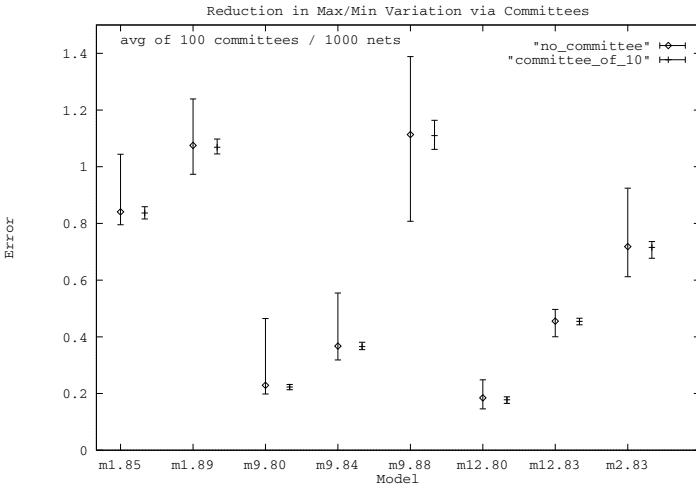


Fig. 16.7. Reduction in error variance for prediction of the U.S.Index of Industrial Production by use of combining forecasts (or committees) [38, 28]. Abscissa points are various combinations of prediction horizon and test period. For example, “m12.80” denotes networks trained to make 12 month forecasts on the ten years prior to 1979 and tested by making true *ex ante* forecasts on the year 1980. Performance metric is normalized mean square error (NMSE) computed over the particular year. All training sets have length 10 years. For each point, bars show range of values for either 1000 individual models, or 100 committees of 10. The individual networks each have three sigmoidal internal units, one linear output, and typically a dozen or so input variables selected by the δ -test from an initial set of 48 candidate variables.

the training criterion with the regularizer is

$$\mathcal{E} = \frac{1}{N} \sum_{t=1}^N \|Z(t) - \hat{Z}(W, I(t))\|^2 + \lambda \rho_\tau^2(W),$$

where $W = \{U, V, A\}$ is the network parameter set, $Z(t)$ are the targets, $I(t) = \{X(s), s = 1, 2, \dots, t\}$ represents the current and all historical input information, N is the size of the training data set, $\rho_\tau^2(W)$ is the regularizer, and λ is a regularization parameter. The closed-form expression for the regularizer for time-lagged recurrent networks is:

$$\rho_\tau(W) = \frac{\gamma \|U\| \|V\|}{1 - \gamma \|A\|} \left[1 - e^{\frac{\gamma \|A\| - 1}{\tau}} \right],$$

where $\|\cdot\|$ is the Euclidean matrix norm and γ is a factor which depends upon the maximal value of the first derivatives of the internal unit activations $g(\cdot)$. Simplifications of the regularizer can be obtained for simultaneous recurrent nets ($\tau \mapsto 0$), two-layer feedforward nets, and one layer linear nets. We have

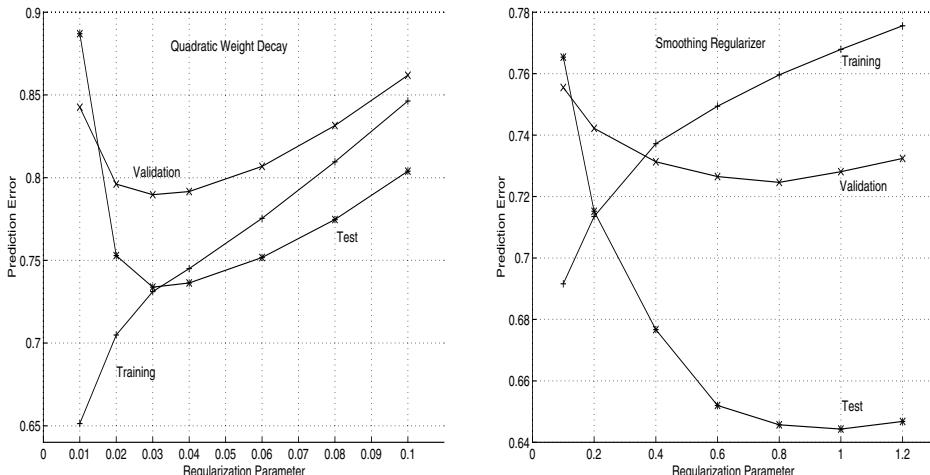


Fig. 16.8. Regularization parameter vs. normalized prediction errors for the task of predicting the one month rates of change of the U.S. Index of Industrial Production [45]. The example given is for a recurrent network trained with standard weight decay (left) or with the new recurrent smoothing regularizer (right). For standard weight decay, the optimal regularization parameter is 0.03 corresponding to a test error of 0.734. For the new smoothing regularizer, the optimal regularization parameter which leads to the least validation error is 0.8 corresponding to a test error of 0.646. The new recurrent regularizer thus yields a 12% reduction in test error relative to that obtained using quadratic weight decay.

successfully tested this regularizer in a number of case studies and found that it performs better than standard quadratic weight decay. A comparison of this recurrent regularizer to quadratic weight decay for 1 month forecasts of IP is shown in figure 16.8.

16.4 Model Selection and Interpretation

In this section, we provide a more in-depth discussion of several issues and techniques for neural network model selection, including the problem of selecting inputs. We describe techniques for selecting architectures via estimates of the prediction risk, especially the *generalized prediction error (GPE)* and *nonlinear cross-validation (NCV)*. We present *sensitivity-based pruning (SBP)* methods for selecting input variables, and demonstrate the use of these methods for predicting the U.S. Index of Industrial Production. Finally, we discuss some approaches to model interpretation and visualization that enable an understanding of economic relationships.

16.4.1 Improving Forecasts via Architecture and Input Selection

For the discussion of architecture selection in this paper, we focus on the most widely used neural network architecture, the two-layer *perceptron* (or *backpropagation*) network. The response function for such a network with architecture λ having I_λ input variables, H_λ internal (hidden) neurons, and a single output is:

$$\hat{y}_\lambda(\mathbf{x}) = h(u_0 + \sum_{j=1}^{H_\lambda} u_j g(v_{j0} + \sum_{i=1}^{I_\lambda} v_{ji} x_i)) . \quad (16.2)$$

Here, h and g are typically sigmoidal nonlinearities, the v_{ji} and v_{j0} are input weights and thresholds, the u_j and u_0 are the output weights and threshold, and the index λ is an abstract label for the specific two layer perceptron network architecture. While we consider for simplicity this restricted class of perceptron networks in this section, our approach can be easily generalized to networks with multiple outputs and multiple layers.

For two layer perceptrons, the *architecture selection problem* is to find a good, near-optimal architecture λ for modeling a given data set. The architecture λ is characterized by the number of hidden units H_λ , the subset of input variables I_λ , and the subset of weights u_j and v_{ji} that are non-zero. If all of the u_j and v_{ji} are non-zero, the network is referred to as *fully connected*. Since an exhaustive search over the space of possible architectures is impossible, the procedure for selecting this architecture requires a heuristic search. See Figure 16.9 for examples of heuristic search strategies and [25] and [30] for additional discussion.

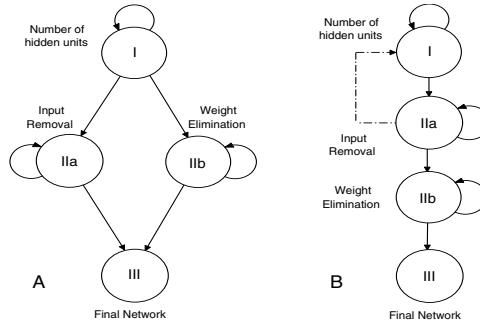


Fig. 16.9. Heuristic Search Strategies: After selecting the number of hidden units H_λ , the input removal and weight elimination can be carried out in parallel (A) or sequentially (B). In (B), the selection of the number of hidden units and removal of inputs may be iterated (dashed line).

In this section, we focus on selecting the “best subset” of input variables for predicting the U.S. Index of Industrial Production. In order to avoid an exhaustive search over the exponentially-large space of architectures obtained by considering

all possible combinations of inputs, we employ a directed search strategy using the *sensitivity-based input pruning (SBP)* algorithm (see section 16.4.6).

16.4.2 Architecture Selection via the Prediction Risk

The notion of “best fits” can be captured via an objective criterion; such as *maximum a posteriori probability (MAP)*, minimum *Bayesian information criterion (BIC)*, *minimum description length (MDL)*, or *generalization ability*. The generalization ability can be defined precisely as the *prediction risk* P_λ , the expected performance of an estimator in predicting new observations. In this section, we use the prediction risk as our selection criterion for two reasons. First, it is straightforward to compute, and second, it provides more information than MAP, BIC, or MDL, since it tells us how much confidence to put in predictions produced by our best model.

Consider a set of observations $D = \{(\mathbf{x}_j, t_j); j = 1 \dots N\}$ that are assumed to be generated as $t_j = \mu(\mathbf{x}_j) + \epsilon_j$ where $\mu(\mathbf{x})$ is an unknown function, the inputs \mathbf{x}_j are drawn independently with an unknown stationary probability density function $p(\mathbf{x})$, the ϵ_j are independent random variables with zero mean ($\bar{\epsilon} = 0$) and variance σ_ϵ^2 , and the t_j are the observed target values. The learning or regression problem is to find an estimate $\hat{\mu}_\lambda(\mathbf{x}; D)$ of $\mu(\mathbf{x})$ given the data set D from a class of predictors or models $\mu_\lambda(\mathbf{x})$ indexed by λ . In general, $\lambda \in \Lambda = (S, A, W)$, where $S \subset X$ denotes a chosen subset of the set of available input variables X , A is a selected architecture within a class of model architectures \mathcal{A} , and W are the adjustable parameters (weights) of architecture A .

The *prediction risk* P_λ (defined above) can be approximated by the expected performance on a finite test set. P_λ can be defined for a variety of loss functions. For the special case of squared error, it is:

$$P_\lambda = \int d\mathbf{x} p(\mathbf{x}) [\mu(\mathbf{x}) - \hat{\mu}(\mathbf{x})]^2 + \sigma_\epsilon^2 \quad (16.3)$$

$$\approx E\left\{ \frac{1}{N} \sum_{j=1}^N (t_j^* - \hat{\mu}_\lambda(\mathbf{x}_j^*))^2 \right\} \quad (16.4)$$

where (\mathbf{x}_j^*, t_j^*) are new observations that were not used in constructing $\hat{\mu}_\lambda(\mathbf{x})$. In what follows, we shall use P_λ as a measure of the generalization ability of a model. Our strategy is to choose an architecture λ in the model space Λ which minimizes an estimate of the prediction risk P_λ .

16.4.3 Estimation of Prediction Risk

Since it is not possible to exactly calculate the prediction risk P_λ given only a finite sample of data, we have to estimate it. The restriction of limited data makes the model selection and prediction risk estimation problems more difficult. This is the typical situation in economic forecasting, where the time series are short.

A limited training set results in a more severe bias/variance (or underfitting vs overfitting) tradeoff (see e.g. [9]), so the model selection problem is both more challenging and more crucial. In particular, it is easier to overfit a small training set, so care must be taken to select a model that is not too large. Also, limited data sets make prediction risk estimation more difficult if there is not enough data available to hold out a sufficiently large independent test sample. In such situations, one must use alternative approaches which enable the estimation of prediction risk from the training data, such as data resampling and algebraic estimation techniques. Data resampling methods include nonlinear refinements of ν -fold cross-validation (*NCV*) and bootstrap estimation, while algebraic estimates (in the regression context) include Akaike's final prediction error (*FPE*) [1], for linear models, and the recently proposed generalized prediction error (*GPE*) for nonlinear models [31, 24, 25], which is identical to the independently-derived network information criterion [34]. For comprehensive discussions of prediction risk estimation, see [8, 16, 43, 25].

16.4.4 Algebraic Estimates of Prediction Risk

Predicted Squared Error for Linear Models. For linear regression models with the squared error loss function, a number of useful algebraic estimates for the prediction risk have been derived. These include the well known generalized cross-validation (*GCV*) [7, 11] and Akaike's final prediction error (*FPE*) [1] formulas:

$$GCV_\lambda = ASE_\lambda \frac{1}{\left(1 - \frac{Q_\lambda}{N}\right)^2} \quad FPE_\lambda = ASE_\lambda \left(\frac{1 + \frac{Q_\lambda}{N}}{1 - \frac{Q_\lambda}{N}} \right). \quad (16.5)$$

Q_λ denotes the number of weights of model λ (ASE_λ denotes the average squared error). Note that although *GCV* and *FPE* are slightly different for small sample sizes, they are asymptotically equivalent for large N :

$$GCV_\lambda \approx FPE_\lambda \approx ASE_\lambda \left(1 + 2 \frac{Q_\lambda}{N} \right) \quad (16.6)$$

A more general expression of predicted squared error (*PSE*) is:

$$PSE_\lambda = ASE_\lambda + 2\hat{\sigma}^2 \frac{Q_\lambda}{N}, \quad (16.7)$$

where $\hat{\sigma}^2$ is an estimate of the noise variance in the data. Estimation strategies for (16.7) and its statistical properties have been analyzed by [3]. *FPE* is obtained as special case of *PSE* by setting $\hat{\sigma}^2 \equiv ASE_\lambda/(N - Q_\lambda)$. See [8, 16, 43] for tutorial treatments.

It should be noted that *PSE*, *FPE* and *GCV* are asymptotically unbiased estimates of the prediction risk for the neural network models considered here under certain conditions. These are: (1) the noise ϵ_j in the observed targets t_j is

independent and identically distributed, (2) the resulting model is unbiased, (3) weight decay is not used, and (4) the nonlinearity in the model can be neglected. For PSE, we further require that an asymptotically unbiased estimate of $\hat{\sigma}^2$ is used. In practice, however, essentially all neural network fits to data will be biased and/or have significant nonlinearity.

Although PSE, FPE and GCV are asymptotically unbiased only under the above assumptions, they are much cheaper to compute than NCV since no re-training is required.

Generalized Prediction Error (GPE) for Nonlinear Models. The predicted squared error PSE, and therefore the final prediction error FPE, are special cases of the *generalized prediction error GPE* [31, 24, 25]. We present an abbreviated description here.

GPE estimates the prediction risk for biased nonlinear models which may use general loss functions and include regularizers such as weight decay. The algebraic form is

$$GPE_\lambda \equiv \mathcal{E}_{\lambda\text{train}} + \frac{2}{N} \text{tr } \hat{V} \hat{G}_\lambda , \quad (16.8)$$

where $\mathcal{E}_{\lambda\text{train}}$ is the training set error (average value of loss function on training set), \hat{V} is a nonlinear generalization of the estimated *noise covariance matrix* of the observed targets, and \hat{G}_λ is the estimated *generalized influence matrix*, a nonlinear analog of the standard influence or hat matrix.

GPE can be expressed in an equivalent form as:

$$GPE_\lambda = \mathcal{E}_{\lambda\text{train}} + 2 \hat{\sigma}_{\text{eff}}^2 \frac{\hat{Q}_{\lambda\text{eff}}}{N} , \quad (16.9)$$

where $\hat{Q}_{\lambda\text{eff}} \equiv \text{tr } \hat{G}$ is the estimated *effective number of model parameters*, and $\hat{\sigma}_{\text{eff}}^2 \equiv (\text{tr } \hat{V} \hat{G}) / (\text{tr } \hat{G})$ is the estimated effective noise variance in the data. For nonlinear and/or regularized models, $\hat{Q}_{\lambda\text{eff}}$ is generally not equal to the number of weights Q_λ .

When the noise in the target variables is assumed to be independent with uniform variance and the squared error loss function is used, (16.9) simplifies to:

$$GPE_\lambda = ASE_\lambda + 2\hat{\sigma}^2 \frac{\hat{Q}_{\lambda\text{eff}}}{N} . \quad (16.10)$$

Note that replacing $\hat{Q}_{\lambda\text{eff}}$ with Q_λ gives the expression for PSE. Various other special cases of (16.8) and (16.10) have been derived by other authors and can be found in [8, 16, 43]. GPE was independently derived by [34], who called it the Network Information Criterion (NIC).

16.4.5 NCV: Cross-Validation for Nonlinear Models

Cross-validation (CV) is a sample re-use method for estimating prediction risk; it makes maximally efficient use of the available data. A perturbative refinement of CV for nonlinear models is called *nonlinear cross-validation (NCV)* [25, 30].

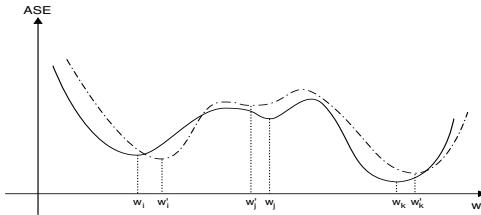


Fig. 16.10. A nonlinear model can have many local minima in the error function. Each local minimum w_i , w_j and w_k (solid curve) corresponds to a different set of parameters and thus to a different model. Training on a different finite sample of data or retraining on a subsample, as in nonlinear cross-validation, gives rise to a slightly different error curve (dashed) and perturbed minima w'_i , w'_j and w'_k . Variations due to data sampling in error curves and their minima are termed *model variance*.

Let the data D be divided into ν randomly selected disjoint subsets D_j of roughly equal size: $\cup_{j=1}^{\nu} D_j = D$ and $\forall i \neq j$, $D_i \cap D_j = \emptyset$. Let N_j denote the number of observations in subset D_j . Let $\hat{\mu}_{\lambda(D_j)}(\mathbf{x})$ be an estimator trained on all data except for $(\mathbf{x}, t) \in D_j$. Then, the cross-validation average squared error for subset j is defined as

$$CV_{D_j}(\lambda) = \frac{1}{N_j} \sum_{(\mathbf{x}_k, t_k) \in D_j} (t_k - \hat{\mu}_{\lambda(D_j)}(\mathbf{x}_k))^2 . \quad (16.11)$$

These are averaged over j to obtain the ν -fold cross-validation estimate of prediction risk:

$$CV(\lambda) = \frac{1}{\nu} \sum_j CV_{D_j}(\lambda) . \quad (16.12)$$

Typical choices for ν are 5 and 10. Leave-one-out CV is obtained in the limit $\nu = N$. CV is a nonparametric estimate of the prediction risk that relies only on the available data.

The frequent occurrence of multiple minima in nonlinear models (see Figure 16.10), each of which represents a different predictor, requires a refinement of the cross-validation procedure. This refinement, *nonlinear cross-validation (NCV)*, is illustrated in Figure 16.11 for $\nu = 5$.

A network is trained on the entire data set D to obtain a model $\hat{\mu}_{\lambda}(\mathbf{x})$ with weights W_0 . These weights are used as the starting point for the ν -fold cross-validation procedure. Each subset D_j is removed from the training data in turn. The network is re-trained using the remaining data starting at W_0 (rather than using random initial weights). Under the assumption that deleting a subset from the training data does not lead to a large difference in the locally-optimal weights, the retraining from W_0 “perturbs” the weights to obtain W_i , $i = 1 \dots \nu$. The Cross-Validation error computed for the “perturbed models” $\hat{\mu}_{\lambda(D_j)}(\mathbf{x})$ thus estimates the prediction risk for the model with locally-optimal weights W_0 as desired, and not the performance of other predictors at other local minima.

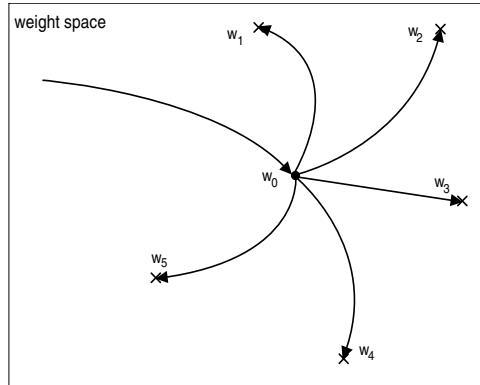


Fig. 16.11. Illustration of the computation of 5-fold *nonlinear cross-validation* (*NCV*). First, the network is trained on all data to obtain weights w_0 which are used as starting point for the cross-validation. Each data subset $D_i, i = 1 \dots 5$ is removed from the training data D in turn. The network is trained, starting at W_0 , using the remaining data. This “perturbs” the weights to obtain w_i . The test error of the “perturbed model” w_i is computed on the hold-out sample D_i . The average of these errors is the 5-fold CV estimate of the prediction risk for the model with weights w_0 .

If the network would be trained from random initial weights for each subset, it could converge to a different minimum corresponding to W_i different from the one corresponding to W_0 . This would correspond to a different model. Thus, starting from W_0 assures us that the cross-validation estimates the prediction risk for a particular model in question corresponding to $W \approx W_0$.

16.4.6 Pruning Inputs via Directed Search and Sensitivity Analysis

Selecting a “best subset” of input variables is a critical part of model selection for forecasting. This is especially true when the number of available input series is large, and exhaustive search through all combinations of variables is computationally infeasible. Inclusion of irrelevant variables not only does not help prediction, but can reduce forecast accuracy through added noise or systematic bias. A model-dependent method for input variable selection is *sensitivity-based pruning (SBP)* [41, 32, 30, 25]. Extensions to this approach are presented in [21]. With this algorithm, candidate architectures are constructed by evaluating the effect of removing an input variable from the fully connected network. These are ranked in order of increasing training error. Inputs are then removed following a “Best First” strategy, i.e. selecting the input that, when removed, increases the training error least.

The SBP algorithm computes a *sensitivity measure* S_i to evaluate the change in training error that would result if input x_i were removed from the network.

One such sensitivity measure is the *delta error*, defined as:

$$\text{Delta Error (DE)} \quad S_i = \frac{1}{N} \sum_{j=1}^N S_{ij} \quad (16.13)$$

where S_{ij} is the sensitivity computed for exemplar x_j . Since there are usually many fewer inputs than weights, a direct evaluation of S_i is feasible:

$$S_{ij} = SE(\bar{x}_i, w_\lambda) - SE(x_{ij}, w_\lambda) \quad (16.14)$$

$$\bar{x}_i = \frac{1}{N} \sum_{j=1}^N x_{ij}$$

S_i measures the effect on the training squared error (SE) of replacing the i^{th} input x_i by its average \bar{x}_i for all exemplars (replacement of a variable by its average value removes its influence on the network output).

Note that in computing S_i , no retraining is done in evaluating $SE(\bar{x}_i, w_\lambda)$. Also note that it is not sufficient to just set $x_{ij} = 0 \forall j$, because the value of the bias of each hidden unit was determined during training and would not be offset properly by setting the input arbitrarily to zero. Of course, if the inputs are normalized to have zero mean prior to training, then setting an input variable to zero is equivalent to replacing it by its mean.

If S_i is large, the network error will be significantly increased by replacing the i^{th} input variable with its mean. If S_i is small, we need the help of other measures to decide whether the i^{th} input variable is useful. Three additional sensitivity measures [21] can be computed based on perturbing an input or a hidden variable and monitoring network output variations:

$$\begin{aligned} \text{Average Gradient (AG)} \quad S_i &= \frac{1}{N} \sum_{j=1}^N \frac{\partial f^{(j)}}{\partial x_i}, \\ \text{Average Absolute Gradient (AAG)} \quad S_i &= \frac{1}{N} \sum_{j=1}^N \left| \frac{\partial f^{(j)}}{\partial x_i} \right|, \\ \text{RMS Gradient (RMSG)} \quad S_i &= \sqrt{\frac{1}{N} \sum_{j=1}^N \left[\frac{\partial f^{(j)}}{\partial x_i} \right]^2}, \end{aligned}$$

where for ease of notation we define $f(\mathbf{x}) \equiv \hat{\mu}_\lambda(\mathbf{x})$ and $f^{(j)} \equiv f(x_{1j}, \dots, x_{ij}, \dots, x_{dj})$ is the network output given the j^{th} input data pattern.

These three sensitivity measures together offer useful information. If S_i^{AG} is positive and large, then on average the change of the direction of the network output f is the same as that of the i^{th} input variable. If S_i^{AG} is negative and has large magnitude, the change of the direction of f on average is opposite to that of the i^{th} input variable. When S_i^{AG} is close to zero, we can not get much

information from this measure. If S_i^{AAG} is large, the output f is sensitive to the i^{th} input variable; if S_i^{AAG} is small, f is not sensitive to the i^{th} input variable. If S_i^{RMSG} is very different from S_i^{AAG} , the i^{th} input series could be very noisy and have a lot of outliers.

16.4.7 Empirical Example

As described in [42], we construct neural network models for predicting the rate of change of the U.S. Index of Industrial Production (IP). The prediction horizon for the IP results presented here is 12 months. Following previous work [27, 20], the results reported here use networks with three sigmoidal units and a single linear output unit.

The data set consists of monthly observations of IP and other macroeconomic and financial series for the period from January 1950 to December 1989. The data set thus has a total of 480 exemplars. Input series are derived from around ten raw time series, including IP, the Index of Leading Indicators, the Standard & Poors 500 Index, and so on. Both the “unfiltered” series and various “filtered” versions are considered for inclusion in the model, for a total of 48 possible input variables. The target series and all 48 candidate input series are normalized to zero mean and unit standard deviation. Figures 16.12 and 16.13 show the results of the sensitivity analysis for the case where the training-set consists of 360 exemplars randomly chosen from the 40 year period; the remaining 120 monthly observations constitute the test-set.

Local optima for the number of inputs are found at 15 on the FPE curve and 13 on the NCV curve. Due to the variability in the FPE and NCV estimates (readily apparent in figure 16.13 for NCV), we favor choosing the first good local minimum for these curves rather than a slightly better global minimum. This local minimum for NCV corresponds to a global minimum for the test error. Choosing it leads to a reduction of 35 (from 48 to 13) in the number of input series and a reduction in the number of network weights from 151 to 46. Inclusion of additional input variables, while decreasing the training error, does not improve the test-set performance.

This empirical example demonstrates the effectiveness of the *sensitivity-based pruning (SBP)* algorithm guided by an estimate of prediction risk, such as the *nonlinear cross-validation (NCV)* algorithm, for selecting a small subset of input variables from a large number of available inputs. The resulting network models exhibit better prediction performances, as measured by either estimates of prediction risk or errors on actual test sets, than models that make use of all 48 input series.

16.4.8 Gaining Economic Understanding through Model Visualization

Although this chapter has focussed on data-driven time series models for economic forecasting, it is possible to extract information from these models about

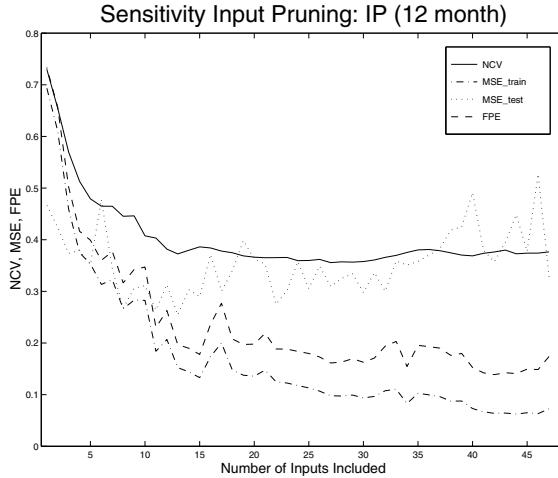


Fig. 16.12. Sensitivity-Based Pruning (SBP) method for selecting a subset of input variables for a neural net forecasting model [42]. The original network was trained on all 48 input variables to predict the 12 month percentage changes in Industrial Production (IP). The variables have been ranked in order of decreasing importance according to a sensitivity measure. The input variables are pruned one-by-one from the network; at each stage, the network is retrained. The figure shows four curves: the Training Error, Akaike Final Prediction Error (FPE), *Nonlinear Cross-Validation Error* (NCV) [30, 25], and the actual Test Error. NCV is used as a selection criterion and suggests that only 13 of the variables should be included. NCV predicts the actual test error quite well relative to FPE.

the structure of the economy. The sensitivity analyses presented above in section 16.4.6 provide a global understanding about which inputs are important for predicting quantities of interest, such as the business cycle. Further information can be gained, however, by examining the evolution of sensitivities over time [21].

Sensitivity analysis performed for an individual exemplar provides information about which input features play an important role in producing the current prediction. Two sensitivity measures for individual exemplars can be defined as:

$$\begin{aligned} \text{Delta Output (DO)} \quad S_i &= \Delta f_i^{(j)} \\ &\equiv f(x_{1j}, \dots, x_{ij}, \dots, x_{dj}) \\ &\quad - f(x_{1j}, \dots, \bar{x}_i, \dots, x_{dj}), \\ \text{Output Gradient (OG)} \quad S_i &= \frac{\partial f^{(j)}}{\partial x_{ij}}, \end{aligned}$$

where as above we define $f^{(j)} \equiv f(x_{1j}, \dots, x_{ij}, \dots, x_{dj})$. If S_i^{DO} or S_i^{OG} is large, then the i^{th} variable plays an important role in making the current prediction,

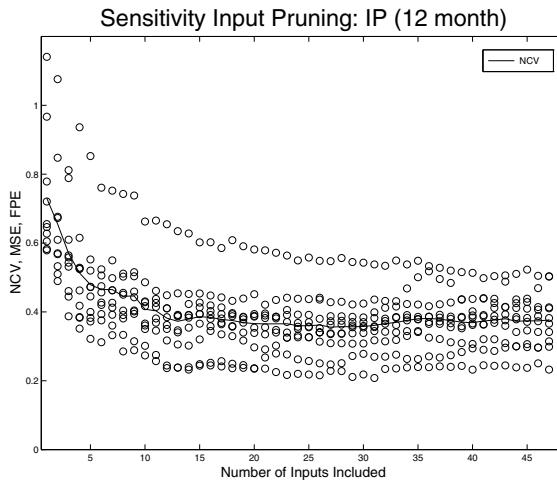


Fig. 16.13. Sensitivity Input Pruning for IP (12 month prediction horizon). The figure illustrates the spread in test-set error for each of the 10 subsets used to calculate NCV (denoted by circles). The NCV error is the average of these test-set errors.

and slightly changing the value of the variable may cause a large change in the network output. Figure 16.14 gives an example of a graphical display of the individual exemplar sensitivities.

Using this graphical display, we can observe which input variables play important roles in producing the current forecast, or which input variables, when we change them, can significantly increase or decrease the forecast error. We can also observe how the roles of different input variables change through time. For example, in Figure 16.14, for exemplars with indices from 56 to 60, the third input variable has large negative sensitivity measures. Starting with the 65th exemplar, the sixth input variable starts to play an important role, and this lasts until the 71st exemplar. This kind of display provides insight into the dynamics of the economy, as learned from the data by the trained neural network model.

16.5 Discussion

In concluding this brief survey of the algorithms for improving forecast accuracy with neural networks, it is important to note that many other potentially useful techniques have been proposed (see also chapter 17). Also, the empirical results presented herein are intended to be illustrative, rather than definitive. Further work on both the algorithms and forecasting models may yield additional improvements. As a final comment, I would like to emphasize that given the difficulty of macroeconomic forecasting, no single technique for reducing prediction risk is sufficient for obtaining optimal performance. Rather, a combination of techniques is required.

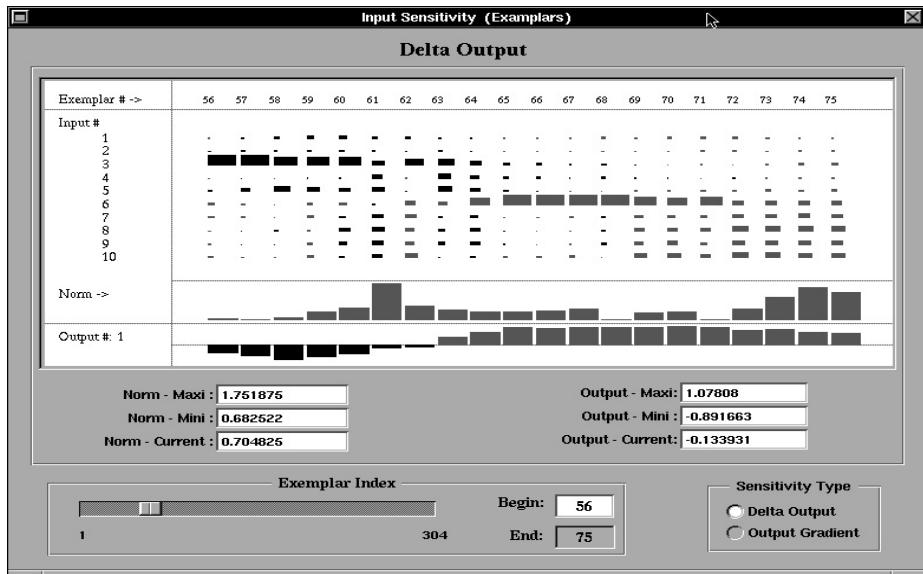


Fig. 16.14. Sensitivity analysis results for individual exemplars for a 10 input model for predicting the U.S. Index of Industrial Production [21]. Black and gray represent negative and positive respectively. The size of a rectangle represents the magnitude of a value. The monthly time index changes along the horizontal direction. The indices of input variables are plotted vertically. This type of graph shows which input variables are important for making forecasts at various points in time. This enables an understanding of economic relationships.

Acknowledgements. The author wishes to thank Todd Leen, Asriel Levin, Yuansong Liao, Hong Pi, Steve Rehfuss, Thorsteinn Rögnvaldsson, Matthew Saffell, Joachim Utans, Lizhong Wu and Howard Yang for their many contributions to this research. This chapter is an expanded version of [26]. This work was supported at OGI by ONR/ARPA grants N00014-92-J-4062 and N00014-94-1-0071, NSF grants CDA-9309728 and CDA-9503968, and at Nonlinear Prediction Systems by DARPA contracts DAAH01-92-CR361 and DAAH01-96-CR026.

References

- [1] Akaike, H.: Statistical predictor identification. *Ann. Inst. Statist. Math.* 22, 203–217 (1970)
- [2] Ash, T.: Dynamic node creation in backpropagation neural networks. *Connection Science* 1(4), 365–375 (1989)
- [3] Barron, A.: Predicted squared error: a criterion for automatic model selection. In: Farlow, S. (ed.) *Self-Organizing Methods in Modeling*. Marcel Dekker, New York (1984)
- [4] Battiti, R.: Using mutual information for selecting features in supervised neural net learning. *IEEE Trans. on Neural Networks* 5(4), 537–550 (1994)

- [5] Bonnlander, B.: Nonparametric selection of input variables for connectionist learning. Technical report, PhD Thesis. Department of Computer Science, University of Colorado (1996)
- [6] Clemen, R.T.: Combining forecasts: A review and annotated bibliography. International Journal of Forecasting (5), 559–583 (1989)
- [7] Craven, P., Wahba, G.: Smoothing noisy data with spline functions: Estimating the correct degree of smoothing by the method of generalized cross-validation. Numer. Math. 31, 377–403 (1979)
- [8] Eubank, R.L.: Spline Smoothing and Nonparametric Regression. Marcel Dekker, Inc. (1988)
- [9] Geman, S., Bienenstock, E., Doursat, R.: Neural networks and the bias/variance dilemma. Neural Computation 4(1), 1–58 (1992)
- [10] Girosi, F., Jones, M., Poggio, T.: Regularization theory and neural network architectures. Neural Computation 7, 219–269 (1995)
- [11] Golub, G., Heath, H., Wahba, G.: Generalized cross validation as a method for choosing a good ridge parameter. Technometrics 21, 215–224 (1979)
- [12] Granger, C.W.J., Newbold, P.: Forecasting Economic Time Series, 2nd edn. Academic Press, San Diego (1986)
- [13] Granger, C.W.J., Teräsvirta, T.: Modelling Nonlinear Economic Relationships. Oxford University Press (1993)
- [14] Hamilton, J.D.: Time Series Analysis. Princeton University Press (1994)
- [15] Hassibi, B., Stork, D.G.: Second order derivatives for network pruning: Optimal brain surgeon. In: Hanson, S.J., Cowan, J.D., Giles, C.L. (eds.) Advances in Neural Information Processing Systems, vol. 5, pp. 164–171. Morgan Kaufmann Publishers, San Mateo (1993)
- [16] Hastie, T.J., Tibshirani, R.J.: *Generalized Additive Models*. Monographs on Statistics and Applied Probability, vol. 43. Chapman and Hall (1990)
- [17] Hoerl, A.E., Kennard, R.W.: Ridge regression: applications to nonorthogonal problems. Technometrics 12, 69–82 (1970)
- [18] Hoerl, A.E., Kennard, R.W.: Ridge regression: biased estimation for nonorthogonal problems. Technometrics 12, 55–67 (1970)
- [19] LeCun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In: Touretzky, D.S. (ed.) Advances in Neural Information Processing Systems, vol. 2. Morgan Kaufmann Publishers (1990)
- [20] Levin, A.U., Leen, T.K., Moody, J.E.: Fast pruning using principal components. In: Cowan, J., Tesauro, G., Alspector, J. (eds.) Advances in Neural Information Processing Systems, vol. 6. Morgan Kaufmann Publishers, San Francisco (1994)
- [21] Liao, Y., Moody, J.E.: A neural network visualization and sensitivity analysis toolkit. In: Amari, S.I., Xu, L., Chan, L.-W., King, I., Leung, K.-S. (eds.) Proceedings of the International Conference on Neural Information Processing, pp. 1069–1074. Springer-Verlag Singapore Pte. Ltd. (1996)
- [22] Litterman, R.B.: Forecasting with Bayesian vector autoregressions – five years of experience. Journal of Business and Economic Statistics 4(1), 25–38 (1986)
- [23] Moody, J.: Challenges of Economic Forecasting: Noise, Nonstationarity, and Nonlinearity. Invited talk presented at Machines that Learn., Snowbird Utah (April 1994)
- [24] Moody, J.: The effective number of parameters: an analysis of generalization and regularization in nonlinear learning systems. In: Moody, J.E., Hanson, S.J., Lippmann, R.P. (eds.) Advances in Neural Information Processing Systems, vol. 4, pp. 847–854. Morgan Kaufmann Publishers, San Mateo (1992)

- [25] Moody, J.: Prediction risk and neural network architecture selection. In: Cherkassky, V., Friedman, J.H., Wechsler, H. (eds.) *From Statistics to Neural Networks: Theory and Pattern Recognition Applications*. Springer (1994)
- [26] Moody, J.: Macroeconomic Forecasting: Challenges and Neural Network Solutions. In: *Proceedings of the International Symposium on Artificial Neural Networks*, Hsinchu, Taiwan (1995) (Invited keynote address)
- [27] Moody, J., Levin, A., Rehfuss, S.: Predicting the U.S. index of industrial production. *Neural Network World* 3(6), 791–794 (1993); Special Issue: *Proceedings of Parallel Applications in Statistics and Economics 1993*
- [28] Moody, J., Rehfuss, S., Saffell, M.: Macroeconomic forecasting with neural networks (1999) (manuscript in preparation)
- [29] Moody, J., Rögnvaldsson, T.: Smoothing regularizers for projective basis function networks. In: *Proceedings of Advances in Neural Information Processing Systems*, NIPS 1996, vol. 9. MIT Press, Cambridge (1997)
- [30] Moody, J., Utans, J.: Architecture selection strategies for neural networks: Application to corporate bond rating prediction. In: Refenes, A.N. (ed.) *Neural Networks in the Capital Markets*. John Wiley & Sons (1994)
- [31] Moody, J.E.: Note on generalization, regularization and architecture selection in nonlinear learning systems. In: Juang, B.H., Kung, S.Y., Kamm, C.A. (eds.) *Neural Networks for Signal Processing*, pp. 1–10. IEEE Signal Processing Society (1991)
- [32] Moody, J.E., Utans, J.: Principled architecture selection for neural networks: Application to corporate bond rating prediction. In: Moody, J.E., Hanson, S.J., Lippmann, R.P. (eds.) *Advances in Neural Information Processing Systems*, vol. 4, pp. 683–690. Morgan Kaufmann Publishers, San Mateo (1992)
- [33] Mozer, M.C., Smolensky, P.: Skeletonization: A technique for trimming the fat from a network via relevance assessment. In: Touretzky, D.S. (ed.) *Advances in Neural Information Processing Systems*, vol. 1. Morgan Kaufmann Publishers, San Mateo (1990)
- [34] Murata, N., Yoshizawa, S., Amari, S.: Network information criterion – determining the number of hidden units for an artificial neural network model. *IEEE Transactions on Neural Networks* 5(6), 865–872 (1994)
- [35] Natter, M., Haefke, C., Soni, T., Otruba, H.: Macroeconomic forecasting using neural networks. In: *Neural Networks in the Capital Markets 1994* (1994)
- [36] Pi, H., Peterson, C.: Finding the embedding dimension and variable dependencies in time series. *Neural Computation*, 509–520 (1994)
- [37] Plaut, D., Nowlan, S., Hinton, G.: Experiments on learning by back propagation. Technical Report CMU-CS-86-126, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania (1986)
- [38] Rehfuss, S.: Macroeconomic forecasting with neural networks (1994) (unpublished simulations)
- [39] Rehkugler, H., Zimmermann, H.G. (eds.): *Neuronale Netze in der Ökonomie*. Verlag Vahlen (1994)
- [40] Swanson, N.R., White, H.: A model selection approach to real-time macroeconomic forecasting using linear models and artificial neural networks. Discussion paper, Department of Economics, Pennsylvania State University (1995)
- [41] Utans, J., Moody, J.: Selecting neural network architectures via the prediction risk: Application to corporate bond rating prediction. In: *Proceedings of the First International Conference on Artificial Intelligence Applications on Wall Street*. IEEE Computer Society Press, Los Alamitos (1991)

- [42] Utans, J., Moody, J., Rehfuss, S.: Selecting input variables via sensitivity analysis: Application to predicting the U.S. business cycle. In: Proceedings of Computational Intelligence in Financial Engineering. IEEE Press (1995)
- [43] Wahba, G.: Spline models for observational data. CBMS-NSF Regional Conference Series in Applied Mathematics (1990)
- [44] Winkler, R.L., Makridakis, S.: The combination of forecasts. Journal of Royal Statistical Society (146) (1983)
- [45] Wu, L., Moody, J.: A smoothing regularizer for feedforward and recurrent networks. Neural Computation 8(2) (1996)
- [46] Yang, H., Moody, J.: Input variable selection based on joint mutual information. Technical report, Department of Computer Science, Oregon Graduate Institute (1998)

How to Train Neural Networks^{*}

Ralph Neuneier and Hans Georg Zimmermann

Siemens AG, Corporate Technology, D-81730 München, Germany

{Ralph.Neuneier,Georg.Zimmermann}@mchp.siemens.de

http://w2.siemens.de/zfe_nn/homepage.html

Abstract. The purpose of this paper is to give a guidance in neural network modeling. Starting with the preprocessing of the data, we discuss different types of network architecture and show how these can be combined effectively. We analyze several cost functions to avoid unstable learning due to outliers and heteroscedasticity. The Observer - Observation Dilemma is solved by forcing the network to construct smooth approximation functions. Furthermore, we propose some pruning algorithms to optimize the network architecture. All these features and techniques are linked up to a complete and consistent training procedure (see figure 17.25 for an overview), such that the synergy of the methods is maximized.

17.1 Introduction

The use of neural networks in system identification or regression tasks is often motivated by the theoretical result that in principle a three layer network can approximate any structure contained in a data set [14]. Consequently, the characteristics of the available data determine the quality of the resulting model. The authors believe that this is a misleading point of view, especially, if the amount of useful information that can be extracted from the data is small. This situation arises typically for problems with a low signal to noise ratio and a relative small training data set at hand. Neural networks are such a rich class of functions, that the control of the optimization process, i. e. the learning algorithm, pruning, architecture, cost functions and so forth, is a central part of the modeling process. The statement that “The neural network solution is not better than [any classical] method” has been used too often in order to describe the results of neural network modeling. At any rate, the assessment of this evaluation presupposes a precise knowledge of the procedure involved in the neural network solution that has been achieved. This is the case because a great variety of additional features and techniques can be applied at the different stages of the process to prevent all the known problems like overfitting and sensitivity to outliers concerning neural networks. Due to the lack of a general recipe, one can often find a statement

* Previously published in: Orr, G.B. and Müller, K.-R. (Eds.): LNCS 1524, ISBN 978-3-540-65311-0 (1998).

declaring that the quality of the neural network model depends strongly on the person who generated the model, which is usually perceived as negative. In contrast, we consider the additional features an outstanding advantage of neural networks compared to classical methods, which typically do not allow such a sophisticated control of their optimization. The aim of our article is to provide a set of techniques to efficiently exploit the capabilities of neural networks. More important, these features will be combined in such a way that we will achieve a maximal effect of synergy by their application.

First, we begin with the preprocessing of the data and define a network architecture. Then, we analyze the interaction between the data and the architecture (Observer - Observation Dilemma) and discuss several pruning techniques to optimize the network topology. Finally, we conclude by combining the proposed features into a unified training procedure (see fig. 17.25 in section 17.8).

Most of the paper is relevant to nonlinear regression in general. Some considerations are focussed on time series modeling and forecasting. All the proposed approaches have been tested on diverse tasks we have to solve for our clients, e. g. forecasting financial markets. The typical problem can be characterized by a relative small set of very noisy data and a high dimensional input vector to cover the complexity of the underlying dynamical system. The paper gives an overview of the unified training procedure we have developed to solve such problems.

17.2 Preprocessing

Besides the obvious scaling of the data (in the following abbreviated by $\text{scale}(\cdot)$), which transforms the different time series such that each series has a mean value of zero and a statistical variance of one, some authors have proposed complicated preprocessing functions. In the field of financial forecasting, these functions are often derived from technical analysis in order to capture some of the underlying dynamics of the financial markets (see [37] and [25] for some examples).

After many experiments with real data, we have settled with the following simple transformations. If the original time series which has been selected as an input, is changing very slowly with respect to the prediction horizon, i. e. there is no clearly identifiable mean reverting equilibrium, then an indicator for the inertia and an information of the driving force has been proven to be very informative. The inertia can be described by a momentum (relative change, eq. 17.1) and the force by the acceleration of the time series (normalized curvature, eq. 17.2). If we have a prediction horizon of n steps into the future the original time series x_t is transformed in the following way:

$$\text{momentum: } \tilde{x}_t = \text{scale} \left(\frac{x_t - x_{t-n}}{x_{t-n}} \right), \quad (17.1)$$

$$\text{force: } \hat{x}_t = \text{scale} \left(\frac{x_t - 2x_{t-n} + x_{t-2n}}{x_{t-n}} \right). \quad (17.2)$$

In eq. 17.1, the relative difference is computed to eliminate exponential trends which, for example, may be caused by inflationary influences. Using only the pre-processing functions of eq. 17.1 typically leads to poor models which only follow obvious trends. The forces, i. e. the transformations by eq. 17.2, are important to characterize the turning points of the time series.

A time series may be fast in returning to its equilibrium state after new information has entered the market, as is the case for most prices of goods and stock rates. In this case, we substitute eq. 17.2 by a description of the forces which drive the price back to the estimated equilibrium. A simple way to estimate the underlying price equilibrium is to take the average over some past values of the time series. Instead of using the relative difference between the estimate and the current value, we look at the difference between the equilibrium and the past value, which lies in the middle of the averaging window. In our example, this is formulated as

$$\hat{x}_t = \text{scale} \left(\frac{x_{t-n} - \frac{1}{2n+1} \sum_{\tau=0}^{2n} x_{t-\tau}}{x_{t-n}} \right). \quad (17.3)$$

Note that in eq. 17.3 we use x_{t-n} instead of the present value x_t leading to an estimation of the equilibrium centered around x_{t-n} . Since, in fact, we are interested in measuring the tension between a price level and the underlying market equilibrium, this estimation offers a more appropriate description at the cost of ignoring the newest possible point information. This concept, known as *mean reverting* dynamics in economics, is analog to the behavior of a pendulum in physics.

17.3 Architectures

We will present several separate architectural building blocks (figures 17.1 to 17.7), which will finally be combined into a unified neural network architecture for time series analysis (figure 17.8). Most of the structural elements can be used in general regression.

17.3.1 Net Internal Preprocessing by a Diagonal Connector

In order to have potentially very different inputs on a comparable scale, we standardize the input data to zero mean and unit variance. A problem with this common approach is that outliers in the input can have a large impact. This is particularly serious for data that contain large shocks, as in finance and economics. To avoid this problem, we propose an additional, *net internal* (short for *network internal*) preprocessing of the inputs by scaling them according to

$$x' = \tanh(wx). \quad (17.4)$$

The implementation of this preprocessing layer is shown in fig. 17.1. This layer has the same number of hidden units as the input layer and uses standard tanh

squashing functions. The particular weight matrix between the input layer and the preprocessing layer is only a square diagonal matrix.

For short term forecasts (e. g. modeling daily returns of stock markets), we typically initialize the weights with the value of 0.1 to ensure that the tanh is in its linear range, which in turn ensures that the external inputs pass through essentially unchanged. If interested in long term models (e. g. six month forecast horizon), we prefer to start with an initial value of 1. The reason is that monthly data are typically more contaminated by “non-economical” effects like political shocks. The large initial weight values eliminate such outliers from the beginning.

The weights in the diagonal connector are restricted to be positive to avoid fluctuations of the sign of x' during training. This constraint keeps eq. 17.4 as a monotonic transformation with the ability to limit outliers. No bias should be used for the preprocessing layer to prevent numerical ambiguities. We found that these additional constraints improve the training stability of the preprocessing layer.

These weights will be adapted during training in the same way as all the other weights within the network. In practice, we observe both growing and shrinking values for the weights. Growing values cause a larger proportion of the input range to be compressed by the squashing function of the tanh. Very small values of diagonal elements indicate the option to prune the corresponding input.

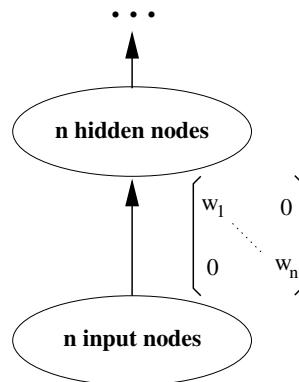


Fig. 17.1. Net internal preprocessing to limit the influence of outliers and to eliminate unimportant inputs

17.3.2 Net Internal Preprocessing by a Bottleneck Network

It is an old idea to use a bottleneck network to shrink the dimensionality of the input vector. This technique was not only used to build encoders [1], but also to perform a principle component analysis [26]. Using a bottleneck as an internal preprocessing building block within a larger network offers the opportunity for us to compress the input information. In addition, the tanh squashing function of the bottleneck layer acts as a limiter of the outliers in the data.

In a first attempt, one may implement a bottleneck architecture as an input-hidden-output-layer sub-network using the inputs also as targets on the output level. An additional connector from the hidden layer is connected to the remaining parts of the network. This design allows the compression of input signals, but there are two major disadvantages implied. If we apply input pruning to the original inputs the decompression becomes disordered. Furthermore, adding noise to the inputs becomes more complex because the disturbed inputs have also to be used as targets at the output layer.

A distinct approach of the bottleneck sub-networks which avoids these difficulties is indicated in fig. 17.2. On the left side of the diagram 17.2 we have the typical compressor-decompressor network. The additional connection is a frozen identity matrix, which duplicates the input layer to the output layer. Since the target for this layer is set to zero, the output of the compressor-decompressor network will adopt the negative values of the inputs in order to compensate for the identical copy of the inputs. Even though the sign of the input signals has been reversed, this circumstance does not imply any consequence for the information flow through the bottleneck layer. This architecture allows an appropriate elimination of inputs as well as the disturbance of the input signals by artificial noise.

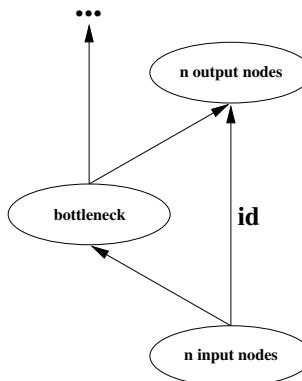


Fig. 17.2. Net internal preprocessing cluster using a bottleneck

In our experiments, we observed that it is possible to train the bottleneck in parallel with the rest of the network. As a consequence, the bottleneck can be shrunk as long as the error on the training set remains at the same level as it would without a compression. As will be described in section 17.6 it is important to use a dynamically controlled input noise as a regularization method during learning. The architecture of fig. 17.2 allows the combination of noise and net internal preprocessing by data compression.

17.3.3 Squared Inputs

Following the suggestion of G. Flake in [10], we also supply the network with the squared values of the inputs which leads to an integration of global and local decision making. The processing of the original inputs and the squared inputs within a tanh activation function enables the network to act as a combination of widely-known neural networks using sigmoidal activation function (MLP) and radial basis function networks (RBF). The output of a typical three-layer MLP is

$$y = \sum_j v_j \tanh \left(\sum_i w_{ji} x_i - \theta_j \right), \quad (17.5)$$

whereas the output of a RBF computes to

$$y = \sum_j v_j e^{-\frac{1}{2} \sum_i \left(\frac{x_i - \mu_{ji}}{\sigma_{ji}} \right)^2}. \quad (17.6)$$

Some research papers typically deal with the comparison of these two types of basis functions whereas Flake proposes the following combining approach

$$y = \sum_j v_j \tanh \left(\sum_i w_{ji} x_i + u_{ji} x_i^2 - \theta_j \right), \quad (17.7)$$

which covers the MLP of eq. 17.5 and simultaneously approximates the RBF output eq. 17.6 to a sufficient level.

Nevertheless, we propose some minor changes to Flake's approach by taking the square of the internally preprocessed inputs instead of supplying the squared original inputs separately. That way, we take advantage of the preprocessing cluster because it limits the outliers. Furthermore, pruning inputs leads to a simultaneous elimination of its linear and squared transformation. A possible implementation is shown in fig. 17.3. The connector **id** is a fixed identity matrix. The cluster above this connector uses square as the nonlinear activation function. The next hidden cluster is able to create MLP and RBF structures with its tanh squashing function.

One might think that the addition of a new connector between the square and the hidden layer with all its additional weights would boost the overfitting. Our experience indicates just the opposite. Furthermore, when optimizing a usual MLP, one typically stops training before the error on the training set gets too small (*early stopping*) in order to increase the probability of good results on the test set. We have observed, that the error on the training set can be very small while at the same time achieving good test results, even if one does not use weight pruning. Our experiences can be understood on the basis of the local modeling features of the architecture. That is, local artifacts in the input space can be encapsulated, so that they have no more global influence which may lead to bad generalization performance. This is especially true for high dimensional input spaces.

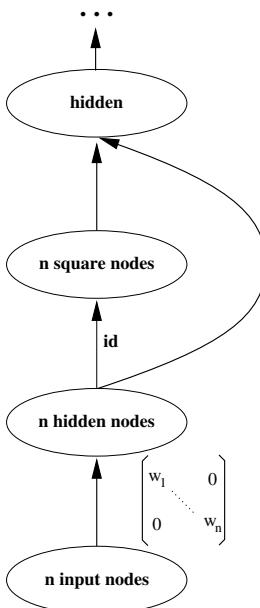


Fig. 17.3. Net internal preprocessing cluster and the square cluster, which produces the input signals for following clusters. Note, that the connector from bias to “hidden” is required but suppressed for visual clarity.

17.3.4 Interaction Layer

This section summarizes the articles of Weigend and Zimmermann in [36, 32]. In most applications of neural networks in financial engineering the number of inputs is huge (of the order of a hundred), but only a single output is used. This situation can be viewed as a large “inverted” funnel; the hard problem is that the only information available for learning is that of a single output. This is one of the reasons for the “data-hungriness” of single-output neural networks, i. e. a large set of training data is often required in order to distinguish nonlinearities from noise. The flip-side of the coin is that small data sets need to be regularized, thus only allowing an identification of a simple model, which implies a bias towards linear models. This circumstance is not to be confused with the bias towards linear models which is a consequence of some techniques for avoiding overfitting such as early stopping or weight decay.

One approach for increasing the information flow from the output side to the input side is to increase the number of output units. In the simplest case, two output units can be used, one to predict the return, the other one to predict the sign of the return [34].

Since we have to model dynamical systems, we would like to provide enough information to characterize the state of the autonomous part of the dynamics on the output side, similar to Takens’ theorem for the notion of state on the input

side [11]. The idea of multiple output units has recently been popularized in the connectionist community in a non-time series context by Caruana [6].

The present paper is concerned with time series, so the embedding of the output can be done analogously to the input side of a tapped “delay” line, indicated in fig. 17.4 as the *point prediction layer*.

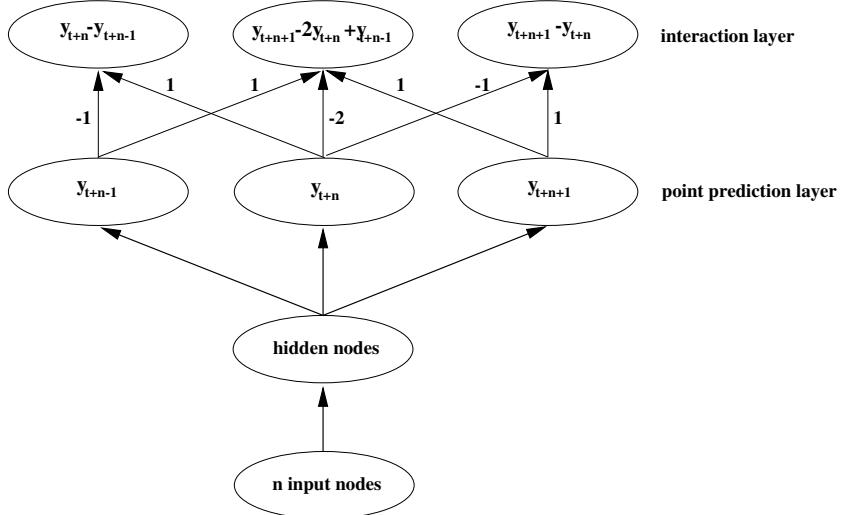


Fig. 17.4. Point predictions followed by the interaction layer

For the forecast we are interested in y_{t+n} , the n -step ahead forecast of variable y . Additionally, we also want to predict y_{t+n-1} and y_{t+n+1} . However, after experimenting with this architecture, we do not consider it to be much of a benefit to our aim, because there seems only very little interaction between the behavior of the outputs as reflected through the implicit transfer by sharing hidden units.

This prompted us to introduce an explicit second output layer, the *interaction layer*. The additional layer computes the next-neighbor derivatives ($y_{t+n} - y_{t+n-1}$) and ($y_{t+n+1} - y_{t+n}$), as well as the curvature ($y_{t+n+1} - 2y_{t+n} + y_{t+n-1}$).

Differences between neighbors are encoded as fixed weights between point prediction and interaction layer, so they do not enlarge the number of parameters to be estimated. The overall cost function is the sum of all six contributions where all individual contributions are weighted evenly. If the target values are not properly scaled, it may be useful to give equal contribution to the error and to scale each error output by the average error of that output unit.

If the point forecasts were perfect, the interaction layer would have no effect at all. To explain the effect of non-zero errors, consider fig. 17.5. Both predictions of the three points have the same pointwise errors at each of the three neighboring

points. However, both the slopes and the curvature are correct in Model 1 (they don't give additional errors), but do add to the errors for Model 2.¹

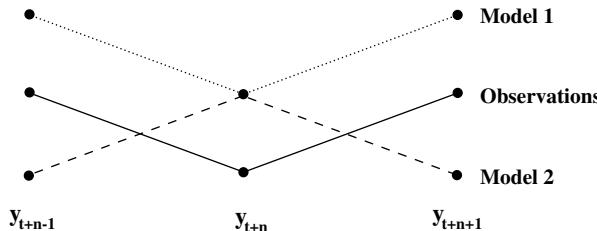


Fig. 17.5. Geometric interpretation of the effect of the interaction layer on the cost function. Given are three curves connecting points at three adjacent steps in time. Whereas Model 1 (connected by dashed line) and Model 2 (connected by the dotted line) have identical pointwise errors to the observations (connected by the solid line), taking derivatives and curvatures into account favors Model 1.

The principle of the interaction layer can also be used to model further relationships on the output side. Let us take as an example a forecasting model of several exchange rates. Between all these forecasts we should guarantee that there is no arbitrage between the assets at the forecast horizon. In other words, at y_{t+n} there is no way to change money in a closed loop and have a positive return. These intermarket relationships can be realized in form of an interaction layer.

In general, the control of intermarket relationships becomes more important if we proceed from forecasting models to portfolio analysis models. In the latter the correct forecast of the interrelationships of the assets is more important than a perfect result on one of the titles.

17.3.5 Averaging

Let us assume, that we have m sub-networks for the same learning task. Different solutions of the sub-networks may be caused by instabilities of the learning or by a different design of the sub-networks. It is a well known principle that averaging the output of several networks may give us a better and more stable result [24, 4].

These advantages can be clearly seen if we define an average error function

$$E_{\text{average}} = \frac{1}{T} \sum_{t=1}^T \left[\left(\frac{1}{m} \sum_{i=1}^m y_{i,t+n} \right) - y_{t+n}^d \right]^2 \quad (17.8)$$

¹ Note that in the case of a quadratic error function, the interaction layer can be substituted by a single output layer of point predictions, combined with a positive definite non-diagonal quadratic form as target function.

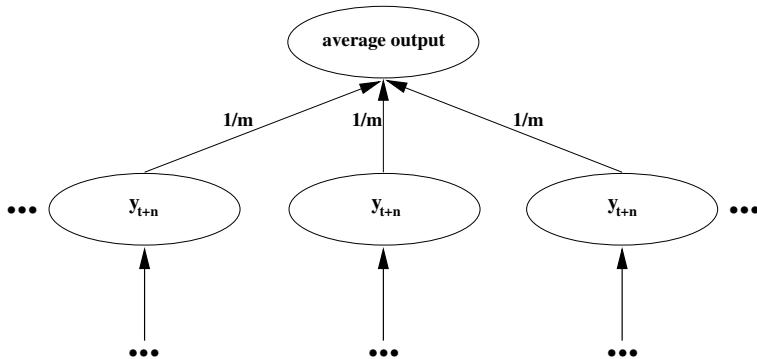


Fig. 17.6. Averaging of several point predictions for the same forecast horizon

with y_{t+n}^d as the target pattern, $y_{i,t+n}$ as the output of sub-network i , m as the number of subnetworks, and T as the number of training patterns. Assuming that the errors of the sub-networks are uncorrelated,

$$\frac{1}{T} \sum_{t=1}^T (y_{i,t+n} - y_{t+n}^d)(y_{j,t+n} - y_{t+n}^d) = 0 , \quad \forall i \neq j \quad (17.9)$$

leads to

$$E_{\text{average}} = \frac{1}{m} \left(\frac{1}{m} \sum_{i=1}^m \left(\frac{1}{T} \sum_{t=1}^T (y_{i,t+n} - y_{t+n}^d)^2 \right) \right) \quad (17.10)$$

$$= \frac{1}{m} \text{ average}(E_{\text{sub-networks}}) . \quad (17.11)$$

According to this argument, the error due to the uncertainty of the training can be reduced. Finally, it is to be noted that averaging adds no additional information about the specific application of our network.

17.3.6 Regularization by Random Targets

It is known in the neural network community that the addition of random targets can improve the learning behavior². An architectural realization of this idea is shown fig. 17.7.

In economical applications we typically use large input vectors in order to capture all probably relevant indicators. The resulting large number of parameters if connecting the input layer to a hidden layer of an appropriate size is the source

² The authors would appreciate any useful citation.

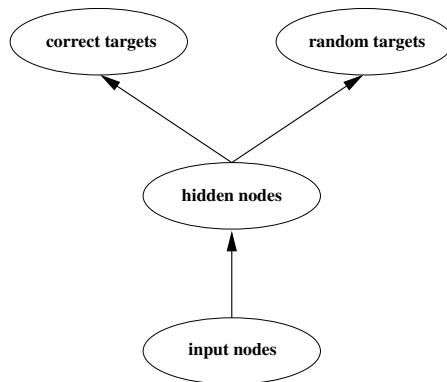


Fig. 17.7. A simple architecture using the original targets (upper left branch) and the random targets for regularization (upper right branch)

of overfitting. To partly remedy this effect, one can extend the neural network with additional but random targets. As a decreasing error of this outputs can only be achieved by memorizing these random events, this technique absorbs a part of the overparametrization of the network. It should not be confused with artificial noise on the output side because the additional patterns are randomly selected according a probability distribution before the training and are held fixed during the learning.

The question if enough or too many additional random targets are supplied can be answered by observing the learning behavior. During training the error with respect to these random targets is steadily decreasing until convergence. If the number of additional targets is too small one may observe overfitting effects (e. g. using a validation set) on the real targets after this convergence. On the other hand, if the the number of additional targets is too large the learning slows down.

One may suspect the parameters focusing on the random targets may have a unpredictable effect on the generalization set but we could not observe such a behavior. If using squared inputs (see sec. 17.3.3), their local learning possibilities supports this by encapsulating local artifacts.

17.3.7 An Integrated Network Architecture for Forecasting Problems

As a next step in our investigation, we suppose our task to be an economic forecasting model. In the following section we integrate the architectural building blocks discussed above into an eleven layer network specialized to fulfill our purpose (see fig. 17.8). As indicated in section 17.2, the net external preprocessing contains at least two inputs per original time series: the momentum in the form

of the relative difference (eq. 17.1) and a force indicator in the form of a curvature or mean reverting description (eq. 17.2).

The lowest part of the network shows the net internal preprocessing by a diagonal connector. Alternatively, we could also use the bottleneck network. By the square layer, we allow our network to cover the difference and similarity analysis of MLP- and RBF- networks. The signals from the internally preprocessed inputs and their squared values are used as inputs, weighted by the associated parameters, to the hidden embedding and hidden force layers (see fig. 17.8).

In contrast to typical neural networks, the upper part of the net is organized in a new way. The underlying dynamical system is supposed to be characterized by an estimation of different features around the forecast horizon. We distinguish these indicators by two particular characteristics, embeddings and forces. The forecast of these different features has been separated in two branches of the network in order to avoid interferences during training. Instead of directly forecasting our final target which has the form:

$$\frac{y_{t+n} - y_t}{y_t}, \quad (17.12)$$

we use the following indicators as targets in these two output layers:
embeddings:

$$\begin{aligned} u_i &= \frac{y_{t+n+i} + y_{t+n} + y_{t+n-i}}{3y_t} - 1 \quad i = 1, \dots, m \\ v_i &= \frac{\frac{1}{2i+1} \sum_{j=-i}^i y_{t+n+j} - y_t}{y_t} \quad i = 1, \dots, m \end{aligned} \quad (17.13)$$

forces:

$$\begin{aligned} r_i &= \frac{-y_{t+n+i} + 2y_{t+n} - y_{t+n-i}}{3y_t} \quad i = 1, \dots, m \\ s_i &= \frac{y_{t+n} - \frac{1}{2i+1} \sum_{j=-i}^i y_{t+n+j}}{y_t} \quad i = 1, \dots, m \end{aligned}$$

Target u_i describes a normalized 3-point embedding with respect to our forecast horizon $t + n$ while v_i represents a complete average around $t + n$ versus present time. The forces r_i and s_i are formulated as curvatures or mean reverting (see also section 17.2). Similar to the embeddings they describe features of a dynamical system with an increasing width. The different widths in turn allow a complete characterization of the time series, analogous to the characterization by points in Takens' Theorem [29].

The motivation of this design is contained in the step from the embedding and force output layers to the multiforecast output layer. We have

$$\begin{aligned}\frac{y_{t+n} - y_t}{y_t} &= u_i + r_i \quad i = 1, \dots, m \\ \frac{y_{t+n} - y_t}{y_t} &= v_i + s_i \quad i = 1, \dots, m.\end{aligned}\tag{17.14}$$

That means, we get $2m$ estimations of our final target by simply adding up the embeddings and their associated forces in pairs. In the network this is easy to realize by two identity connectors. After the multiforecast layer we can add an averaging connector to get our final output. This averaging can be done by fixed weights $1/2m$ or by learning after finishing the training of the lower network.

The control-embedding and control-force clusters are motivated by the following observation. On the embedding / force output level the network has to estimate only slightly different features depending on the width parameter. Neural networks have a tendency to estimate these features too similarly. To counteract this behavior we have to add two additional clusters which control the difference between the individual outputs inside the embedding and the force cluster. Thus, the network has not only to learn u_i and u_{i+1} on the embedding output level but also the difference $u_i - u_{i+1}$ on the control embedding level. The same is valid for v_i, r_i, s_i .

From a formal viewpoint the multiforecast cluster, the control-embedding cluster and control-force cluster are interaction layers supporting the identification of the underlying dynamical system in form of embeddings and forces. Keep in mind, that although the full network seems to be relatively complex, most of the connectors are fixed during training. Those are only used to produce the appropriate information flows whose design is the real focus of this section.

Our proposed network design allows for an intuitive evaluation of the dimension of the target indicators *embeddings* and *forces*: how to choose m in eq. 17.13? Start with a relative large m and train the network to the minimal error on the training set as will be described in section 17.8. Then train only the weights of the up to now fixed connector between cluster multi-forecast and average forecast (fig. 17.8). If the dimension m has been chosen too large training may lead to such weights which suppress the long range embeddings and forces. Thus, it is possible to achieve an optimal dimension m of embedding and forces. For the case of a six month forecast horizon we were successful with a value of $m = 6$.

The eleven layer network automatically integrates aspects of section 17.3.6 concerning random targets. If the dimension m has been chosen too large, then the extreme target indicators act as random targets. On the other hand, if the forecast problem is characterized by high noise in the short term, the indicators for smaller m values generate random targets. Thus, choosing m too large does not harm our network design as discussed in section 17.3.6, but can improve generalization by partly adsorbing the overparametrization of the network.

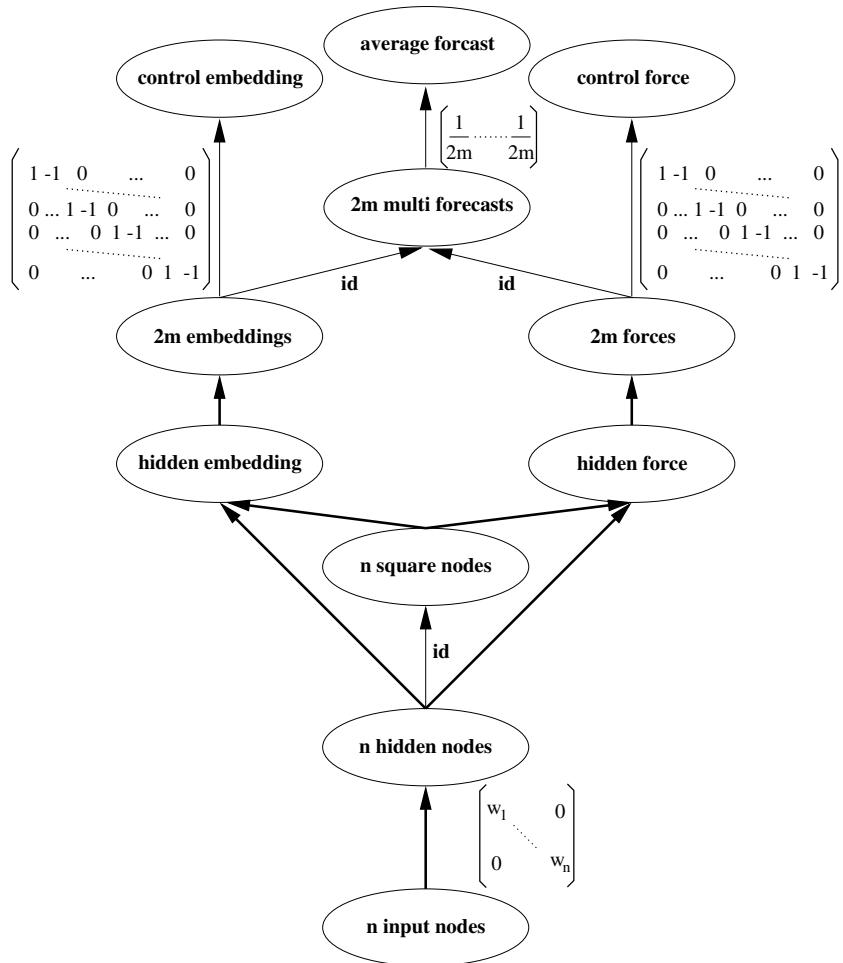


Fig. 17.8. The integrating eleven layer architecture

The different forecasts o_i of the multiforecast cluster in fig. 17.8 can be used to estimate structural instability s of the network model by

$$s = \sum_{i=1}^{2m} |o_i - \bar{o}| \quad \text{with} \quad \bar{o} = \frac{1}{2m} \sum_{i=1}^{2m} o_i . \quad (17.15)$$

A subsequent decision support system using this network can interpret the measurements s as indicators how much one can trust the model. Note that these values of uncertainty must not be identified with error bars as described in section 17.5 because the s merely quantify the instability of the learning.

17.4 Cost Functions

Typical error functions can be written as a sum of individual terms over all T training patterns,

$$E = \frac{1}{T} \sum_{t=1}^T E_t, \quad (17.16)$$

with the individual error E_t depending on the network output $y(x_t, w)$ and the given target data y_t^d . The often used square error,

$$E_t = \frac{1}{2} (y(x_t, w) - y_t^d)^2, \quad (17.17)$$

can be derived from the maximum-likelihood principle and a Gaussian noise model. Eq. 17.17 yields relatively simple error derivatives and results in asymptotically best estimators under certain distribution assumptions, i. e. homoscedasticity. In practical applications, however, several of these assumptions are commonly violated, which may dramatically reduce the prediction reliability of the neural network. A problem arising from this violation is the large impact outliers in the target data can have on the learning, which is also a result of scaling the original time series to zero mean and unit variance. This effect is particularly serious for data in finance and economics which contain large shocks. Another cause of difficulties in financial time series analysis is heteroscedasticity, i.e. situations where the variance of target variable changes over time. We will especially consider cases where the variance is input-dependent: $\sigma_t^2 = \sigma^2(x_t)$.

We propose two approaches to reduce the problems resulting from outliers and heteroscedasticity.

17.4.1 Robust Estimation with LnCosh

Outliers are common in financial time series and are usually caused by “information shocks” such as announcements of government data or dividends paid by companies that are out of line with market expectations. These shocks appear as discontinuities in the trajectory of an affected asset. To be robust against such shocks, typical cost functions like

$$E_t = |y(x_t, w) - y_t^d| \quad (17.18)$$

are used which do not overweight large errors. A smoother version is given by

$$E_t = \frac{1}{a} \log \cosh (a (y(x_t, w) - y_t^d)), \quad (17.19)$$

with parameter $a > 1$. We typically use $a \in [3, 4]$. This function approximates the parabola of the squared errors for small differences (Gaussian noise model), and is proportional to the absolute value of the difference for larger values of the difference (Laplacian noise model). The assumed noise model, the cost function 17.19 and its derivative for the case $a = 5$ is shown in fig. 17.9. The function

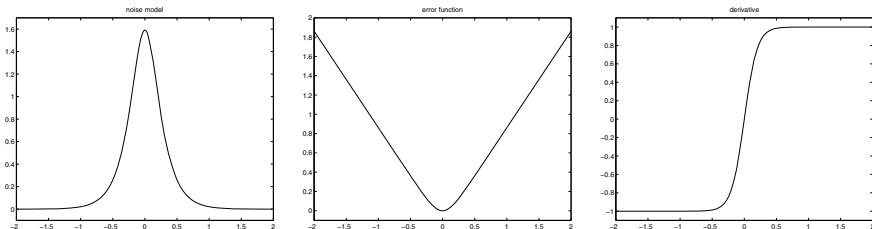


Fig. 17.9. The Laplacian-like noise model, left, the log cosh error function 17.19, middle, and its corresponding derivative, right

log cosh is motivated by the observation that the derivative of $|x|$ is $\text{sign}(x)$ and $\tanh(ax)$ is a smooth approximation of this step function with the integral $\int \tanh(az)dz = \frac{1}{a} \log \cosh(az)$.

17.4.2 Robust Estimation with CDEN

This section describes a more general framework for robust estimation which is based on the theory of density estimation. The advantages are twofold. First, any parameters of the cost function (e. g. the a for eq. 17.19) can be determined by learning which avoids an artificial bias by setting those parameters to predefined values. Second, the proposed methods allow the modeling of heteroscedastic time series whose variance changes over time.

Probability density estimating neural networks have recently gained major attention in the neural network community as a more adequate tool to describe probabilistic relations than common feed-forward networks (see [35], [22] and [20]). Interest has hereby focussed on exploiting the additional information which is inherent to the conditional density, for example the conditional variance as a measure of prediction reliability [28], the representation of multi-valued mappings in the form of multi-modal densities to approach inverse problems [3], or the use of the conditional densities for optimal portfolio construction [18]. In this paper, we will use the *Conditional Density Estimation Network (CDEN)*, which is, among various other density estimation methods, extensively discussed in [22] and [20].

A possible architecture of the CDEN is shown in fig. 17.10. It is assumed that $p(y|x)$, the conditional density to be identified, may be formulated as the parametric density $p(y|\phi(x))$. The condition on x is realized by the parameter vector ϕ which determines the form of the probability distribution $p(y|\phi(x))$. Both $\phi(x)$ and $p(y|\cdot)$ may be implemented as neural networks with the output of the first determining the weights of the latter. Denoting the weights contained in the parameter prediction network $\phi(x)$ as w , we may thus write $p(y|x, w) = p(y|\phi_w(x))$. Assuming independence such that $p(y_1, \dots, y_T|\cdot) = p(y_1|\cdot) \cdots p(y_T|\cdot)$ we minimize the negative Log-Likelihood error function.

$$E = -\log p(y_1, \dots, y_T | \cdot)$$

$$\begin{aligned} &= -\log \prod_{t=1}^T p(y_t | x_t, w) \\ &= -\sum_{t=1}^T \log p(y_t | x_t, w) \end{aligned} \quad (17.20)$$

by performing gradient descent using a variant of the common backpropagation algorithm, we give way to a maximum likelihood estimate of the weights in the parameter prediction network [22, 20].

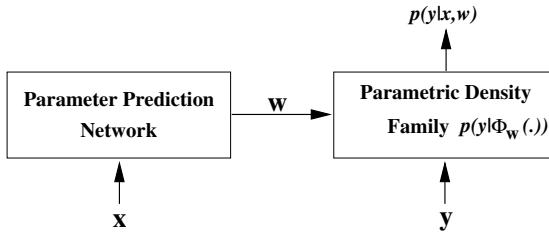


Fig. 17.10. Conditional Density Estimation Network, CDEN

Specific problems with the discussed model are first approached by determining an appropriate density family $p(y|\phi(x))$. A powerful choice is a Gaussian mixture

$$p(y|\phi(x)) = \sum_{i=1}^n P_i(x) p(y|\mu_i(x), \sigma_i(x)), \quad P_i(x) \geq 0, \quad \sum_i^n P_i(x) = 1, \quad (17.21)$$

because they cover a wide range of probability models. For the univariate case (one output), the $p(y|\mu_i(x), \sigma_i(x))$ are normal density functions:

$$p(y|\mu_i(x), \sigma_i(x)) = \frac{1}{\sqrt{2\pi}\sigma_i(x)} e^{-\frac{1}{2}\left(\frac{y-\mu_i(x)}{\sigma_i(x)}\right)^2}. \quad (17.22)$$

There are several ways to determine the individual density parameters contained in $(P_i, \mu_i, \sigma_i)_{i=1}^n$. Either they are set as the output of the parameter prediction network, or they are trained as x -independent, adaptable weights of the density network, or some of them may be given by prior knowledge (e. g. clustering, neuro-fuzzy).

A probability model $p(y|\cdot)$ based on the CDEN architecture in fig. 17.11 which perceives the presence of outliers and thus leads to robust estimators is illustrated in the left part of figure 17.12. The CDEN consists of two Gaussians with identical estimation for their mean $\mu(x)$. While its narrow Gaussian represents the distribution of the non-outliers part of the data, the wider one expresses the assumption that some data are located at larger distances from the prediction. The fact that outliers are basically exceptions may be reflected by an appropriate choice of the mixture weightings P_i , which may be regarded as prior probabilities for each Gaussian distribution. Using this probability model yields the following norm for the maximum-likelihood minimization:

$$E_t = -\log \left[\frac{P_1}{\sigma_1} e^{-\frac{1}{2} \left(\frac{y(x_{t,w}) - y_t^d}{\sigma_1} \right)^2} + \frac{P_2}{\sigma_2} e^{-\frac{1}{2} \left(\frac{y(x_{t,w}) - y_t^d}{\sigma_2} \right)^2} \right]. \quad (17.23)$$

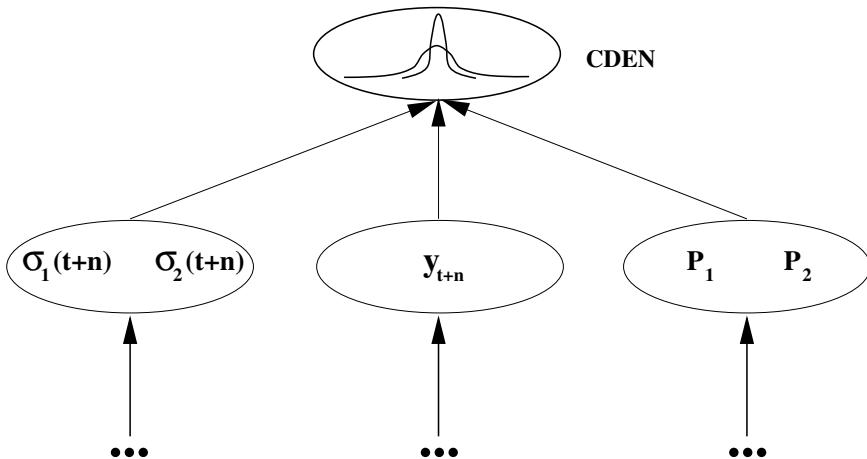


Fig. 17.11. A CDEN for limiting the influence of outliers

The qualitative behavior of E_t in the one-dimensional case is illustrated in the middle and left part of figure 17.12 (compare also fig. 17.9). One may clearly see how the mixture limits the influence of outliers. Norms with influence-limiting properties are called M-estimators in regression [5].

The difficulty involved in using M-estimators is that they usually possess a set of parameters which have to be properly determined. In our case, the parameters are P_1, P_2, σ_1 and σ_2 . In the framework of the CDEN architecture, they are considered as adaptable weights of the density network. The advantage is that unlike classical M-estimators, the parameters are determined by the data during training and thus do not bias the solution. One method which has proven to be successful in regression is to substitute eq. 17.23 by a mixture of a quadratic

error function and a linear function of the absolute error to limit the influence of the outliers. An adaptive version of such an error measure may easily be implemented in the CDEN framework by substituting a mixture of Gaussian and Laplace distributions in equation 17.21. Other limiting error functions may be constructed alike.

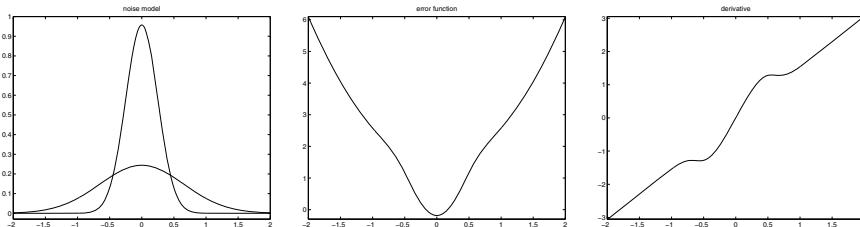


Fig. 17.12. The Gaussian mixture noise model, left, the error function 17.23, middle, and its corresponding derivative, right

Heteroscedasticity may arise due to changes in the risk associated with an investment. In the stock market, for example, the variance of a stock's return is commonly related to the company's debt-equity ratio due to the well-known leverage effect of a varying income on its return on equity. Heteroscedasticity has been extensively studied in time series analysis and is commonly approached using the (G)ARCH methodology. While the latter explains the conditional variances based on past residuals, the CDEN particularly accounts for nonlinear dependencies of the variances on past observations.

If we assume a normally distributed noise model with zero mean and variance σ^2 , an appropriate representation is a single Gaussian with variable scale and location parameter. An implementation in the CDEN is straightforward. We use a parameter prediction network with two outputs, one for the conditional expectation and another one for the conditional variance. This special case of the CDEN has also been extensively investigated by Nix and Weigend [21]. During the training of the network, the weights are optimized with respect to

$$E_t = \left[\log \sqrt{2\pi} \sigma_t(x_t, w) + \frac{1}{2\sigma_t(x_t, w)^2} (y(x_t, w) - y_t^d)^2 \right]. \quad (17.24)$$

Minimization according to eq. 17.17 and eq. 17.24 obviously only differs in that the individual errors $(y(x_t, w) - y_t^d)$ are weighted by estimates of the inverse variances $1/\sigma_t^2(x_t, w)$ in the CDEN. Training the CDEN thus corresponds to deriving a *generalized least square estimator (GLSE)*, except that we use an estimate $\hat{\sigma}_t(x_t, w)$ instead of the unknown σ_t .³

³ Definition and properties of the GLSE are extensively discussed in [5]. The case where an estimate of σ is used during optimization is commonly denoted as a *two-stage estimation*.

17.5 Error Bar Estimation with CDEN

In addition to a more robust learning, the CDEN approach can be used to estimate the uncertainty associated with the prediction of the expected value $y_{t+n} := y(x_t, w)$. Here we assume a normally distributed error and have to optimize a likelihood function of the form

$$E = \frac{1}{T} \sum_{t=1}^T \left[\log \left(\sqrt{2\pi} \sigma(x_t, w_\sigma) \right) + \frac{(y(x_t, w) - y_t^d)^2}{2\sigma^2(x_t, w_\sigma)} \right]. \quad (17.25)$$

If we assume a Laplacian noise model, the cost function is

$$E = \frac{1}{T} \sum_{t=1}^T \left[\log (2\sigma(x_t, w_\sigma)) + \frac{|y(x_t, w) - y_t^d|}{\sigma(x_t, w_\sigma)} \right]. \quad (17.26)$$

The log cosh-approximation of the Laplacian model has the form

$$E = \frac{1}{T} \sum_{t=1}^T \left[\log (\pi\sigma(x_t, w_\sigma)) + \log \cosh \left(\frac{y(x_t, w) - y_t^d}{\sigma(x_t, w_\sigma)} \right) \right]. \quad (17.27)$$

In fig. 17.13 we show a network architecture which we found useful to apply such error bar estimations. The architecture combines net internal preprocessing with the squared input approach and estimates in two branches the expected value $y(x_t, w)$ and the standard derivation $\sigma(x_t, w_\sigma)$. The CDEN cluster combines these pieces of information and computes the flow of the error.

We have found that the mixture of local and global analysis as shown in fig. 17.13 harmonizes well with the aim of solving a forecasting problem including error bars. To assure positive values for $\sigma(x)$ a suitable activation function $e^{(\cdot)}$ is used in the σ cluster. By using a positive offset (bias) one can avoid the singularities in the likelihood target function which are caused by very small $\sigma(x)$.

In fig. 17.14 the CDEN with one Gauss-function is combined with the architecture of section 17.3.7. Here we assume that the estimated “forces” contain the necessary information to approximate the uncertainty of our averaged forcecast:

$$\sigma^2(y_{t+n}|x_t) = \sigma^2(y_{t+n}|\text{forces}(x_t)) \quad (17.28)$$

or, more specifically, using the acceleration and mean reverting forces of fig. 17.14:

$$\sigma^2(y_{t+n}|x_t) = \sum_{i=1}^{2m} w_i \cdot \text{force}_i^2(x_t). \quad (17.29)$$

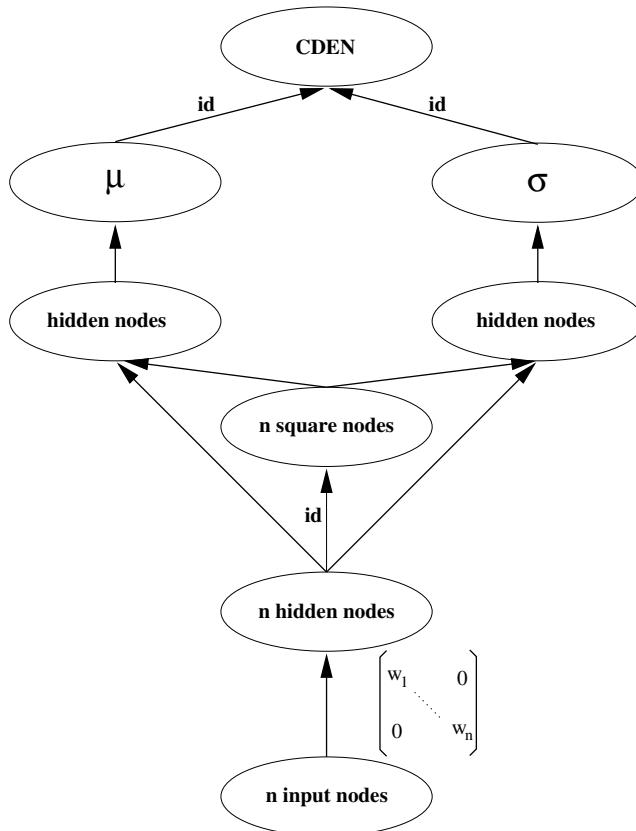


Fig. 17.13. Error bar estimation with CDEN

By a linear combination of the squared values of the forces, this architecture learns the input dependent variance $\sigma(y_{t+n}|x_t)$ (see fig. 17.14 and eq. 17.13 for a description of forces). Thus, we are able to achieve a forecast as well as an error bar.

The interesting point in this combination is not only to be seen in the possibility of using the general frame as the means to identify dynamical systems. In this environment we can even analyze the responsibility of the forces, long range or short range, with respect to the uncertainty of the forecast. In several monthly forecast models we have found a monotonic increase of importance from the short to the long range forces.

Error bars and variance estimates are an essential piece of information for the typically used mean-variance approach in portfolio management [7]. The aim is

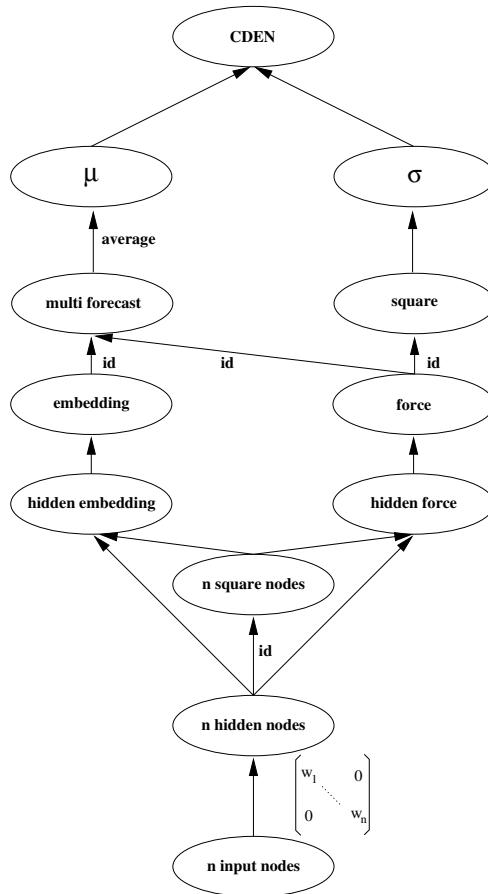


Fig. 17.14. Error bar estimation with CDEN using the architecture of section 17.3. The control-embedding and control-force clusters are suppressed for visual clarity.

to compute efficient portfolios which allocate the investor's capital to several assets in order to maximize the return of the investment for a certain level of risk. The variance is often estimated by linear models or is given a priori by an expert. In contrast, the CDEN approximates $\sigma(y_{t+n}|x_t)$ as a function of the current state of the financial market and of the forecast of the neural network.

Covariances can also be estimated with CDEN using a multivariate Gaussian in eq. 17.22. Implementation details can be found in [17, 19]. If one approximates the conditional density $p(y|\phi(x))$ with several Gaussians, the estimated expected mean, variance and covariance may be computed using typical moment generating transformations [23, 19].

17.6 Data Meets Structure

17.6.1 The Observer-Observation Dilemma

Human beings believe that they are able to solve a psychological version of the *Observer-Observation Dilemma*. On the one hand, they use their observations to constitute an understanding of the laws of the world, on the other hand, they use this understanding to evaluate the correctness of the incoming pieces of information. Of course, as everybody knows, human beings are not free from making mistakes in this psychological dilemma. We encounter a similar situation when we try to build a mathematical model using data. Learning relationships from the data is only one part of the model building process. Overrating this part often leads to the phenomenon of overfitting in many applications (especially in economic forecasting). In practice, evaluation of the data is often done by external knowledge, i. e. by optimizing the model under constraints of smoothness and regularization [16]. If we assume that our model summarizes the best knowledge of the system to be identified, why shouldn't we use the model itself to evaluate the correctness of the data? One approach to do this is called Clearning [33]. In this paper, we present a unified approach of the interaction between the data and a neural network (see also [38]). It includes a new symmetric view on the optimization algorithms, here learning and cleaning, and their control by parameter and data noise.

17.6.2 Learning Reviewed

We are especially interested in using the output of a neural network $y(x, w)$, given the input pattern, x , and the weight vector, w , as a forecast of financial time series. In the context of neural networks learning normally means the minimization of an error function E by changing the weight vector w in order to achieve good generalization performance. Again, we assume that the error function can be written as a sum of individual terms over all T training patterns, $E = \frac{1}{T} \sum_{t=1}^T E_t$. The often used sum-of-square error can be derived from the maximum-likelihood principle and a Gaussian noise model:

$$E_t = \frac{1}{2} (y(x, w) - y_t^d)^2, \quad (17.30)$$

with y_t^d as the given target pattern. If the error function is a nonlinear function of the parameters, learning has to be done iteratively by a search through the weight space, changing the weights from step τ to $\tau + 1$ according to:

$$w^{(\tau+1)} = w^{(\tau)} + \Delta w^{(\tau)}. \quad (17.31)$$

There are several algorithms for choosing the weight increment $\Delta w^{(\tau)}$, the easiest being *gradient descent*. After each presentation of an input pattern, the gradient $g_t := \nabla E_t|_w$ of the error function with respect to the weights is computed. In

the batch version of gradient descent the increments are based on all training patterns

$$\Delta w^{(\tau)} = -\eta g = -\eta \frac{1}{T} \sum_{t=1}^T g_t, \quad (17.32)$$

whereas the pattern-by-pattern version changes the weights after each presentation of a pattern x_t (often randomly chosen from the training set):

$$\Delta w^{(\tau)} = -\eta g_t. \quad (17.33)$$

The learning rate η is typically held constant or follows an annealing procedure during training to assure convergence.

Our experiments have shown that small batches are most useful, especially in combination with Vario-Eta, a stochastic approximation of a Quasi-Newton method [9]:

$$\Delta w^{(\tau)} = -\frac{\eta}{\sqrt{\frac{1}{T} \sum (g_t - g)^2}} \cdot \frac{1}{N} \sum_{t=1}^N g_t, \quad (17.34)$$

with $N \leq 20$.

Let us assume, that the error function of a specific problem is characterized by a minimum in a narrow valley whose boundaries are parallel to the axes of the weight space. For the two dimensional case, such a situation is shown in fig. 17.15.

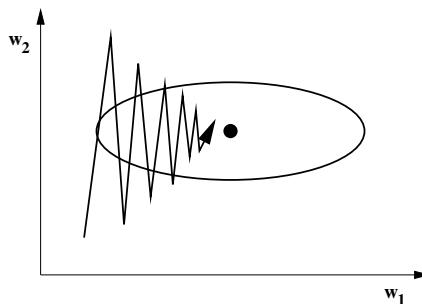


Fig. 17.15. Vario-Eta, a stochastic approximation of a Quasi-Newton method

A gradient approach would follow a “zigzagging” track and would approximate the minimum very slowly. With Vario-Eta, the zigzagging along w_2 is damped and the drift along w_1 is accelerated. This behavior is similar to the weight trajectories classical Newton methods show in such a situation. The actual implementation uses stochastic approximations to compute the standard deviation.

The use of the standard deviation instead of the variance in Vario-Eta means an additional advantage in training large networks. Passing a long sequence of layers in a neural network, the error signals contain less and less information. The normalization in eq. 17.34 rescales the learning information for every weight. If one designed Vario-Eta as close as possible to second order methods, it would be appropriate to use the variance instead the standard deviation in the denominator, but then we would also lose the scaling property. In section 17.6.3, we will provide a further advantage of using the standard deviation.

Learning pattern-by-pattern or with small batches can be viewed as a stochastic search process because we can write the weight increments as:

$$\Delta w^{(\tau)} = -\eta \left[g + \left(\frac{1}{N} \sum_{t=1}^N g_t - g \right) \right]. \quad (17.35)$$

These increments consist of the terms g with a drift to a local minimum and of noise terms $(\frac{1}{N} \sum_{t=1}^N g_t - g)$ disturbing this drift.

17.6.3 Parameter Noise as an Implicit Penalty Function

Consider the Taylor expansion of $E(w)$ around some point w in the weight space

$$E(w + \Delta w) = E(w) + \sum_i \frac{\partial E}{\partial w_i} \Delta w_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 E}{\partial w_i \partial w_j} \Delta w_i \Delta w_j + \dots . \quad (17.36)$$

Assume a given sequence of T disturbance vectors Δw_t , whose elements are uncorrelated over t with zero mean and variance (row-)vector $\text{var}(\Delta w_i)$. The expected value $\langle E(w) \rangle$ can then be approximated by

$$\langle E(w) \rangle \approx \frac{1}{T} \sum_t E(w + \Delta w_t) = E(w) + \frac{1}{2} \sum_i \text{var}(\Delta w_i) \frac{\partial^2 E}{\partial w_i^2} \quad (17.37)$$

assuming that the first and second derivatives of E are stable if we are close to a local minimum. In eq. 17.37, noise on the weights acts implicitly as a penalty term to the error function given by the second derivatives $\frac{\partial^2 E}{\partial w_i^2}$. The noise variances $\text{var}(\Delta w_i)$ operate as penalty parameters. As a consequence, flat minima solutions which may be important for achieving good generalization performance are favored [13].

Learning pattern-by-pattern introduces automatically such noise in the training procedure i.e., $\Delta w_t = -\eta \cdot g_t$. Close to convergence, we can assume that g_t is i.i.d. with zero mean and variance vector $\text{var}(g_i)$ so that the expected value can be approximated by

$$\langle E(w) \rangle \approx E(w) + \frac{\eta^2}{2} \sum_i \text{var}(g_i) \frac{\partial^2 E}{\partial w_i^2} . \quad (17.38)$$

This type of learning introduces a local penalty parameter $\text{var}(g_i)$, characterizing the stability of the weights $w = [w_i]_{i=1,\dots,k}$. In a local minimum the sum of gradients for the weight w_i is $\sum g_{it} = 0$ whereas the variance $\text{var}(g_i)$ may be large. In this case the solution is very sensitive against resampling of the data and therefore unstable. To improve generalization the curvature of the error function around such weights with high variance should be strongly penalized. This is automatically done by pattern-by-pattern learning.

The noise effects due to Vario-Eta learning $\Delta w_t(i) = -\frac{\eta}{\sqrt{\sigma_i^2}} \cdot g_{ti}$ leads to an expected value

$$\langle E(w) \rangle \approx E(w) + \frac{\eta^2}{2} \sum_i \frac{\partial^2 E}{\partial w_i^2}. \quad (17.39)$$

By canceling the term $\text{var}(g_i)$ in eq. 17.38, Vario-Eta achieves a simplified uniform penalty parameter, which depends only on the learning rate η . Whereas pattern-by-pattern learning is a slow algorithm with a locally adjusted penalty control, Vario-Eta is fast only at the cost of a simplified uniform penalty term.

local	learning rate (Vario-Eta)	\rightarrow	global	penalty
global	learning rate (pattern by pattern)	\rightarrow	local	penalty

Following these thoughts, we will now show that typical Newton methods should not be used in stochastic learning. Let us assume that we are close to a local minimum and that the Hessian H of the error function is not significantly changing anymore. On this supposition, the implied noise would be of the form $\Delta w_t = \eta H^{-1} g_t$. Because of the stable Hessian, the mean of the noise is zero so that the expected error function becomes

$$\langle E(w) \rangle \approx E(w) + \frac{\eta^2}{2} \sum_i \text{var}(g_i) \left(\frac{\partial^2 E}{\partial w_i^2} \right)^{-1}. \quad (17.40)$$

In this case, we have again a local control of the penalty parameter through the variance of the gradients $\text{var}(g_i)$, like in the pattern by pattern learning. But now, we are penalizing weights at points in the weight space where the inverse of the curvature is large. This means, we penalize flat minima solutions, which counters our goal of searching for stable solutions.

Typically Newton methods are used as cumulative learning methods so that the previous arguments do not apply. Therefore, we conclude, that second order methods should not be used in stochastic search algorithms. To support global

Table 17.1. Structure-Speed-Dilemma

learning	structure	speed
pattern by pattern	+	-
VarioEta	-	+

learning and to exploit the bias to flat minima solution of such algorithms, we only use pattern-by-pattern learning or Vario-Eta in the following.

We summarize this section by giving some advice on how to achieve flat minima solutions (see also table 17.1):

- Train the network to a minimal training error solution with Vario-Eta, which is a stochastic approximation of a Newton method and therefore fast.
- Add a final phase of pattern-by-pattern learning with a uniform learning rate to fine tune the local curvature structure by the local penalty parameters (eq. 17.38). For networks with many layers, this step should be omitted because the gradients will vanish due to the long signal flows. Only Vario-Eta with its scaling capability can solve such optimization problems appropriately.
- Use a learning rate η as high as possible to keep the penalty effective. The training error may vary a bit, but the inclusion of the implicit penalty is more important.

We want to point out that the decision of which learning algorithm to use not only influences the speed and global behavior of the learning, but also, for fixed learning rates, leads to different structural solutions. This structural consequence has also to be taken into account if one analyzes and compares stochastic learning algorithms.

17.6.4 Cleaning Reviewed

When training neural networks, one typically assumes that the input data is noise-free and one forces the network to fit the data exactly. Even the control procedures to minimize overfitting effects (i.e., pruning) consider the inputs as exact values. However, this assumption is often violated, especially in the field of financial analysis, and we are taught by the phenomenon of overfitting not to follow the data exactly. Clearning, as a combination of cleaning and learning, has been introduced in [33]. In the following, we focus on the cleaning aspects. The motivation was to minimize overfitting effects by considering the input data as being corrupted by noise whose distribution has to be learned also.

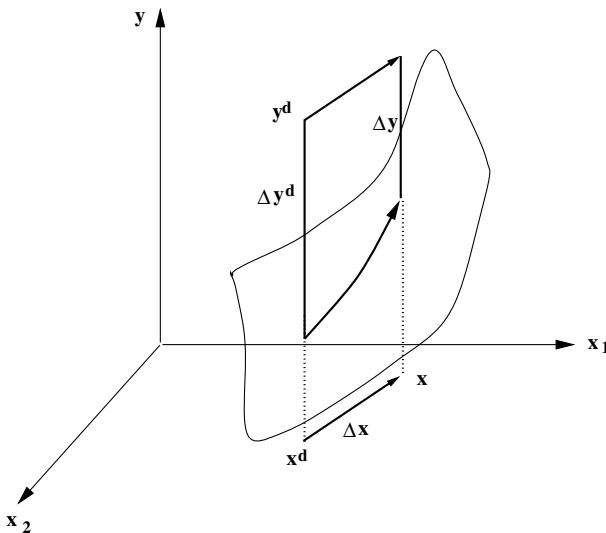


Fig. 17.16. If the slope of the modeled function is large, then a small shift in the input data decreases the output error dramatically

The cleaning error function for the pattern t is given by the sum of two terms assuming same variance levels for input and output

$$E_t^{y,x} = \frac{1}{2} \left[(y_t - y_t^d)^2 + (x_t - x_t^d)^2 \right] = E_t^y + E_t^x \quad (17.41)$$

with x_t^d, y_t^d as the observed data point. In the pattern-by-pattern learning, the network output $y(x_t, w)$ determines the weight adaptation as usual,

$$w^{(\tau+1)} = w^{(\tau)} - \eta \frac{\partial E^y}{\partial w} . \quad (17.42)$$

We also must memorize correction vectors Δx_t for all input data of the training set in order to present the cleaned input x_t to the network,

$$x_t = x_t^d + \Delta x_t . \quad (17.43)$$

The update rule for the corrections, initialized with $\Delta x_t^{(0)} = 0$ can be derived from typical adaptation sequences $x_t^{(\tau+1)} = x_t^{(\tau)} - \eta \frac{\partial E^{y,x}}{\partial x}$ leading to

$$\Delta x_t^{(\tau+1)} = (1 - \eta) \Delta x_t^{(\tau)} - \eta (y_t - y_t^d) \frac{\partial y}{\partial x} . \quad (17.44)$$

This is a nonlinear version of the error-in-variables concept in statistics [27] (see also fig. 17.16 for the two dimensional case).

All of the necessary quantities, i. e. $(y_t - y_t^d) \frac{\partial y(x, w)}{\partial x}$ are computed by typical back-propagation algorithms anyway. We found that the algorithms work well if the same learning rate η is used for both the weight and cleaning updates. For regression, cleaning forces the acceptance of a small error in x , which can in turn decrease the error in y dramatically, especially in the case of outliers. Successful applications of cleaning are reported in [33] and [30].

Although the network may learn an optimal model for the cleaned input data, there is no easy way to work with cleaned data on the test set because for this data we do not know the output target difference for computing eq. 17.44. As a consequence, the model is evaluated on a test set with a different noise characteristic compared to the training set. We will later propose a combination of learning with noise and cleaning to work around this serious disadvantage.

17.6.5 Data Noise Reviewed

Artificial noise on the input data is often used during training because it creates an infinite number of training examples and expands the data to empty parts of the input space. As a result, the tendency of learning by heart may be limited because smoother regression functions are produced.

Now, we consider again the Taylor expansion, this time applied to $E(x)$ around some point x in the input space. The expected value $\langle E(x) \rangle$ is approximated by

$$\langle E(x) \rangle \approx \frac{1}{T} \sum_t E(x + \Delta x_t) = E(x) + \frac{1}{2} \sum_j \text{var}(\Delta x_j) \frac{\partial^2 E}{\partial x_j^2} \quad (17.45)$$

where $\frac{\partial^2 E}{\partial x_j^2}$ are the diagonal elements of the Hessian H_{xx} of the error function with respect to the inputs x . Again, in eq. 17.45, noise on the inputs acts implicitly as a penalty term to the error function with the noise variances $\text{var}(\Delta x_j)$ operating as penalty parameters (compare eq. 17.37). Noise on the input improves generalization behavior by favoring smooth models [3].

The noise levels can be set to a constant value, e. g. given by a priori knowledge, or adaptive as described now. We will concentrate on a uniform or normal noise distribution. Then, the adaptive noise level ξ_j is estimated for each input j individually. Suppressing pattern indices, we define the average residual errors ξ_j and ξ_j^2 as:

$$\text{uniform residual error: } \xi_j = \frac{1}{T} \sum_t \left| \frac{\partial E^y}{\partial x_j} \right|, \quad (17.46)$$

$$\text{Gaussian residual error: } \xi_j^2 = \frac{1}{T} \sum_t \left(\frac{\partial E^y}{\partial x_j} \right)^2. \quad (17.47)$$

Actual implementations use stochastic approximation, e. g. for the uniform residual error

$$\xi_j^{(\tau+1)} = (1 - \frac{1}{T})\xi_j^{(\tau)} + \frac{1}{T} \left| \frac{\partial E^y}{\partial x_j} \right|. \quad (17.48)$$

The different residual error levels can be interpreted as follows (table 17.2): A small level ξ_j may indicate an unimportant input j or a perfect fit of the network concerning this input j . In both cases, a small noise level is appropriate. On the other hand, a high value of ξ_j for an input j indicates an important but imperfectly fitted input. In this case high noise levels are advisable. High values of ξ_j lead to a stiffer regression model and may therefore increase the generalization performance of the network. Therefore, we use ξ_j or ξ_j^2 as parameter to control the level of noise for input j .

Table 17.2. The interpretation of different levels of the residual errors ξ

observation of the residual error	interpretation	advice for the noise control
ξ small	perfect fit or unimportant input	use low noise
ξ large	imperfect fit but important input	use high noise

17.6.6 Cleaning with Noise

Typically, training with noisy inputs involves taking a data point and adding a random variable drawn from a fixed or adaptive distribution. This new data point x_t is used as an input to the network. If we assume, that the data is corrupted by outliers and other influences, it is preferable to add the noise term to the cleaned input. For the case of Gaussian noise the resulting new input is:

$$x_t = x_t^d + \Delta x_t + \xi \phi, \quad (17.49)$$

with ϕ drawn from the normal distribution. The cleaning of the data leads to a corrected mean of the data and therefore to a more symmetric noise distribution, which also covers the observed data x_t .

We propose a variant which allows more complicated and problem dependent noise distributions:

$$x_t = x_t^d + \Delta x_t - \Delta x_k, \quad (17.50)$$

where k is a random number drawn from the indices of the memorized correction vectors $[\Delta x_t]_{t=1,\dots,T}$. By this, we exploit the distribution properties of the correction vectors in order to generate a possibly asymmetric and/or dependent noise distribution, which still covers $x_t = x_t^d$ if $k = t$.

One might wonder why we want to disturb the cleaned input $x_t^d + \Delta x_t$ with an additional noisy term Δx_k . The reason for this is that we want to benefit from representing the whole input distribution to the network instead of only using one particular realization. This approach supplies a solution to the cleaning problem when switching from the training set to the test set as described in section 17.6.4.

17.6.7 A Unifying Approach: The Separation of Structure and Noise

In the previous sections we explained how the data can be separated into a cleaned part and an unexplainable noisy part. Analogously, the neural network is described as a time invariant structure (otherwise no forecasting is possible) and a noisy part.

data	\rightarrow	cleaned data	$+$	time invariant data noise
neural network	\rightarrow	time invariant parameters	$+$	parameter noise

We propose using cleaning and adaptive noise to separate the data and using learning and stochastic search to separate the structure of the neural network.

data	\leftarrow	cleaning (neural network)	$+$	adaptive noise (neural network)
neural network	\leftarrow	learning (data)	$+$	stochastic search (data)

The algorithms analyzing the data depend directly on the network whereas the methods searching for structure are directly related to the data. It should be clear that the model building process should combine both aspects in an alternating or simultaneous manner. We normally use learning and cleaning simultaneously. The interaction of the data analysis and network structure algorithms is a direct embodiment of the the concept of the Observer-Observation Dilemma.

The aim of the unified approach can be described, exemplary assuming here a Gaussian noise model, as the minimization of the error due to both, the structure and the data:

$$\frac{1}{2T} \sum_{t=1}^T \left[(y_t - y_t^d)^2 + (x_t - x_t^d)^2 \right] \rightarrow \min_{x_t, w} \quad (17.51)$$

Combining the algorithms and approximating the cumulative gradient g by \tilde{g} using an exponential smoothing over patterns, we obtain

data		
$\Delta x_t^{(\tau+1)} = (1 - \eta) \Delta x_t^{(\tau)} - \eta(y_t - y_t^d) \frac{\partial y}{\partial x}$		
$x_t = x_t^d + \underbrace{\Delta x_t^{(\tau)}}_{\text{cleaning}} - \underbrace{\Delta x_k^{(\tau)}}_{\text{noise}}$		
(17.52)		
structure		
$\tilde{g}^{(\tau+1)} = (1 - \alpha) \tilde{g}^{(\tau)} + \alpha(y_t - y_t^d) \frac{\partial y}{\partial w}$		
$w^{(\tau+1)} = w^{(\tau)} - \underbrace{\eta \tilde{g}^{(\tau)}}_{\text{learning}} - \underbrace{\eta(g_t - \tilde{g}^{(\tau)})}_{\text{noise}}$		

The cleaning of the data by the network computes an individual correction term for each training pattern. The adaptive noise procedure according to eq. 17.50 generates a potentially asymmetric and dependent noise distribution which also covers the observed data. The implied curvature penalty, whose strength depends on the individual liability of the input variables, can improve the generalization performance of the neural network.

The learning of the structure involves a search for time invariant parameters characterized by $\frac{1}{T} \sum g_t = 0$. The parameter noise supports this exploration as a stochastic search to find better “global” minima. Additionally, the generalization performance may be further improved by the implied curvature penalty depending on the local liability of the parameters. Note that, although the description of the weight updates collapses to the simple form of eq. 17.33, we preferred the formula above to emphasize the analogy between the mechanism which handles the data and the structure.

In searching for an optimal combination of data and parameters, we have needed to model in both. This is not an indication of our failure to build a perfect model but rather it is an important element for controlling the interaction of data and structure.

17.7 Architectural Optimization

The initial network can only be a guess on the appropriate final architecture. One way to handle this inadequacy is to use a growing network. As a possibility in the opposite direction, pruning can be applied to shrink the network. From our experience the advantage of fast learning of growing networks is more than counterbalanced by the better learning properties of large architectures. At least in the first learning steps, large networks are not trapped as quickly in local minima. When applying the pruning methods according the *late stopping* concept, which trains the weights of the network until the error function converges (see section 17.8.1), the learning procedure had a sufficient number of adaptation steps to adjust the broad network structure. Another advantage of the pruning technique is the greater flexibility in generating sparse and possibly irregular network architectures.

17.7.1 Node-Pruning

We evaluate the importance of input- or hidden nodes by using as a test value

$$\text{test}_i = E(x_1, \dots, x_i = \mu(x_i), \dots, x_n) - E(x_1, \dots, x_n), \quad (17.53)$$

with $\mu(x_i)$ describing the mean value of the i -th input in the time series for the training set. This test value is a measure of the increase of the error if we omit one of the input series.

The creation of the bottleneck structure in the net internal preprocessing (section 17.2) is best performed by applying this test for the hidden neurons in the bottleneck on the training set. Thereby the disturbance of the learning is reduced to the minimum. In one of the steps of the training procedure (section 17.8) we will show that the pruning of the input neurons should be done on the validation set to improve the time stability of the forecasting.

In addition to the deletion of nodes the ranking of the inputs can give us a deeper insight into the task (which inputs are the most relevant and so on).

17.7.2 Weight-Pruning

The neural network topology represents only a hypothesis of the true underlying class of functions. Due to possible mis-specification, we may have defects of the parameter noise distribution. Pruning algorithms not only limit the memory of the network, but they also appear to be useful for correcting the noise distribution in different ways.

Stochastic-Pruning

Stochastic-Pruning [8] is basically a t-test on the weights w ,

$$\text{test}_w = \frac{|w + g|}{\sqrt{\frac{1}{T} \sum (g_t - g)^2}}, \quad (17.54)$$

with $g = \frac{1}{T} \sum_t g_t$. Weights with low test_w values constitute candidates for pruning. From the viewpoint of our approach, this pruning algorithm is equivalent to the cancellation of weights with low *liability* as measured by the size of the weight divided by the standard deviation of its fluctuations. By this, we get a stabilization of the learning against resampling of the training data.

This easy to compute method worked well in combination with the early stopping concept in contrast to insufficient results with the late stopping concept. For early stopping Stochastic-Pruning acts like a t-test, whereas for late stopping the gradients of larger weights do not fluctuate enough to give useful test values. Thus, the pruning procedure includes an artificial bias to nonlinear models. Furthermore, Stochastic-Pruning is also able to revive already pruned weights.

Early-Brain-Damage

A further weight pruning method is EBD, *Early-Brain-Damage* [31], which is based on the often cited OBD pruning method [15]. In contrast to OBD, EBD allows its application before the training has reached a local minimum.

For every weight, EBD computes as a test value an approximation of the difference between the error function for $w = 0$ versus the value of the error function for the best situation this weight can have

$$\text{test}_w = E(0) - E(w_{\min}) = -gw + \frac{1}{2}w' H w + \frac{1}{2}gH^{-1}g'. \quad (17.55)$$

The above approximation is motivated by a Taylor expansion of the error function. From

$$E(\tilde{w}) = E(w) + g(\tilde{w} - w) + \frac{1}{2}(\tilde{w} - w)' H (\tilde{w} - w) \quad (17.56)$$

we get

$$E(0) = E(w) - gw + \frac{1}{2}w' H w \quad (17.57)$$

and as a solution to the minimum problem $E(\tilde{w}) \rightarrow \min$ we have

$$w_{\min} = w - H^{-1}g' \text{ together with } E(w_{\min}) = E(w) - \frac{1}{2}gH^{-1}g'. \quad (17.58)$$

The difference of these two error values is the proposed EBD test. The Hessian H in this approach is computed in the same way as in the original OBD calculus [15].

One of the advantages of EBD over OBD is the possibility of performing the testing while being slightly away from a local minimum. In our training procedure we propose using noise even in the final part of learning so that we are only near a local minimum. Furthermore, EBD is also able to revive already pruned weights.

Similar to Stochastic Pruning, EBD favors weights with a low rate of fluctuations. If a weight is pushed around by a high noise, the implicit curvature

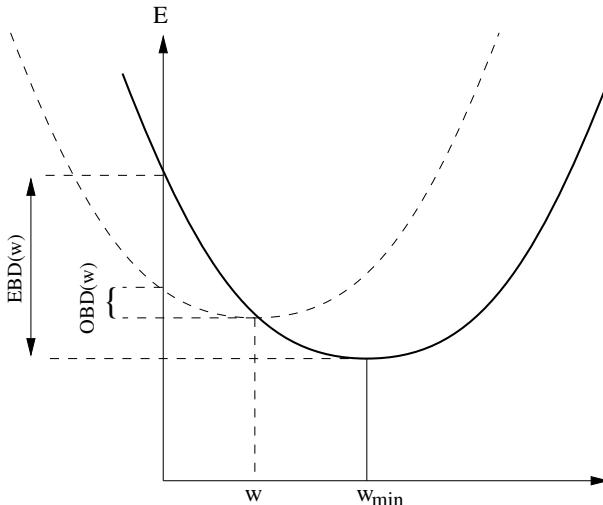


Fig. 17.17. EBD versus ODB weight pruning

penalty would favor a flat minimum around this weight leading to its elimination by EBD.

Our practical experience shows that EBD pruning allows the creation of extremely sparse networks. We had examples where we could prune an initial network with 3000 weights down to a structure with around 50 weights. The first iterations of EBD pruning would typically give no improvement in generalization. This is due to the fact that EBD is testing the importance of a weight regarding the error function, i. e. the same information which is used by the backpropagation algorithm. To say it in another way: EBD cancels out only weights which are not disturbing the learning. Only at the end of the training procedure does the network have to give up a part of the coded structure which leads to an improvement in generalization.

Inverse-Kurtosis

A third pruning method we want to discuss is a method which we call *Inverse-Kurtosis*. The motivation follows an analysis of the following examples of possible distributions of gradient impulses forcing a weight shown in fig. 17.18.

If the network is trained to a local minimum the mean of all gradients by definition is equal to zero. Nevertheless the distribution of the gradients may differ. Now we have to analyze the difference of a peaked or very broad distribution versus a normal distribution. It is our understanding that the peaked distribution indicates a weight which reacts only to a small number of training patterns. A broader distribution, on the other hand, is a sign that many training patterns

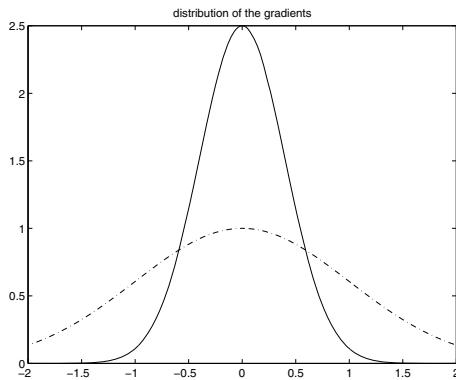


Fig. 17.18. Possible distributions of the gradients g_t , if weights are fixed

focus on the optimization of this weight. In other words, a weight with a peaked distribution has learned by heart special events of the time series but it has not modeled a general underlying structure of the data. Therefore, Inverse-Kurtosis pruning harmonizes well with our network architecture which can encapsulate local structures with its squared layer. Furthermore, a weight with a very broad distribution of its gradients is pushed around by random events because it reacts to almost every pattern with a similar strength. A straight forward way to distinguish between the above distributions is based on the kurtosis to measure the difference to a normal distribution:

$$\text{distance}_w = \left(\frac{\frac{1}{T} \sum_{t=1}^T (g_t - \bar{g})^4}{\left(\frac{1}{T} \sum_{t=1}^T (g_t - \bar{g})^2 \right)^2} - 3 \right)^2. \quad (17.59)$$

To rank importance of the network weights based on this difference to a normal distribution we define the test values as

$$\text{test}_w = \frac{1}{\varepsilon + |\text{distance}_w|}, \quad (17.60)$$

with the small term $\varepsilon \approx 0.001$ to avoid numerical problems.

Similar to the previously mentioned pruning techniques we have now large test values for weights we do not want to eliminate. Note that in this test, we have neglected the size of the gradients and only taken into account the form of the distribution. The weights themselves are not a part of the evaluation, and we have not observed that this method has much effect on the distribution of the weight sizes found by the learning. In contrast, Stochastic Pruning and EBD have a tendency to prune out small weights because they explicitly refer to the size of the weights. Basically, Inverse-Kurtosis computes low test values

(i. e. probable pruning candidates) for weights which are encapsulating only local artifacts or random events in the data instead of modeling a general structure.

In our experience we found that this technique can give a strong improvement of generalization in the first several iterations of pruning.

Instability-Pruning

The local consequence of our learning until the minimal training error is that the cumulative gradient g for every weight is zero on the training set,

$$g = \frac{1}{T} \sum_{t \in T} g_t = 0. \quad (17.61)$$

If this condition was valid for the validation set our model would be perfectly stable:

$$g_V = \frac{1}{V} \sum_{t \in V} g_t = 0. \quad (17.62)$$

Since g_V typically is different from zero we have to check if the measured difference is significant enough to indicate instability. In the language of statistics this is a two sample test for the equality of the mean of two distributions. The following test value (Welch-test) measures this difference:

$$\text{distance}_w = \frac{g_V - g}{\sqrt{\frac{\sigma_V^2}{V} + \frac{\sigma_T^2}{T}}}, \quad (17.63)$$

which is approximately normally distributed with zero mean and unit variance. As a pruning test value we define

$$\text{test}_w = \frac{1}{\varepsilon + |\text{distance}_w|}. \quad (17.64)$$

This test value can be used to construct a ranking of the weights in the following way: Train the network until weights are near a local optimum and compute the test values. Then, take out 5% of the most unstable weights as measured by this test criterion and redo the learning step. Eliminating more than 5% of the weights is not recommandable because Instability-Pruning is not referring to the cost function used for learning, and thus may have a large impact on the error. This test of stability allows the definition of an interesting criterion when to stop the model building process. After pruning weights until a given stability level, we can check if the model is still approximating well enough. This final stopping criterion is possible due to the asymptotic normal distribution of the test. For example, if we define weights with $|\text{distance}_w| < 1$ as stable, then we prune weights with $\text{test}_w < 1$.

One may argue that the usage of the validation set in this approach is an implicit modeling of the validation set and therefore the validation set allows no

error estimation on the generalization set. On the other hand, the typical need for an validation set is the estimation of a final stopping point in the pruning procedure. We have tried an alternative approach: Instead of using the validation set in terms of g_V and σ_V^2 in our comparison let us take a weighted measurement of the training set such that most recent data is given more importance:

$$\tilde{g} = \frac{2}{T(T+1)} \sum_{t \in T} t g_t, \quad (17.65)$$

$$\tilde{\sigma}^2 = \frac{2}{T(T+1)} \sum_{t \in T} t(g_t - \tilde{g})^2. \quad (17.66)$$

Checking the stability over the training set by substituting g_v resp. σ_V^2 with eq. 17.65 resp. 17.66 in eq. 17.63 avoids using the validation set. In our first experiences we observed that this version works as well as the previous one although the evaluation still needs more testing.

In comparison to EBD, which eliminates weights with only a small effect on the error function, Instability-Pruning introduces a new feature in the process of model building: stability over time. Additional information about when to choose which pruning method will be given in the following section 17.8.4.

17.8 The Training Procedure

The authors believe that complex real world problems should be attacked by a combination of various methods in order to handle the different types of difficulties which may arise during the optimization of neural networks. Thus, the aim of this section is to link all the previously described features to a complete and consistent training procedure.

17.8.1 Training Paradigms: Early vs. Late Stopping

One of the most well known techniques for attacking the overfitting problem is the early stopping procedure. In this procedure, we start the learning with a network initialized by small weights. In its most simple version one uses a validation set to estimate the beginning of overfitting, at which point the network learning is stopped. A more sophisticated variant of the procedure is to learn up to the start of overfitting and then to prune a part of the network structure, by say 10% of the weights. Then, one restarts the smaller network and repeats the steps. This sequence, which is schematically shown in fig. 17.19, has to be iterated until a stable structure with no signs of overfitting is achieved.

In principle this procedure may work and it will generate a model with good generalization performance, but in many cases it will fail to do so, as we have observed in our experiments during the last several years. Difficulties with early stopping arise because the stopping point turns up after a few learning epochs through the training data. The authors have worked on examples where the

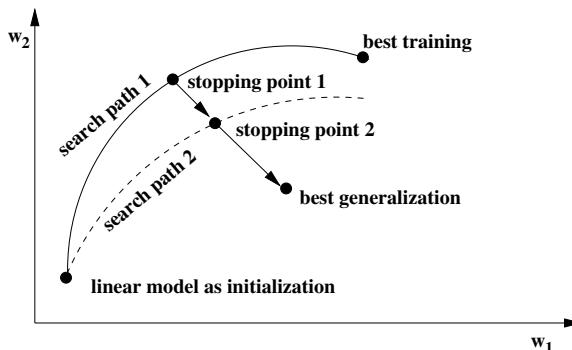


Fig. 17.19. Early stopping: After initialization with small weights the network is trained until the error on the validation set starts to increase (search path 1 to stopping point 1). Then, some weights are pruned, the remaining weights are reinitialized and training starts again until stopping point 2. This procedure has to be iterated until a stable structure with no signs of overfitting is achieved.

stopping point appeared as early as after the second epoch of learning. In this case, the solution is restricted to linear models, since the network has not been offered any chance to learn a complex nonlinearity from the data. A decreased learning rate does not mean a reduction of this bias, because it only slows down the movement away from the linear models. Using initial weights with larger values is also problematic for two reasons. The random initial (probably incorrect) specification of the network may lead to decreasing error curves due to shrinking weight values, but after a while overfitting will probably start again. Another important critique of this initialization is the intrinsic dependency of the solution on the starting point.

The same argument is true for some of the penalty term approaches for regularizing the networks. Weight decay, for example, is a regularizer with a bias towards linear models. Most of the other penalty functions (like data independent smoothness) set additional constraints on the model building process. These restrictions are not necessarily related to the task to be solved (see [16] for smoothing regularizers for neural networks with sigmoidal activation function in the hidden layer).

Following these thoughts, in this section we present a sequence of steps that follow a late stopping concept. We start with small weights to lower the influence of the random initialization. Then, we learn until the minimal error on the training data is achieved. Typically, the network shows clear overfitting at this point but we can be sure that the maximum nonlinearity which can be represented by the chosen network, has been learned from the data. At this point we must describe the way back to a solution with good generalization performance, as indicated in fig. 17.20.

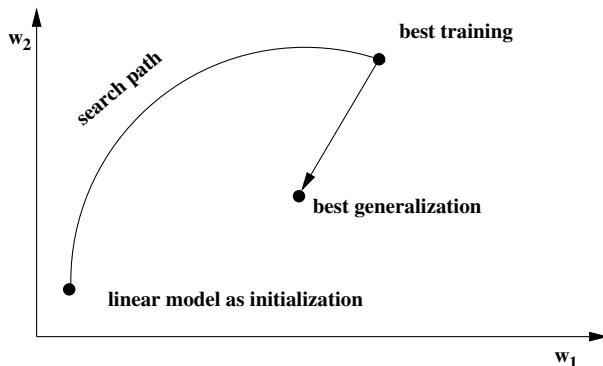


Fig. 17.20. Late stopping: After initialization with small weights the network is trained until the minimal training error. Subsequent optimization of the network structure by pruning increases generalization performance. Then, the network is trained again until the minimal training error.

Basically the two parts of the training, learning to the minimal training error and extracting an appropriate generalizing model, can be understood as a generation of a structural hypothesis followed by a falsification of the generated structure. These steps will be explained in detail in the next paragraphs.

17.8.2 Setup Steps

Before starting the learning process we have to specify the network architecture. Let us assume that our aim is the identification of a dynamical system. Then, we propose using the network architecture (fig. 17.8) of section 17.3, may be with the CDEN extension (fig. 17.14) of section 17.5 in order to additionally estimate error bars.

A further setup decision is the separation of the data set into a training set, a validation set and a generalization set. Possible separations are shown in fig. 17.21.

Considering forecasting models of time series (at least in economic applications), the sets should be explicitly separated in the order indicated in fig. 17.21.1. Otherwise there is no chance to test the model stability over time. If we randomly took out the validation set of the training set (fig. 17.21.2), we would have no chance to test this stability because the validation patterns are always embedded in training patterns.

In the proposed sequence we do not use the most recent data before the test set in the learning. As a consequence, one might tend to choose the validation patterns from the oldest part while using the most recent data in the training set (fig. 17.21.3). This leads to a better basis for the generation of our structural hypothesis by learning. Basically, the pruning methods are a falsification of the generated structure. Using the separation in fig. 17.21.3 with the validation set

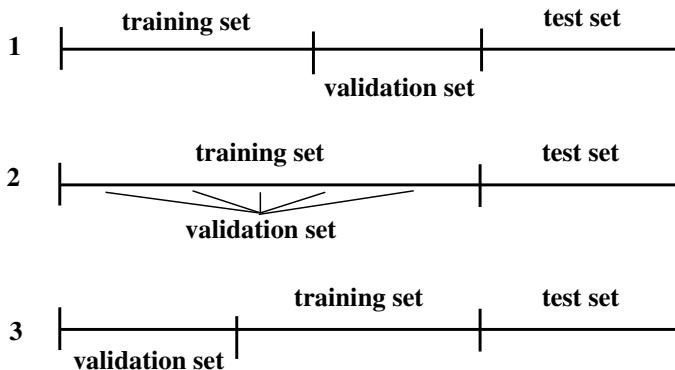


Fig. 17.21. Separation of the data

including the oldest data can make this falsification misleading. In a fast changing world, model stability over time is an important performance characteristic of a good network model. Thus, we prefer the separation in fig. 17.21.1.

The several preprocessed time series which serve as inputs have to be checked for correlation because highly correlated inputs only serve to increase the amount of numerical instability in our model. To avoid this, we can introduce a bottleneck substructure to perform a principle component analysis. For our typical application with about 50 inputs, we use net internal preprocessing by the diagonal matrix. For models with a very large number of inputs (some hundreds) the bottleneck approach may be superior.

If using the net internal preprocessing by the diagonal matrix, it is necessary to check the pairwise correlation of the inputs. Following the common guideline of using lots of chart indicators in financial modeling it is typical that many of these indicators have a high correlation. Keeping them in the training set will cause the solutions of different optimization runs to give us different answers regarding the importance of the input factors.

Since we often start with a number of weights which is large relative to the number of training patterns, we also propose using a small weight decay during learning. The penalty parameter should be in the range of 10% of the learning rate because we are not interested in reducing the number of effective parameters. With this small value of the decay parameter, only those weights which simply learn nothing are pulled to zero. By this, we still achieve the minimal training error, but eliminate all the unnecessary weights which have an unpredictable effect on the test set.

17.8.3 Learning: Generation of Structural Hypothesis

If confronted with a large data set, such as in the task of forecasting daily returns, we propose training the weights with the VarioEta adaptation algorithm. As a

stochastic approximation to a Newton method this is a fast learning technique. Obeying the arguments about the implicit penalty of section 17.6 we should let the VarioEta training be followed by a simple pattern by pattern learning with a constant learning rate to achieve structurally correct learning. As mentioned, it is valuable to hold the learning rate as high as possible to benefit from the implied penalty term. However, if interested in monthly forecasting models, one may use only the pattern-by-pattern learning because learning speed is not relevant due to the small data set. On the other hand, the implied curvature penalty is more important to generate good models (see section 17.6).

We propose to start the cleaning and the cleaning noise procedure of section 17.6 from the beginning. In this way, one can observe the following interesting relationship and interaction between the stochastic learning and the cleaning noise which improves the learning behavior. Since the initial noise variance is set to zero, the noise to the input variables will start with a low level and will then increase rapidly during the first learning epochs. After several epochs, when the network has captured some of the structure in the data, the noise decays in parallel with the residual input error of the network. As a consequence, in the beginning of the training, the network can learn only the global structure in the data. Later in the training process, more and more detailed features of the data are extracted which leads simultaneously to lower cleaning noise levels (fig. 17.22). Cleaning⁴ improves the learning process by sequencing the data structure from global to increasingly specialized features. Furthermore, noise improves the stochastic search in the weight space and therefore reduces the problem of local minima.

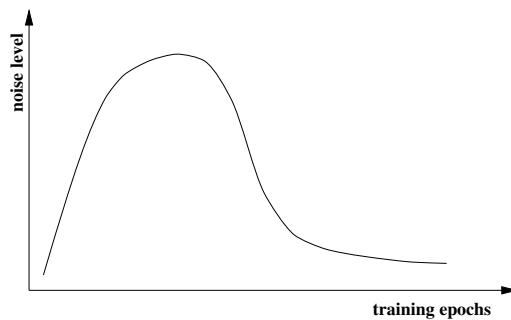


Fig. 17.22. Due to the large noise level in the beginning of the training phase, the network first learns global features. At later training steps, it is able to extract more and more details.

Depending on the problem, we observed that the implicitly controlled noise level can go down close to zero or it can converge to an intermediate level. We finish the step of learning to a minimal training error when we observe a stable

⁴ Even in an early stopping concept Cleaning may improve the learning.

behavior of the error function on the training and validation set simultaneously with the stable behavior of the mean noise level for all inputs on the training set. The final model of this step is a structural hypothesis about the dynamical system we are analyzing.

17.8.4 Pruning: Falsification of the generated Structure

Next, we have to test the stability of our model over time. Especially in economic modeling, it is not evident that the structure of the capital markets we are extracting from the seventies or eighties are still valid now. We can check this stability by testing the network behavior on the validation set which follows the training set on the time axis.

By using input pruning on the validation set (see section 17.7), we are able to identify input series that may be of high relevance in the training set but may also have no or even a counterproductive effect on the validation data. Note that, the pruning of non-relevant time series is not possible before achieving the minimal training error because there exists no well defined relationship between all the inputs and the target variables before. On the other hand, unimportant or inconsistent inputs with respect to the validation set should not be eliminated later in order to facilitate the construction of models with high generalization performance by subsequent learning.

An alternative way to check the stability of the model is to perform weight pruning on the validation set using the Instability-Pruning algorithm. If the data set is very large, weight pruning has a significant advantage because only one pass through the data set is necessary, in comparison to n passes to get the ranking of the input pruning with n as the number of inputs. This becomes very important in the field of data-mining where we have to deal with hundreds or thousands of megabytes of training data.

Each time after pruning of weights or inputs, the network is trained again to achieve a stable behavior of the error curves on the training / validation set (fig. 17.24) and of the noise level (fig. 17.23). Here, by stable behavior of the error curve we mean that there is no significant downward trend (i. e. improving of the model) or upward trend (i. e. restructuring of the model). In the diagram shown in fig. 17.25, these stop conditions are marked with (*).

To eliminate local artifacts in the network structure at this stage, we can include some iterations of Inverse-Kurtosis pruning and subsequent retraining. The process of iteratively pruning and training ends if a *drastic* decay, i. e. 5%, of this criterion is no longer observable. For several iterations of training and pruning, this strategy leads to a rapidly decreasing error on the validation set.

Due to the fact that the shrinking number of network parameters leads to increasing gradients, pattern by pattern learning with a constant learning rate is most useful because the implicit local curvature penalty becomes increasingly important.

One may criticize the proposed pruning policy because it is indeed an optimization on the validation set. Thus, the error on the validation set does not represent a good estimation of the generalization error anymore. We believe that there is no way to omit this pruning step since the test of the time stability is important for achieving models with high generalization performance.

In the subsequent part of the network learning, we use “Occam’s Razor” by weight pruning based on test values computed on the training set. This may be EBD or Instability-Pruning (see section 17.7.2). The performance of both methods based on hundreds of experiments is comparable. After pruning about 10% resp. 5% of the active weights, we train again until stable error curves and noise levels are obtained.

One can combine these pruning methods by first eliminating weights with Instability-Pruning and generating very sparse networks using EBD afterwards. Instability-Pruning favors model stability over time whereas EBD allows sparse models which still approximate well.

If using the eleven layer architecture of fig. 17.8, weight pruning should only be applied to the weights connecting the preprocessing or square layer to the hidden layers for two reasons. First, these connectors represent the majority of the weights in the neural network. Second, it is important to apply the pruning techniques only to such weights which have the same distance to the output neurons. Thus, the test values of the gradient based pruning algorithms are comparable which leads to a reliable ranking of the weights.⁵

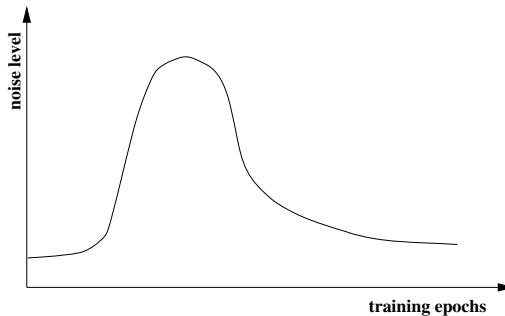


Fig. 17.23. Interaction between pruning and the adaptive noise level

⁵ If one decides not to use the eleven layer architecture but a typical neural network extended by random targets, one has to consider the following consequences for input and weight pruning. The test values have to be computed only with respect to the correct targets. By this, weights which are modeling random targets will lose their importance. EBD pruning works very well in identifying such flat minima weights (section 17.7.2), and thus, is our favorite pruning method for such a modeling approach.

There is an interesting interaction between the pruning and the adaptive noise procedure (see fig. 17.23). At each pruning step, we eliminate some parameters which have memorized some of the structure in the data. Thereby, the residual input errors will increase, which leads to increased noise levels on the inputs. Consequently, after each pruning step the learning has to focus on more global structures of the data. Owing to the fact that by pruning the network memory gradually decreases, the network cannot rely on any particular features of the data but rather is forced to concentrate more and more on the general underlying structure. Interestingly, this part of the procedure can be viewed as being the opposite of learning until the minimal training error. There the network is forced by the cleaning noise to learn the global structures first before trying to extract specific characteristics from the data.

17.8.5 Final Stopping Criteria of the Training

The last question we have to answer is the definition of the final stopping point. Obviously, after many weight pruning and training iterations, the error will increase for the training and the validation set. Thus, the minimal error on the validation set should give us an estimate of when to stop the optimization. In practice, the error curves during learning are not smooth and monotonic curves with only one minimum. The most simple advice is to prune until all weights are eliminated while keeping a trace to store the best intermediate solution on the validation set. Fig. 17.24 displays the error behavior in the different phases of the training procedure assuming the worst situation from the viewpoint of an early stopping method due to the immediately increasing error on the validation set.

Instability-Pruning offers an alternative definition of a final stopping point. If we substitute EBD by this pruning method for applying Occam's Razor, each

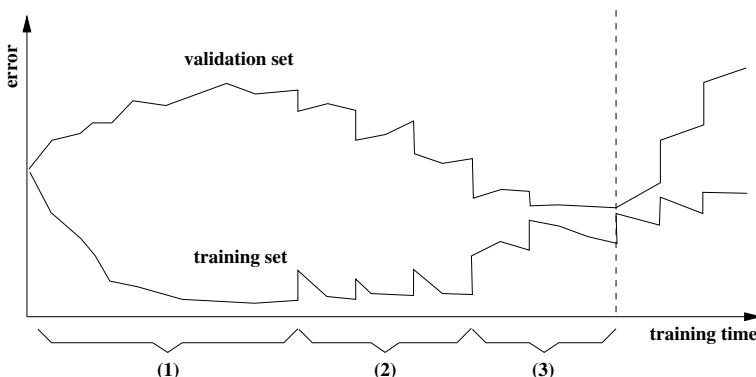


Fig. 17.24. The error behavior during the training. Region (1) describes the way to a minimal training error, (2) is the typical behavior during the input pruning on the validation set and (3) shows the consequence of Occam's razor in form of the pruning.

pruning / retraining iteration will increase the stability of the model because Instability-Pruning deletes unstable weights (see section 17.7.2).

17.8.6 Diagram of the Training Procedure

The following diagram in fig. 17.25 shows the training steps combined into a unified training procedure.

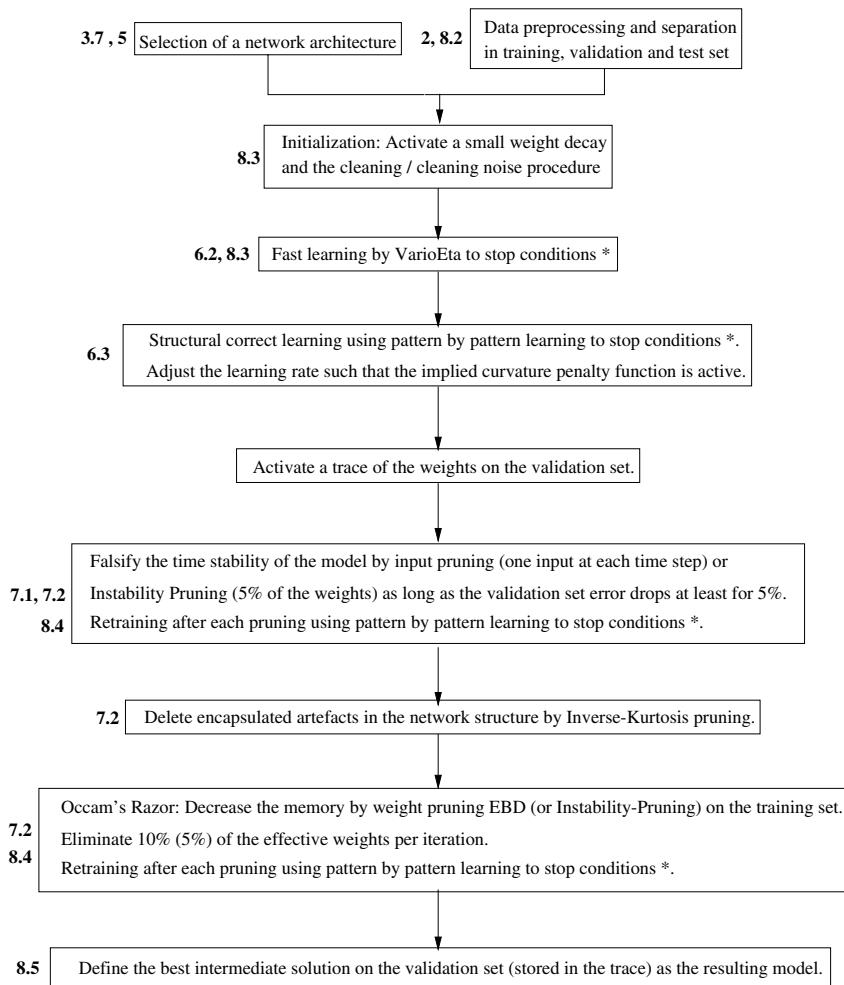


Fig. 17.25. The diagram of the training procedure. The marks (*) indicates the stopping points (section 17.8.3 and 17.8.4). Box numbers correspond to sections.

17.9 Experiments

In a research project sponsored by the European Community we are using the proposed approach to estimate the returns of 3 financial markets for each of the G7 countries [12]. These estimations will be subsequently used in an asset allocation scheme to create a Markowitz-optimal portfolio. In this paper, we only report the results of the estimation of the German bond rate, which is one of the more difficult tasks due to the reunification of Germany and GDR. Here, we have to predict the return of the bond rate 6 months ahead. The inputs consist of 39 variables obtained by preprocessing 16 relevant financial time series⁶. The training set covers the time from April 1974 to December 1991, the validation set from January 1992 to May 1994. The results are collected using the out-of-sample data which runs from June 1994 to May 1996. To demonstrate the behavior of the algorithms, we compare our approach with a standard neural network with one hidden layer (20 neurons, tanh transfer function) and one linear output (eq. 17.30 as error function). First, we trained the neural network until convergence with pattern-by-pattern learning using a small batch size of 20 patterns. We refer to this network as a classical approach. Then, we trained the 11-layer network (fig. 17.8) using the unified approach as described in section 4.1. Due to small data sets we used pattern-by-pattern learning without VarioEta. The data were manipulated by the cleaning and noise method of eq. 17.50. We compare the resulting predictions of the networks on the basis of three performance measures (see tab. 17.3). First, the hit rate counts how often the sign of the return of the bond has been correctly predicted. As for the other measures, the step from the forecast model to a trading system is here kept very simple. If the output is positive, we buy shares of the bond, otherwise we sell them. The potential realized is the ratio of the return to the maximum possible return over the training (test) set. The annualized return is the average yearly profit of the trading systems. Our approach turns out to be superior. For example, we almost doubled the annualized return from 4.5% to 8.5% on the test set. In fig. 17.26, we compare the accumulated return of the two approaches on the test set. The unified approach not only shows a higher profitability, but also has by far a less maximal draw down.

Table 17.3. Comparison of the hit rate, the realized potential and the annualized return of the two networks for the training (test) data

network	hit rate	realized potential	annualized return
our approach	96% (81%)	100% (75%)	11.22% (8.5%)
classical approach	93% (66%)	96% (44%)	10.08% (4.5%)

⁶ At the moment, we are not allowed to be more specific on the actual data we used.

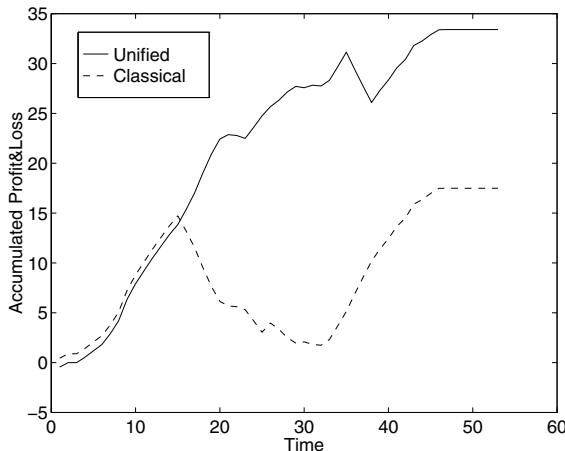


Fig. 17.26. Comparison of the accumulated profit&loss curve of the two approaches

17.10 Conclusion

A typical regression analysis by neural networks begins with the statement: “In principal a neural network can model everything”. If the data is not very reliable due to noise or missing factors, learning is only one part of the model building procedure. Often, task-independent regularization is used as weapon to reduce the uncertainty introduced by the data. More generally, the model building process demands more information (prior knowledge) about the specific task.

We have shown that there are a large number of techniques for including additional constraints which depend on the problem. Consequently, the resulting model is not only based on the training data but also on the additional constraints and also on the exact steps of the training procedure. From a Bayesian viewpoint, this can be described as an integration of priors in the model building process. Thus, this article can also be interpreted as a discussion of valuable priors and of how to combine them in order to achieve a maximal synergy.

The authors believe that the additional features are an outstanding advantage of neural networks and will lead to more robust and successful models. This paper gives an overview about some of the features we experienced as useful in our applications.

Finally, the unified training procedure is a recipe for building a model with neural networks by way of a relatively simple sequence of steps. An important aspect of this paper has been the study of the interactions between the different techniques. The described algorithms are integrated in the *Simulation Environment for Neural Networks*, SENN, a product of Siemens AG. More information can be found on the web page <http://www.senn.sni.de>.

References

- [1] Ackley, D.H., Hinton, G.E., Sejnowski, T.J.: A learning algorithm for Boltzmann machines. *Cognitive Science* 9, 147–169 (1985); Reprinted in [2]
- [2] Anderson, J.A., Rosenfeld, E. (eds.): *Neurocomputing: Foundations of Research*. The MIT Press, Cambridge (1988)
- [3] Bishop, C.M.: *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford (1995)
- [4] Breiman, L.: Bagging predictors. Technical Report TR No. 421, Department of Statistics, University of California (1994)
- [5] Bunke, H., Bunke, O.: *Nonlinear Regression, Functional Analysis and Robust Methods*, vol. 2. John Wiley and Sons (1989)
- [6] Caruana, R.: Multitask learning. *Machine Learning* 28, 41 (1997)
- [7] Elton, E.J., Gruber, M.J.: *Modern Portfolio Theory and Investment Analysis*. John Wiley & Sons (1995)
- [8] Finnoff, W., Hergert, F., Zimmermann, H.G.: Improving generalization performance by nonconvergent model selection methods. In: Aleksander, I., Taylor, J. (eds.) *Proc. of the Inter. Conference on Artificial Neural Networks, ICANN 1992*, vol. 2, pp. 233–236 (1992)
- [9] Finnoff, W., Hergert, F., Zimmermann, H.G.: Neuronale Lernverfahren mit variabler Schrittweite, Tech. report, Siemens AG (1993)
- [10] Flake, G.W.: Square Unit Augmented, Radially Extended, Multilayer Perceptrons. In: Orr, G.B., Müller, K.-R. (eds.) *NN: Tricks of the Trade*, 1st edn. LNCS, vol. 7700, pp. 143–161. Springer, Heidelberg (2012)
- [11] Gershenfeld, N.A.: An experimentalist’s introduction to the observation of dynamical systems. In: Hao, B.L. (ed.) *Directions in Chaos*, vol. 2, pp. 310–384. World Scientific, Singapore (1989)
- [12] Herve, P., Naim, P., Zimmermann, H.G.: Advanced Adaptive Architectures for Asset Allocation: A Trial Application. In: *Forecasting Financial Markets* (1996)
- [13] Hochreiter, S., Schmidhuber, J.: Flat minima. *Neural Computation* 9(1), 1–42 (1997)
- [14] Hornik, K.: Approximation Capabilities of Multilayer Feedforward Networks. *Neural Networks* 4, 251–257 (1991)
- [15] le Cun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In: Touretzky, D.S. (ed.) *Advances in Neural Information Processing Systems, NIPS 1989*, vol. 2, pp. 598–605. Morgan Kaufmann, San Mateo (1990)
- [16] Moody, J.E., Rögnvaldsson, T.S.: Smoothing regularizers for projective basis function networks. In: Mozer, M.C., Jordan, M.I., Petsche, T. (eds.) *Advances in Neural Information Processing Systems*, vol. 9, p. 585. The MIT Press (1997)
- [17] Williams, P.M.: Using Neural Networks to Model Conditional Multivariate Densities. Technical Report CSRP 371, School of Cognitive and Computing Sciences, Univ. of Sussex (February 1995)
- [18] Neuneier, R.: Optimal asset allocation using adaptive dynamic programming. In: *Advances in Neural Information Processing Systems*, vol. 8. MIT Press (1996)
- [19] Neuneier, R.: Optimale Investitionsentscheidungen mit Neuronalen Netzen. PhD thesis, Universität Kaiserslautern, Institut für Informatik (1998)
- [20] Neuneier, R., Finnoff, W., Hergert, F., Ormoneit, D.: Estimation of Conditional Densities: A Comparison of Neural Network Approaches. In: *Intern. Conf. on Artificial Neural Networks, ICANN*, vol. 1, pp. 689–692. Springer (1994)
- [21] Nix, D.A., Weigend, A.S.: Estimating the mean and variance of the target probability distribution. In: *World Congress of Neural Networks*. Lawrence Erlbaum Associates (1994)

- [22] Ormoneit, D.: Estimation of Probability Densities using Neural Networks. Master's thesis, Fakultät für Informatik, Technische Universität München (1993)
- [23] Papoulis, A.: Probability, Random Variables, and Stochastic Processes, 3rd edn. McGraw Hill, Inc. (1991)
- [24] Perrone, M.P.: Improving Regression Estimates: Averaging Methods for Variance Reduction with Extensions to General Convex Measure Optimization. PhD thesis, Brown University (1993)
- [25] Refenes, A.P. (ed.): Neural Networks in the Capital Market. Wiley & Sons (1994)
- [26] Sanger, T.D.: Optimal unsupervised learning in a single-layer linear feedforward network. *Neural Networks* 2, 459–473 (1989)
- [27] Seber, G.A.F., Wild, C.J.: Nonlinear Regression. John Wiley & Sons, New York (1989)
- [28] Srivastava, A.N., Weigend, A.S.: Computing the probability density in connectionist regression. In: Marinaro, M., Morasso, P.G. (eds.) *Proceedings of the International Conference on Artificial Neural Networks*, Sorrento, Italy (ICANN 1994), pp. 685–688. Springer (1994); Also in *Proceedings of the IEEE International Conference on Neural Networks*, Orlando, FL (IEEE-ICNN 1994), pp. 3786–3789. IEEE Press (1994)
- [29] Takens, F.: Detecting Strange Attractors in Turbulence. In: Rand, D.A., Young, L.S. (eds.) *Dynamical Systems and Turbulence*. Lecture Notes in Mathematics, vol. 898, pp. 366–381. Springer (1981)
- [30] Tang, B., Hsieh, W., Tangang, F.: Clearning neural networks with continuity constraints for prediction of noisy time series. In: *Progres in Neural Information Processing* (ICONIP 1996), pp. 722–725. Springer, Berlin (1996)
- [31] Tresp, V., Neuneier, R., Zimmermann, H.G.: Early brain damage. In: *Advances in Neural Information Processing Systems*, vol. 9. MIT Press (1997)
- [32] Weigend, A.S., Zimmermann, H.G.: Exploiting local relations as soft constraints to improve forecasting. *Computational Intelligence in Finance* 6(1) (January 1998)
- [33] Weigend, A.S., Zimmermann, H.G., Neuneier, R.: The observer-observation dilemma in neuro-forecasting: Reliable models from unreliable data through clearning. In: Freedman, R. (ed.) *AI Applications on Wall Street*, pp. 308–317. Software Engineering Press, New York (1995)
- [34] Weigend, A.S., Rumelhart, D.E., Huberman, B.A.: Generalization by weight-elimination with application to forecasting. In: Lippmann, R.P., Moody, J.E., Touretzky, D.S. (eds.) *Advances in Neural Information Processing Systems*, vol. 3, pp. 875–882. Morgan Kaufmann, San Mateo (1991)
- [35] White, H.: Parametrical statistical estimation with artificial neural networks. Technical report, University of California, San Diego (1991)
- [36] Zimmermann, H.G., Weigend, A.S.: Representing dynamical systems in feed-forward networks: A six layer architecture. In: Weigend, A.S., Abu-Mostafa, Y., Refenes, A.-P.N. (eds.) *Decision Technologies for Financial Engineering: Proceedings of the Fourth International Conference on Neural Networks in the Capital Markets* (NNCM 1996). World Scientific, Singapore (1997)
- [37] Zimmermann, H.G.: Neuronale Netze als Entscheidungskalkül. In: Rehkugler, H., Zimmermann, H.G. (eds.) *Neuronale Netze in der Ökonomie*. Verlag Franz Vahlen (1994)
- [38] Zimmermann, H.G., Neuneier, R.: The observer-observation dilemma in neuro-forecasting. In: *Advances in Neural Information Processing Systems*, vol. 10. MIT Press (1998)

Big Learning and Deep Neural Networks

Preface

More data and compute resources opens the way to “big learning”, that is, scaling up machine learning to large data sets and complex problems. In order to solve these new problems, we need to identify the complex dependencies that interrelate inputs and outputs [1]. Achieving this goal requires powerful algorithms in terms of both representational power and ability to absorb and distill information from large data flows. More and more it becomes apparent that neural networks provide an excellent toolset to scale learning. This holds true for simple linear systems as well as today’s grand challenges where the underlying application problem requires high nonlinearity and complex structured representations.

The following four chapters introduce the basic toolbox for solving these complex learning problems. They include first and second-order optimization methods, the best practice of training neural networks and an introduction to *Torch7*, a neural network library for implementing large-scale learning problems.

A first challenge is to feed the model with enough data in order to properly identify the rich set of dependencies. Chapter 18 [2] presents the stochastic gradient descent algorithm (learning one example at the time), showing that it can minimize the objective function at a rate that is under certain conditions asymptotically optimal.

Neural networks are a useful framework to carry out such optimization. The *correspondence principle for neural networks* that will be detailed later in Chapter 28 [7] states that we can associate to an equation or an optimization problem, a neural network architecture that reduces the learning problem to an exchange of local signals between neighboring units of the network. Therefore, the tractability of the optimization problem is much dependent on the neural network architecture and its multiple hyperparameters [3]. These hyperparameters and the methods to choose them are dissected in Chapter 19 [5].

In some cases, the multiple hyperparameters of the neural network are insufficient to produce a well-conditioned optimization problem. Difficult optimization problems include, for example, recurrent neural networks or deep neural networks, both involving long-range dependencies. Exploiting second order information (i.e. the second derivatives of the objective function) can lead to faster convergence. Second-order methods are much more sensitive to noise than first order methods and require carefully tuned optimization parameters in order to achieve a good speed-stability tradeoff. Second-order methods for deep and recurrent neural networks are discussed in much depth in Chapter 20 [6].

Finally, algorithms must be translated into code that can be run by the machine. Chapter 21 [4] presents *Torch7*, an efficient and easy to use software for building and training neural networks. The software provides a large number of neural network primitives and is highly modular so that it can be applied to a wide range of problems including computer vision, natural language processing, speech recognition and many more.

Grégoire & Klaus

References

- [1] Bengio, Y., LeCun, Y.: Scaling learning algorithms towards AI. In: Large Scale Kernel Machines (2007)
- [2] Bottou, L.: Stochastic Gradient Descent Tricks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 421–436. Springer, Heidelberg (2012)
- [3] LeCun, Y., Bottou, L., Orr, G.B., Müller, K.-R.: Efficient BackProp. In: Orr, G.B., Müller, K.-R. (eds.) NIPS-WS 1996. LNCS, vol. 1524, pp. 9–50. Springer, Heidelberg (1998)
- [4] Collobert, R., Kavukcuoglu, K., Farabet, C.: Implementing Neural Networks Efficiently. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 537–557. Springer, Heidelberg (2012)
- [5] Bengio, Y.: Practical Recommendations for Gradient-based Training of Deep Architectures. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 437–478. Springer, Heidelberg (2012)
- [6] Martens, J., Sutskever, I.: Training Deep and Recurrent Networks with Hessian-free Optimization. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 479–535. Springer, Heidelberg (2012)
- [7] Zimmermann, H.-G., Tietz, C., Grothmann, R.: Forecasting with Recurrent Neural Networks: 12 Tricks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 687–707. Springer, Heidelberg (2012)

18

Stochastic Gradient Descent Tricks

Léon Bottou

Microsoft Research, Redmond, WA
leon@bottou.org
<http://leon.bottou.org>

Abstract. Chapter 1 strongly advocates the *stochastic back-propagation* method to train neural networks. This is in fact an instance of a more general technique called *stochastic gradient descent* (SGD). This chapter provides background material, explains why SGD is a good learning algorithm when the training set is large, and provides useful recommendations.

18.1 Introduction

Chapter 1 strongly advocates the *stochastic back-propagation* method to train neural networks. This is in fact an instance of a more general technique called *stochastic gradient descent* (SGD). This chapter provides background material, explains why SGD is a good learning algorithm when the training set is large, and provides useful recommendations.

18.2 What Is Stochastic Gradient Descent?

Let us first consider a simple supervised learning setup. Each example z is a pair (x, y) composed of an arbitrary input x and a scalar output y . We consider a *loss function* $\ell(\hat{y}, y)$ that measures the cost of predicting \hat{y} when the actual answer is y , and we choose a family \mathcal{F} of functions $f_w(x)$ parametrized by a weight vector w . We seek the function $f \in \mathcal{F}$ that minimizes the loss $Q(z, w) = \ell(f_w(x), y)$ averaged on the examples. Although we would like to average over the unknown distribution $dP(z)$ that embodies the Laws of Nature, we must often settle for computing the average on a sample $z_1 \dots z_n$.

$$E(f) = \int \ell(f(x), y) dP(z) \quad E_n(f) = \frac{1}{n} \sum_{i=1}^n \ell(f(x_i), y_i) \quad (18.1)$$

The *empirical risk* $E_n(f)$ measures the training set performance. The *expected risk* $E(f)$ measures the generalization performance, that is, the expected performance on future examples. The statistical learning theory [25] justifies minimizing the empirical risk instead of the expected risk when the chosen family \mathcal{F} is sufficiently restrictive.

18.2.1 Gradient Descent

It has often been proposed (e.g., [18]) to minimize the empirical risk $E_n(f_w)$ using *gradient descent* (GD). Each iteration updates the weights w on the basis of the gradient of $E_n(f_w)$,

$$w_{t+1} = w_t - \gamma \frac{1}{n} \sum_{i=1}^n \nabla_w Q(z_i, w_t), \quad (18.2)$$

where γ is an adequately chosen learning rate. Under sufficient regularity assumptions, when the initial estimate w_0 is close enough to the optimum, and when the learning rate γ is sufficiently small, this algorithm achieves *linear convergence* [6], that is, $-\log \rho \sim t$, where ρ represents the residual error.¹

Much better optimization algorithms can be designed by replacing the scalar learning rate γ by a positive definite matrix Γ_t that approaches the inverse of the Hessian of the cost at the optimum :

$$w_{t+1} = w_t - \Gamma_t \frac{1}{n} \sum_{i=1}^n \nabla_w Q(z_i, w_t). \quad (18.3)$$

This *second order gradient descent* (2GD) is a variant of the well known Newton algorithm. Under sufficiently optimistic regularity assumptions, and provided that w_0 is sufficiently close to the optimum, second order gradient descent achieves *quadratic convergence*. When the cost is quadratic and the scaling matrix Γ is exact, the algorithm reaches the optimum after a single iteration. Otherwise, assuming sufficient smoothness, we have $-\log \log \rho \sim t$.

18.2.2 Stochastic Gradient Descent

The *stochastic gradient descent* (SGD) algorithm is a drastic simplification. Instead of computing the gradient of $E_n(f_w)$ exactly, each iteration estimates this gradient on the basis of a single randomly picked example z_t :

$$w_{t+1} = w_t - \gamma_t \nabla_w Q(z_t, w_t). \quad (18.4)$$

The stochastic process $\{w_t, t=1, \dots\}$ depends on the examples randomly picked at each iteration. It is hoped that (18.4) behaves like its expectation (18.2) despite the noise introduced by this simplified procedure.

Since the stochastic algorithm does not need to remember which examples were visited during the previous iterations, it can process examples on the fly in a deployed system. In such a situation, the stochastic gradient descent directly optimizes the expected risk, since the examples are randomly drawn from the ground truth distribution.

¹ For mostly historical reasons, *linear convergence* means that the residual error asymptotically decreases exponentially, and *quadratic convergence* denotes an even faster asymptotic convergence. Both convergence rates are considerably faster than the SGD convergence rates discussed in section 18.2.3.

Table 18.1. Stochastic gradient algorithms for various learning systems

Loss	Stochastic gradient algorithm
Adaline [26]	
$Q_{\text{adaline}} = \frac{1}{2}(y - w^\top \Phi(x))^2$ Features $\Phi(x) \in \mathbb{R}^d$, Classes $y = \pm 1$	$w \leftarrow w + \gamma_t (y_t - w^\top \Phi(x_t)) \Phi(x_t)$
Perceptron [17]	
$Q_{\text{perceptron}} = \max\{0, -y w^\top \Phi(x)\}$ Features $\Phi(x) \in \mathbb{R}^d$, Classes $y = \pm 1$	$w \leftarrow w + \gamma_t \begin{cases} y_t \Phi(x_t) & \text{if } y_t w^\top \Phi(x_t) \leq 0 \\ 0 & \text{otherwise} \end{cases}$
K-Means [12]	
$Q_{\text{kmeans}} = \min_k \frac{1}{2}(z - w_k)^2$ Data $z \in \mathbb{R}^d$ Centroids $w_1 \dots w_k \in \mathbb{R}^d$ Counts $n_1 \dots n_k \in \mathbb{N}$, initially 0	$k^* = \arg \min_k (z_t - w_k)^2$ $n_{k^*} \leftarrow n_{k^*} + 1$ $w_{k^*} \leftarrow w_{k^*} + \frac{1}{n_{k^*}}(z_t - w_{k^*})$ (counts provide optimal learning rates!)
SVM [5]	
$Q_{\text{svm}} = \lambda w^2 + \max\{0, 1 - y w^\top \Phi(x)\}$ Features $\Phi(x) \in \mathbb{R}^d$, Classes $y = \pm 1$ Hyperparameter $\lambda > 0$	$w \leftarrow w - \gamma_t \begin{cases} \lambda w & \text{if } y_t w^\top \Phi(x_t) > 1, \\ \lambda w - y_t \Phi(x_t) & \text{otherwise.} \end{cases}$
Lasso [23]	
$Q_{\text{lasso}} = \lambda w _1 + \frac{1}{2}(y - w^\top \Phi(x))^2$ $w = (u_1 - v_1, \dots, u_d - v_d)$ Features $\Phi(x) \in \mathbb{R}^d$, Classes $y = \pm 1$ Hyperparameter $\lambda > 0$	$u_i \leftarrow [u_i - \gamma_t (\lambda - (y_t - w^\top \Phi(x_t)) \Phi_i(x_t))]_+$ $v_i \leftarrow [v_i - \gamma_t (\lambda + (y_t - w^\top \Phi(x_t)) \Phi_i(x_t))]_+$ with notation $[x]_+ = \max\{0, x\}$.

Table 18.1 illustrates stochastic gradient descent algorithms for a number of classic machine learning schemes. The stochastic gradient descent for the Perceptron, for the Adaline, and for k -Means match the algorithms proposed in the original papers. The SVM and the Lasso were first described with traditional optimization techniques. Both Q_{svm} and Q_{lasso} include a regularization term controlled by the hyper-parameter λ . The K-means algorithm converges to a local minimum because Q_{kmeans} is nonconvex. On the other hand, the proposed update rule uses second order learning rates that ensure a fast convergence. The proposed Lasso algorithm represents each weight as the difference of two positive variables. Applying the stochastic gradient rule to these variables and enforcing their positivity leads to sparser solutions.

18.2.3 The Convergence of Stochastic Gradient Descent

The convergence of stochastic gradient descent has been studied extensively in the stochastic approximation literature. Convergence results usually require decreasing learning rates satisfying the conditions $\sum_t \gamma_t^2 < \infty$ and $\sum_t \gamma_t = \infty$. The Robbins-Siegmund theorem [16] provides the means to establish almost sure convergence under surprisingly mild conditions [3], including cases where the loss function is non smooth.

The convergence speed of stochastic gradient descent is in fact limited by the noisy approximation of the true gradient. When the learning rates decrease too slowly, the variance of the parameter estimate w_t decreases equally slowly. When the learning rates decrease too quickly, the expectation of the parameter estimate w_t takes a very long time to approach the optimum.

- When the Hessian matrix of the cost function at the optimum is strictly positive definite, the best convergence speed is achieved using learning rates $\gamma_t \sim t^{-1}$ (e.g. [14]). The expectation of the residual error then decreases with similar speed, that is, $\mathbb{E}(\rho) \sim t^{-1}$. These theoretical convergence rates are frequently observed in practice.
- When we relax these regularity assumptions, the theory suggests slower asymptotic convergence rates, typically like $\mathbb{E}(\rho) \sim t^{-1/2}$ (e.g., [28]). In practice, the convergence only slows down during the final stage of the optimization process. This may not matter in practice because one often stops the optimization before reaching this stage (see section 18.3.1.)

Second order stochastic gradient descent (2SGD) multiplies the gradients by a positive definite matrix Γ_t approaching the inverse of the Hessian :

$$w_{t+1} = w_t - \gamma_t \Gamma_t \nabla_w Q(z_t, w_t). \quad (18.5)$$

Unfortunately, this modification does not reduce the stochastic noise and therefore does not significantly improve the variance of w_t . Although constants are improved, the expectation of the residual error still decreases like t^{-1} , that is, $\mathbb{E}(\rho) \sim t^{-1}$ at best, (e.g. [1], appendix).

Therefore, as an optimization algorithm, stochastic gradient descent is asymptotically *much slower* than a typical batch algorithm. However, this is not the whole story...

18.3 When to Use Stochastic Gradient Descent?

During the last decade, the data sizes have grown faster than the speed of processors. In this context, the capabilities of statistical machine learning methods is limited by the computing time rather than the sample size. The analysis presented in this section shows that stochastic gradient descent performs very well in this context.

Use stochastic gradient descent
when training time is the bottleneck.

18.3.1 The Trade-Offs of Large Scale Learning

Let $f^* = \arg \min_f E(f)$ be the best possible prediction function. Since we seek the prediction function from a parametrized family of functions \mathcal{F} , let

$f_{\mathcal{F}}^* = \arg \min_{f \in \mathcal{F}} E(f)$ be the best function in this family. Since we optimize the empirical risk instead of the expected risk, let $f_n = \arg \min_{f \in \mathcal{F}} E_n(f)$ be the empirical optimum. Since this optimization can be costly, let us stop the algorithm when it reaches a solution \tilde{f}_n that minimizes the objective function with a predefined accuracy $E_n(\tilde{f}_n) < E_n(f_n) + \rho$. The excess error $\mathcal{E} = \mathbb{E}[E(\tilde{f}_n) - E(f^*)]$ can then be decomposed in three terms [2]:

$$\mathcal{E} = \underbrace{\mathbb{E}[E(f_{\mathcal{F}}^*) - E(f^*)]}_{\mathcal{E}_{\text{app}}} + \underbrace{\mathbb{E}[E(f_n) - E(f_{\mathcal{F}}^*)]}_{\mathcal{E}_{\text{est}}} + \underbrace{\mathbb{E}[E(\tilde{f}_n) - E(f_n)]}_{\mathcal{E}_{\text{opt}}}. \quad (18.6)$$

- The approximation error $\mathcal{E}_{\text{app}} = \mathbb{E}[E(f_{\mathcal{F}}^*) - E(f^*)]$ measures how closely functions in \mathcal{F} can approximate the optimal solution f^* . The approximation error can be reduced by choosing a larger family of functions.
- The estimation error $\mathcal{E}_{\text{est}} = \mathbb{E}[E(f_n) - E(f_{\mathcal{F}}^*)]$ measures the effect of minimizing the empirical risk $E_n(f)$ instead of the expected risk $E(f)$. The estimation error can be reduced by choosing a smaller family of functions or by increasing the size of the training set.
- The optimization error $\mathcal{E}_{\text{opt}} = \mathbb{E}[E(\tilde{f}_n) - E(f_n)]$ measures the impact of the approximate optimization on the expected risk. The optimization error can be reduced by running the optimizer longer. The additional computing time depends of course on the family of function and on the size of the training set.

Given constraints on the maximal computation time T_{\max} and the maximal training set size n_{\max} , this decomposition outlines a trade-off involving the size of the family of functions \mathcal{F} , the optimization accuracy ρ , and the number of examples n effectively processed by the optimization algorithm.

$$\min_{\mathcal{F}, \rho, n} \mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}} \quad \text{subject to} \quad \begin{cases} n \leq n_{\max} \\ T(\mathcal{F}, \rho, n) \leq T_{\max} \end{cases} \quad (18.7)$$

Two cases should be distinguished:

- *Small-scale learning problems* are first constrained by the maximal number of examples. Since the computing time is not an issue, we can reduce the optimization error \mathcal{E}_{opt} to insignificant levels by choosing ρ arbitrarily small, and we can minimize the estimation error \mathcal{E}_{est} by choosing $n = n_{\max}$. We then recover the approximation-estimation trade-off that has been widely studied in statistics and in learning theory.
- *Large-scale learning problems* are constrained by the maximal computing time, usually because the supply of training examples is very large. Approximate optimization can achieve better expected risk because more training examples can be processed during the allowed time. The specifics depend on the computational properties of the chosen optimization algorithm.

18.3.2 Asymptotic Analysis of the Large-Scale Case

Solving (18.7) in the asymptotic regime amounts to ensuring that the terms of the decomposition (18.6) decrease at similar rates. Since the asymptotic convergence rate of the excess error (18.6) is the convergence rate of its slowest term, the computational effort required to make a term decrease faster would be wasted.

For simplicity, we assume in this section that the Vapnik-Chervonenkis dimensions of the families of functions \mathcal{F} are bounded by a common constant. We also assume that the optimization algorithms satisfy all the assumptions required to achieve the convergence rates discussed in section 18.2. Similar analyses can be carried out for specific algorithms under weaker assumptions (e.g. [22]).

A simple application of the uniform convergence results of [25] gives then the upper bound

$$\mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}} = \mathcal{E}_{\text{app}} + \mathcal{O}\left(\sqrt{\frac{\log n}{n}} + \rho\right).$$

Unfortunately the convergence rate of this bound is too pessimistic. Faster convergence occurs when the loss function has strong convexity properties [9] or when the data distribution satisfies certain assumptions [24]. The equivalence

$$\mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}} \sim \mathcal{E}_{\text{app}} + \left(\frac{\log n}{n}\right)^\alpha + \rho, \quad \text{for some } \alpha \in [\frac{1}{2}, 1], \quad (18.8)$$

provides a more realistic view of the asymptotic behavior of the excess error (e.g. [13, 4]). Since the three components of the excess error should decrease at the same rate, the solution of the trade-off problem (18.7) must then obey the multiple asymptotic equivalences

$$\mathcal{E} \sim \mathcal{E}_{\text{app}} \sim \mathcal{E}_{\text{est}} \sim \mathcal{E}_{\text{opt}} \sim \left(\frac{\log n}{n}\right)^\alpha \sim \rho. \quad (18.9)$$

Table 18.2 summarizes the asymptotic behavior of the four gradient algorithms described in section 18.2. The first three rows list the computational cost of each iteration, the number of iterations required to reach an optimization accuracy ρ , and the corresponding computational cost. The last row provides a more interesting measure for large scale machine learning purposes. Assuming we operate at the optimum of the approximation-estimation-optimization trade-off (18.7), this line indicates the computational cost necessary to reach a predefined value of the excess error, and therefore of the expected risk. This is computed by applying the equivalences (18.9) to eliminate the variables n and ρ from the third row results.²

Although the stochastic gradient algorithms, SGD and 2SGD, are clearly the worst optimization algorithms (third row), they need less time than the

² Note that $\varepsilon^{1/\alpha} \sim \log(n)/n$ implies both $\alpha^{-1} \log \varepsilon \sim \log \log(n) - \log(n) \sim -\log(n)$ and $n \sim \varepsilon^{-1/\alpha} \log n$. Replacing $\log(n)$ in the latter gives $n \sim \varepsilon^{-1/\alpha} \log(1/\varepsilon)$.

Table 18.2. Asymptotic equivalents for various optimization algorithms: gradient descent (GD, eq. 18.2), second order gradient descent (2GD, eq. 18.3), stochastic gradient descent (SGD, eq. 18.4), and second order stochastic gradient descent (2SGD, eq. 18.5). Although they are the worst optimization algorithms, SGD and 2SGD achieve the fastest convergence speed on the expected risk. They differ only by constant factors not shown in this table, such as condition numbers and weight vector dimension.

	GD	2GD	SGD	2SGD
Time per iteration:	n	n	1	1
Iterations to accuracy ρ :	$\log \frac{1}{\rho}$	$\log \log \frac{1}{\rho}$	$1/\rho$	$1/\rho$
Time to accuracy ρ :	$n \log \frac{1}{\rho}$	$n \log \log \frac{1}{\rho}$	$1/\rho$	$1/\rho$
Time to excess error ε :	$\frac{1}{\varepsilon^{1/\alpha}} \log \frac{1}{\varepsilon}$	$\frac{1}{\varepsilon^{1/\alpha}} \log \frac{1}{\varepsilon} \log \log \frac{1}{\varepsilon}$	$1/\varepsilon$	$1/\varepsilon$

other algorithms to reach a predefined expected risk (fourth row). Therefore, in the large scale setup, that is, when the limiting factor is the computing time rather than the number of examples, the stochastic learning algorithms performs asymptotically better!

18.4 General Recommendations

The rest of this contribution provides a series of recommendations for using stochastic gradient algorithms. Although some of these recommendations seem trivial, experience has shown again and again how easily they can be overlooked.

18.4.1 Preparing the Data

Randomly shuffle the training examples.

Although the theory calls for picking examples randomly, it is usually faster to zip sequentially through the training set. But this does not work if the examples are grouped by class or come in a particular order. Randomly shuffling the examples eliminates this source of problems. Section 1.4.2 provides an additional discussion.

Use preconditioning techniques.

Stochastic gradient descent is a first-order algorithm and therefore suffers dramatically when it reaches an area where the Hessian is ill-conditioned. Fortunately, many simple preprocessing techniques can vastly improve the situation. Sections 1.4.3 and 1.5.3 provide many useful tips.

18.4.2 Monitoring and Debugging

**Monitor both the training cost
and the validation error.**

Since stochastic gradient descent is useful when the training time is the primary concern, we can spare some training examples to build a decent validation set. It is important to periodically evaluate the validation error during training because we can stop training when we observe that the validation error has not improved in a long time.

It is also important to periodically compute the training cost because stochastic gradient descent is an iterative optimization algorithm. Since the training cost is exactly what the algorithm seeks to optimize, the training cost should be generally decreasing.

A good approach is to repeat the following operations:

1. Zip once through the shuffled training set and perform the stochastic gradient descent updates (18.4).
2. With an additional loop over the training set, compute the training cost. Training cost here means the criterion that the algorithm seeks to optimize. You can take advantage of the loop to compute other metrics, but the training cost is the one to watch
3. With an additional loop over the validation set, to compute the validation set error. Error here means the performance measure of interest, such as the classification error. You can also take advantage of this loop to cheaply compute other metrics.

Computing the training cost and the validation error represent a significant computational effort because it requires additional passes over the training and validation data. But this beats running blind.

Check the gradients using finite differences.

When the computation of the gradients is slightly incorrect, stochastic gradient descent often works slowly and erratically. This has led many to believe that slow and erratic is the normal operation of the algorithm.

During the last twenty years, I have often been approached for advice in setting the learning rates γ_t of some rebellious stochastic gradient descent program. My advice is to forget about the learning rates and check that the gradients are computed correctly. This reply is biased because people who compute the gradients correctly quickly find that setting small enough learning rates is easy. Those who ask usually have incorrect gradients. Carefully checking each line of the gradient computation code is the wrong way to check the gradients. Use finite differences:

1. Pick an example z .
2. Compute the loss $Q(z, w)$ for the current w .
3. Compute the gradient $g = \nabla_w Q(z, w)$.
4. Apply a slight perturbation $w' = w + \delta$. For instance, change a single weight by a small increment, or use $\delta = -\gamma g$ with γ small enough.
5. Compute the new loss $Q(z, w')$ and verify that $Q(z, w') \approx Q(z, w) + \delta g$.

This process can be automated and should be repeated for many examples z , many perturbations δ , and many initial weights w . Flaws in the gradient computation tend to only appear when peculiar conditions are met. It is not uncommon to discover such bugs in SGD code that has been quietly used for years.

Experiment with the learning rates γ_t using a small sample of the training set.

The mathematics of stochastic gradient descent are amazingly independent of the training set size. In particular, the asymptotic SGD convergence rates [14] are independent from the sample size. Therefore, assuming the gradients are correct, the best way to determine the correct learning rates is to perform experiments using a small but representative sample of the training set. Because the sample is small, it is also possible to run traditional optimization algorithms on this same dataset in order to obtain reference point and set the training cost target.

When the algorithm performs well on the training cost of the small dataset, keep the same learning rates, and let it soldier on the full training set. Expect the validation performance to plateau after a number of epochs roughly comparable to the number of epochs needed to reach this point on the small training set.

18.5 Linear Models with L_2 Regularization

This section provides specific recommendations for training large linear models with L_2 regularization. The training objective of such models has the form

$$E_n(w) = \frac{\lambda}{2} \|w\|^2 + \frac{1}{n} \sum_{i=1}^n \ell(y_i w x_i) \quad (18.10)$$

where $y_i = \pm 1$, and where the function $\ell(m)$ is convex. The corresponding stochastic gradient update is then obtained by approximating the derivative of the sum by the derivative of the loss with respect to a single example

$$w_{t+1} = (1 - \gamma_t \lambda) w_t - \gamma_t y_t x_t \ell'(y_t w_t x_t) \quad (18.11)$$

Examples:

- Support Vector Machines (SVM) use the non differentiable *hinge loss* [5]:

$$\ell(m) = \max\{0, 1 - m\}.$$

- It is often more convenient in the linear case to use the *log-loss*:

$$\ell(m) = \log(1 + e^{-m}).$$

The differentiable log-loss is more suitable for the gradient algorithms discussed here. This choice leads to a logistic regression algorithm: probability estimates can be derived using the logistic function:

$$P(y = +1|x) \approx \frac{1}{1 + e^{-wx}}.$$

- All statistical models with linear parametrization are in fact amenable to stochastic gradient descent, using the log-likelihood of the model as the loss function $Q(z, w)$. For instance, results for Conditional Random Fields (CRF) [8] are reported in Sec. 18.5.4.

18.5.1 Sparsity

Leverage the sparsity of the training examples $\{x_t\}$.

- Represent w_t as a product $s_t W_t$ where $s_t \in \mathbb{R}$.

The training examples often are very high dimensional vectors with only a few non zero coefficients. The stochastic gradient update (18.11)

$$w_{t+1} = (1 - \gamma_t \lambda) w_t - \gamma_t y_t x_t \ell'(y_t w_t x_t)$$

is then inconvenient because it first rescales *all* coefficients of vector w by factor $(1 - \gamma_t \lambda)$. In contrast, the rest of the update only involves the weight coefficients corresponding to a nonzero coefficient in the pattern x_t .

Expressing the vector w_t as the product $s_t W_t$, where s is a scalar, provides a workaround [21]. The stochastic gradient update (18.11) can then be divided into operations whose complexity scales with the number of nonzero terms in x_t :

$$\begin{aligned} g_t &= \ell'(y_t s_t W_t x_t), \\ s_{t+1} &= (1 - \gamma_t \lambda) s_t, \\ W_{t+1} &= W_t - \gamma_t y_t g_t x_t / s_{t+1}. \end{aligned}$$

18.5.2 Learning Rates

Use learning rates of the form $\gamma_t = \gamma_0 (1 + \gamma_0 \lambda t)^{-1}$

- Determine the best γ_0 using a small training data sample.

When the Hessian matrix of the cost function at the optimum is strictly positive, the best convergence speed is achieved using learning rates of the form $(\lambda_{\min} t)^{-1}$ where λ_{\min} is the smallest eigenvalue of the Hessian [14]. The theoretical analysis also shows that overestimating λ_{\min} by more than a factor two leads to very slow convergence. Although we do not know the exact value of λ_{\min} , the L_2 regularization term in the training objective function means that $\lambda_{\min} \geq \lambda$. Therefore we can safely use learning rates that asymptotically decrease like $(\lambda t)^{-1}$.

Unfortunately, simply using $\gamma_t = (\lambda t)^{-1}$ leads to very large learning rates in the beginning of the optimization. It is possible to use an additional projection step [21] to contain the damage until the learning rates reach reasonable values. However it is simply better to start with reasonable learning rates. The formula $\gamma_t = \gamma_0(1 + \gamma_0 \lambda t)^{-1}$ ensures that the learning rates γ_t start from a predefined value γ_0 and asymptotically decrease like $(\lambda t)^{-1}$.

The most robust approach is to determine the best γ_0 as explained earlier, using a small sample of the training set. This is justified because the asymptotic SGD convergence rates [14] are independent from the sample size. In order to make the method more robust, I often use a γ_0 slightly smaller than the best value observed on the small training sample.

Such learning rates have been found to be effective in situations that far exceed the scope of this particular analysis. For instance, they work well with nondifferentiable loss functions such as the hinge loss [21]. They also work well when one adds an unregularized bias term to the model. However it is then wise to use smaller learning rates for the bias term itself.

18.5.3 Averaged Stochastic Gradient Descent

The *stochastic gradient descent/averaged SGD* (ASGD) algorithm [19] performs the normal stochastic gradient update (18.4) and computes the average

$$\bar{w}_t = \frac{1}{t - t_0} \sum_{i=t_0+1}^t w_t .$$

This average can be computed efficiently using a recursive formula. For instance, in the case of the L_2 regularized training objective (18.10), the following weight updates implement the ASGD algorithm:

$$\begin{aligned} w_{t+1} &= (1 - \gamma_t \lambda) w_t - \gamma_t y_t x_t \ell'(y_t w_t x_t) \\ \bar{w}_{t+1} &= \bar{w}_t + \mu_t (w_{t+1} - \bar{w}_t) \end{aligned}$$

with the averaging rate

$$\mu_t = 1 / \max\{1, t - t_0\} .$$

When one uses learning rates γ_t that decrease slower than t^{-1} , the theoretical analysis of ASGD shows that the training error $E_n(\bar{w}_t)$ decreases like t^{-1} with the *optimal constant* [15]. This is as good as the second order stochastic gradient descent (2SGD) for a fraction of the computational cost of (18.5).

Unfortunately, ASGD typically starts more slowly than the plain SGD and can take a long time to reach the optimal asymptotic convergence speed. Although an adequate choice of the learning rates helps [27], the problem worsens when the dimension d of the inputs x_t increases. Unfortunately, there are no clear guidelines for selecting the time t_0 that determines when we engage the averaging process.

Try averaged stochastic gradient with

- Learning rates $\gamma_t = \gamma_0(1 + \gamma_0\lambda t)^{-3/4}$
- Averaging rates $\mu_t = 1 / \max\{1, t - d, t - n\}$

Similar to the trick explained in Sec. 18.5.1, there is an efficient method to implement averaged stochastic gradient descent for sparse training data. The idea is to represent the variables w_t and \bar{w}_t as

$$\begin{aligned} w_t &= s_t W_t \\ \bar{w}_t &= (A_t + \alpha_t W_t) / \beta_t \end{aligned}$$

where η_t , α_t and β_t are scalars. The average stochastic gradient update equations can then be rewritten in the manner that only involve scalars or sparse operations [27] :

$$\begin{aligned} g_t &= \ell'(y_t s_t W_t x_t), \\ s_{t+1} &= (1 - \gamma_t \lambda) s_t \\ W_{t+1} &= W_t - \gamma_t y_t x_t g_t / s_{t+1} \\ A_{t+1} &= A_t + \gamma_t \alpha_t y_t x_t g_t / s_{t+1} \\ \beta_{t+1} &= \beta_t / (1 - \mu_t) \\ \alpha_{t+1} &= \alpha_t + \mu_t \beta_{t+1} s_{t+1} \end{aligned}$$

18.5.4 Experiments

This section briefly reports experimental results illustrating the actual performance of SGD and ASGD on a variety of linear systems. The source code is available at <http://leon.bottou.org/projects/sgd>. All learning rates were determined as explained in section 18.5.2.

Figure 18.1 reports results achieved using SGD for a linear SVM trained for the recognition of the CCAT category in the RCV1 dataset [10] using both the hinge loss and the log loss. The training set contains 781,265 documents represented by 47,152 relatively sparse TF/IDF features. SGD runs considerably faster than either the standard SVM solvers SVMLIGHT and SVMPERF [7] or the super-linear optimization algorithm TRON [11].

Figure 18.2 reports results achieved for a linear model trained on the ALPHA task of the 2008 Pascal Large Scale Learning Challenge using the squared hinge

Algorithm	Time	Test Error
<i>Hinge loss SVM, $\lambda = 10^{-4}$.</i>		
SVMLIGHT	23,642 s.	6.02 %
SVMPerf	66 s.	6.03 %
SGD	1.4 s.	6.02 %
<i>Log loss SVM, $\lambda = 10^{-5}$.</i>		
TRON (-e0.01)	30 s.	5.68 %
TRON (-e0.001)	44 s.	5.70 %
SGD	2.3 s.	5.66 %

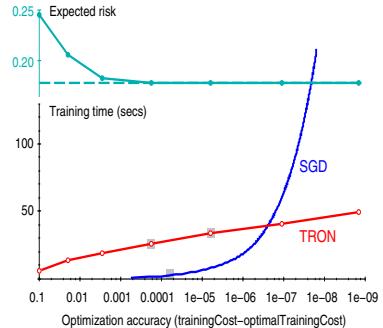


Fig. 18.1. Results achieved with a L_2 regularized linear model trained on the RCV1 task using both the hinge loss and the log loss. The lower half of the plot shows the time required by SGD and TRON to reach a predefined accuracy ρ on the log loss task. The upper half shows that the expected risk stops improving long before the super-linear optimization algorithm TRON overcomes SGD.

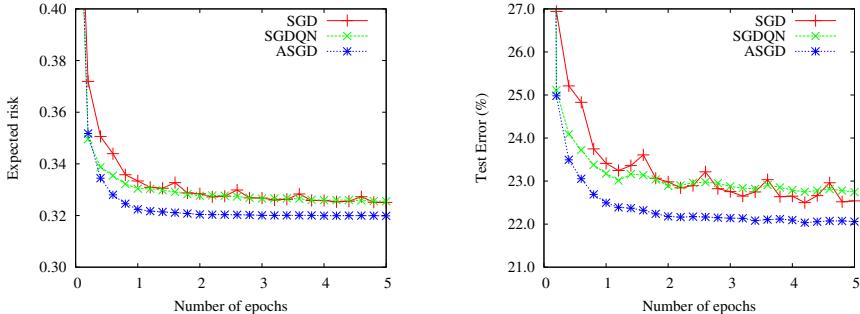


Fig. 18.2. Comparison of the test set performance of SGD, SGDQN, and ASGD for a L_2 regularized linear model trained with the squared hinge loss on the ALPHA task of the 2008 Pascal Large Scale Learning Challenge. ASGD nearly reaches the optimal expected risk after a single pass.

loss $\ell(m) = \max\{0, 1 - m\}^2$. For reference, we also provide the results achieved by the SGDQN algorithm [1] which was one of the winners of this competition, and works by adapting a separate learning rate for each weight. The training set contains 100,000 patterns represented by 500 centered and normalized variables. Performances measured on a separate testing set are plotted against the number of passes over the training set. ASGD achieves near optimal results after one epoch only.

Figure 18.3 reports results achieved using SGD, SGDQN, and ASGD for a CRF [8] trained on the CONLL 2000 Chunking task [20]. The training set contains 8936 sentences for a 1.68×10^6 dimensional parameter space. Performances measured on a separate testing set are plotted against the number of passes over the training set. SGDQN appears more attractive because ASGD

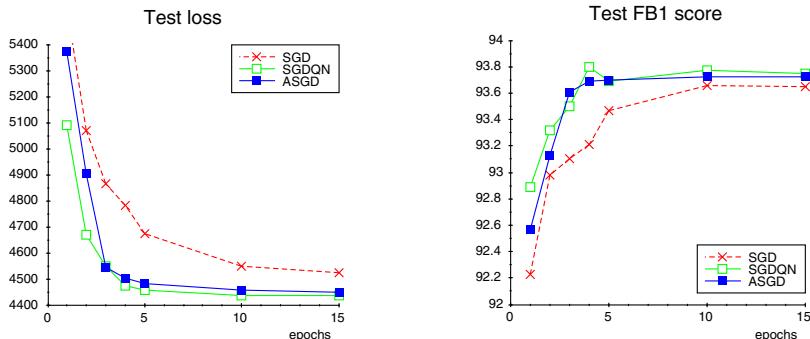


Fig. 18.3. Comparison of the test set performance of SGD, SGDQN, and ASGD on a L_2 regularized CRF trained on the CONLL Chunking task. On this task, SGDQN appears more attractive because ASGD does not fully reach its asymptotic performance.

does not reach its asymptotic performance. All three algorithms reach the best test set performance in a couple minutes. The standard CRF L-BFGS optimizer takes 72 minutes to compute an equivalent solution.

18.6 Conclusion

Stochastic gradient descent and its variants are versatile techniques that have proven invaluable as a learning algorithms for large datasets. The best advice for a successful application of these techniques is (*i*) to perform small-scale experiments with subsets of the training data, and (*ii*) to pay a ruthless attention to the correctness of the gradient computation.

References

- [1] Bordes, A., Bottou, L., Gallinari, P.: SGD-QN: Careful quasi-Newton stochastic gradient descent. *Journal of Machine Learning Research* 10, 1737–1754 (2009); with erratum, JMLR 11, 2229–2240 (2010)
- [2] Bottou, L., Bousquet, O.: The tradeoffs of large scale learning. In: Platt, J., Koller, D., Singer, Y., Roweis, S. (eds.) *Advances in Neural Information Processing Systems*, vol. 20, pp. 161–168. NIPS Foundation (2008), <http://books.nips.cc>
- [3] Bottou, L.: Online algorithms and stochastic approximations. In: Saad, D. (ed.) *Online Learning and Neural Networks*. Cambridge University Press, Cambridge (1998)
- [4] Bousquet, O.: Concentration Inequalities and Empirical Processes Theory Applied to the Analysis of Learning Algorithms. Ph.D. thesis, Ecole Polytechnique, Palaiseau, France (2002)
- [5] Cortes, C., Vapnik, V.: Support-vector network. *Machine Learning* 20(3), 273–297 (1995)
- [6] Dennis, J., Schnabel, R.B.: *Numerical Methods For Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Inc., Englewood Cliffs (1983)

- [7] Joachims, T.: Training linear SVMs in linear time. In: Proceedings of the 12th ACM SIGKDD International Conference, New York (2006)
- [8] Lafferty, J.D., McCallum, A., Pereira, F.C.N.: Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: Brodley, C.E., Danyluk, A.P. (eds.) Proceedings of the Eighteenth International Conference on Machine Learning (ICML), pp. 282–289. Morgan Kaufmann, Williams College (2001)
- [9] Lee, W.S., Bartlett, P.L., Williamson, R.C.: The importance of convexity in learning with squared loss. *IEEE Transactions on Information Theory* 44(5), 1974–1980 (1998)
- [10] Lewis, D.D., Yang, Y., Rose, T.G., Li, F.: RCV1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research* 5, 361–397 (2004)
- [11] Lin, C.J., Weng, R.C., Keerthi, S.S.: Trust region newton methods for large-scale logistic regression. In: Ghahramani, Z. (ed.) Proc. Twenty-Fourth International Conference on Machine Learning (ICML), pp. 561–568. ACM (2007)
- [12] MacQueen, J.: Some methods for classification and analysis of multivariate observations. In: LeCam, L.M., Neyman, J. (eds.) Proceedings of the Fifth Berkeley Symposium on Mathematics, Statistics, and Probabilities, vol. 1, pp. 281–297. University of California Press, Berkeley (1967)
- [13] Massart, P.: Some applications of concentration inequalities to statistics. *Annales de la Faculté des Sciences de Toulouse series 6* 9(2), 245–303 (2000)
- [14] Murata, N.: A statistical study of on-line learning. In: Saad, D. (ed.) *Online Learning and Neural Networks*. Cambridge University Press, Cambridge (1998)
- [15] Polyak, B.T., Juditsky, A.B.: Acceleration of stochastic approximation by averaging. *SIAM J. Control Optim.* 30(4), 838–855 (1992)
- [16] Robbins, H., Siegmund, D.: A convergence theorem for non negative almost supermartingales and some applications. In: Rustagi, J.S. (ed.) *Optimizing Methods in Statistics*. Academic Press (1971)
- [17] Rosenblatt, F.: The perceptron: A perceiving and recognizing automaton. Tech. Rep. 85-460-1, Project PARA, Cornell Aeronautical Lab (1957)
- [18] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In: Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. I, pp. 318–362. Bradford Books, Cambridge (1986)
- [19] Ruppert, D.: Efficient estimations from a slowly convergent robbins-monro process. Tech. Rep. 781, Cornell University Operations Research and Industrial Engineering (1988)
- [20] Sang, E.F.T.K., Buchholz, S.: Introduction to the CoNLL-2000 shared task: Chunking. In: Cardie, C., Daelemans, W., Nedellec, C., Tjong Kim Sang, E.F. (eds.) *Proceedings of CoNLL 2000 and LLL 2000*, Lisbon, Portugal, pp. 127–132 (2000)
- [21] Shalev-Shwartz, S., Singer, Y., Srebro, N.: Pegasos: Primal estimated subgradient solver for SVM. In: Proc. 24th Intl. Conf. on Machine Learning (ICML 2007), pp. 807–814. ACM (2007)
- [22] Shalev-Shwartz, S., Srebro, N.: SVM optimization: inverse dependence on training set size. In: Proceedings of the 25th International Machine Learning Conference (ICML 2008), pp. 928–935. ACM (2008)
- [23] Tibshirani, R.: Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society (Series B)* 58, 267–288 (1996)
- [24] Tsybakov, A.B.: Optimal aggregation of classifiers in statistical learning. *Annals of Statistics* 32(1) (2004)

- [25] Vapnik, V.N., Chervonenkis, A.Y.: On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications* 16(2), 264–280 (1971)
- [26] Widrow, B., Hoff, M.E.: Adaptive switching circuits. In: IRE WESCON Conv. Record, Part 4, pp. 96–104 (1960)
- [27] Xu, W.: Towards optimal one pass large scale learning with averaged stochastic gradient descent (2011), <http://arxiv.org/abs/1107.2490>
- [28] Zinkevich, M.: Online convex programming and generalized infinitesimal gradient ascent. In: Proc. Twentieth International Conference on Machine Learning (2003)

Practical Recommendations for Gradient-Based Training of Deep Architectures

Yoshua Bengio

Université de Montréal

Abstract. Learning algorithms related to artificial neural networks and in particular for Deep Learning may seem to involve many bells and whistles, called hyper-parameters. This chapter is meant as a practical guide with recommendations for some of the most commonly used hyper-parameters, in particular in the context of learning algorithms based on back-propagated gradient and gradient-based optimization. It also discusses how to deal with the fact that more interesting results can be obtained when allowing one to adjust many hyper-parameters. Overall, it describes elements of the practice used to successfully and efficiently train and debug large-scale and often deep multi-layer neural networks. It closes with open questions about the training difficulties observed with deeper architectures.

19.1 Introduction

Following a decade of lower activity, research in artificial neural networks was revived after a 2006 breakthrough [61, 14, 95] in the area of *Deep Learning*, based on greedy layer-wise unsupervised pre-training of each layer of features. See [7] for a review. Many of the practical recommendations that justified the previous edition of this book are still valid, and new elements were added, while some survived longer by virtue of the practical advantages they provided. The panorama presented in this chapter regards some of these surviving or novel elements of practice, focusing on learning algorithms aiming at training deep neural networks, but leaving most of the material specific to the Boltzmann machine family to another chapter [60].

Although such recommendations come out of a living practice that emerged from years of experimentation and to some extent mathematical justification, they should be challenged. They constitute a good starting point for the experimenter and user of learning algorithms but very often have not been formally validated, leaving open many questions that can be answered either by theoretical analysis or by solid comparative experimental work (ideally by both). A good indication of the need for such validation is that different researchers and research groups do not always agree on the practice of training neural networks.

Several of the recommendations presented here can be found implemented in the *Deep Learning Tutorials*¹ and in the related *Pylearn2* library², all based on the **Theano** library (discussed below) written in the *Python* programming language.

The 2006 Deep Learning breakthrough [61, 14, 95] centered on the use of *unsupervised representation learning* to help learning *internal representations*³ by providing a *local training signal* at each level of a hierarchy of features⁴. Unsupervised representation learning algorithms can be applied several times to learn different layers of a deep model. Several unsupervised representation learning algorithms have been proposed since then. Those covered in this chapter (such as auto-encoder variants) retain many of the properties of artificial multi-layer neural networks, relying on the back-propagation algorithm to estimate stochastic gradients. Deep Learning algorithms such as those based on the Boltzmann machine and those based on auto-encoder or sparse coding variants often include a supervised fine-tuning stage. This supervised fine-tuning as well as the gradient descent performed with auto-encoder variants also involves the back-propagation algorithm, just as like when training deterministic feedforward or recurrent artificial neural networks. Hence this chapter also includes recommendations for training ordinary supervised deterministic neural networks or more generally, most machine learning algorithms relying on iterative gradient-based optimization of a parametrized learner with respect to an explicit training criterion.

This chapter assumes that the reader already understands the standard algorithms for training supervised multi-layer neural networks, with the loss gradient computed thanks to the back-propagation algorithm [103]. It starts by explaining basic concepts behind Deep Learning and the greedy layer-wise pretraining strategy (Section 19.1.1), and recent unsupervised pre-training algorithms (denoising and contractive auto-encoders) that are closely related in the way they are trained to standard multi-layer neural networks (Section 19.1.2). It then reviews in Section 19.2 basic concepts in iterative gradient-based optimization and in particular the stochastic gradient method, gradient computation with a flow graph, automatic differentiation. The main section of this chapter is Section 19.3, which explains hyper-parameters in general, their optimization, and specifically covers the main hyper-parameters of neural networks. Section 19.4 briefly describes simple ideas and methods to debug and visualize neural networks, while Section 19.5 covers parallelism, sparse high-dimensional inputs, symbolic inputs

¹ <http://deeplearning.net/tutorial/>

² <http://deeplearning.net/software/pylearn2>

³ A neural network computes a sequence of data transformations, each step encoding the raw input into an intermediate or internal representation, in principle to make the prediction or modeling task of interest easier.

⁴ In standard multi-layer neural networks trained using back-propagated gradients, the only signal that drives parameter updates is provided at the output of the network (and then propagated backwards). Some unsupervised learning algorithms provide a local source of guidance for the parameter update in each layer, based only on the inputs and outputs of that layer.

and embeddings, and multi-relational learning. The chapter closes (Section 19.6) with open questions on the difficulty of training deep architectures and improving the optimization methods for neural networks.

19.1.1 Deep Learning and Greedy Layer-Wise Pretraining

The notion of *reuse*, which explains the power of distributed representations [7], is also at the heart of the theoretical advantages behind *Deep Learning*. Complexity theory of circuits, e.g. [54, 55], (which include neural networks as special cases) has much preceded the recent research on deep learning. The depth of a circuit is the length of the longest path from an input node of the circuit to an output node of the circuit. Formally, one can change the depth of a given circuit by changing the definition of what each node can compute, but only by a constant factor [7]. The typical computations we allow in each node include: weighted sum, product, artificial neuron model (such as a monotone non-linearity on top of an affine transformation), computation of a kernel, or logic gates. Theoretical results [54, 55, 13, 10, 9] clearly identify families of functions where a deep representation can be exponentially more efficient than one that is insufficiently deep. If the same set of functions can be represented from within a family of architectures associated with a smaller VC-dimension (e.g. less hidden units⁵), learning theory would suggest that it can be learned with fewer examples, yielding improvements in both *computational efficiency* and *statistical efficiency*.

Another important motivation for feature learning and Deep Learning is that they can be done with *unlabeled examples*, so long as the factors (unobserved random variables explaining the data) relevant to the questions we will ask later (e.g. classes to be predicted) are somehow salient in the input distribution itself. This is true under the *manifold hypothesis*, which states that natural classes and other high-level concepts in which humans are interested are associated with *low-dimensional* regions in input space (manifolds) near which the distribution concentrates, and that different class manifolds are well-separated by regions of very low density. It means that a small semantic change around a particular example can be captured by changing only a few numbers in a high-level abstract representation space. As a consequence, feature learning and Deep Learning are intimately related to principles of *unsupervised learning*, and they can work in the *semi-supervised setting* (where only a few examples are labeled), as well as in the *transfer learning* and *multi-task* settings (where we aim to generalize to new classes or tasks). The underlying hypothesis is that many of the underlying factors are *shared* across classes or tasks. Since representation learning aims to extract and isolate these factors, representations can be *shared* across classes and tasks.

One of the most commonly used approaches for training deep neural networks is based on *greedy layer-wise pre-training* [14]. The idea, first introduced in Hinton *et al.* [61], is to train one layer of a deep architecture at a time using

⁵ Note that in our experiments, deep architectures tend to generalize very well even when they have quite large numbers of parameters.

unsupervised representation learning. Each level takes as input the representation learned at the previous level and learns a new representation. The learned representation(s) can then be used as input to predict variables of interest, for example to classify objects. After unsupervised pre-training, one can also perform supervised fine-tuning of the whole system⁶, i.e., optimize not just the classifier but also the lower levels of the feature hierarchy with respect to some objective of interest. Combining unsupervised pre-training and supervised fine-tuning usually gives better generalization than pure supervised learning from a purely random initialization. The unsupervised representation learning algorithms for pre-training proposed in 2006 were the Restricted Boltzmann Machine or RBM [61], the auto-encoder [14] and a sparsifying form of auto-encoder similar to sparse coding [95].

19.1.2 Denoising and Contractive Auto-encoders

An auto-encoder has two parts: an encoder function f that maps the input x to a representation $h = f(x)$, and a decoder function g that maps h back in the space of x in order to reconstruct x . In the regular auto-encoder the reconstruction function $r(\cdot) = g(f(\cdot))$ is trained to minimize the average value of a reconstruction loss *on the training examples*. Note that reconstruction loss should be high for most other input configurations⁷. The regularization mechanism makes sure that reconstruction cannot be perfect everywhere, while minimizing the reconstruction loss at training examples digs a hole in reconstruction error where the density of training examples is large. Examples of reconstruction loss functions include $\|x - r(x)\|^2$ (for real-valued inputs) and $-\sum_i x_i \log r_i(x) + (1 - x_i) \log(1 - r_i(x))$ (when interpreting x_i as a bit or a probability of a binary event). Auto-encoders capture the input distribution by learning to *better reconstruct more likely input configurations*. The difference between the reconstruction vector and the input vector can be shown to be related to the log-density gradient as estimated by the learner [114, 16] and the Jacobian matrix of the reconstruction with respect to the input gives information about the second derivative of the density, i.e., in which direction the density remains high when you are on a high-density manifold [99, 16]. In the Denoising Auto-Encoder (DAE) and the Contractive Auto-Encoder (CAE), the training procedure also introduces *robustness* (insensitivity to small variations), respectively in the reconstruction $r(x)$ or in the representation $f(x)$. In the DAE [115, 116], this is achieved by training with stochastically corrupted inputs, but trying to reconstruct the uncorrupted inputs. In the CAE [99], this is achieved by adding an explicit regularizing term in the training criterion, proportional to the norm of the Jacobian of the encoder, $\|\frac{\partial f(x)}{\partial x}\|^2$. But the CAE and the DAE are very related [16]: when the

⁶ The whole system composes the computation of the representation with computation of the predictor's output.

⁷ Different regularization mechanisms have been proposed to push reconstruction error up in low density areas: denoising criterion, contractive criterion, and code sparsity. It has been argued that such constraints play a role similar to the partition function for Boltzmann machines [96].

noise is Gaussian and small, the denoising error minimized by the DAE is equivalent to minimizing the norm of the Jacobian of the reconstruction function $r(\cdot) = g(f(\cdot))$, whereas the CAE minimizes the norm of the Jacobian of the encoder $f(\cdot)$. Besides Gaussian noise, another interesting form of corruption has been very successful with DAEs: it is called the *masking corruption* and consists in randomly zeroing out a large fraction (like 20% or even 50%) of the inputs, where the zeroed out subset is randomly selected for each example. In addition to the contractive effect, it forces the learned encoder to be able to rely only on an arbitrary subset of the input features.

Another way to prevent the auto-encoder from perfectly reconstructing everywhere is to introduce a sparsity penalty on h , discussed below (Section 19.3.1).

19.1.3 Online Learning and Optimization of Generalization Error

The objective of learning is not to minimize training error or even the training criterion. The latter is a surrogate for generalization error, i.e., performance on new (out-of-sample) examples, and there are no hard guarantees that minimizing the training criterion will yield good generalization error: it depends on the appropriateness of the parametrization and training criterion (with the corresponding prior they imply) for the task at hand.

Many learning tasks of interest will require huge quantities of data (most of which will be unlabeled) and as the number of examples increases, so long as capacity is limited (the number of parameters is small compared to the number of examples), training error and generalization approach each other. In the regime of such large datasets, we can consider that the learner sees an unending stream of examples (e.g., think about a process that harvests text and images from the web and feeds it to a machine learning algorithm). In that context, it is most efficient to simply update the parameters of the model after each example or few examples, as they arrive. This is the ideal *online learning* scenario, and in a simplified setting, we can even consider each new example z as being sampled i.i.d. from an unknown generating distribution with probability density $p(z)$. More realistically, examples in online learning do not arrive i.i.d. but instead from an unknown stochastic process which exhibits serial correlation and other temporal dependencies. Many learning algorithms rely on gradient-based numerical optimization of a training criterion. Let $L(z, \theta)$ be the loss incurred on example z when the parameter vector takes value θ . The gradient vector for the loss associated with a single example is $\frac{\partial L(z, \theta)}{\partial \theta}$.

If we consider the simplified case of i.i.d. data, there is an interesting observation to be made: *the online learner is performing stochastic gradient descent on its generalization error*. Indeed, the generalization error C of a learner with parameters θ and loss function L is

$$C = E[L(z, \theta)] = \int p(z)L(z, \theta)dz$$

while the stochastic gradient from sample z is

$$\hat{g} = \frac{\partial L(z, \theta)}{\partial \theta}$$

with z a random variable sampled from p . The gradient of generalization error is

$$\frac{\partial C}{\partial \theta} = \frac{\partial}{\partial \theta} \int p(z) L(z, \theta) dz = \int p(z) \frac{\partial L(z, \theta)}{\partial \theta} dz = E[\hat{g}]$$

showing that the online gradient \hat{g} is an unbiased estimator of the generalization error gradient $\frac{\partial C}{\partial \theta}$. It means that online learners, when given a stream of non-repetitive training data, really optimize (maybe not in the optimal way, i.e., using a first-order gradient technique) what we really care about: generalization error.

19.2 Gradients

19.2.1 Gradient Descent and Learning Rate

The gradient or an estimator of the gradient is used as the core part the computation of parameter updates for gradient-based numerical optimization algorithms. For example, simple online (or stochastic) gradient descent [102, 28] updates the parameters after each example is seen, according to

$$\theta^{(t)} \leftarrow \theta^{(t-1)} - \epsilon_t \frac{\partial L(z_t, \theta)}{\partial \theta}$$

where z_t is an example sampled at iteration t and where ϵ_t is a *hyper-parameter* that is called the *learning rate* and whose choice is crucial. If the learning rate is too large⁸, the average loss will increase. The optimal learning rate is usually close to (by a factor of 2) the *largest learning rate that does not cause divergence of the training criterion*, an observation that can guide heuristics for setting the learning rate [8], e.g., start with a large learning rate and if the training criterion diverges, try again with 3 times smaller learning rate, etc., until no divergence is observed.

See [26] for a deeper treatment of stochastic gradient descent, including suggestions to set learning rate schedule and improve the asymptotic convergence through averaging.

In practice, we use *mini-batch* updates based on an *average* of the gradients⁹ inside each block of B examples:

$$\theta^{(t)} \leftarrow \theta^{(t-1)} - \epsilon_t \frac{1}{B} \sum_{t'=Bt+1}^{B(t+1)} \frac{\partial L(z_{t'}, \theta)}{\partial \theta}. \quad (19.1)$$

⁸ Above a value which is approximately 2 times the largest eigenvalue of the average loss Hessian matrix.

⁹ Compared to a sum, an average makes a small change in B have only a small effect on the optimal learning rate, with an increase in B generally allowing a small increase in the learning rate because of the reduced variance of the gradient.

With $B = 1$ we are back to ordinary online gradient descent, while with B equal to the training set size, this is standard (also called “batch”) gradient descent. With intermediate values of B there is generally a sweet spot. When B increases we can get more multiply-add operations per second by taking advantage of parallelism or efficient matrix-matrix multiplications (instead of separate matrix-vector multiplications), often gaining a factor of 2 in practice in overall training time. On the other hand, as B increases, the number of updates per computation done decreases, which slows down convergence (in terms of error vs number of multiply-add operations performed) because less updates can be done in the same computing time. Combining these two opposing effects yields a typical U-curve with a sweet spot at an intermediate value of B .

Keep in mind that even the true gradient direction (averaging over the whole training set) is only the steepest descent direction locally but may not point in the right direction when considering larger steps. In particular, because the training criterion is not quadratic in the parameters, as one moves in parameter space the optimal descent direction keeps changing. Because the gradient direction is not quite the right direction of descent, there is no point in spending a lot of computation to estimate it precisely for gradient descent. Instead, doing more updates more frequently helps to explore more and faster, especially with large learning rates. In addition, smaller values of B may benefit from more exploration in parameter space and a form of regularization both due to the “noise” injected in the gradient estimator, which may explain the better test results sometimes observed with smaller B .

When the training set is finite, training proceeds by sweeps through the training set called an *epoch*, and full training usually requires many epochs (iterations through the training set). Note that stochastic gradient (either one example at a time or with mini-batches) is different from ordinary *gradient descent*, sometimes called “batch gradient descent”, which corresponds to the case where B equals the training set size, i.e., there is one parameter update per epoch). The great advantage of stochastic gradient descent and other online or minibatch update methods is that their convergence does not depend on the size of the training set, only on the number of updates and the richness of the training distribution. In the limit of a large or infinite training set, a batch method (which updates only after seeing all the examples) is hopeless. In fact, even for ordinary datasets of tens or hundreds of thousands of examples (or more!), stochastic gradient descent converges much faster than ordinary (batch) gradient descent, and beyond some dataset sizes the speed-up is almost linear (i.e., doubling the size almost doubles the gain)¹⁰. It is really important to use the stochastic version in order to get reasonable clock-time convergence speeds.

As for any stochastic gradient descent method (including the mini-batch case), it is important for efficiency of the estimator that each example or mini-batch be sampled approximately independently. Because random access to memory (or even worse, to disk) is expensive, a good approximation, called incremental

¹⁰ On the other hand, batch methods can be parallelized easily, which becomes an important advantage with currently available forms of computing power.

gradient [21], is to visit the examples (or mini-batches) in a fixed order corresponding to their order in memory or disk (repeating the examples in the same order on a second epoch, if we are not in the pure online case where each example is visited only once). In this context, it is safer if the examples or mini-batches are first put in a random order (to make sure this is the case, it could be useful to first shuffle the examples). Faster convergence has been observed if the order in which the mini-batches are visited is changed for each epoch, which can be reasonably efficient if the training set holds in computer memory.

19.2.2 Gradient Computation and Automatic Differentiation

The gradient can be either computed manually or through automatic differentiation. Either way, it helps to structure this computation as a *flow graph*, in order to prevent mathematical mistakes and make sure an implementation is computationally efficient. The computation of the loss $L(z, \theta)$ as a function of θ is laid out in a graph whose nodes correspond to elementary operations such as addition, multiplication, and non-linear operations such as the neural networks activation function (e.g., sigmoid or hyperbolic tangent), possibly at the level of vectors, matrices or tensors. The flow graph is directed and acyclic and has three types of nodes: input nodes, internal nodes, and output nodes. Each of its nodes is associated with a numerical output which is the result of the application of that computation (none in the case of input nodes), taking as input the output of previous nodes in a directed acyclic graph. Example z and parameter vector θ (or their elements) are the input nodes of the graph (i.e., they do not have inputs themselves) and $L(z, \theta)$ is a scalar output of the graph. Note that here, in the supervised case, z can include an input part x (e.g. an image) and a target part y (e.g. a target class associated with an object in the image). In the unsupervised case $z = x$. In a semi-supervised case, there is a mix of labeled and unlabeled examples, and z includes y on the labeled examples but not on the unlabeled ones.

In addition to associating a numerical output o_a to each node a of the flow graph, we can associate a gradient $g_a = \frac{\partial L(z, \theta)}{\partial o_a}$. The gradient will be defined and computed recursively in the graph, in the opposite direction of the computation of the nodes' outputs, i.e., whereas o_a is computed using outputs o_p of *predecessor* nodes p of a , g_a will be computed using the gradients g_s of *successor* nodes s of a . More precisely, the chain rule dictates

$$g_a = \sum_s g_s \frac{\partial o_s}{\partial o_a}$$

where the sum is over immediate successors of a . Only output nodes have no successor, and in particular for the output node that computes L , the gradient is set to 1 since $\frac{\partial L}{\partial L} = 1$, thus initializing the recursion. Manual or automatic differentiation then only requires to define the partial derivative associated with each type of operation performed by any node of the graph. When implementing gradient descent algorithms with manual differentiation the result tends to be verbose, brittle code that lacks modularity – all bad things in terms of software

engineering. A better approach is to express the flow graph in terms of objects that modularize how to compute outputs from inputs as well as how to compute the partial derivatives necessary for gradient descent. One can pre-define the operations of these objects (in a “forward propagation” or `fprop` method) and their partial derivatives (in a “backward propagation” or `bprop` method) and encapsulate these computations in an object that knows how to compute its output given its inputs, and how to compute the gradient with respect to its inputs given the gradient with respect to its output. This is the strategy adopted in the `Theano` library¹¹ with its `Op` objects [18], as well as in libraries such as `Torch`¹² [37] and `Lush`¹³.

Compared to `Torch` and `Lush`, `Theano` adds an interesting ingredient which makes it a full-fledged automatic differentiation tool: symbolic computation. The flow graph itself (without the numerical values attached) can be viewed as a symbolic representation (in a data structure) of a numerical computation. In `Theano`, the gradient computation is first performed symbolically, i.e., each `Op` object knows how to create other `Ops` corresponding to the computation of the partial derivatives associated with that `Op`. Hence the *symbolic differentiation* of the output of a flow graph with respect to any or all of its input nodes can be performed easily in most cases, yielding another flow graph which specifies how to compute these gradients, given the input of the original graph. Since the gradient graph typically contains the original graph (mapping parameters to loss) as a sub-graph, in order to make computations efficient it is important to automate (as done in `Theano`) a number of *simplifications* which are graph transformations preserving the semantics of the output (given the input) but yielding smaller (or more numerically stable or more efficiently computed) graphs (e.g., removing redundant computations). To take advantage of the fact that computing the loss gradient includes as a first step computing the loss itself, it is advantageous to structure the code so that both the loss and its gradient are computed at once, with a single graph having multiple outputs. The advantages of performing gradient computations symbolically are numerous. First of all, one can readily compute gradients over gradients, i.e., second derivatives, which are useful for some learning algorithms. Second, one can define algorithms or training criteria involving gradients themselves, as required for example in the Contractive Auto-Encoder (which uses the norm of a Jacobian matrix in its training criterion, i.e., really requires second derivatives, which here are cheap to compute). Third, it makes it easy to implement other useful graph transformations such as graph simplifications or numerical optimizations and transformations that help making the numerical results more robust and more efficient (such as working in the domain of logarithms of probabilities rather than in the domain of probabilities directly). Other potential beneficial applications of such symbolic manipulations include parallelization and additional differential operators (such as the R-operator, recently implemented in `Theano`, which is very useful to

¹¹ <http://deeplearning.net/software/theano/>

¹² <http://www.torch.ch>

¹³ <http://lush.sourceforge.net>

compute the product of a Jacobian matrix $\frac{\partial f(x)}{\partial x}$ or Hessian matrix $\frac{\partial^2 L(x, \theta)}{\partial \theta^2}$ with a vector without ever having to actually compute and store the matrix itself [90]).

19.3 Hyper-parameters

A pure learning algorithm can be seen as a function taking training data as input and producing as output a function (e.g. a predictor) or model (i.e. a bunch of functions). However, in practice, many learning algorithms involve hyper-parameters, i.e., annoying knobs to be adjusted. In many algorithms such as Deep Learning algorithms the number of hyper-parameters (ten or more!) can make the idea of having to adjust all of them unappealing. In addition, it has been shown that the use of computer clusters for hyper-parameter selection can have an important effect on results [91]. Choosing hyper-parameter values is formally equivalent to the question of *model selection*, i.e., given a family or set of learning algorithms, how to pick the most appropriate one inside the set? *We define a hyper-parameter for a learning algorithm A as a variable to be set prior to the actual application of A to the data, one that is not directly selected by the learning algorithm itself.* It is basically an outside control knob. It can be discrete (as in model selection) or continuous (such as the learning rate discussed above). Of course, one can hide these hyper-parameters by wrapping another learning algorithm, say B, around A, to selects A's hyper-parameters (e.g. to minimize validation set error). We can then call B a hyper-learner, and if B has no hyper-parameters itself then the composition of B over A could be a “pure” learning algorithm, with no hyper-parameter. In the end, to apply a learner to training data, one has to have a pure learning algorithm. The hyper-parameters can be fixed by hand or tuned by an algorithm, but their value has to be selected. The value of some hyper-parameters can be selected based on the performance of A on its training data, but most cannot. For any hyper-parameter that has an impact on the effective capacity of a learner, it makes more sense to select its value based on out-of-sample data (outside the training set), e.g., a validation set performance, online error, or cross-validation error. Note that some learning algorithms (in particular unsupervised learning algorithms such as algorithms for training RBMs by approximate maximum likelihood) are problematic in this respect because we cannot directly measure the quantity that is to be optimized (e.g. the likelihood) because it is intractable. On the other hand, the expected denoising reconstruction error is easy to estimate (by just averaging the denoising error over a validation set).

Once some out-of-sample data has been used for selecting hyper-parameter values, it cannot be used anymore to obtain an unbiased estimator of generalization performance, so one typically uses a test set (or double cross-validation¹⁴, in

¹⁴ Double cross-validation applies recursively the idea of cross-validation, using an outer loop cross-validation to evaluate generalization error and then applying an inner loop cross-validation inside each outer loop split's training subset (i.e., splitting it again into training and validation folds) in order to select hyper-parameters for that split.

the case of small datasets) to estimate generalization error of the pure learning algorithm (with hyper-parameter selection hidden inside).

19.3.1 Neural Network Hyper-parameters

Different learning algorithms involve different sets of hyper-parameters, and it is useful to get a sense of the kinds of choices that practitioners have to make in choosing their values. We focus here mostly on those relevant to neural networks and Deep Learning algorithms.

Hyper-parameters of the Approximate Optimization. First of all, several learning algorithms can be viewed as the combination of two elements: a training criterion and a model (e.g., a family of functions, a parametrization) on the one hand, and on the other hand, a particular procedure for approximately optimizing this criterion. Correspondingly, one should distinguish hyper-parameters associated with the optimizer from hyper-parameters associated with the model itself, i.e., typically the function class, regularizer and loss function. We have already mentioned above some of the hyper-parameters typically associated with gradient-based optimization. Here is a more extensive descriptive list, focusing on those used in stochastic (mini-batch) gradient descent (although number of training iterations is used for all iterative optimization algorithms).

- The **initial learning rate** (ϵ_0 below, Eq.(19.2)). This is often the single most important hyper-parameter and one should always make sure that it has been tuned (up to approximately a factor of 2). Typical values for a neural network with standardized inputs (or inputs mapped to the (0,1) interval) are less than 1 and greater than 10^{-6} but these should not be taken as strict ranges and greatly depend on the parametrization of the model. A default value of 0.01 typically works for standard multi-layer neural networks but it would be foolish to rely exclusively on this default value. If there is only time to optimize one hyper-parameter and one uses stochastic gradient descent, then this is the hyper-parameter that is worth tuning.
- The choice of strategy for decreasing or adapting the **learning rate schedule** (with hyper-parameters such as the time constant τ in Eq. (19.2) below). The default value of $\tau \rightarrow \infty$ means that the learning rate is constant over training iterations. In many cases the benefit of choosing other than this default value is small. An example of $O(1/t)$ learning rate schedule, used in Bergstra and Bengio [17] is

$$\epsilon_t = \frac{\epsilon_0 \tau}{\max(t, \tau)} \quad (19.2)$$

which keeps the learning rate constant for the first τ steps and then decreases it in $O(1/t^\alpha)$, with traditional recommendations (based on asymptotic analysis of the convex case) suggesting $\alpha = 1$. See Bach and Moulines [2] for a recent analysis of the rate of convergence for the general case of $\alpha \leq 1$, suggesting that smaller values of α should be used in the non-convex case,

especially when using a gradient averaging or momentum technique (see below). An adaptive and heuristic way of automatically setting τ above is to keep ϵ_t constant until the training criterion stops decreasing significantly (by more than some relative improvement threshold) from epoch to epoch. That threshold is a less sensitive hyper-parameter than τ itself. An alternative to a fixed schedule with a couple of (global) free hyper-parameters like in the above formula is the use of an *adaptive* learning rate heuristic, e.g., the simple procedure proposed in [26]: at regular intervals during training, using a fixed small subset of the training set (what matters is only the number of examples used, not what fraction of the whole training set it represents), continue training with N different choices of learning rate (all in parallel), and keep the value that gave the best results until the next re-estimation of the optimal learning rate. Other examples of adaptive learning rate strategies are discussed below (Sec. 19.6.2).

- The **mini-batch size** (B in Eq. (19.1)) is typically chosen between 1 and a few hundreds, e.g. $B = 32$ is a good default value, with values above 10 taking advantage of the speed-up of matrix-matrix products over matrix-vector products. The impact of B is mostly computational, i.e., larger B yield faster computation (with appropriate implementations) but requires visiting more examples in order to reach the same error, since there are less updates per epoch. In theory, this hyper-parameter should impact *training time* and not so much *test performance*, so it can be optimized separately of the other hyper-parameters, by comparing training curves (training and validation error vs amount of training time), after the other hyper-parameters (except learning rate) have been selected. B and ϵ_0 may slightly interact with other hyper-parameters so both should be re-optimized at the end. Once B is selected, it can generally be fixed while the other hyper-parameters can be further optimized (except for a *momentum* hyper-parameter, if one is used).
- **Number of training iterations** T (measured in mini-batch updates). This hyper-parameter is particular in that it can be optimized almost for free using the principle of *early stopping*: by keeping track of the out-of-sample error (as for example estimated on a validation set) as training progresses (every N updates), one can decide how long to train for any given setting of all the other hyper-parameters. Early stopping is an inexpensive way to avoid strong overfitting, i.e., even if the other hyper-parameters would yield to overfitting, early stopping will considerably reduce the overfitting damage that would otherwise ensue. It also means that it hides the overfitting effect of other hyper-parameters, possibly obscuring the analysis that one may want to do when trying to figure out the effect of individual hyper-parameters, i.e., it tends to even out the performance obtained by many otherwise overfitting configurations of hyper-parameters by compensating a too large capacity with a smaller training time. For this reason, it might be useful to turn early-stopping off when analyzing the effect of individual hyper-parameters. Now let us turn to implementation details. Practically, one needs to continue training beyond the selected number of training iterations \hat{T} (which should be the point of lowest validation error in the training run) in order to ascertain

that validation error is unlikely to go lower than at the selected point. A heuristic introduced in the *Deep Learning Tutorials*¹⁵ is based on the idea of *patience* (set initially to 10000 examples in the MLP tutorial), which is a minimum number of training examples to see after the candidate selected point \hat{T} before deciding to stop training (i.e. before accepting this candidate as the final answer). As training proceeds and new candidate selected points \hat{T} (new minima of the validation error) are observed, the patience parameter is increased, either multiplicatively or additively on top of the last \hat{T} found. Hence, if we find a new minimum¹⁶ at t , we save the current best model, update $\hat{T} \leftarrow t$ and we increase our patience up to $t + \text{constant}$ or $t \times \text{constant}$. Note that validation error should not be estimated after each training update (that would be really wasteful) but after every N examples, where N is at least as large as the validation set (ideally several times larger so that the early stopping overhead remains small)¹⁷.

- **Momentum** β . It has long been advocated [56, 59] to temporally smooth out the stochastic gradient samples obtained during the stochastic gradient descent. For example, a moving average of the past gradients can be computed with $\bar{g} \leftarrow (1 - \beta)\bar{g} + \beta g$, where g is the instantaneous gradient $\frac{\partial L(z_t, \theta)}{\partial \theta}$ or a minibatch average, and β is a small positive coefficient that controls how fast the old examples get downweighted in the moving average. The simplest momentum trick is to make the updates proportional to this smoothed gradient estimator \bar{g} instead of the instantaneous gradient g . The idea is that it removes some of the noise and oscillations that gradient descent has, in particular in the directions of high curvature of the loss function¹⁸. A default value of $\beta = 1$ (no momentum) works well in many cases but in some cases momentum seems to make a positive difference. Polyak averaging [93] is a related form of parameter averaging¹⁹ that has theoretical advantages and has been advocated and shown to bring improvements on some unsupervised learning procedures such as RBMs [110]. More recently, several mathematically motivated algorithms [88, 75] have been proposed that incorporate some form of momentum and that also ensure much faster convergence (linear rather than sublinear) compared to stochastic gradient

¹⁵ <http://deeplearning.net/tutorial/>

¹⁶ Ideally, we should use a statistical test of significance and accept a new minimum (over a longer training period) only if the improvement is statistically significant, based on the size and variance estimates one can compute for the validation set.

¹⁷ When an extra processor on the same machine is available, validation error can conveniently be recomputed by a processor different from the one performing the training updates, allowing more frequent computation of validation error.

¹⁸ Think about a ball coming down a valley. Since it has not started from the bottom of the valley it will oscillate between its sides as it settles deeper, forcing the learning rate to be small to avoid large oscillations that would kick it out of the valley. Averaging out the local gradients along the way will cancel the opposing forces from each side of the valley.

¹⁹ Polyak averaging uses for predictions a moving average of the parameters found in the trajectory of stochastic gradient descent.

descent, at least for convex optimization problems. See also [26] for an example of averaged SGD with successful empirical speedups in the convex case. Note however that in the pure online case (stream of examples) and under some assumptions, the sublinear rate of convergence of stochastic gradient descent with $O(1/t)$ decrease of learning rate is an optimal rate, at least for convex problems [87]. That would suggest that for really large training sets it may not be possible to obtain better rates than ordinary stochastic gradient descent, albeit the constants in front (which depend on the condition number of the Hessian) may still be greatly reduced by using second-order information online [28, 27].

- **Layer-specific optimization hyper-parameters:** although rarely done, it is possible to use different values of optimization hyper-parameters (such as the learning rate) on different layers of a multi-layer network. This is especially appropriate (and easier to do) in the context of layer-wise unsupervised pre-training, since each layer is trained separately (while the layers below are kept fixed). This would be particularly useful when the number of units per layer varies a lot from layer to layer. See the paragraph below entitled **Layer-wise optimization of hyper-parameters** (Sec. 19.3.3). Some researchers also advocate the use of different learning rates for the different *types* of parameters one finds in the model, such as biases and weights in the standard multi-layer network, but the issue becomes more important when parameters such as precision or variance are included in the lot [38].

Up to now we have only discussed the hyper-parameters in the setup where one trains a neural network by stochastic gradient descent. With other optimization algorithms, some hyper-parameters are typically different. For example, Conjugate Gradient (CG) algorithms typically have a number of line search steps (which is a hyper-parameter) and a tolerance for stopping each line search (another hyper-parameter). An optimization algorithm like L-BFGS (limited-memory Broyden-Fletcher-Goldfarb-Shanno) also has a hyper-parameter controlling the memory usage of the algorithm, the rank of the Hessian approximation kept in memory, which also has an influence on the efficiency of each step. Both CG and L-BFGS are iterative (e.g., one line search per iteration), and the number of iterations can be optimized as described above for stochastic gradient descent, with early stopping.

19.3.2 Hyper-parameters of the Model and Training Criterion

Let us now turn to “model” and “criterion” hyper-parameters typically found in neural networks, especially deep neural networks.

- **Number of hidden units n_h .** Each layer in a multi-layer neural network typically has a size that we are free to set and that controls capacity. Because of early stopping and possibly other regularizers (e.g., weight decay, discussed below), it is mostly important to choose n_h large enough. Larger than optimal values typically do not hurt generalization performance much, but of

course they require proportionally more computation (in $O(n_h^2)$ if scaling all the layers at the same time in a fully connected architecture). Like for many other hyper-parameters, there is the option of allowing a different value of n_h for each hidden layer²⁰ of a deep architecture. See the paragraph below entitled **Layer-wise optimization of hyper-parameters** (Sec. 19.3.3). In a large comparative study [70], we found that using the same size for all layers worked generally better or the same as using a decreasing size (pyramid-like) or increasing size (upside down pyramid), but of course this may be data-dependent. For most tasks that we worked on, we find that an *overcomplete*²¹ first hidden layer works better than an undercomplete one. Another even more often validated empirical observation is that the optimal n_h is much larger when using unsupervised pre-training in a supervised neural network, e.g., going from hundreds of units to thousands of units. A plausible explanation is that after unsupervised pre-training many of the hidden units are carrying information that is irrelevant to the specific supervised task of interest. In order to make sure that the information relevant to the task is captured, larger hidden layers are therefore necessary when using unsupervised pre-training.

- **Weight decay** regularization coefficient λ . A way to reduce overfitting is to add a *regularization term* to the training criterion, which limits the capacity of the learner. The parameters of machine learning models can be regularized by pushing them towards a prior value, which is typically 0. L2 regularization adds a term $\lambda \sum_i \theta_i^2$ to the training criterion, while L1 regularization adds a term $\lambda \sum_i |\theta_i|$. Both types of terms can be included. There is a clean Bayesian justification for such a regularization term: it is the negative log-prior $-\log P(\theta)$ on the parameters θ . The training criterion then corresponds to the negative joint likelihood of data and parameters, $-\log P(data, \theta) = -\log P(data|\theta) - \log P(\theta)$, with the loss function $L(z, \theta)$ being interpreted as $-\log P(z|\theta)$ and $-\log P(data|\theta) = -\sum_{t=1}^T L(z_t, \theta)$ if the *data* consists of T i.i.d. examples z_t . This detail is important to note because when one is doing stochastic gradient-based learning, it makes sense to use an unbiased estimator of the gradient of the total training criterion (including both the total loss and the regularizer), but one only considers a single mini-batch or example at a time. How should the regularizer be weighted in this sum, which is different from the sum of the regularizer and the total loss on all examples? On each mini-batch update, the gradient of the regularization penalty should be multiplied not just by λ but also by $\frac{B}{T}$, i.e., one over the number of updates needed to go once through the training set. When the training set size is not a multiple of B , the last mini-batch will have size $B' < B$ and the contribution of the regularizer to the mini-batch gradient should therefore be modified accordingly (i.e. scaled by $\frac{B'}{B}$ compared to other mini-batches). In the pure online setting (there is no fixed ahead training set size nor iterating again on the examples), it would then

²⁰ A hidden layer is a group of units that is neither an input layer nor an output layer.

²¹ Larger than the input vector.

make sense to use $\frac{B}{t}$ at example t , or one over the number of updates to date. L2 regularization penalizes large values more strongly and corresponds to a Gaussian prior $\propto \exp(-\frac{1}{2} \frac{\|\theta\|^2}{\sigma^2})$ with prior variance $\sigma^2 = 1/(2\lambda)$. Note that there is a connection between early stopping (see above, choosing the number of training iterations) and L2 regularization [34], with one basically playing the same role as the other (but early stopping allowing a much more efficient selection of the hyper-parameter value, which suggests dropping L2 regularization altogether when early-stopping is used). However, L1 regularization behaves differently and can sometimes be useful, acting as a form of feature selection. L1 regularization makes sure that parameters that are not really very useful are driven to zero (i.e. encouraging sparsity of the parameter values), and corresponds to a Laplace density prior $\propto e^{-\frac{\|\theta\|}{s}}$ with scale parameter $s = \frac{1}{\lambda}$. L1 regularization often helps to make the input filters²² cleaner (more spatially localized) and easier to interpret. Stochastic gradient descent will not yield actual zeros but values hovering around zero. If both L1 and L2 regularization are used, a different coefficient (i.e. a different hyper-parameter) should be considered for each, and one may also use a different coefficient for different layers. In particular, the input weights and output weights may be treated differently.

One reason for treating output weights differently (i.e., not relying only on early stopping) is that we know that it is sufficient to regularize only the output weights in order to constrain capacity: in the limit case of the number of hidden units going to infinity, L2 regularization corresponds to Support Vector Machines (SVM) while L1 regularization corresponds to boosting [12]. Another reason for treating inputs and outputs differently from hidden units is because they may be sparse. For example, some input features may be 0 most of the time while others are non-zero frequently. In that case, there are fewer examples that inform the model about that rarely active input feature, and the corresponding parameters (weights outgoing from the corresponding input units) should be more regularized than the parameters associated with frequently observed inputs. A similar situation may occur with target variables that are sparse (e.g., trying to predict rarely observed events). In both cases, the effective number of meaningful updates seen by these parameters is less than the actual number of updates. This suggests to scale the regularization coefficient of these parameters by one over the effective number of updates seen by the parameter. A related formula turns up in Bayesian probit regression applied to sparse inputs [53]. Some practitioners also choose to penalize only the weights w and not the biases b associated with the hidden unit activations $w'z + b$ for a unit taking the vector of values z as input. This guarantees that even with strong regularization, the predictor would converge to the optimal constant predictor, rather than the one corresponding to 0 activation. For example, with the mean-square loss and the cross-entropy loss, the optimal constant predictor is the output average.

²² The input weights of a 1st layer neuron are often called “filters” because of analogies with signal processing techniques such as convolutions.

- **Sparsity of activation** regularization coefficient α . A common practice in the Deep Learning literature [95, 97, 81, 82, 3, 49, 33, 52] consists in adding a penalty term to the training criterion that encourages the hidden units to be sparse, i.e., with values at or near 0. Although the L1 penalty (discussed above in the case of weights) can also be applied to hidden units activations, this is mathematically very different from the L1 regularization term on parameters. Whereas the latter corresponds to a prior on the parameters, the former does not because it involves the training distribution (since we are looking at data-dependent hidden units outputs). Although we will not discuss this much here, the inspiration for a sparse representation in Deep Learning comes from the earlier work on *sparse coding* [89]. As discussed in Goodfellow *et al.* [51] sparse representations may be advantageous because they encourage representations that *disentangle* the underlying factors of representation. A sparsity-inducing penalty is also a way to regularize (in the sense of reducing the number of examples that the learner can learn by heart) [97], which means that the sparsity coefficient is likely to interact with the many other hyper-parameters which influence capacity. In general, increased sparsity can be compensated by a larger number of hidden units. Several approaches have been proposed to induce a sparse representation (or with more hidden units whose activation is closer to 0). One approach [97, 72, 120] is simply to penalize the L1 norm of the representation or another function of the hidden units' activation (such as the student-t log-prior). This typically makes sense for non-linearities such as the sigmoid which have a saturating output around 0, but not for the hyperbolic tangent non-linearity (whose saturation is near the -1 and 1 interval borders rather than near the origin). Another option is to penalize the biases of the hidden units, to make them more negative [95, 81, 51, 69]. Note that penalizing the bias runs the danger that the weights could compensate for the bias²³, which could hurt the numerical optimization of parameters. When directly penalizing the hidden unit outputs, several variants can be found in the literature, but no clear comparative analysis has been published to evaluate which one works better. Although the L1 penalty (i.e., simply α times the sum of output elements h_j in the case of sigmoid non-linearity) would seem the most natural (because of its use in sparse coding), it is used in few papers involving sparse auto-encoders. A close cousin of the L1 penalty is the Student-t penalty ($\log(1 + h_j^2)$), originally proposed for sparse coding [89]. Several researchers penalize the *average* output \bar{h}_j (e.g. over a mini-batch), and instead of pushing it to 0, encourage it to approach a fixed target ρ . This can be done through a mean-square error penalty such as $\sum_j (\rho - \bar{h}_j)^2$, or maybe more sensibly (because h_j behaves like a probability), a Kullback-Liebler divergence with respect to the binomial distribution with probability ρ , $-\rho \log \bar{h}_j - (1 - \rho) \log(1 - \bar{h}_j) + \text{constant}$, e.g., with $\rho = 0.05$, as in [59]. In addition to the regularization penalty itself, the choice of activation function

²³ Because the input to the layer generally has a non-zero average, that when multiplied by the weights acts like a bias.

can have a strong impact on the sparsity obtained. In particular, rectifying non-linearities (such as $\max(0, x)$, instead of a sigmoid) have been very successful in several instances [64, 86, 49, 84, 50]. The rectifier also relates to the hard tanh [35], whose derivatives are also 0 or 1. In sparse coding and sparse predictive coding [65] the activations are directly optimized and actual zeros are the expected result of the optimization. In that case, ordinary stochastic gradient is not guaranteed to find these zeros (it will oscillate around) and other methods such as proximal gradient are more appropriate [21].

- **Neuron non-linearity.** The typical neuron output is $s(a) = s(w'x + b)$, where x is the vector of inputs into the neuron, w the vector of weights and b the offset or bias parameter, while s is a scalar non-linear function. Several non-linearities have been proposed and some choices of non-linearities have been shown to be more successful [64, 48, 49]. The most commonly used by the author, for hidden units, are the sigmoid $1/(1 + e^{-a})$, the hyperbolic tangent $\frac{e^a - e^{-a}}{e^a + e^{-a}}$, the rectifier $\max(0, a)$ and the hard tanh [35]. Note that the sigmoid was shown to yield serious optimization difficulties when used as the top hidden layer of a deep supervised network [48] without unsupervised pre-training, but works well for auto-encoder variants²⁴. For output (or reconstruction) units, hard neuron non-linearities like the rectifier do not make sense because when the unit is saturated (e.g. $a < 0$ for the rectifier) and associated with a loss, no gradient is propagated inside the network, i.e., there is no chance to correct the error²⁵. In the case of hidden layers the gradient manages to go through a subset of the hidden units, even if the others are saturated. For output units a good trick is to obtain the output non-linearity and the loss by considering the associated negative log-likelihood and choosing an appropriate (conditional) output probability model, usually in the exponential family. For example, one can typically take squared error and linear outputs to correspond to a Gaussian output model, cross-entropy and sigmoids to correspond to a binomial output model, and $-\log \text{output}[\text{target class}]$ with softmax outputs to correspond to multinomial output variables. For reasons yet to be elucidated, having a sigmoidal non-linearity on the output (reconstruction) units (along with target inputs normalized in the (0,1) interval) seems to be helpful when training the contractive auto-encoder.
- **Weights initialization scaling coefficient.** Biases can generally be initialized to zero but weights need to be initialized carefully to break the

²⁴ The author hypothesizes that this discrepancy is due to the fact that the weight matrix W of an auto-encoder of the form $r(x) = W^T \text{sigmoid}(Wx)$ is pulled towards being orthonormal since this would make the auto-encoder closer to the identity function, because $W^T W x \approx x$ when W is orthonormal and x is in the span of the rows of W .

²⁵ A hard non-linearity for the output units non-linearity is very different from a hard non-linearity in the loss function, such as the hinge loss. In the latter case the derivative is 0 only when there is no error.

symmetry between hidden units of the same layer²⁶. Because different output units receive different gradient signals, this symmetry breaking issue does not concern the output weights (into the output units), which can therefore also be set to zero. Although several tricks [79, 48] for initializing the weights into hidden layers have been proposed (i.e. a hyper-parameter is the discrete choice between them), Bergstra and Bengio [17] also inserted as an extra hyper-parameter a scaling coefficient for the initialization range. These tricks are based on the idea that units with more inputs (the *fan-in* of the unit) should have smaller weights. Both LeCun *et al.* [79] and Glorot and Bengio [48] recommend scaling by the inverse of the *square root* of the fan-in, although Glorot and Bengio [48] and the Deep Learning Tutorials use a combination of the fan-in and fan-out, e.g., sample a $\text{Uniform}(-r, r)$ with $r = \sqrt{6/(\text{fan-in} + \text{fan-out})}$ for hyperbolic tangent units and $r = 4\sqrt{6/(\text{fan-in} + \text{fan-out})}$ for sigmoid units. We have found that we could avoid any hyper-parameter related to initialization using these formulas (and the derivation in Glorot and Bengio [48] can be used to derive the formula for other settings). Note however that in the case of RBMs, a zero-mean Gaussian with a small standard deviation around 0.1 or 0.01 works well [59] to initialize the weights, while visible biases are typically set to their optimal value if the weights were 0, i.e., $\log(\bar{x}/(1 - \bar{x}))$ in the case of a binomial visible unit whose corresponding binary input feature has empirical mean \bar{x} in the training set.

An important choice is whether one should use *unsupervised pre-training* (and which unsupervised feature learning algorithm to use) in order to initialize parameters. In most settings we have found unsupervised pre-training to help and very rarely to hurt, but of course that implies additional training time and additional hyper-parameters.

- **Random seeds.** There are often several sources of randomness in the training of neural networks and deep learners (such as for random initialization, sampling examples, sampling hidden units in stochastic models such as RBMs, or sampling corruption noise in denoising auto-encoders). Some random seeds could therefore yield better results than others. Because of the presence of local minima in the training criterion of neural networks (except in the linear case or with fixed lower layers), parameter initialization matters. See Erhan *et al.* [44] for an example of histograms of test errors for hundreds of different random seeds. Typically, the choice of random seed only has a slight effect on the result and can mostly be ignored in general or for most of the hyper-parameter search process. If computing power is available, then a final set of jobs with different random seeds (5 to 10) for a small set of best choices of hyper-parameter values can squeeze a bit more performance. Another way to exploit computing power to push performance a bit is model averaging, as in Bagging [29] and Bayesian methods. After training them,

²⁶ By symmetry, if hidden units of the same layer share the same input and output weights, they will compute the same output and receive the same gradient, hence performing the same update and remaining identical, thus wasting capacity.

the outputs of different networks (or in general different learning algorithms) can be averaged. For example, the difference between the neural networks being averaged into a committee may come from the different seeds used for parameter initialization, or the use of different subsets of input variables, or different subsets of training examples (the latter being called Bagging).

- **Preprocessing.** Many preprocessing steps have been proposed to massage raw data into appropriate inputs for neural networks and model selection must also choose among them. In addition to element-wise standardization (subtract mean and divide by standard deviation), Principal Components Analysis (PCA) has often been advocated [79, 17] and also allows dimensionality reduction, at the price of an extra hyper-parameter (the number of principal components retained, or the proportion of variance explained). A convenient non-linear preprocessing is the *uniformization* [84] of each feature (which estimates its cumulative distribution F_i and then transforms each feature x_i by its quantile $F_i^{-1}(x_i)$, i.e., returns an approximate normalized rank or quantile for the value x_i). A simpler to compute transform that may help reduce the tails of input features is a non-linearity such as the logarithm or the square root, in an attempt to make them more Gaussian-like.

In addition to the above somewhat generic choices, more choices arise with different architectures and learning algorithms. For example, the denoising auto-encoder has a hyper-parameter scaling the amount of input corruption and the contractive auto-encoder has as hyper-parameter a coefficient scaling the norm of the Jacobian of the encoder, i.e., controlling the importance of the contraction penalty. The latter seems to be a rather sensitive hyper-parameter that must be tuned carefully. The contractive auto-encoder’s success also seems sensitive to the **weight tying** constraint used in many auto-encoder architectures: the decoder’s weight matrix is equal to the transpose of the encoder’s weight matrix. The specific architecture used in the contractive auto-encoder (with tied weights, sigmoid non-linearities on hidden and reconstruction units, along with squared loss or cross-entropy loss) works quite well but other related variants do not always train well, for reasons that remain to be understood.

There are also many architectural choices that are relevant in the case of convolutional architectures (e.g. for modeling images, time-series or sound) [78, 80, 71] in which hidden units have local receptive fields.

19.3.3 Manual Search and Grid Search

Many of the hyper-parameters or model choices described above can be ignored by picking a standard trick suggested here or in some other paper. Still, one remains with a substantial number of choices to be made, which may give the impression of neural network training as an art. With modern computing facilities based on large computer clusters, it is however possible to make the optimization of hyper-parameters a more reproducible and automated process, using techniques such as grid search or better, random search, or even hyper-parameter optimization, discussed below.

General Guidance for the Exploration of Hyper-parameters. First of all, let us consider recommendations for exploring hyper-parameter settings, whether with manual search, with an automated procedure, or with a combination of both. We call a *numerical hyper-parameter* one that involves choosing a real number or an integer (where order matters), as opposed to making a discrete symbolic choice from an unordered set. Examples of numerical hyper-parameters are regularization coefficients, number of hidden units, number of training iterations, etc. One has to think of hyper-parameter selection as a *difficult form of learning*: there is both an optimization problem (looking for hyper-parameter configurations that yield low validation error) and a generalization problem: there is uncertainty about the expected generalization after optimizing validation performance, and it is possible to overfit the validation error and get optimistically biased estimators of performance when comparing many hyper-parameter configurations. The training criterion for this learning is typically the validation set error, which is a proxy for generalization error. Unfortunately, the relation between hyper-parameters and validation error can be complicated. Although to first approximation we expect a kind of U-shaped curve (when considering only a single hyper-parameter, the others being fixed), this curve can also have noisy variations, in part due to the use of finite data sets.

- **Best value on the border.** When considering the validation error obtained for different values of a numerical hyper-parameter one should pay attention as to whether or not the best value found is near the border of the investigated interval. If it is near the border, then this suggests that better values can be found with values beyond the border: it is recommended in that case to explore further, beyond that border. Because the relation between a hyper-parameter and validation error can be noisy, it is generally not enough to try very few values. For instance, trying only 3 values for a numerical hyper-parameter is insufficient, even if the best value found is the middle one.
- **Scale of values considered.** Exploring values of a numerical hyper-parameter entails choosing a *starting interval* to be searched, which is therefore a kind of hyper-hyper-parameter. By choosing the interval large enough to start with, but based on previous experience with this hyper-parameter, we ensure that we do not get completely wrong results. Now instead of choosing the intermediate values *linearly* in the chosen interval, it often makes much more sense to consider a linear or uniform sampling in the *log-domain* (in the space of the logarithm of the hyper-parameter). For example, the results obtained with a learning rate of 0.01 are likely to be very similar to the results with 0.011 while results with 0.001 could be quite different from results with 0.002 even though the absolute difference is the same in both cases. The *ratio* between different values is often a better guide of the expected impact of the change. That is why exploring uniformly or regularly-spaced values in the space of the *logarithm* of the numerical hyper-parameter is typically preferred for positive-valued numerical hyper-parameters.

– **Computational considerations.** Validation error is actually not the only measure to consider in selecting hyper-parameters. Often, one has to consider computational cost, either of training or prediction. Computing resources for training and prediction are limited and generally condition the choice of intervals of considered values: for example increasing the number of hidden units or number of training iterations also scales up computation. An interesting idea is to use *computationally cheap estimators* of validation error to select some hyper-parameters. For example, Saxe *et al.* [105] showed that the architecture hyper-parameters of convolutional networks could be selected using *random weights* in the lower layers of the network (filters of the convolution). While this yields a noisy and biased (pessimistic) estimator of the validation error which would otherwise be obtained with full training, this cheap estimator appears to be correlated with the expensive validation error. Hence this cheap estimator is enough for selecting some hyper-parameters (or for keeping under consideration for further and more expensive evaluation only the few best choices found). Even without cheap estimators of generalization error, high-throughput computing (e.g., on clusters, GPUs, or clusters of GPUs) can be exploited to run not just hundreds but thousands of training jobs, something not conceivable only a few years ago, with each job taking on the order of hours or days for larger datasets. With computationally cheap surrogates, some researchers have run on the order of ten thousands trials, and we can expect future advances in parallelized computing power to boost these numbers.

Coordinate Descent and Multi-resolution Search. When performing a manual search and with access to only a single computer, a reasonable strategy is *coordinate descent*: change only one hyper-parameter at a time, always making a change from the best configuration of hyper-parameters found up to now. Instead of a standard coordinate descent (which systematically cycles through all the variables to be optimized) one can make sure to regularly fine-tune the most sensitive variables, such as the learning rate.

Another important idea is that there is no point in exploring the effect of fine changes before one or more reasonably good settings have been found. The idea of *multi-resolution search* is to start the search by considering only a few values of the numerical hyper-parameters (over a large range), or considering large changes each time a new value is tried. One can then start from the one or few best configurations found and explore more locally around them with smaller variations around these values.

Automated and Semi-automated Grid Search. Once some interval or set of values has been selected for each hyper-parameter (thus defining a search space), a simple strategy that exploits parallel computing is the **grid search**. One first needs to convert the numerical intervals into lists of values (e.g., K regularly-spaced values in the log-domain of the hyper-parameter). The grid search is simply an exhaustive search through all the combinations of these values. The cross-product of these lists contains a number of elements that

is unfortunately exponential in the number of hyper-parameters (e.g., with 5 hyper-parameters, each allowed to take 6 different values, one gets $6^5 = 7776$ configurations). In section 19.3.4 below we consider an approach that works more efficiently than the grid search when the number of hyper-parameters increases beyond 2 or 3.

The advantage of the grid search, compared to many other optimization strategies (such as coordinate descent), is that it is fully parallelizable. If a large computer cluster is available, it is tempting to choose a model selection strategy that can take advantage of parallelization. One practical disadvantage of grid search (especially against random search, Sec. 19.3.4), with a parallelized set of jobs on a cluster, is that if only one of the jobs fails²⁷ then one has to launch another volley of jobs to complete the grid (and yet a third one if any of these fails, etc.), thus multiplying the overall computing time.

Typically, a single grid search is not enough and practitioners tend to proceed with a sequence of grid searches, each time adjusting the ranges of values considered based on the previous results obtained. Although this can be done manually, this procedure can also be automated by considering the idea of multi-resolution search to guide this outer loop. Different, more local, grid searches can be launched in the neighborhood of the best solutions found previously. In addition, the idea of coordinate descent can also be thrown in, by making each grid search focus on only a few of the hyper-parameters. For example, it is common practice to start by exploring the initial learning rate while keeping fixed (and initially constant) the learning rate descent schedule. Once the shape of the schedule has been chosen, it may be possible to further refine the learning rate, but in a smaller interval around the best value found.

Humans can get very good at performing hyper-parameter search, and having a human in the loop also has the advantage that it can help detect bugs or unwanted or unexpected behavior of a learning algorithm. However, for the sake of reproducibility, machine learning researchers should strive to use procedures that do not involve human decisions in the middle, only at the outset (e.g., setting hyper-parameter ranges, which can be specified in a paper describing the experiments).

Layer-Wise Optimization of Hyper-parameters. In the case of Deep Learning with unsupervised pre-training there is an opportunity for combining coordinate descent and cheap relative validation set performance evaluation associated with some hyper-parameter choices. The idea, described by Mesnil *et al.* [84], Bengio [8], is to perform greedy choices for the hyper-parameters associated with lower layers (near the input) before training the higher layers. One first trains (unsupervised) the first layer with different hyper-parameter values and somehow estimates the relative validation error that would be obtained from these different configurations if the final network only had this single layer as internal representation. In the common case where the ultimate task is supervised, it means training a simple supervised predictor (e.g. a linear classifier) on

²⁷ For all kinds of hardware and software reasons, a job failing is very common.

top of the learned representation. In the case of a linear predictor (e.g. regression or logistic regression) this can even be done on the fly while unsupervised training of the representation progresses (i.e. can be used for early stopping as well), as in [70]. Once a set of apparently good (according to this greedy evaluation) hyper-parameters values has been found (or possibly using only the best one found), these good values can be used as starting point to train (and hyper-optimize) a second layer in the same way, etc. The completely greedy approach is to keep only the best configuration up to now (for the lower layers), but keeping the K best configurations overall only multiplies computational costs of hyper-parameter selection by K for layers beyond the first one, because we would still keep only the best K configurations from all the 1st layer and 2nd layer hyper-parameters as starting points for exploring 3rd layer hyper-parameters, etc. This procedure is formalized in the Algorithm 19.1 below. Since greedy layer-wise pre-training does not modify the lower layers when pre-training the upper layers, this is also very efficient computationally. This procedure allows one to set the hyper-parameters associated with the unsupervised pre-training stage, and then there remains hyper-parameters to be selected for the supervised fine-tuning stage, if one is desired. A final supervised fine-tuning stage is strongly suggested, especially when there are many labeled examples [67].

19.3.4 Random Sampling of Hyper-parameters

A serious problem with the grid search approach to find good hyper-parameter configurations is that it scales exponentially badly with the number of hyper-parameters considered. In the above sections we have discussed numerous hyper-parameters and if all of them were to be explored at the same time it would be impossible to use only a grid search to do so.

One may think that there are no other options simply because this is an instance of the curse of dimensionality. But like we have found in our work on Deep Learning [7], if there is some structure in a target function we are trying to discover, then there is a chance to find good solutions without paying an exponential price. It turns out that in many practical cases we have encountered, there is a kind of structure that *random sampling* can exploit [17]. The idea of random sampling is to replace the regular grid by a random (typically uniform) sampling. Each tested hyper-parameter configuration is selected by independently sampling each hyper-parameter from a prior distribution (typically uniform in the log-domain, inside the interval of interest). For a discrete hyper-parameter, a multinomial distribution can be defined according to our prior beliefs on the likely good values. At worse, i.e., with no prior preference at all, this would be a uniform distribution across the allowed values. In fact, we can use our prior knowledge to make this prior distribution quite sophisticated. For example, we can readily include knowledge that some values of some hyper-parameters only make sense in the context of other particular values of hyper-parameters. This is a practical consideration for example when considering layer-specific hyper-parameters when the number of layers itself is a hyper-parameter.

Algorithm 19.1 Greedy layer-wise hyper-parameter optimization.

```

input  $K$ : number of best configurations to keep at each level.
input  $NLEVELS$ : number of levels of the deep architecture
input  $LEVELSETTINGS$ : list of hyper-parameter settings to be considered
for unsupervised pre-training of a level
input  $SFTSETTINGS$ : list of hyper-parameter settings to be considered for
supervised fine-tuning

Initialize set of best configurations  $S = \emptyset$ 
for  $L = 1$  to  $NLEVELS$  do
    for  $C$  in  $LEVELSETTINGS$  do
        for  $H$  in ( $S$  or  $\{\emptyset\}$ ) do
            * Pretrain level  $L$  using hyper-parameter setting  $C$  for level  $L$  and
            the parameters obtained with setting  $H$  for lower levels.
            * Evaluate target task performance  $\mathcal{L}$  using this depth- $L$  pre-trained
            architecture (e.g. train a linear classifier on top of these layers and estimate
            validation error).
            * Push the pair  $(C \cup H, \mathcal{L})$  into  $S$  if it is among the  $K$  best performing
            of  $S$ .
        end for
    end for
end for
for  $C$  in  $SFTSETTINGS$  do
    for  $H$  in  $S$  do
        * Supervised fine-tuning of the pre-trained architecture associated with
         $H$ , using supervised fine-tuning hyper-parameter setting  $C$ .
        * Evaluate target task performance  $\mathcal{L}$  of this fine-tuned predictor (e.g.
        validation error).
        * Push the pair  $(C \cup H, \mathcal{L})$  into  $S$  if it is among the  $K$  best performing
        of  $S$ .
    end for
end for
output  $S$  the set of  $K$  best-performing models with their settings and vali-
dation performance.

```

The experiments performed [17] show that random sampling can be many times more efficient than grid search as soon as the number of hyper-parameters goes beyond the 2 or 3 typically seen with SVMs and vanilla neural networks. The main reason why faster convergence is observed is because it allows one to explore more values for each hyper-parameter, whereas in grid search, the same value of a hyper-parameter is repeated in exponentially many configurations (of all the other hyper-parameters). In particular, if only a small subset of the hyper-parameters really matters, then this procedure can be shown to be exponentially more efficient. What we found is that for different datasets and architectures, the subset of hyper-parameters that mattered most was different, but it was often the case that a few hyper-parameters made a big difference (and the learning rate is always one of them!). When marginalizing (by averaging or minimizing) the validation performance to visualize the effect of one or two hyper-parameters, we get a more noisy picture using a random search compared to a grid search, because of the random variations of the other hyper-parameters but one with much more resolution, because so many more different values have been considered. Practically, one can plot the curves of best validation error as the number of random trials²⁸ is increased (with mean and standard deviation, obtained by considering, for each choice of number of trials, all possible same-size subsets of trials), and this curve tells us that we are approaching a plateau, i.e., it tells us whether it is worth it or not to continue launching jobs, i.e., we can perform a kind of early stopping in the outer optimization over hyper-parameters. Note that one should distinguish the curve of the “best trial in first N trials” with the curve of the mean (and standard deviation) of the “best in a subset of size N”. The latter is a better statistical representative of the improvements we should expect if we increase the number of trials. Even if the former has a plateau, the latter may still be on the increase, pointing for the need to more hyper-parameter configuration samples, i.e., more trials [17]. Comparing these curves with the equivalent obtained from grid search we see faster convergence with random search. On the other hand, note that one advantage of grid search compared to random sampling is that the qualitative analysis of results is easier because one can consider variations of a single hyper-parameter with all the other hyper-parameters being fixed. It may remain a valid option to do a small grid search around the best solutions found by random search, considering only the hyper-parameters that were found to matter or which concern a scientific question of interest²⁹.

Random search maintains the advantage of easy parallelization provided by grid search and improves on it. Indeed, a practical advantage of random search compared to grid search is that if one of the jobs fails then there is no need to re-launch that job. It also means that if one has launched 100 random search

²⁸ Each random trial corresponding to a training job with a particular choice of hyper-parameter values.

²⁹ This is often the case in machine learning research, e.g., does depth of architecture matter? then we need to control accurately for the effect of depth, with all other hyper-parameters optimized for each value of depth.

jobs, and finds that the convergence curve still has an interesting slope, one can launch another 50 or 100 without wasting the first 100. It is not that simple to combine the results of two grid searches because they are not always compatible (i.e., one is not a subset of the other).

Finally, although random search is a useful addition to the toolbox of the practitioner, semi-automatic exploration is still helpful and one will often iterate between launching a new volley of jobs and analysis of the results obtained with the previous volley in order to guide model design and research. What we need is more, and more efficient, automation of hyper-parameter optimization. There are some interesting steps in this direction [62, 19, 63, 109] but much more needs to done.

19.4 Debugging and Analysis

19.4.1 Gradient Checking and Controlled Overfitting

A very useful debugging step consists in verifying that the implementation of the gradient $\frac{\partial L}{\partial \theta}$ is compatible with the computation of L as a function of θ . If the analytically computed gradient does not match the one obtained by a finite difference approximation, this signals that a bug is probably present somewhere. First of all, looking at for which i one gets important relative change between $\frac{\partial L}{\partial \theta_i}$ and its finite difference approximation, we can get hints as to where the problem may be. An error in sign is particularly troubling, of course. A good next step is then to verify in the same way intermediate gradients $\frac{\partial L}{\partial a}$ with a some quantities that depend on the faulty θ , such as intervening neuron activations.

As many researchers know, the gradient can be approximated by a finite difference approximation obtained from the first-order Taylor expansion of a scalar function f with respect to a scalar argument x :

$$\frac{\partial f(x)}{\partial x} = \frac{f(x + \varepsilon) - f(x)}{\varepsilon} + o(\varepsilon)$$

But a less known fact is that a second order approximation can be achieved by considering the following alternative formula:

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x + \varepsilon) - f(x - \varepsilon)}{2\varepsilon} + o(\varepsilon^2).$$

The second order terms of the Taylor expansion of $f(x + \varepsilon)$ and $f(x - \varepsilon)$ cancel each other because they are even, leaving only 3rd or higher order terms, i.e., $o(\varepsilon^2)$ error after dividing the difference by ε . Hence this formula is twice more expensive (not a big deal while debugging) but provides quadratically more precision.

Note that because of finite precision in the computation, there will be a difference between the analytic (even correct) and finite difference gradient. Contrary to naive expectations, the relative difference may *grow* if we choose an ε that is too small, i.e., the error should first decrease as ε is decreased, and then may

worsen when numerical precision kicks in, due to non-linearities. We have often used a value of $\varepsilon = 10^{-4}$ in neural networks, a value that is sufficiently small to detect most bugs.

Once the gradient is known to be well computed, another sanity check is that gradient descent (or any other gradient-based optimization) should be able to overfit on a small training set³⁰. In particular, to factor out effects of SGD hyper-parameters, a good sanity check for the code (and the other hyper-parameters) is to verify that one can overfit on a small training set using a powerful second order method such as L-BFGS. For any optimizer, though, as the number of examples is increased, the degradation of training error should be gradual while validation error should improve. And one typically sees the advantages of SGD over batch second-order methods like L-BFGS increase as the training set size increases. The break-even point may depend on the task, parallelization (multi-core or GPU, see Sec.19.5 below), and architecture (number of computations compared to number of parameters, per example).

Of course, the real goal of learning is to achieve good generalization error, and the latter can be estimated by measuring performance on an independent test set. When test error is considered too high, the first question to ask is whether it is because of a difficulty in optimizing the training criterion or because of overfitting. Comparing training error and test error (and how they change as we change hyper-parameters that influence capacity, such as the number of training iterations) helps to answer that question. Depending on the answer, of course, the appropriate ways to improve test error are different. Optimization difficulties can be fixed by looking for bugs in the training code, inappropriate values of optimization hyper-parameters, or simply insufficient capacity (e.g. not enough degrees of freedom, hidden units, embedding sizes, etc.). Overfitting difficulties can be addressed by collecting more training data, introducing more or better regularization terms, multi-task training, unsupervised pre-training, unsupervised term in the training criterion, or considering different function families (or neural network architectures). In a multi-layer neural network, both problems can be simultaneously present. For example, as discussed in Bengio *et al.* [14], Bengio [7], it is possible to have zero training error with a large top-level hidden layer that allows the output layer to overfit, while the lower layer are not doing a good job of extracting useful features because they were not properly optimized.

Unless using a framework such as Theano which automatically handles the efficient allocation of buffers for intermediate results, it is important to pay attention to such buffers in the design of the code. The first objective is to avoid memory allocation in the middle of the training loop, i.e., all memory buffers should be allocated once and for all. Careless reuse of the same memory

³⁰ In principle, bad local minima could prevent that, but in the overfitting regime, e.g., with more hidden units than examples, the global minimum of the training error can generally be reached almost surely from random initialization, presumably because the training criterion becomes convex in the parameters that suffice to get the training error to zero [12], i.e., the output weights of the neural network.

buffers for different uses can however lead to bugs, which can be checked, in the debugging phase, by initializing buffers to the NaN (Not-A-Number) value, which propagates into downstream computation (making it easy to detect that uninitialized values were used)³¹.

19.4.2 Visualizations and Statistics

The most basic statistics that should be measured during training are error statistics. The *average loss* on the training set and the validation set and their evolution during training are very useful to monitor progress and differentiate overfitting from poor optimization. To make comparisons easier, it may be useful to compare neural networks during training in terms of their “age” (number of updates made times mini-batch size B , i.e., number of examples visited) rather than in terms of number of epochs (which is very sensitive to the training set size).

When using unsupervised training to learn the first few layers of a deep architecture, a very common debugging and analysis tool is the *visualization of filters*, i.e., of the weight vectors associated with individual hidden units. This is simplest in the case of the first layer and where the inputs are images (or image patches), time-series, or spectrograms (all of which are visually interpretable). Several recipes have been proposed to extend this idea to visualize the preferred input of hidden units in layers that follow the first one [81, 43]. In the case of the first layer, since one often obtains Gabor filters, a parametric fit of these filters to the weight vector can be done so as to visualize the distribution of orientations, positions and scales of the learned filters. An interesting special case of visualizing first-layer weights is the visualization of *word embeddings* (see Section 19.5.3 below) using a dimensionality reduction technique such as t-SNE [113].

An extension of the idea of visualizing filters (which can apply to non-linear or deeper features) is that of visualizing local (around the given test point) leading tangent vectors, i.e., the main directions in input space to which the representation (at a given layer) is most sensitive to [100].

In the case where the inputs are not images or easily visualizable, or to get a sense of the weight values in different hidden units, Hinton diagrams [58] are also very useful, using small squares whose color (black or white) indicates a weight’s sign and whose area represents its magnitude.

Another way to visualize what has been learned by an unsupervised (or joint label-input) model is to look at samples from the model. Sampling procedures have been defined at the outset for RBMs, Deep Belief Nets, and Deep Boltzmann Machines, for example based on Gibbs sampling. When weights become larger, mixing between modes can become very slow with Gibbs sampling. An interesting alternative is rates-FPCD [112, 30] which appears to be more robust to this problem and generally mixes faster, but at the cost of losing theoretical guarantees.

³¹ Personal communication from David Warde-Farley, who learned this trick from Sam Roweis.

In the case of auto-encoder variants, it was not clear until recently whether they were really capturing the underlying density (since they are not optimized with respect to the maximum likelihood principle or an approximation of it). It was therefore even less clear if there existed appropriate sampling algorithms for auto-encoders, but a recent proposal for sampling from contractive auto-encoders appears to be working very well [101], based on arguments about the geometric interpretation of the first derivative of the encoder [16], showing that denoising and contractive auto-encoders capture local moments (first and second) of the training density.

To get a sense of what individual hidden units represent, it has also been proposed to vary only one unit while keeping the others fixed, e.g., to the value obtained by finding the hidden units representation associated with a particular input example.

Another interesting technique is the *visualization of the learning trajectory in function space* [44]. The idea is to associate the function (as opposed to simply the parameters) computed by a neural network with a low-dimensional (2-D or 3-D) representation, e.g., with the t-SNE [113] or Isomap [111] algorithms, and then plot the evolution of this function during training, or the population of such trajectories for different initializations. This provides visualization of *effective local minima*³² and shows that no two different random initializations ended up in the same effective local minimum.

Finally, another useful type of visualization is to display statistics (e.g., histogram, mean and standard deviation) of activations (inputs and outputs of the non-linearities at each layer), activation gradients, parameters and parameter gradients, by groups (e.g. different layers, biases vs weights) and across training iterations. See Glorot and Bengio [48] for a practical example. A particularly interesting quantity to monitor is the discriminative ability of the representations learnt at each layer, as discussed in [85], and ultimately leading to an analysis of the disentangled factors captured by the different layers as we consider deeper architectures.

19.5 Other Recommendations

19.5.1 Multi-core Machines, BLAS and GPUs

Matrix operations are the most time-consuming in efficient implementations of many machine learning algorithms and this is particularly true of neural networks and deep architectures. The basic operations are matrix-vector products (forward propagation and back-propagation) and vector times vector outer products (resulting in a matrix of weight gradients). Matrix-matrix multiplications can be done substantially faster than the equivalent sequence of matrix-vector products for two reasons: by smart caching mechanisms such as implemented in the BLAS library (which is called from many higher-level environments such as

³² It is difficult to know for sure if it is a true local minima or if it appears like one because the optimization algorithm is stuck.

python’s numpy and Theano, Matlab, Torch or Lush), and thanks to parallelism. Appropriate versions of BLAS can take advantage of multi-core machines to distribute these computations on multi-core machines. The speed-up is however generally a fraction of the total speedup one can hope for (e.g. $4\times$ on a 4-core machine), because of communication overheads and because not all computation is parallelized. Parallelism becomes more efficient when the sizes of these matrices is increased, which is why mini-batch updates can be computationally advantageous, and more so when more cores are present.

The extreme multi-core machines are the GPUs (Graphics Processing Units), with hundreds of cores. Unfortunately, they also come with constraints and specialized compilers which make it more difficult to fully take advantage of their potential. On 512-core machines, we are routinely able to get speed-ups of $4\times$ to $40\times$ for large neural networks. To make the use of GPUs practical, it really helps to use existing libraries that efficiently implement computations on GPUs. See Bergstra *et al.* [18] for a comparative study of the Theano library (which compiles numpy-like code for GPUs). One practical issue is that only the GPU-compiled operations will typically be done on the GPU, and that transfers between the GPU and CPU considerably slow things down. It is important to use a profiler to find out what is done on the GPU and how efficient these operations are in order to quickly invest one’s time where needed to make an implementation GPU-efficient and keep most operations on the GPU card.

19.5.2 Sparse High-Dimensional Inputs

Sparse high-dimensional inputs can be efficiently handled by traditional supervised neural networks by using a sparse matrix multiplication. Typically, the input is a sparse vector while the weights are in a dense matrix, and one should use an efficient implementation made for just this case in order to optimally take advantage of sparsity. There is still going to be an overhead on the order of $2\times$ or more (on the multiply-add operations, not the others) compared to a dense implementation of the matrix-vector product.

For many unsupervised learning algorithms there is unfortunately a difficulty. The computation for these learning algorithms usually involves some kind of reconstruction of the input (like for all auto-encoder variants, but also for RBMs and sparse coding variants), as if the inputs were in the output space of the learner. Two exceptions to this problem are semi-supervised embedding [117] and Slow Feature Analysis [119, 20]. The former pulls the representation of nearby examples near each other and pushes dissimilar points apart, while also tuning the representation for a supervised learning task. The latter maximizes the learned features’ variance while minimizing their covariance and maximizing their temporal auto-correlation.

For algorithms that do need a form of input reconstruction, an efficient approach based on *sampled reconstruction* [39] has been proposed, successfully implemented and evaluated for the case of auto-encoders and denoising auto-encoders. The first idea is that on each example (or mini-batch), one samples a subset of the elements of the reconstruction vector, along with the associated

reconstruction loss. One only needs to compute the reconstruction and the loss associated with these sampled elements (or features), as well as the associated back-propagation operations into hidden units and reconstruction weights. That alone would multiplicatively reduce the computational cost by the amount of sparsity but make the gradient much more noisy and possibly biased as well, if the sampling distribution was chosen not uniform. To reduce the variance of that estimator, the idea is to guess for which features the reconstruction loss will be larger and to sample with higher probability these features (and their loss). In particular, the authors always sample the features with a non-zero in the input (or the corrupted input, in the denoising case), and uniformly sample an equal number of those with a zero in the input and corrupted input. To make the estimator unbiased now requires introducing a weight on the reconstruction loss associated with each sampled feature, inversely proportional to the probability of sampling it, i.e., this is an importance sampling scheme. The experiments show that the speed-up increases linearly with the amount of sparsity while the average loss is optimized as well as in the deterministic full-computation case.

19.5.3 Symbolic Variables, Embeddings, Multi-task Learning and Multi-relational Learning

Parameter sharing [68, 77, 68, 31, 4, 5] is an old neural network technique for increasing statistical power: if a parameter is used in N times more contexts (different tasks, different parts of the input, etc.) then it may be as if we had N times more training examples for tuning its value. More examples to estimate a parameter reduces its variance (with respect to sampling of training examples), which is directly influencing generalization error: for example the generalization mean squared error can be decomposed as the sum of a bias term and a variance term [46]. The **reuse** idea was first exploited by applying the same parameter to different parts of the input, as in convolutional neural networks [68, 77]. Reuse was also exploited by sharing the lower layers of a network (and the representation of the input that they capture) across multiple tasks associated with different outputs of the network [31, 4, 5]. This idea is also one of the key motivations behind Deep Learning [7] because one can think of the intermediate features computed in higher (deeper) layers as different tasks that can share the sub-features computed in lower layers (nearer the input). This very basic notion of reuse is key to improving generalization in many settings, guiding the design of neural network architectures in practical applications as well.

An interesting special case of these ideas is in the context of learning with symbolic data. If some input variables are symbolic, taking value in a finite alphabet, they can be represented as neural network inputs by a one-hot sub-vector of the input vector (with a 0 everywhere except at the position associated with the particular symbol). Now, sometimes different input variables refer to different instances of the same *type* of symbol. A patent example is with neural language models [11, 6], where the input is a sequence of words. In these models, the same input layer weights are reused for words at different positions in the input sequence (as in convolutional networks). The product of a one-hot

sub-vector with this shared weight matrix is a generally dense vector, and this associates each symbol in the alphabet with a point in a vector space³³, which we call its *embedding*. The idea of vector space representations for words and symbols is older [40] and is a particular case of the notion of *distributed representation* [57, 58] central to the connectionist approaches. Learned embeddings of symbols (or other objects) can be conveniently visualized using a dimensionality reduction algorithm such as t-SNE [113].

In addition to sharing the embedding parameters across positions of words in an input sentence, Collobert *et al.* [36] share them across natural language processing tasks such as Part-Of-Speech tagging, chunking and semantic role labeling. Parameter sharing is a key idea behind convolutional nets, recurrent neural networks and dynamic Bayes nets, in which the same parameters are used for different temporal or spatial slices of the data. This idea has been generalized from sequences and 2-D images to arbitrary graphs with recursive neural networks or recursive graphical models [92, 45, 25, 108], Markov Logic Networks [98] and relational learning [47]. A relational database can be seen as a set of objects (or typed values) and relations between them, of the form (object1, relation-type, object2). The same global set of parameters can be shared to characterize such relations, across relations (which can be seen as tasks) and objects. Object-specific parameters are the parameters specifying the embedding of a particular discrete object. One can think of the elements of each embedding vector as *implicit learned attributes*. Different tasks may demand different attributes, so that objects which share some underlying characteristics and behavior should end up having similar values of some of their attributes. For example, words appearing in semantically and syntactically similar contexts end up getting a very close embedding [36]. If the same attributes can be useful for several tasks, then statistical power is gained through parameter sharing, and transfer of information between tasks can happen, making the data of some task informative for generalizing properly on another task.

The idea proposed in Bordes *et al.* [23, 24] is to learn an energy function that is lower for positive (valid) relations present in the training set, and parametrized in two parts: on the one hand the symbol embeddings and on the other hand the rest of the neural network that maps them to a scalar energy. In addition, by considering relation types themselves as particular symbolic objects, the model can reason about relations themselves and have relations between relation types. For example, ‘To be’ can act as a relation type (in subject-attribute relations) but in the statement “‘To be’ is a verb” it appears both as a relation type and as an object of the relation.

Such multi-relational learning opens the door to the application of neural networks outside of their traditional applications, which was based on a single homogeneous source of data, often seen as a matrix with one row per example and one column (or group of columns) per random variable. Instead, one often has multiple heterogeneous sources of data (typically providing examples seen

³³ The result of the matrix multiplication, which equals one of the columns of the matrix.

as a tuple of values), each involving different random variables. So long as these different sources *share some variables*, then the above multi-relational multi-task learning approaches can be applied. Each variable can be associated with its embedding function (that maps the value of a variable to a generic representation space that is valid *across tasks and data sources*). This framework can be applied not only to symbolic data but to mixed symbolic/numeric data if the mapping from object to embedding is generalized from a table look-up to a parametrized function (the simplest being a linear mapping) from its raw attributes (e.g., image features) to its embedding. This has been exploited successfully to design image search systems in which images and queries are mapped to the same semantic space [118].

19.6 Open Questions

19.6.1 On the Added Difficulty of Training Deeper Architectures

There are experimental results which provide some evidence that, at least in some circumstances, deeper neural networks are more difficult to train than shallow ones, in the sense that there is a greater chance of missing out on better minima when starting from random initialization. This is borne out by all the experiments where we find that some initialization scheme can drastically improve performance. In the Deep Learning literature this has been shown with the use of unsupervised pre-training (supervised or not), both applied to supervised tasks — training a neural network for classification [61, 14, 95] — and unsupervised tasks — training a Deep Boltzmann Machine to model the data distribution [104].

The learning trajectories visualizations of Erhan *et al.* [44] have shown that even when starting from nearby configurations in function space, different initializations seem to always fall in a different effective local minimum. Furthermore, the same study showed that the minima found when using unsupervised pre-training were far in function space from those found from random initialization, in addition to giving better generalization error. Both of these findings highlight the importance of initialization, hence of local minima effects, in deep networks. Finally, it has been shown that these effects were both increased when considering deeper architectures [44].

There are also results showing that specific ways of setting the initial distribution and ordering of examples (“curriculum learning”) can yield better solutions [42, 15, 66]. This also suggest that very particular ways of initializing parameters, very different from uniformly sampled, can have a strong impact on the solutions found by gradient descent. The hypothesis proposed in [15] is that curriculum learning can act similarly to a *continuation method*, i.e., starting from an easier optimization task (e.g. convex) and tracking the local minimum as the learning task is gradually made more difficult and closer to the real task of interest.

Why would training deeper networks be more difficult? This is clearly still an open question. A plausible partial answer is that deeper networks are also

more non-linear (since each layer composes more non-linearity on top of the previous ones), making gradient-based methods less efficient. It may also be that the number and structure of local minima both change qualitatively as we increase depth. Theoretical arguments support a potentially exponential gain in expressive power of deeper architectures [7, 9] and it would be plausible that with this added expressive power coming from the combinatorics of composed reuse of sub-functions could come a corresponding increase in the number (and possibly quality) of local minima. But the best ones could then also be more difficult to find.

On the practical side, several experimental results point to factors that may help training deep architectures:

- **A local training signal.** What many successful procedures for training deep networks have in common is that they involve a local training signal that helps each layer decide what to do without requiring the back-propagation of gradients through many non-linearities. This includes of course the many variants of greedy layer-wise pre-training but also the less well-known semi-supervised embedding algorithm [117].
- **Initialization in the right range.** Based on the idea that both activations and gradients should be able to flow well through a deep architecture without significant reduction in variance, Glorot and Bengio [48] proposed setting up the initial weights to make the Jacobian of each layer have singular values near 1 (or preserve variance in both directions). In their experiments this clearly helped greatly reducing the gap between purely supervised and pre-trained deep networks.
- **Choice of non-linearities.** In the same study [48] and a follow-up [49] it was shown that the choice of hidden layer non-linearities interacted with depth. In particular, without unsupervised pre-training, a deep neural network with sigmoids in the top hidden layer would get stuck for a long time on a plateau and generally produce inferior results, due to the special role of 0 and of the initial gradients from the output units. Symmetric non-linearities like the hyperbolic tangent did not suffer from that problem, while softer non-linearities (without exponential tails) such as the *softsign* function $s(a) = \frac{a}{1+|a|}$ worked even better. In Glorot *et al.* [49] it was shown that an asymmetric but hard-limiting non-linearity such as the rectifier ($s(a) = \max(0, a)$, see also [86]) actually worked very well (but should not be used for output units), in spite of the prior belief that the fact that when hidden units are saturated, gradients would not flow well into lower layers. In fact gradients flow very well, but on selected paths, possibly making the credit assignment (which parameters should change to handle the current error) sharper and the Hessian condition number better. A recent heuristic that is related to the difficulty of gradient propagation through neural net non-linearities is the idea of “centering” the non-linear operation such that each hidden unit has zero average output and zero average slope [107, 94].

19.6.2 Adaptive Learning Rates and Second-Order Methods

To improve convergence and remove learning rates from the list of hyper-parameters, many authors have advocated exploring adaptive learning rate methods, either for a global learning rate [32], a layer-wise learning rate, a neuron-wise learning rate, or a parameter-wise learning rate [22] (which then starts to look like a diagonal Newton method). LeCun [76], LeCun *et al.* [79] advocate the use of a second-order diagonal Newton (always positive) approximation, with one learning rate per parameter (associated with the approximated inverse second derivative of the loss with respect to the parameter). Hinton [59] proposes scaling learning rates so that the average weight update is on the order of 1/1000th of the weight magnitude. LeCun *et al.* [79] also propose a simple power method in order to estimate the largest eigenvalue of the Hessian (which would be the optimal learning rate). An interesting alternative to variants of Newton's method are variants of the *natural gradient* method [1], but like the basic Newton method it is computationally too expensive, requiring operations on a too large square matrix (number of parameters by number of parameters). Diagonal and low-rank online approximations of natural gradient [73, 74] have been proposed and shown to speed-up training in some contexts. Several adaptive learning rate procedures have been proposed recently and merit more attention and evaluations in the neural network context, such as *adagrad* [41] and the adaptive learning rate method from Schaul *et al.* [106] which claims to remove completely the need for a learning rate hyper-parameter.

Whereas stochastic gradient descent converges very quickly initially it is generally slower than second-order methods for the final convergence, and this may be important in some applications. As a consequence, batch training algorithms (performing only one update after seeing the whole training set) such as the Conjugate Gradient method (a second order method) have dominated stochastic gradient descent for not too large datasets (e.g. less than thousands or tens of thousands of examples). Furthermore, it has recently been proposed and successfully applied to use second-order methods over *large mini-batches* [72, 83]. The idea is to do just a few iterations of the second-order methods on each mini-batch and then move on to the next mini-batch, starting from the best previous point found. A useful twist is to start training with one or more epoch of SGD, since SGD remains the fastest optimizer early on in training.

At this point in time however, although the second-order and natural gradient methods are appealing conceptually, have demonstrably helped in the studied cases and may in the end prove to be very important, they have not yet become a standard for neural networks optimization and need to be validated and maybe improved by other researchers, before displacing simple (mini-batch) stochastic gradient descent variants.

19.7 Conclusion

In spite of decades of experimental and theoretical work on artificial neural networks, and with all the impressive progress made since the first edition of

this book, in particular in the area of Deep Learning, there is still much to be done to better train neural networks and better understand the underlying issues that can make the training task difficult. As stated in the introduction, the wisdom distilled here should be taken as a guideline, to be tried and challenged, not as a practice set in stone. The practice summarized here, coupled with the increase in available computing power, now allows researchers to train neural networks on a scale that is far beyond what was possible at the time of the first edition of this book, helping to move us closer to artificial intelligence.

Acknowledgements. The author is grateful for the comments and feedback provided by Nicolas Le Roux, Ian Goodfellow, James Bergstra, Guillaume Desjardins, Razvan Pascanu, David Warde-Farley, Eric Larsen, Frederic Bastien, and Sina Honari, as well as for the financial support of NSERC, FQRNT, CIFAR, and the Canada Research Chairs.

References

- [1] Amari, S.: Natural gradient works efficiently in learning. *Neural Computation* 10(2), 251–276 (1998)
- [2] Bach, F., Moulines, E.: Non-asymptotic analysis of stochastic approximation algorithms. In: NIPS 2011 (2011)
- [3] Bagnell, J.A., Bradley, D.M.: Differentiable sparse coding. In: NIPS 2009, pp. 113–120 (2009)
- [4] Baxter, J.: Learning internal representations. In: COLT 1995, pp. 311–320 (1995)
- [5] Baxter, J.: A Bayesian/information theoretic model of learning via multiple task sampling. *Machine Learning* 28, 7–40 (1997)
- [6] Bengio, Y.: Neural net language models. *Scholarpedia* 3(1), 3881 (2008)
- [7] Bengio, Y.: Learning deep architectures for AI. Now Publishers (2009)
- [8] Bengio, Y.: Deep learning of representations for unsupervised and transfer learning. In: JMLR W&CP: Proc. Unsupervised and Transfer Learning (2011)
- [9] Bengio, Y., Delalleau, O.: On the Expressive Power of Deep Architectures. In: Kivinen, J., Szepesvári, C., Ukkonen, E., Zeugmann, T. (eds.) ALT 2011. LNCS, vol. 6925, pp. 18–36. Springer, Heidelberg (2011)
- [10] Bengio, Y., LeCun, Y.: Scaling learning algorithms towards AI. In: Large Scale Kernel Machines (2007)
- [11] Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C.: A neural probabilistic language model. *JMLR* 3, 1137–1155 (2003)
- [12] Bengio, Y., Le Roux, N., Vincent, P., Delalleau, O., Marcotte, P.: Convex neural networks. In: NIPS 2005, pp. 123–130 (2006a)
- [13] Bengio, Y., Delalleau, O., Le Roux, N.: The curse of highly variable functions for local kernel machines. In: NIPS 2005, pp. 107–114 (2006b)
- [14] Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H.: Greedy layer-wise training of deep networks. In: NIPS 2006 (2007)
- [15] Bengio, Y., Louradour, J., Collobert, R., Weston, J.: Curriculum learning. In: ICML 2009 (2009)
- [16] Bengio, Y., Alain, G., Rifai, S.: Implicit density estimation by local moment matching to sample from auto-encoders. Technical report, arXiv:1207.0057 (2012)
- [17] Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. *J. Machine Learning Res.* 13, 281–305 (2012)

- [18] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., Bengio, Y.: Theano: a CPU and GPU math expression compiler. In: Proc. Python for Scientific Comp. Conf. (SciPy) (2010)
- [19] Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: NIPS 2011 (2011)
- [20] Berkes, P., Wiskott, L.: Applying Slow Feature Analysis to Image Sequences Yields a Rich Repertoire of Complex Cell Properties. In: Dorronsoro, J.R. (ed.) ICANN 2002. LNCS, vol. 2415, pp. 81–86. Springer, Heidelberg (2002)
- [21] Bertsekas, D.P.: Incremental gradient, subgradient, and proximal methods for convex optimization: a survey. Technical Report 2848, LIDS (2010)
- [22] Bordes, A., Bottou, L., Gallinari, P.: Sgd-qn: Careful quasi-newton stochastic gradient descent. Journal of Machine Learning Research 10, 1737–1754 (2009)
- [23] Bordes, A., Weston, J., Collobert, R., Bengio, Y. (2011). Learning structured embeddings of knowledge bases. In: AAAI (2011)
- [24] Bordes, A., Glorot, X., Weston, J., Bengio, Y.: Joint learning of words and meaning representations for open-text semantic parsing. In: AISTATS 2012 (2012)
- [25] Bottou, L.: From machine learning to machine reasoning. Technical report, arXiv.1102 (2011)
- [26] Bottou, L.: Stochastic Gradient Descent Tricks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 421–436. Springer, Heidelberg (2012)
- [27] Bottou, L., Bousquet, O.: The tradeoffs of large scale learning. In: NIPS 2008 (2008)
- [28] Bottou, L., LeCun, Y.: Large-scale on-line learning. In: NIPS 2003 (2004)
- [29] Breiman, L.: Bagging predictors. Machine Learning 24(2), 123–140 (1994)
- [30] Breuleux, O., Bengio, Y., Vincent, P.: Quickly generating representative samples from an rbm-derived process. Neural Computation 23(8), 2053–2073 (2011)
- [31] Caruana, R.: Multitask connectionist learning. In: Proceedings of the 1993 Connectionist Models Summer School, pp. 372–379 (1993)
- [32] Cho, K., Raiko, T., Ilin, A.: Enhanced gradient and adaptive learning rate for training restricted boltzmann machines. In: ICML 2011, pp. 105–112 (2011)
- [33] Coates, A., Ng, A.Y.: The importance of encoding versus training with sparse coding and vector quantization. In: ICML 2011 (2011)
- [34] Collobert, R., Bengio, S.: Links between perceptrons, MLPs and SVMs. In: ICML 2004 (2004a)
- [35] Collobert, R., Bengio, S.: Links between perceptrons, MLPs and SVMs. In: International Conference on Machine Learning, ICML (2004b)
- [36] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. Journal of Machine Learning Research 12, 2493–2537 (2011a)
- [37] Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A matlab-like environment for machine learning. In: BigLearn, NIPS Workshop (2011b)
- [38] Courville, A., Bergstra, J., Bengio, Y.: Unsupervised models of images by spike-and-slab RBMs. In: ICML 2011 (2011)
- [39] Dauphin, Y., Glorot, X., Bengio, Y.: Sampled reconstruction for large-scale learning of embeddings. In: Proc. ICML 2011 (2011)
- [40] Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. J. Am. Soc. Information Science 41(6), 391–407 (1990)
- [41] Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research (2011)

- [42] Elman, J.L.: Learning and development in neural networks: The importance of starting small. *Cognition* 48, 781–799 (1993)
- [43] Erhan, D., Courville, A., Bengio, Y.: Understanding representations learned in deep architectures. Technical Report 1355, Université de Montréal/DIRO (2010a)
- [44] Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., Bengio, S.: Why does unsupervised pre-training help deep learning? *J. Machine Learning Res.* 11, 625–660 (2010b)
- [45] Frasconi, P., Gori, M., Sperduti, A.: A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks* 9(5), 768–786 (1998)
- [46] Geman, S., Bienenstock, E., Doursat, R.: Neural networks and the bias/variance dilemma. *Neural Computation* 4(1), 1–58 (1992)
- [47] Getoor, L., Taskar, B.: *Introduction to Statistical Relational Learning*. MIT Press (2006)
- [48] Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: *AISTATS 2010*, pp. 249–256 (2010)
- [49] Glorot, X., Bordes, A., Bengio, Y. (2011a). Deep sparse rectifier neural networks. In: *AISTATS 2011* (2011)
- [50] Glorot, X., Bordes, A., Bengio, Y.: Domain adaptation for large-scale sentiment classification: A deep learning approach. In: *ICML 2011* (2011b)
- [51] Goodfellow, I., Le, Q., Saxe, A., Ng, A.: Measuring invariances in deep networks. In: *NIPS 2009*, pp. 646–654 (2009)
- [52] Goodfellow, I., Courville, A., Bengio, Y.: Spike-and-slab sparse coding for unsupervised feature discovery. In: *NIPS Workshop on Challenges in Learning Hierarchical Models* (2011)
- [53] Graepel, T., Candela, J.Q., Borchert, T., Herbrich, R.: Web-scale Bayesian click-through rate prediction for sponsored search advertising in microsoft’s bing search engine. In: *ICML* (2010)
- [54] Håstad, J.: Almost optimal lower bounds for small depth circuits. In: *STOC 1986*, pp. 6–20 (1986)
- [55] Håstad, J., Goldmann, M.: On the power of small-depth threshold circuits. *Computational Complexity* 1, 113–129 (1991)
- [56] Hinton, G.E.: Relaxation and its role in vision. Ph.D. thesis, University of Edinburgh (1978)
- [57] Hinton, G.E.: Learning distributed representations of concepts. In: *Proc. 8th Annual Conf. Cog. Sc. Society*, pp. 1–12 (1986)
- [58] Hinton, G.E.: Connectionist learning procedures. *Artificial Intelligence* 40, 185–234 (1989)
- [59] Hinton, G.E.: A practical guide to training restricted Boltzmann machines. Technical Report UTML TR 2010-003, Department of Computer Science, University of Toronto (2010)
- [60] Hinton, G.E.: A Practical Guide to Training Restricted Boltzmann Machines. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) *NN: Tricks of the Trade*, 2nd edn. LNCS, vol. 7700, pp. 599–619. Springer, Heidelberg (2012)
- [61] Hinton, G.E., Osindero, S., Teh, Y.-W.: A fast learning algorithm for deep belief nets. *Neural Computation* 18, 1527–1554 (2006)
- [62] Hutter, F.: *Automated Configuration of Algorithms for Solving Hard Computational Problems*. Ph.D. thesis, University of British Columbia (2009)
- [63] Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello Coello, C.A. (ed.) *LION 5*. LNCS, vol. 6683, pp. 507–523. Springer, Heidelberg (2011)

- [64] Jarrett, K., Kavukcuoglu, K., Ranzato, M., LeCun, Y.: What is the best multi-stage architecture for object recognition? In: ICCV (2009)
- [65] Kavukcuoglu, K., Ranzato, M.-A., Fergus, R., LeCun, Y.: Learning invariant features through topographic filter maps. In: CVPR 2009 (2009)
- [66] Krueger, K.A., Dayan, P.: Flexible shaping: how learning in small steps helps. *Cognition* 110, 380–394 (2009)
- [67] Lamblin, P., Bengio, Y.: Important gains from supervised fine-tuning of deep architectures on large labeled sets. In: NIPS 2010 Deep Learning and Unsupervised Feature Learning Workshop (2010)
- [68] Lang, K.J., Hinton, G.E.: The development of the time-delay neural network architecture for speech recognition. Technical Report CMU-CS-88-152, Carnegie-Mellon University (1988)
- [69] Larochelle, H., Bengio, Y.: Classification using discriminative restricted Boltzmann machines. In: ICML 2008 (2008)
- [70] Larochelle, H., Bengio, Y., Louradour, J., Lamblin, P.: Exploring strategies for training deep neural networks. *J. Machine Learning Res.* 10, 1–40 (2009)
- [71] Le, Q., Ngiam, J., Chen, Z., Hao Chia, D.J., Koh, P.W., Ng, A.: Tiled convolutional neural networks. In: NIPS 2010 (2010)
- [72] Le, Q., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., Ng, A.: On optimization methods for deep learning. In: ICML 2011 (2011)
- [73] Le Roux, N., Manzagol, P.-A., Bengio, Y.: Topmoumoute online natural gradient algorithm. In: NIPS 2007 (2008)
- [74] Le Roux, N., Bengio, Y., Fitzgibbon, A.: Improving first and second-order methods by modeling uncertainty. In: Optimization for Machine Learning. MIT Press (2011)
- [75] Le Roux, N., Schmidt, M., Bach, F.: A stochastic gradient method with an exponential convergence rate for strongly-convex optimization with finite training sets. Technical report, arXiv:1202.6258 (2012)
- [76] LeCun, Y.: Modèles connexionnistes de l'apprentissage. Ph.D. thesis, Université de Paris VI (1987)
- [77] LeCun, Y.: Generalization and network design strategies. Technical Report CRG-TR-89-4, University of Toronto (1989)
- [78] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Backpropagation applied to handwritten zip code recognition. *Neural Computation* 1(4), 541–551 (1989)
- [79] LeCun, Y.A., Bottou, L., Orr, G.B., Müller, K.-R.: Efficient BackProp. In: Orr, G.B., Müller, K.-R. (eds.) NIPS-WS 1996. LNCS, vol. 1524, pp. 9–50. Springer, Heidelberg (1998a)
- [80] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient based learning applied to document recognition. *IEEE* 86(11), 2278–2324 (1998b)
- [81] Lee, H., Ekanadham, C., Ng, A. (2008). Sparse deep belief net model for visual area V2. In: NIPS 2007 (2007)
- [82] Lee, H., Grosse, R., Ranganath, R., Ng, A.Y.: Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In: ICML 2009 (2009)
- [83] Martens, J.: Deep learning via Hessian-free optimization. In: ICML 2010, pp. 735–742 (2010)

- [84] Mesnil, G., Dauphin, Y., Glorot, X., Rifai, S., Bengio, Y., Goodfellow, I., Lavoie, E., Muller, X., Desjardins, G., Warde-Farley, D., Vincent, P., Courville, A., Bergstra, J.: Unsupervised and transfer learning challenge: a deep learning approach. In: Proc. Unsupervised and Transfer Learning, JMLR W&CP, vol. 7 (2011)
- [85] Montavon, G., Braun, M.L., Müller, K.-R.: Deep Boltzmann machines as feed-forward hierarchies. In: AISTATS 2012 (2012)
- [86] Nair, V., Hinton, G.E.: Rectified linear units improve restricted Boltzmann machines. In: ICML 2010 (2010)
- [87] Nemirovski, A., Yudin, D.: Problem complexity and method efficiency in optimization. Wiley (1983)
- [88] Nesterov, Y.: Primal-dual subgradient methods for convex problems. Mathematical Programming 120(1), 221–259 (2009)
- [89] Olshausen, B.A., Field, D.J.: Sparse coding with an overcomplete basis set: a strategy employed by V1? Vision Research 37, 3311–3325 (1997)
- [90] Pearlmutter, B.: Fast exact multiplication by the Hessian. Neural Computation 6(1), 147–160 (1994)
- [91] Pinto, N., Doukhan, D., DiCarlo, J.J., Cox, D.D.: A high-throughput screening approach to discovering good forms of biologically inspired visual representation. PLoS Comput. Biol. 5(11), e1000579 (2009)
- [92] Pollack, J.B.: Recursive distributed representations. Artificial Intelligence 46(1), 77–105 (1990)
- [93] Polyak, B., Juditsky, A.: Acceleration of stochastic approximation by averaging. SIAM J. Control and Optimization 30(4), 838–855 (1992)
- [94] Raiko, T., Valpola, H., LeCun, Y. (2012). Deep learning made easier by linear transformations in perceptrons. In: AISTATS 2012 (2012)
- [95] Ranzato, M., Poultney, C., Chopra, S., LeCun, Y.: Efficient learning of sparse representations with an energy-based model. In: NIPS 2006 (2007)
- [96] Ranzato, M., Boureau, Y.-L., LeCun, Y.: Sparse feature learning for deep belief networks. In: Platt, J., Koller, D., Singer, Y., Roweis, S. (eds.) Advances in Neural Information Processing Systems (NIPS 2007), vol. 20, pp. 1185–1192. MIT Press, Cambridge (2008a)
- [97] Ranzato, M., Boureau, Y., LeCun, Y.: Sparse feature learning for deep belief networks. In: NIPS 2007 (2008b)
- [98] Richardson, M., Domingos, P.: Markov logic networks. Machine Learning 62, 107–136 (2006)
- [99] Rifai, S., Vincent, P., Muller, X., Glorot, X., Bengio, Y.: Contracting auto-encoders: Explicit invariance during feature extraction. In: ICML 2011 (2011a)
- [100] Rifai, S., Dauphin, Y., Vincent, P., Bengio, Y., Muller, X.: The manifold tangent classifier. In: NIPS 2011 (2011b)
- [101] Rifai, S., Bengio, Y., Dauphin, Y., Vincent, P.: A generative process for sampling contractive auto-encoders. In: ICML 2012 (2012)
- [102] Robbins, H., Monro, S.: A stochastic approximation method. Annals of Mathematical Statistics 22, 400–407 (1951)
- [103] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature 323, 533–536 (1986)
- [104] Salakhutdinov, R., Hinton, G.: Deep Boltzmann machines. In: AISTATS 2009 (2009)
- [105] Saxe, A.M., Koh, P.W., Chen, Z., Bhand, M., Suresh, B., Ng, A.: On random weights and unsupervised feature learning. In: ICML 2011 (2011)

- [106] Schaul, T., Zhang, S., LeCun, Y.: No More Pesky Learning Rates. Technical report (2012)
- [107] Schraudolph, N.N.: Centering Neural Network Gradient Factors. In: Orr, G.B., Müller, K.-R. (eds.) NIPS-WS 1996. LNCS, vol. 1524, pp. 207–548. Springer, Heidelberg (1998)
- [108] Socher, R., Manning, C., Ng, A.Y.: Parsing natural scenes and natural language with recursive neural networks. In: ICML 2011 (2011)
- [109] Srinivasan, A., Ramakrishnan, G.: Parameter screening and optimisation for ILP using designed experiments. Journal of Machine Learning Research 12, 627–662 (2011)
- [110] Swersky, K., Chen, B., Marlin, B., de Freitas, N.: A tutorial on stochastic approximation algorithms for training restricted boltzmann machines and deep belief nets. In: Information Theory and Applications Workshop (2010)
- [111] Tenenbaum, J., de Silva, V., Langford, J.C.: A global geometric framework for nonlinear dimensionality reduction. Science 290(5500), 2319–2323 (2000)
- [112] Tieleman, T., Hinton, G.: Using fast weights to improve persistent contrastive divergence. In: ICML 2009 (2009)
- [113] van der Maaten, L., Hinton, G.E.: Visualizing data using t-sne. J. Machine Learning Res. 9 (2008)
- [114] Vincent, P.: A connection between score matching and denoising autoencoders. Neural Computation 23(7) (2011)
- [115] Vincent, P., Larochelle, H., Bengio, Y., Manzagol, P.-A.: Extracting and composing robust features with denoising autoencoders. In: ICML 2008 (2008)
- [116] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., Manzagol, P.-A.: Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. J. Machine Learning Res. 11 (2010)
- [117] Weston, J., Ratle, F., Collobert, R.: Deep learning via semi-supervised embedding. In: ICML 2008 (2008)
- [118] Weston, J., Bengio, S., Usunier, N.: Wsabie: Scaling up to large vocabulary image annotation. In: Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI (2011)
- [119] Wiskott, L., Sejnowski, T.J.: Slow feature analysis: Unsupervised learning of invariances. Neural Computation 14(4), 715–770 (2002)
- [120] Zou, W.Y., Ng, A.Y., Yu, K.: Unsupervised learning of visual invariance with temporal coherence. In: NIPS 2011 Workshop on Deep Learning and Unsupervised Feature Learning (2011)

20

Training Deep and Recurrent Networks with Hessian-Free Optimization

James Martens and Ilya Sutskever

University of Toronto, Department of Computer Science
`{jmartens,ilya}@cs.utoronto.ca`

Abstract. In this chapter we will first describe the basic HF approach, and then examine well-known performance-improving techniques such as preconditioning which we have found to be beneficial for neural network training, as well as others of a more heuristic nature which are harder to justify, but which we have found to work well in practice. We will also provide practical tips for creating efficient and bug-free implementations and discuss various pitfalls which may arise when designing and using an HF-type approach in a particular application.

20.1 Introduction

Hessian-Free optimization (HF) is an approach for unconstrained minimization of real-valued smooth objective functions. Like standard Newton’s method, it uses local quadratic approximations to generate update proposals. It belongs to the broad class of approximate Newton methods that are practical for problems of very high dimensionality, such as the training objectives of large neural networks. Different algorithms that use many of the same key principles have appeared in the literatures of various communities under different names such as Newton-CG, CG-Steihaug, Newton-Lanczos, and Truncated Newton [27, 28], but applications to machine learning and especially neural networks, have been limited or non-existent until recently. With the work of Martens [22] and later Martens and Sutskever [23] it has been demonstrated that such an approach, if carefully designed and implemented, can work very well for optimizing non-convex functions such as the training objective for deep neural networks and recurrent neural networks (RNNs), given sensible random initializations. This was significant because gradient descent methods have been observed to be very slow and sometimes completely ineffective [17, 4, 18] on these problems, unless special non-random initializations schemes like layer-wise pre-training [17, 16, 3] are used. HF, which is a general optimization-based approach, can be used in conjunction with or as an alternative to existing pre-training methods and is more widely applicable, since it relies on fewer assumptions about the specific structure of the network.

In this report we will first describe the basic HF approach, and then examine well-known general purpose performance-improving techniques as well as others

that are specific to HF (versus other Truncated-Newton type approaches) or to neural networks. We will also provide practical tips for creating efficient and correct implementations, and discuss the pitfalls which may arise when designing and using an HF-based approach in a particular application.

Table 20.1. A summary of the notation used. Note we will occasionally use some of these symbols to describe certain concepts that are local to a given sub-section. The subscripts “ k ” and “ $k-1$ ”, will often be dropped for compactness where they are implied from the context.

Notation	Description
$[x]_i$	The i -th entry of a vector x
$[A]_{i,j}$	The (i,j) -th entry a matrix A
$\mathbf{1}_m$	A vector of length m whose entries are 1
$\text{sq}(\cdot)$	The element-wise square of a vector or a matrix
$\text{vec}(A)$	The vectorization of a matrix A
f	The objective function
f_i	The objective function on case i
k	the current iteration of HF
θ_k	The parameter setting at the k -th HF iteration
n	The dimension of θ
δ_k	The variable being optimized by CG at the k -th HF iteration
M_{k-1}	A local quadratic approximation of f at θ_{k-1}
\hat{M}_{k-1}	A “damped” version of the above
B_{k-1}	The curvature matrix of M_{k-1}
\hat{B}_{k-1}	The curvature matrix of \hat{M}_{k-1}
$h', \nabla h$	The gradient of a scalar function h
$h'', \nabla^2 h$	The Hessian of a scalar function h
$L(\cdot)$	The loss function
ρ	The reduction ratio $\frac{f(\theta_k) - f(\theta_{k-1})}{M_{k-1}(\delta_k)}$
$F(\theta)$	A function that maps parameters to predictions on all training cases
D	A damping matrix
P	A preconditioning matrix
$K_i(A, r_0)$	The subspace $\text{span}\{r_0, Ar_0, \dots, A^{i-1}r_0\}$
ℓ	The number of layers of a feedforward net
z	The output of the network
m	The dimension of z
T	The number of time-steps of an RNN
λ	Strength constant for penalty damping terms
λ_j	j -th eigenvalue of curvature matrix
$\text{diag}(A)$	A vector consisting of the diagonal of the matrix A
$\text{diag}(v)$	A diagonal matrix A satisfying $[A]_{i,i} = [v]_i$

20.2 Feedforward Neural Networks

We now formalize feedforward neural networks (FNNs). Given an input x and setting of the parameters θ that determine weight matrices and the biases $(W_1, \dots, W_{\ell-1}, b_1, \dots, b_{\ell-1})$, the FNN computes its output y_ℓ by the following recurrence:

$$y_{i+1} = s_i(W_i y_i + b_i)$$

where $y_1 = x$. The vectors y_i are the activations of the neural network, and the activation functions $s_i(\cdot)$ are some nonlinear functions, typically sigmoid or a tanh functions applied coordinate-wise.

Given a matching target t , the FNN's training objective on a single case $f(\theta; (x, t))$ is given by

$$f(\theta; (x, t)) = L(y_\ell; t)$$

where $L(z; t)$ is a loss function which quantifies how bad z is at predicting the target t . Note that L may not compare z directly to t , but instead may transform it first into some prediction vector p .

Finally, the training error, which is the objective of interest for learning, is obtained by averaging the losses $f(\theta; (x, t))$ over a set S of input-output pairs (aka training cases):

$$f(\theta) = \frac{1}{|S|} \sum_{(x, t) \in S} f(\theta; (x, t))$$

Algorithm 20.1 . An algorithm for computing the gradient of a feedforward neural network

```

input:  $y_0$ ;  $\theta$  mapped to  $(W_1, \dots, W_{\ell-1}, b_1, \dots, b_{\ell-1})$ .
for all  $i$  from 1 to  $\ell - 1$  do
     $x_{i+1} \leftarrow W_i y_i + b_i$ 
     $y_{i+1} \leftarrow s_{i+1}(x_{i+1})$ 
end for
 $d y_\ell \leftarrow \partial L(y_\ell; t_\ell) / \partial y_\ell$  ( $t_\ell$  is the target)
for all  $i$  from  $\ell - 1$  downto 1 do
     $d x_{i+1} \leftarrow d y_{i+1} s'_{i+1}(x_{i+1})$ 
     $d W_i \leftarrow d x_{i+1} y_i^\top$ 
     $d b_i \leftarrow d x_{i+1}$ 
     $d y_i \leftarrow W_i^\top d x_{i+1}$ 
end for
output:  $\nabla f(\theta)$  as mapped from  $(d W_1, \dots, d W_{\ell-1}, d b_1, \dots, d b_{\ell-1})$ .
```

20.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are the time-series analog of feed-forward neural networks. RNNs model the mapping from an input sequence to an output sequence, and possess feedback connections in their hidden units that allow them to use information about past inputs to inform the predictions of future outputs. They may also be viewed as a special kind of feed-forward net with a “layer” for each time-step of the sequence. But unlike in a feed-forward network where each layer has its own parameters, the “layers” of an RNN share their parameters.

Their high-dimensional hidden state and nonlinear dynamics allow RNNs to learn very general and versatile representations, and to express highly complex sequential relationships. This representational power makes it possible, in principle, for RNNs to learn compact solutions for very difficult sequence modeling and labeling tasks. But despite their attractive qualities, RNNs did not enjoy widespread adoption after their initial discovery due to the perception that they were too difficult to properly train. The vanishing gradients problem [4, 18], where the derivative terms can exponentially decay to zero or explode during back-propagation through time is cited as one of the main reasons for this difficulty. In the case of decay, important back-propagated error signals from the output at future time-steps may decay nearly to zero by the time they have been back-propagated far enough to reach the relevant inputs. This makes the unmodified gradient a poor direction to follow if the RNN is to learn to exploit long-range input-output dependencies in the certain datasets.

Recent work by Martens & Sutskever [23] has demonstrated that HF is a viable method for optimizing RNNs on datasets that exhibit pathological long range dependencies that were believed difficult or impossible to learn with gradient descent. These problems were first examined by Hochreiter & Schmidhuber [19] where the proposed solution was to modify the RNN architecture with special memory units.

Basic RNNs are parameterized by three matrices and a special initial hidden state vector, so that $\theta \equiv (W_{xh}, W_{hh}, W_{zh}, h_0)$, where W_{xh} are the connections from the inputs to the hidden units, W_{hh} are the recurrent connections, and W_{zh} are the hidden-to-output connections. Given a sequence of vector-valued inputs $x = (x_1, \dots, x_T)$ and vector-valued target outputs $t = (t_1, \dots, t_T)$, the RNN computes a sequence of hidden states and predictions according to:

$$\begin{aligned} h_\tau &= s(W_{xh}x_\tau + W_{hh}h_{\tau-1}) \\ z_\tau &= W_{zh}h_\tau \end{aligned}$$

where h_0 is a special parameter vector of the initial state and $s(\cdot)$ is a nonlinear activation function (typically evaluated coordinate-wise).

The RNN learning objective for a single input-output pair of sequences (x, t) is given by:

$$f(\theta; (x, t)) = L(z; t) \equiv \sum_{\tau=1}^T L_\tau(z_\tau; t_\tau)$$

where L_τ is a loss function as in the previous section. As with FNNs, the objective function is obtained by averaging the loss over the training cases:

$$f(\theta) = \frac{1}{|S|} \sum_{(x,t) \in S} f(\theta; (x, t))$$

20.4 Hessian-Free Optimization Basics

We consider the setting of unconstrained minimization of a twice-differentiable objective function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ w.r.t. to a vector of real-valued parameters $\theta \in \mathbb{R}^n$. 2nd-order optimizers such as HF are derived from the classical Newton's method (a.k.a. the Newton-Raphson method), an approach based on the idea of iteratively optimizing a sequence of local quadratic models/approximations of the objective function in order to produce updates to θ . In the simplest situation, given the previous setting of the parameters θ_{k-1} , iteration k produces a new iterate θ_k by minimizing a local quadratic model $M_{k-1}(\delta)$ of the objective $f(\theta_{k-1} + \delta)$, which is formed using gradient and curvature information local to θ_{k-1} . More precisely, we define

$$M_{k-1}(\delta) = f(\theta_{k-1}) + \nabla f(\theta_{k-1})^\top \delta + \frac{1}{2} \delta^\top B_{k-1} \delta \quad (20.1)$$

where B_{k-1} is the “curvature matrix”, and is chosen to be the Hessian $H(\theta_{k-1})$ of f at θ_{k-1} in the case of standard Newton's method. The new iterate θ_k is computed as $\theta_{k-1} + \alpha_k \delta_k$ where δ_k^* is the minimizer of 20.1, and $\alpha_k \in [0, 1]$ is chosen typically chosen via a line-search, with a preference for $\alpha_k = 1$. A standard efficient method for performing this kind of line search will be briefly discussed in section 20.8.8. The multiplication of δ_k by α_k can be viewed as a crude instance of a general technique called “update damping”, which we will introduce next, and later discuss in depth in section 20.8.

When B_{k-1} is positive definite (PD), $M(\delta_k)$ will be bounded below and so its minimizer will exist, and will be given by $\delta_k^* = x_k - B_{k-1}^{-1} \nabla f(\theta_{k-1})$, which is the standard Newton step. Unfortunately, for many good choices of B_{k-1} , such as the Hessian at θ_{k-1} , even computing the entire $n \times n$ curvature matrix B_{k-1} , let alone inverting it/solving the system $B_{k-1} \delta_k = -f(\theta_{k-1})$ (at a cost of $O(n^3)$), will be impractical for all but very small neural networks.

The main idea in Truncated-Newton methods such as HF is to avoid this costly inversion by partially optimizing the quadratic function M using the linear conjugate gradient algorithm (CG) [15], and using the resulting approximate minimizer δ_k to update θ . CG is a specialized optimizer created specifically for quadratic objectives of the form $q(x) = \frac{1}{2} x^\top A x - b^\top x$ where $A \in \mathbb{R}^{n \times n}$ is positive semi-definite (PSD), and $b \in \mathbb{R}^n$. CG works by constructing the update from a sequence of vectors which have the property that they are “ A -conjugate” and can thus be optimized independently in sequence. To apply CG to eqn. 20.1 we

take $x = \delta$, $A = B_{k-1}$ and $b = \nabla f(\theta_{k-1})$, noting that the constant term $f(\theta_{k-1})$ can be ignored.

Note: From this point forward, we will abbreviate M_{k-1} with M and B_{k-1} with B when the subscript is implied by the context.

CG has the nice property that it only requires access to matrix-vectors products with the curvature matrix B (which can be computed *much* more efficiently than the entire matrix in many cases, as we will discuss in section 20.5), and it has a fixed-size storage overhead of a few n -dimensional vectors. Moreover, CG is a very powerful algorithm, which after i iterations, will find the provably optimal solution of any convex quadratic function $q(x)$ over the Krylov subspace $K_i(A, r_0) \equiv \text{span}\{r_0, Ar_0, A^2r_0, \dots, A^{i-1}r_0\}$, where $r_0 = Ax_0 - b$ and x_0 is the initial solution [32]. Any other gradient based method applied directly to a quadratic function like M , even a very powerful one like Nesterov's accelerated gradient descent [29], can also be shown to produce solutions which lie in the Krylov subspace, and thus will always be strictly outperformed by CG given the same number of iterations¹.

Fortunately, in addition to these strong optimality properties, CG works extremely well in practice and may often converge in a number of iterations $i \ll n$, depending on the structure of B . But even when it does not converge it tends to make very good partial progress.

The preconditioned CG algorithm is given in alg. 20.2. Note that Ap_i only needs to be computed once in each iteration of the main loop, and the quadratic objective $q(x_i)$ can be cheaply computed as $q(x_i) = \frac{1}{2}(r_i - b)^\top x_i$. Also note that any notation such as α_i or y_i should not be confused with the other uses of these symbols that occur elsewhere in this report. The preconditioning matrix P allows CG to operate within a transformed coordinate system and a good choice of P can substantially accelerate the method. This is possible despite the previously claimed optimality of CG because P induces a transformed Krylov subspace. Preconditioning, methods for implementing it, its role within HF, and its subtle interaction with other parts of the HF approach, will be discussed in section 20.11.

With practicality in mind, one can terminate CG according to various criteria, balancing the quality of the solution with the number of iterations required to obtain it (and hence number of matrix vector products – the main computational expense of the method). The approach taken by Martens [22] was to terminate CG based on a measure of relative progress optimizing M , computed as:

$$s_j = \frac{M(x_j) - M(x_{j-k})}{M(x_j)}$$

¹ This being said, it is possible to construct quadratic optimization problems where CG will perform essentially no better than accelerated gradient descent. Although it is also possible to construct ones where CG converge in only a few iteration while accelerated gradient descent will take much longer.

Algorithm 20.2 . Preconditioned conjugate gradient algorithm (PCG)

```

inputs:  $b, A, x_0, P$ 
 $r_0 \leftarrow Ax_0 - b$ 
 $y_0 \leftarrow \text{solution of } Py = r_0$ 
 $p_0 \leftarrow -y_0$ 
 $i \leftarrow 0$ 
while termination conditions do not apply do
     $\alpha_i \leftarrow \frac{r_i^\top y_i}{p_i^\top Ap_i}$ 
     $x_{i+1} \leftarrow x_i + \alpha_i p_i$ 
     $r_{i+1} \leftarrow r_i + \alpha_i Ap_i$ 
     $y_{i+1} \leftarrow \text{solution of } Py = r_{i+1}$ 
     $\beta_{i+1} \leftarrow \frac{r_{i+1}^\top y_{i+1}}{r_i^\top y_i}$ 
     $p_{i+1} \leftarrow -y_{i+1} + \beta_{i+1} p_i$ 
     $i \leftarrow i + 1$ 
end while
output:  $x_i$ 

```

where x_j is the j -th iterate of CG and k is the size of the window over which the average is computed, which should be increased with j . A reasonable choice that works well in practice is $k = \max(10, j/10)$. CG can be terminated at iteration j when

$$s_j < 0.0001 \quad (20.2)$$

or some other such constant. Depending on the situation it may make more sense to truncate earlier to find a more economical trade-off between relative progress and computation.

However, deciding when to terminate CG turns out to be a much more complex and subtle issue than implied by the above discussion, and in section 20.8.7 of this paper we will discuss additional reasons to terminate CG that have nothing directly to do with the value of M . In particular, earlier truncations may sometimes have a beneficial damping effect, producing updates that give a better improvement in f than would be obtained by a fully converged solution (or equivalently, one produced by exact inversion of the curvature matrix).

When f is non-convex (as it is with neural networks), B will sometimes be indefinite, and so the minimizer of M may not exist. In particular, progressively larger δ 's may produce arbitrarily low values of M , leading to nonsensical or undefined updates. This issue can be viewed as an extreme example of the general problem that the quadratic model M is only a crude local approximation to f , and so its minimizer (assuming it even exists), might lie in a region of \mathbb{R}^n where the approximation breaks down, sometimes catastrophically. While the aforementioned line-search can remedy this problem to some degree, this is a general problem with 2nd-order optimization that must be carefully addressed. Ways to do this are sometimes called “damping methods”, a term which we shall

use here, and include such techniques as restriction of the optimization over $M(\cdot)$ to a “trust-region”, and the augmentation of M by penalty terms which are designed to encourage the minimizer of M to be somewhere in \mathbb{R}^n where M remains a good approximation to f . Such approaches must be used with care, since restricting/penalizing the optimization of M too much will result in very reliable updates which are nonetheless useless due to being too “small”. In section 20.8 we will discuss various general damping methods in 2nd-order optimization, and some which are more specific to HF.

While the damping methods such as those mentioned above allow one to optimize M even when B is indefinite, there is another way to deal with the indefiniteness problem directly. The classical Gauss–Newton algorithm for non-linear least squares uses a positive semi-definite curvature matrix which is viewed as an approximation to the Hessian, and Schraudolph [11] was able to generalize this idea to cover a much larger class of objective functions that include most neural network training objectives. This “generalized Gauss–Newton matrix” (GGN), is also guaranteed to be positive semi-definite, and tends to work much better than the Hessian in practice as a curvature matrix when optimizing non-convex objectives. While using the GGN matrix will not eliminate the need for damping, Martens [22] nonetheless found that it was easier to use than the Hessian, producing better updates and requiring less damping. The computational and theoretical aspects of the GGN matrix and its use within HF will be discussed in detail in section 20.6.

Objective functions $f(\theta)$ that appear in machine learning are almost always defined as arithmetic averages over a training set S , and thus so can the gradient and the curvature-matrix vector products:

$$\begin{aligned} f(\theta) &= \frac{1}{|S|} \sum_{(x,t) \in S} f(\theta; (x,t)) \\ \nabla f(\theta) &= \frac{1}{|S|} \sum_{(x,t) \in S} \nabla f(\theta; (x,t)) \\ B(\theta)v &= \frac{1}{|S|} \sum_{(x,t) \in S} B(\theta; (x,t))v \end{aligned}$$

where $f(\theta; (x,t))$ is the objective and $B(\theta; (x,t))$ the curvature matrix associated with the training pair (x,t) .

In order to make HF practical for large datasets it is necessary to estimate the gradient and curvature matrix-vector products using subsets of the training data, called “minibatches.” And while it may seem natural to compute the matrix-vector products required by CG using a newly sampled minibatch at each iteration of alg. 20.2, CG is unfortunately not designed to handle this kind of “stochasticity” and its theory depends very much on a stable definition of B for concepts like B -conjugacy to even make sense. And in practice, we have found that such an approach does not seem to work very well, and results in CG itself diverging in some cases. The solution advocated by Martens [22] and independently by Byrd et al. [8] is to fix the minibatch used to define B for the entire

run of CG. Minibatches and the practical issues which arise when using them will be discussed in more depth in section 20.12.

Algorithm 20.3. High-level outline for the basic Hessian-free approach. Various details have been purposefully left unstated, and some aspects will be subject to change throughout this report.

```

inputs:  $\theta_0, \lambda$ 
Set  $\delta_0 \leftarrow \mathbf{0}$ 
 $k \leftarrow 1$ 
while solution is not satisfactory do
    Select a set of points  $S$  for the gradient ..... sec. 20.12
     $b \leftarrow -\nabla f(\theta_{k-1})$  on  $S$  ..... sec. 20.2
    Select a set of points  $S'$  for the curvature ..... sec. 20.12
    Compute a preconditioner  $P$  at  $\theta_k$  ..... sec. 20.11
    Compute a damping matrix  $D_k$  ..... sec. 20.8
    Define  $A(v) \equiv G(\theta_{k-1})v + \lambda D_k v$  on  $S'$  ..... sec. 20.6
    Choose a decay constant  $\zeta \in [0, 1]$  ..... sec. 20.10
     $\delta_k \leftarrow \text{PCG}(b, A, \zeta \delta_{k-1}, P)$  ..... alg. 20.2
    Update  $\lambda$  with the Levenberg-Marquardt method ..... sec. 20.8.5
    Choose/compute a step-size  $\alpha$  ..... sec. 20.8.8
     $\theta_k \leftarrow \theta_{k-1} + \alpha \delta_k$ 
     $k \leftarrow k + 1$ 
end while

```

20.5 Exact Multiplication by the Hessian

To use the Hessian H of f as the curvature matrix B within HF we need an algorithm to efficiently compute matrix-vector products with arbitrary vectors $v \in \mathbb{R}^n$. Noting that the Hessian is the Jacobian of the gradient, we have that the Hessian-vector product $H(\theta)v$ is the directional derivative of the gradient $\nabla f(\theta)$ in the direction v , and so by the definition of directions derivatives,

$$H(\theta)v = \lim_{\varepsilon \rightarrow 0} \frac{\nabla f(\theta + \varepsilon v) - \nabla f(\theta)}{\varepsilon}$$

This equation implies a finite-differences algorithm for computing Hv at the cost of a single extra gradient evaluation. But in practice, and in particular when dealing with highly nonlinear functions like neural network training objectives, methods that use finite differences suffer from significant numerical issues, which can make them generally undesirable and perhaps even unusable in some situations.

Fortunately, there is a method for computing the sought-after directional derivative in a numerically stable way that does not resort to finite differences. In the optimization theory literature, the method is known as “forward-differentiation” [34, 30], although we follow the exposition of Pearlmutter [31], who rediscovered it for neural networks and other related models. The idea is to make repeated use of the chain rule, much like in the backpropagation algorithm, to differentiate the value of every node in the computational graph of the gradient. We formalize this notion by introducing the R_v -notation. Let $R_v X$ denote the directional derivative of X in direction v :

$$R_v X = \lim_{\varepsilon \rightarrow 0} \frac{X(\theta + \varepsilon v) - X(\theta)}{\varepsilon} = \frac{\partial X}{\partial \theta} v \quad (20.3)$$

Being a derivative, the $R_v(\cdot)$ operator obeys the usual rules of differentiation:

$$R_v(X + Y) = R_v X + R_v Y \quad \text{linearity} \quad (20.4)$$

$$R_v(XY) = (R_v X)Y + X R_v Y \quad \text{product rule} \quad (20.5)$$

$$R_v(h(X)) = (R_v X)h'(X) \quad \text{chain rule} \quad (20.6)$$

where h' denotes the Jacobian of h . From this point on we will abbreviate R_v as simply “R” to keep the notation compact.

Noting that $Hv = R\{\nabla f(\theta)\}$, computing the Hessian-vector product amounts to computing $R\{\nabla f(\theta)\}$ by applying these rules recursively to the computational graph for $\nabla f(\theta)$, in a way analogous to back-propagation (but operating forward instead of backwards).

To make this precise, we will formalize the notion of a computational graph for an arbitrary vector-valued function $h(\theta)$, which can be thought of as a special kind of graph which implements the computation of a given function by breaking it down as a collection of simpler operations, represented by M nodes, with various input-output dependencies between the nodes indicated by directed edges. The nodes of the computational graph are vector valued, and each node i computes an arbitrary differentiable functions $a_i = \gamma_i(z_i)$ of their input z_i . Each input vector z_i is formally the concatenation the output of each of its parent nodes $a_j \in P_i$. The input θ is distributed over a set of input nodes $\mathcal{I} \subset \{1, \dots, M\}$ and the outputs are computed at output nodes $\mathcal{O} \subset \{1, \dots, M\}$.

In summary, the function $h(\theta)$ is computed according to the following procedure:

1. For each $i \in \mathcal{I}$ set a_i according to entries of θ
2. For i from 1 to M such that $i \notin \mathcal{I}$:

$$\begin{aligned} z_i &= \text{concat}_{j \in P_i} a_j \\ a_i &= \gamma_i(z_i) \end{aligned}$$

3. Output $h(\theta)$ according to the values in $\{a_i\}_{i \in \mathcal{O}}$

where P_i is the set of parents of node i .

The advantage of the computational graph formalism is that it allows the application of the R-operator to be performed in a fool-proof and mechanical way that can be automated. In particular, our function $R(h(\theta))$ can be computed as follows:

1. For each $i \in \mathcal{I}$ set Ra_i according to entries of v (which correspond to entries of θ)
2. For i from 1 to M such that $i \notin \mathcal{I}$:

$$Rz_i = \text{concat}_{j \in P_i} Ra_j \quad (20.7)$$

$$Ra_i = \gamma'_i(z_i) Rz_i \quad (20.8)$$

3. Set output $R(h(\theta))$ according to the values in $\{Rz_i\}_{i \in \mathcal{O}}$

where $\gamma'_i(z_i)$ is the Jacobian of γ_i .

In general, computing $\gamma'_i(z_i)$ (or more simply multiplying it by a vector) is simple² and is of comparable cost to computing $\gamma_i(z_i)$, which makes computing the Hessian-vector product using this method comparable to the cost of the gradient. Notice however that we need to have each z_i available in order to evaluate $\gamma'_i(z_i)$ in general, so all of the z_i 's (or equivalently all of the a_i 's) must either be computed in tandem with the Ra_i 's and Rz_i 's (making the cost of the Hessian-vector product roughly comparable to the cost of two evaluations of the gradient), or be precomputed and cached (e.g. during the initial computation of the gradient).

When using an iterative algorithm like CG that requires multiple Hessian-vector products for the same θ , caching can save considerable computation, but as discussed in section 20.7 may require considerable extra storage when computing matrix-vector products over large minibatches.

Algorithm 20.5 gives the pseudo-code for computing the Hessian-vector product associated with the feedforward neural network defined in section 20.2. The parameter vector θ defines the weight matrices and the biases $(W_1, \dots, W_{\ell-1}, b_1, \dots, b_{\ell-1})$ and v maps analogously to $(RW_1, \dots, RW_{\ell-1}, Rb_1, \dots, Rb_{\ell-1})$. This algorithm was derived by applying the rules 20.4–20.6 to each line of alg. 20.2, where various required quantities such as y_i are assumed to be available either because they are cached, or by running the corresponding lines of alg. 20.2 in tandem.

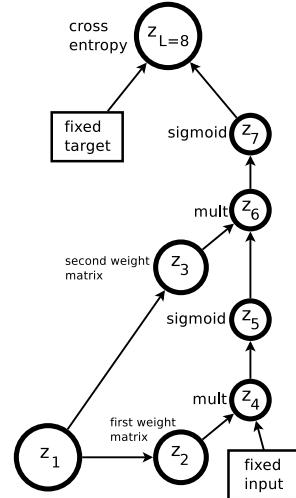


Fig. 20.1. An example of a computational graph of the loss of a neural network objective. The weights are considered the inputs here.

20.6 The Generalized Gauss-Newton Matrix

The indefiniteness of the Hessian is problematic for 2nd-order optimization of non-convex functions because an indefinite curvature matrix B may result in a

² If this is not the case then node i should be split into several simpler operations.

Algorithm 20.4. An algorithm for computing $H(\theta)v$ in feedforward neural networks.

input: v mapped to $(RW_1, \dots, RW_{\ell-1}, Rb_1, \dots, Rb_{\ell-1})$

$$Ry_0 \leftarrow 0 \quad (\text{since } y_0 \text{ is not a function of the parameters})$$

for all i **from** 1 **to** $\ell - 1$ **do**

$$Rx_{i+1} \leftarrow RW_i y_i + W_i Ry_i + Rb_i \quad (\text{product rule})$$

$$Ry_{i+1} \leftarrow Rx_{i+1} s'_{i+1}(x_{i+1}) \quad (\text{chain rule})$$

end for

$$Rdy_\ell \leftarrow R \left(\frac{\partial L(y_\ell; t_\ell)}{\partial y_\ell} \right) = \frac{\partial \{ \partial L(y_\ell; t_\ell) / \partial y_\ell \}}{\partial y_\ell} Ry_\ell = \frac{\partial^2 L(y_\ell; t_\ell)}{\partial y_\ell^2} Ry_\ell$$

for all i **from** $\ell - 1$ **downto** 1 **do**

$$Rdx_{i+1} \leftarrow Rdy_{i+1} s'_{i+1}(x_{i+1}) + dy_{i+1} R \{ s'_{i+1}(x_{i+1}) \} \quad (\text{product rule})$$

$$= dy_{i+1} s''_{i+1}(x_{i+1}) Rx_{i+1} \quad (\text{chain rule})$$

$$RdW_i \leftarrow Rdx_{i+1} y_i^\top + dx_{i+1} Ry_i^\top \quad (\text{product rule})$$

$$Rdb_i \leftarrow Rdy_i$$

$$Rdy_i \leftarrow RW_i^\top dx_{i+1} + W_i^\top Rdx_{i+1} \quad (\text{product rule})$$

end for

output: $H(\theta)v$ as mapped from $(RdW_1, \dots, RdW_{\ell-1}, Rdb_1, \dots, Rdb_{\ell-1})$.

quadratic M which is not bounded below and thus does not have a minimizer to use as the update δ . This problem can be addressed in a multitude of ways. For example, imposing a trust-region (sec. 20.8.6) will constrain the optimization, or a penalty-based damping method (sec. 20.8.1) will effectively add a positive semi-definite (PSD) contribution to B which may render it positive definite (PD). Another solution specific to truncated Newton methods is to truncate CG as soon as it generates a conjugate direction with negative curvature (i.e., when $p_i^\top A p_i < 0$ in alg. 20.2), a solution which may be useful in some applications but which we have not found to be particularly effective for neural network training.

Based on our experience, the best solution to the indefiniteness problem is to instead use the generalized Gauss-Newton (GGN) matrix proposed by Schraudolph [11], which is a provably positive semidefinite curvature matrix that can be viewed as an approximation to the Hessian. We will denote this matrix as G .

The generalized Gauss-Newton matrix can be derived in at least two ways, and both require that the objective $f(\theta)$ be expressed as the composition of two functions as $f(\theta) = L(F(\theta))$ where L is convex. In a neural network setting, F maps the parameters θ to a m -dimensional vector of the neural network's outputs $z \equiv F(\theta)$, and $L(z)$ is a convex "loss function" which typically measures the difference between the network's outputs (which may be further transformed within L to produce "predictions" p) and the targets. For RNNs, z will be a vector of the outputs from *all* the time-steps and L computes the sum over losses at each one of them.

One way to view the GGN matrix is as an approximation of H where we drop certain terms that involve the 2nd-derivatives of F . Applying the chain rule to compute the Hessian of f (at θ_{k-1}), we get:

$$\begin{aligned} f &= L(F(\theta)) \\ \nabla f(\theta) &= J^\top \nabla L \\ f''(\theta) &= J^\top L'' J + \sum_{i=1}^m [\nabla L]_i ([F]_i)'' \end{aligned}$$

where J denotes the Jacobian of F , ∇L is the gradient of $L(z)$ w.r.t. z , and all 1st and 2nd derivatives are evaluated at θ_{k-1} . The first term $J^\top L'' J$ is a positive definite matrix whenever $L(z)$ is convex in z , and is defined as the GGN matrix. Note that in the special case where $L(z) = \frac{1}{2}\|z\|^2$ (so that $L'' = I$) we recover the standard Gauss-Newton matrix usually seen in the context of non-linear least squares optimization and the Levenberg-Marquardt algorithm [25].

Martens and Sutskever [23] showed that the GGN matrix can also be viewed as the Hessian of a particular approximation of f constructed by replacing F with its 1st-order approximation. Consider a local convex approximation \hat{f} to f at θ_{k-1} that is obtained by taking the first-order approximation $F(\theta) \approx F(\theta_{k-1}) + J\delta$ (where $\delta = \theta - \theta_{k-1}$):

$$\hat{f}(\delta) = L(F(\theta_{k-1}) + J\delta) \quad (20.9)$$

The approximation \hat{f} is convex because it is a composition of a convex function and an affine function. It is easy to see that \hat{f} and f have the same derivative when $\delta = 0$, because

$$\nabla \hat{f} = J^\top \nabla L = J^\top \nabla L$$

which is precisely the derivative of f at θ_{k-1} . And the Hessian of \hat{f} at $\delta = 0$ is precisely the GGN matrix:

$$\hat{f}'' = J^\top L'' J = G$$

Note that it may be possible to represent a function f with multiple distinct compositions of the form $L(F(\theta))$, and each of these will give rise to a slightly different GGN matrix. For neural networks, a natural choice for the output vector z is often just to identify it as the output of the final layer (i.e., y_ℓ), however this may not always result in a convex L . As a rule of thumb, it is best to define L and F in way that L performs “as much of the computation of f as possible” (but this is a problematic concept due to the existence of multiple distinct sequences of operations for computing f). For the case of neural networks with a softmax output layer and cross-entropy error, it is best to define L so that it performs both the softmax *and* then the cross-entropy, while F computes only the inputs to the soft-max function. This is also the recommendation made by Schraudolph [11]. A possible reason that this choice works best is due to the fact that F

is being replaced with its first-order approximation whose range is unbounded. Hence the GGN matrix makes sense only when L 's input domain is \mathbb{R}^m (as opposed to $[0, 1]^m$ for the cross-entropy error), since this is the range of the 1st-order approximation of F .

20.6.1 Multiplying by the Gauss-Newton Matrix

For the GGN matrix to be useful in the context of HF, we need an efficient algorithm for computing the Gv products. Methods for multiplying by the classical Gauss-Newton matrix are well-known in the optimization literature [30], and these methods were generalized by Schraudolph [11] for the GGN matrix, using an approach which we will now describe.

We know from the previous section that the GGN matrix can be expressed as the product of three matrices: $Gv = J^\top L'' J v$. Thus multiplication of a vector v by the GGN matrix amounts to the sequential multiplication of that vector by these 3 matrices. First, the product Jv is a Jacobian times vector and is therefore precisely equal to the directional derivative $R_v\{F(\theta)\}$, and thus can be efficiently computed with the R-method as in section 20.5. Next, given that the loss function L is usually simple, multiplication of Jv by L'' is also simple (sec. 20.6.2). Finally, we multiply the vector $L''Jv$ by the matrix J^\top using the backpropagation algorithm. Note that the backpropagation algorithm takes the derivatives w.r.t. the predictions (∇L) as inputs, and returns the derivative w.r.t. the parameters, namely $J^\top \nabla L$, but we can replace ∇L with any vector we want.

Algorithm 20.5 . An algorithm for computing Gv of a feedforward neural network.

```

input:  $RW_1, \dots, RW_{\ell-1}, Rb_1, \dots, Rb_{\ell-1}$ .
 $Ry_0 \leftarrow 0$  ( $y_0$  is not a function of the parameters)
for all  $i$  from 1 to  $\ell - 1$  do
     $Rx_{i+1} \leftarrow RW_i y_i + W_i Ry_i + Rb_i$  (product rule)
     $Ry_{i+1} \leftarrow Rx_{i+1} s'_{i+1}(x_{i+1})$ 
end for
 $Rdy_\ell \leftarrow \frac{\partial^2 L(y_\ell; t_\ell)}{\partial y_\ell^2} Ry_\ell$ 
for all  $i$  from  $\ell - 1$  downto 1 do
     $Rdx_{i+1} \leftarrow Rdy_{i+1} s'_{i+1}(x_{i+1})$ 
     $RdW_i \leftarrow Rdx_{i+1} y_i^\top$ 
     $Rdb_i \leftarrow Rdx_{i+1}$ 
     $Rdy_i \leftarrow RW_i^\top dx_{i+1}$ 
end for
output:  $(RdW_1, \dots, RdW_{\ell-1}, Rb_1, \dots, Rb_{\ell-1})$ .

```

As observed by Martens and Sutskever [23], the second interpretation of the GGN matrix given in the previous section immediately implies an alternative

method for computing Gv products. In particular, we can use the R method from sec. 20.5 to efficiently multiply by the Hessian of \hat{f} , given a computational graph for $\nabla \hat{f}$. While doing this would require one to replace the part of the forward pass corresponding to F with a multiplication by the analogous Jacobian evaluated at θ_{k-1} (which can be done using the R operator method applied to f), a simpler approach is just to modify the algorithm for computing ∇f so that all derivative terms involving intermediate quantities in the back-propagation through F are treated as “constants”, which while they are computed from θ_{k-1} , are formally independent of θ . This version will only compute \hat{f} properly for $\theta = \theta_{k-1}$, but this is fine for our purposes since this is the point at which we wish to evaluate the GGN matrix.

Algorithm 20.5 multiplies by the GGN matrix for the special case of a feed-forward neural network and is derived using this second technique.

20.6.2 Typical Losses

In this section we present a number of typical loss functions and their Hessians (table 20.2). The function $p(z)$ computes the predictions p from the network outputs z . These losses are convex and it is easy to multiply by their Hessians

Table 20.2. Typical losses with their derivatives and Hessians. The loss L and the nonlinearity $p(z)$ are “matching”, which means that the Hessian is independent of the target t and is PSD.

Name	$L(z; t)$	$\nabla L(z; t)$	$L''(z; t)$	p
Squared error	$\frac{1}{2} \ p - t\ ^2$	$-(p - t)$	I	$p = z$
Cross-entropy error	$-t \log p - (1-t) \log(1-p)$	$-(p - t)$	$\text{diag}(p(1-p))$	$p = \text{Sigmoid}(z)$
Cross-entropy error (multi-dim)	$-\sum_i [t]_i \log[p]_i$	$-(p - t)$	$\text{diag}(p) - pp^\top$	$p = \text{Softmax}(z)$

without explicitly forming the matrix, since they are each either diagonal or the sum of a diagonal and a rank-1 term.

When applying this formulation to FNNs, note that because it formally includes the computation of the predictions p from the network outputs z (assumed to lie anywhere in \mathbb{R}^m) in the loss function itself (instead of in the activation function s_ℓ at the output layer), s_ℓ should be set to the identity function.

20.6.3 Dealing with Non-convex Losses

We may sometimes want to have a non-convex loss function. The generalized Gauss-Newton matrix construction will not produce a positive definite matrix in this case because the GGN matrix $J^\top L'' J$ will usually be PSD only when L'' is, which is a problem that can be addressed in one of several ways. For example, if our loss is $L(y; t) = \|\tanh(y) - t\|^2/2$, which is non-convex, we could formally treat the \tanh nonlinearity as being part of F (replacing F with $\tanh \circ F$), and redefine the loss L as $\|y - t\|^2/2$. Another trick which may work would be to approximate

the loss-Hessian L'' with a positive definite matrix, which could be done, say, by adding a scaled multiple of the diagonal to L'' , or by taking the eigen-decomposition of L'' and discarding the eigenvectors that have negative eigenvalues.

20.7 Implementation Details

20.7.1 Efficiency via Parallelism

A good implementation of HF can make fruitful use of parallelization in two ways.

First, it can benefit from model parallelism, which is the ability to perform the input and output computations associated with each neuron in a given layer parallel. Although model parallelism accelerates any optimization algorithm that is applied to neural networks, current hardware is incapable of fully taking advantage of it, mostly because weights are stored in a centralized memory with very limited bandwidth.

Second, an implementation of HF can benefit from data parallelism, where the computation of the gradient or curvature matrix vector products is performed independently and in parallel across the training cases in the current minibatch. Data parallelism is much easier to exploit in current hardware because it requires minimal communication, in stark contrast to model parallelism, which requires frequent and rapid communication of unit activations. The potential speedup offered by data parallelism is limited by the gains that can be derived from using larger minibatches to compute updates in HF, as well as the sheer amount of parallel computing power available.

HF tends to benefit from using relatively large minibatches, especially compared to first-order methods like stochastic gradient descent, and so exploiting data parallelism may bring significant reductions in computation time. Nonetheless, there is a point of diminishing returns after which making the minibatch larger provides limited or no benefit in terms of the quality of the update proposals (as measured by how much they reduce f).

Data parallelism is typically implemented using vectorization, which is a way of specifying a single computational process that is independently performed on every element of a vector. Since most implementations that use vectorization (e.g. GPU code) become more efficient per case as the size of the minibatch increases, there is a distinct benefit to using larger minibatches (up until the aforementioned point of diminishing returns, or the point where the implementations parallel computing resources are fully utilized).

Most of the computation performed by HF consists of computing the GGN vector products and fortunately it is possible to obtain a 50% speedup over a naive implementation of the GGN vector products using activity caching. Recall that a multiplication by the GGN matrix consists of a multiplication by J which is followed by a multiplication by J^\top , both of which require the neural network's unit activations (the y_i 's in FNNs or y_τ 's in RNNs). However, given that the network's activations are a function of only θ , and that CG multiplies different

vectors by the same GGN matrix (so its setting of θ is fixed), it is possible to cache the network’s activations y_i and to reuse them for all the GGN-vector products made during an entire run of CG.

When a model is very large, which is the case for a large RNN with a large number of time-steps T , the unit activations produced by even a modestly-sized minibatch may become too numerous to fit in memory. This is especially a problem for GPU implementations since GPUs typically have much less memory available than CPUs. This has two undesirable consequences. First, activity caching becomes impossible, and second, it necessitates the splitting of a large minibatch into many smaller “computational minibatches” (the results from which will be summed up after each has been processed), which can greatly reduce the cost-effectiveness of vectorization.

The problem can be addressed in at least 2 ways. One is to cache the activations in a larger but slower memory storage (e.g. the CPU memory), and to retrieve them as needed. This is often faster than the use of many smaller minibatches.

Another way involves reducing the storage requirements at the cost of performing re-computation of some of the states. In particular, we store the hidden states at every multiple of \sqrt{T} time-steps (thus reducing the storage requirement by a factor of \sqrt{T}), and recompute sequences of \sqrt{T} between these “check-points” as they become needed, discarding them immediately after use. Due to the way the forward and backwards passes involved in computing gradients and matrix-vector products go through the time-steps in linear order, the state at each time-step needs to be recomputed at most once in the case of the gradient, and twice in the case of the matrix-vector product.

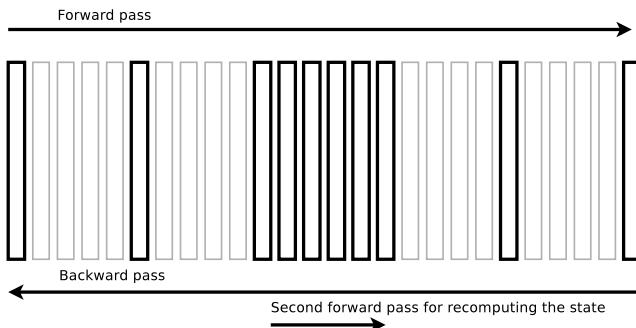


Fig. 20.2. An illustration of the method for conserving the memory of the RNN. Each column represents a hidden state of an RNN, and only the highlighted columns reside in memory at any given time.

20.7.2 Verifying the Correctness of G Products

A well-known pitfall for neural networks practitioners is an incorrect implementation for computing the gradient, which is hard to diagnose without having a correct implementation to compare against. The usual procedure is to

re-implement the gradient computation using finite differences and verify that the two implementations agree, up to some reasonable precision.

To verify the correctness of an implementation of Truncated Newton optimizer like HF, as we must also verify the correctness of the curvature-matrix vector products. When $B = H$, there are well-known black-box finite differentiation implementations available which can be used for this purpose. Thus we will concentrate on how to verify the correctness of the Gv products.

Given that $G = J^\top L'' J$ so $Gv = J^\top (L''(Jv))$, computing the G -vector products via finite differences reduces to doing this for Jw , $L''w$ and $J^\top w$ for arbitrary vectors w of appropriate dimension (not necessarily the same for each).

1. For Jw we compute $(F(\theta + \varepsilon w) - F(\theta - \varepsilon w))/(2\varepsilon)$ for a small ε .
2. For $L''w$ we can simply approximate L'' using one of the aforementioned finite-differences implementations that are available for approximating Hessians.
3. For $J^\top w$ we exploit $[J]_{j,i}^\top = [Je_j]_i$ where e_j is the j -th standard basis vector, and use the method in point 1 to approximate Je_j

To be especially thorough, one should probably test that Ge_j agrees with its finite differences version for each j , effectively constructing the whole matrix G .

For this kind of finite-differences numerical differentiation to be practical it is important to use small toy versions of the target networks, with much fewer units in each layer, and smaller values for the depth ℓ or sequence length T (such as 4). In most situations, a good value of ε is often around 10^{-4} , and it is possible to achieve a relative estimation error from the finite differences approximation of around 10^{-6} , assuming a high-precision floating point implementation (i.e. float64 rather than float32).

It is also important to use random θ 's that are of a reasonable scale. Parameters that are too small will fail to engage the nonlinearities, leaving them in their “linear regions” and making them behave like linear functions, while parameters that are too large may cause “saturation” of the units of the network, making them behave like step-functions (the opposite extreme). In either case, a proposed implementation of some exact derivative computation could match the finite differences versions to high precision despite being incorrect, as the local derivatives of the activation functions may be constant or even zero.

Another option to consider when implementing complex gradient/matrix computations is to use an automatic differentiation system package such as Theano [5]. This approach has the advantage of being mostly fool proof, at the possible cost of customization and efficiency (e.g. it may be hard to cache the activities using previously discussed techniques).

20.8 Damping

While unmodified Newton's method may work well for certain objectives, it tends to do very poorly if applied directly to highly nonlinear objective functions, such as those which arise when training neural networks. The reason for this failure

has to do with the fact that the minimizer δ^* of the quadratic approximation M may be very large and “aggressive” in the early and the intermediate stages of the optimization, in the sense that it is often located far beyond the region where the quadratic approximation is reasonably trust-worthy.

The convergence theory for non-convex smooth optimization problems (which include neural net training objectives) describes what happens only when the optimization process gets close enough to a local minimum so that the steps taken are small compared to the change in curvature (e.g. as measured by the Lipschitz constant of the Hessian). In such a situation, the quadratic model will always be highly accurate at δ^* , and so one can fully optimize M and generate a sequence of updates which will converge “quadratically” to the local minimum of f . And for some very simply optimization problems which can arise in practice it may even be possible to apply unmodified Newton’s method without any trouble, ignoring the theoretical requirement of proximity to a local minimum. However, for neural network training objectives, and in particular deep feed-forward networks and RNNs, the necessity of these proximity assumptions quickly becomes clear after basic experiments, where such naive 2nd-order optimization tends to diverge rapidly from most sensible random initializations of θ .

The solution that is sometimes advocated for this problem is to use a more stable and reliable method, like gradient-descent for the beginning of optimization, and then switch later to 2nd-order methods for “fine convergence”. Optimization theory guarantees that as long as the learning rate constant is sufficiently small, gradient descent will converge from any starting point. But precise convergence is often not necessary, or even undesirable (due to issues of overfitting). Instead, if we believe that making use of curvature information can be beneficial in constructing updates long before the “fine convergence” regime described by local convergence theory sets in, it may be worthwhile to consider how to make more careful and conservative use of curvature information in order to construct large but still sensible update proposals, instead of defaulting to 1st-order ones out of necessity.

“Damping”, a term used mostly in the engineering literature, and one which we will adopt here, refers to methods which modify M or constrain the optimization over it in order to make it more likely that the resulting update δ will lie in a region where M remains a reasonable approximation to f and hence yield a substantial reduction. The key difficulty with damping methods is that if they are overused or improperly calibrated, the resulting updates will be “reliable” but also be too small and insignificant (as measured by the reduction in f).

An effective damping method is of critical importance to the performance of a 2nd-order method, and obtaining the best results will likely require the use of a variety of different techniques, whose usefulness depends both on the particular application and the underlying 2nd-order method. In this section we will discuss some generic damping methods that can be used in 2nd-order optimizers and how to apply them in HF (either separately or in some combination) along with methods which are specific to neural networks and HF.

One thing to keep in mind when reading this section is that while the immediate goal of damping methods is to increase the quality of the parameter update produced by optimizing M (as measured by the immediate improvement in the objective f), damping methods can and will have an important influence on the global optimization performance of 2nd-order optimizers when applied to multimodal objectives functions, in ways that are sometimes difficult to predict or explain, and will be problem dependent. For example, we have observed empirically that on difficult neural-net training objectives, damping schemes which tend to produce updates that give the best reductions in f in the short term, may not always yield the best global optimization performance in the long term.

20.8.1 Tikhonov Damping

“Tikhonov regularization” or Tikhonov damping³ is arguably the most well-known damping method, and works by penalizing the squared magnitude $\|\delta\|^2$ of the update δ by introducing an additional quadratic penalty term into the quadratic model M . Thus, instead of minimizing M , we minimize a “damped” quadratic

$$\hat{M}(\delta) \equiv M(\delta) + \frac{\lambda}{2} \delta^\top \delta = f(\theta) + \nabla f(\theta)^\top \delta + \frac{1}{2} \delta^\top \hat{B} \delta$$

where $\hat{B} = B + \lambda I$, where $\lambda \geq 0$ is a scalar parameter determining the “strength” of the damping. Computing the matrix-vector product with \hat{B} is straightforward since $\hat{B}v = (B + \lambda I)v = Bv + \lambda v$.

As $\lambda \rightarrow \infty$, the damped curvature matrix \hat{B} tends to a multiple of the identity and the minimizer δ^* has the property that $\delta^* \rightarrow \nabla f(\theta)/\lambda$, meaning the overall optimization process reduces to gradient descent with a particular learning rate.

To better understand the effect of the Tikhonov damping, note that the addition of a scalar multiple of the identity matrix to B has the effect of increasing each of the eigenvalues by precisely λ . This can be seen by noting that if $B = V\Sigma V^\top$ where $V = [v_1 | v_2 | \dots | v_n]$ are eigenvectors of B (which are orthonormal since B is symmetric), and $\Sigma \equiv \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ the diagonal matrix of eigenvalues, then $\hat{B} = V\Sigma V^\top + \lambda I = V\Sigma V^\top + \lambda VV^\top = V(\Sigma + \lambda I)V^\top$. Thus the curvature associated with each eigenvector v_j in the damped matrix is given by $v_j^\top \hat{B} v_j = \lambda_j + \lambda$.

This modulation of the curvature has profound effect on the inverse of \hat{B} since $\hat{B}^{-1} = V^\top (\Sigma + \lambda I)^{-1} V$, where $(\Sigma + \lambda I)^{-1} = \text{diag}((\lambda_1 + \lambda)^{-1}, (\lambda_2 + \lambda)^{-1}, \dots, (\lambda_n + \lambda)^{-1})$ and this will be particularly significant for λ_j ’s that are small compared to λ , since $(\lambda_j + \lambda)^{-1}$ will generally be much smaller than λ_j^{-1} in such cases.

³ A name which we will use to avoid confusion with the other meaning of term regularization in the learning context.

The effect on the minimizer δ^* of \hat{M} can be seen by noting that

$$\delta^* = - \sum_j \frac{v_j^\top \nabla f(\theta_{k-1})}{\lambda_j + \lambda} v_j$$

so the distance $v_j^\top \delta^*$ that δ^* moves θ in the direction v_j will be effectively multiplied by $\frac{\lambda_j}{\lambda_j + \lambda}$. Thus, Tikhonov damping should be appropriate when the quadratic model is most untrustworthy along directions of very low-curvature (along which δ^* will tend to travel very far in the absence of damping).

Picking a good value of λ is critical to the success of a Tikhonov damping approach. Too high, and the update will resemble gradient descent with a very small learning rate and most of the power of 2nd-order optimization will be lost, with the low-curvature directions particularly affected. Conversely, if λ is too small, the quadratic model \hat{M} will be too aggressively optimized by CG, resulting in a very large parameter update (particular in directions of low curvature) which may cause an increase in f instead of a decrease. Unfortunately, determining a good value of λ is a nontrivial problem, which is sensitive to the overall scale of the objective function (i.e. using $\lambda = 1$ for f gives the same update as $\lambda = 2$ would for $2f$), and other more subtle properties of f , many of which will vary over the parameter space. It is in fact very rarely the case that a single value of λ will be appropriate at all θ 's.

A method for dynamically adapting λ during optimization, which we have found works reasonably well in practice, will be discussed in section 20.8.5. Note that Tikhonov damping is the method used by LeCun et al. [21], where the constant “ μ ” (which is *not* adapted) plays the role of λ .

It is worth noting that Vinyals and Povey [33] have recently developed an alternative approach to Tikhonov damping, based on the idea of directly optimizing f over a K -dimensional Krylov basis generated by CG (or equivalently a Lanczos iteration). Because the Krylov subspace generated using a $\hat{B} = B + \lambda I$ doesn't depend on λ (assuming a CG initialization of $x_0 = 0$), this method searches over a space of solutions that contain all those which would be found by optimizing a Tikhonov-damped \hat{M} for some λ . Because of this it can find solutions which will give more reduction in f than CG could obtain for *any* value of λ . The downsides of the approach are that the searching must be performed using a general-purpose 2nd-order optimizer like BFGS, which will require extra gradient and function evaluations, that a basis for the entire Krylov subspace must be stored in memory (which may not always be practical when n is large), and finally that CG initializations cannot influence the construction of the Krylov subspace.

20.8.2 Problems with Tikhonov Damping

For standard parameterizations of neural networks, where entries of the weight-matrices and bias vectors are precisely the entries of θ , and the regularization is the standard spherical L2 penalty $\beta \|\theta\|^2$, Tikhonov damping appears to be

a natural choice, and works pretty well in practice. This is because for certain nicely behaved and also useful areas of the parameter space, the effective scale at which each parameter operates is (very) roughly equal. But imagine a simple reparameterization of a FNN so that at some particular layer j , θ parameterizes $10^4 W_j$ instead of W_j . Now the objective function is 10^4 times more sensitive than it was before to changes in the parameters associated with layer j and only layer j , and imposing a Tikhonov damping penalty consisting of an equally weighted sum of squared changes over *all* entries of θ (given by $\lambda/2\|\delta\|^2 = \lambda/2\sum_{i=1}^n \delta_i^2$) no longer seems like a good idea.

For an even more extreme example, consider the case where we would like to constrain some of the weights of the network to be positive, and do this by a simple reparameterization via the \exp function, so that for each component $[\theta]_i$ of θ corresponding to one of these weights w , we have $w = \exp([\theta]_i)$ instead $w = [\theta]_i$. By applying the chain rule we see that in the new parameterization, the i -th component of the gradient, and the i -th row and column of the GGN matrix are both effectively multiplied by $\exp([\theta]_i)$, resulting in the update δ^* changing by a factor $\exp([\theta]_i)^{-1}$ in entry i .

More formally, if we define $C \in \mathbb{R}^{n \times n}$ to be Jacobian of the function ϕ which maps the new parameters back to the default ones, then the gradient and GGN matrix in the new parameterization can be expressed in terms of those from the original parameterization as $C^\top \nabla f$ and $C^\top G C$ respectively⁴. The optimal update thus becomes:

$$\delta^* = (C^\top B C)^{-1} C^\top \nabla f = C^{-\top} B^{-1} \nabla f$$

For our particular example, C is a diagonal matrix satisfying $[C]_{i,i} = \exp([\theta]_i)$ for reparameterized entries of θ , and $[C]_{i,j} = 1$ for the rest.

Assuming that the original 2nd-order update was a reasonable one in the original parameterization, the 2nd-order update as computed in the new parameterization should also reasonable (when taken in the new parameterization). In particular, a reparameterized weight w (and hence f) will become exponentially more sensitive to changes in $[\theta]_i$ as $[\theta]_i$ itself grows, and exponentially less sensitive as it shrinks, so an extra multiplicative factor of $\exp([\theta]_i)^{-1}$ compensates for this nicely. This should be contrasted with gradient descent, where the update will change in exactly the opposite way (being multiplied by $\exp([\theta]_i)$) thus further compounding the sensitivity problem.

Unfortunately, if we use standard Tikhonov damping directly in the reparameterized space, the assumption that all parameters operate at similar scales will be strongly violated, and we will lose the nice self-rescaling property of our update. For example, the curvature associated with $[\theta]_i$, which is equal to the curvature for w multiplied by $\exp([\theta]_i)^2$, may be completely overwhelmed by the addition of λ to the diagonal of G when $[\theta]_i$ is below zero, resulting in an update which will fail to make a substantial change in $[\theta]_i$. Conversely, if $[\theta]_i$ is large

⁴ Note that this result holds for smooth and invertible ϕ , as long as we use the GGN matrix. If we use the Hessian, it holds only if ϕ is affine.

then the Tikhonov damping contribution won't properly penalize large changes to $[\theta]_i$ which may lead to a very large and untrustworthy update.

We could hope that a sensible scheme for adapting λ would compensate by adjusting λ in proportion with $\exp([\theta]_i)$, but the issue is that there are many other components of θ , such as other exp-reparameterized weights, and these may easily be small or larger than $[\theta]_i$, and thus operate at vastly different scales. In practice, what will mostly likely happen is that any sensible scheme for dynamically adjusting λ will cause it to increase until it matches the scale of the largest of these reparameterized weights, resulting in updates which make virtually no changes to the other weights of the network.

In general, Tikhonov and any of the other quadratic penalty based damping methods we will discuss in the following sections, can all be made arbitrarily strong through the choice of λ , thus constraining the optimization of \hat{M} to a degree sufficient to ensure that the update will not leave the region where M is a sensible approximation. What differentiates good approaches from bad ones is how they weigh different directions relative to each other. Schemes that tend to assign more weight to directions associated with more serious violations of the approximation quality of M will get away with using smaller values of λ , thus allowing the sub-optimization of \hat{M} to be less constrained and thus produce larger and more useful updates to θ .

20.8.3 Scale-Sensitive Damping

The scale sensitivity of the Tikhonov damping is similar to the scale sensitivity that plagues 1st-order methods, and is precisely the type of issue we would like to avoid by moving to 2nd-order methods. Tikhonov damping makes the same implicit assumptions about scale that are made by first-order methods: that the default norm $\|\cdot\|$ on \mathbb{R}^n is a reasonable way to measure change in θ and a reasonable quantity to penalize when searching for a suitable update to θ . 1st-order methods can even be viewed as a special case of 2nd-order methods where the curvature term is given entirely by a Tikhonov-type damping penalty, so that $\hat{B} = \lambda I$ and $\delta^* = -1/\lambda \nabla f(\theta)$.

One solution to this problem is to only use parameterizations which exhibit approximately uniform sensitivity properties, but this is limiting and it may be hard to tell at-a-glance if such a property holds for a particular network and associated parameterization.

A potential way to address this problem is to use a quadratic penalty function which depends on the current position in parameter space (θ_{k-1}) and is designed to better respect the local scale properties of f at θ_{k-1} . In particular, instead of adding the penalty term $\lambda/2\|\delta\|^2$ to M we may instead add $\lambda/2\|\delta\|_{D_{k-1}}^2 = \lambda/2\delta^\top D_{k-1}\delta$, where D_{k-1} is some symmetric positive definite (PD) matrix that depends on θ_{k-1} . Such a term may provide a more meaningful measure of change in θ , by accounting for the sensitivity properties of f more precisely. We call the matrix D_{k-1} , the damping matrix, and will drop the subscript $k-1$ for brevity. Scale sensitive damping is implemented in HF by working with a “damped” curvature matrix given $\hat{B} = B + \lambda D$, where the required matrix-vector products

can be computed using as $\hat{B}v = Bv + \lambda Dv$, assuming an efficient algorithm for computing matrix-vector products with D .

A specific damping matrix which may work well in the case of the *exp*-reparameterized network discussed in the previous sub-section would be $D = C^\top C$ (for a definition of C , see the previous sub-section). With such a choice we find that the update δ^* produced by fully optimizing \hat{M} is equal to C^{-1} times the update which would have been obtained with the original parameterization and standard Tikhonov damping with strength λ . Similarly to the undamped case, this is true because:

$$(C^\top BC + \lambda C^\top C)^{-1} C^\top g = C^{-1}(B + \lambda I)^{-1} C^{-\top} C^\top g = C^{-1}(B + \lambda I)^{-1} g$$

It should also be noted that this choice of damping matrix corresponds to a penalty function $\frac{1}{2}\|\delta\|_{C^\top C}^2$ which is precisely the Gauss-Newton approximation of $\frac{\lambda}{2}\|\theta^\dagger\|^2 = \frac{\lambda}{2}\|\phi(\theta)\|^2$ w.r.t. to the new parameters θ , where $\theta^\dagger = \phi(\theta)$ are the default/original parameters. The interpretation is that we are penalizing change in the original parameters (which are assumed to have a very roughly uniform scale), despite performing optimization w.r.t. the new ones.

While we were able to design a sensible custom scheme in this example, exploiting the fact that the default parameterization of a neural network gives parameters which tend to operate at approximately similar scales (in most areas of the parameter space anyway), it would be nice to have a more generic and self-adaptive approach in the cases where we do not have such a property. One possible approach is to set D to be the diagonal matrix formed by taking the diagonal of B (i.e. $D = \text{diag}(\text{diag}(B))$), a choice made in the classical Levenberg-Marquardt algorithm. With this choice, the update δ^* produced by fully optimizing the damped quadratic \hat{M} will be invariant to diagonal linear reparameterizations of θ .

Another nice property of this choice of D is that it produces an update which is invariant to rescaling of f (i.e. optimizing βf instead of f for some $\beta > 0$). By contrast, a pure Tikhonov damping scheme would rely on the careful adjustment of λ to achieve such an invariance.

One obvious way to overcome the deficiencies of a damping approach based on a diagonal matrix would be to use a non-diagonal one, such as the original curvature matrix B itself. Such a choice for D produces updates that share all of the desirable invariance properties associated with a pure undamped Newton approach (assuming full optimization of \hat{M}), such as invariance to *arbitrary* linear reparameterizations, and rescalings of f . This is because with this choice, the damping-modified curvature matrix \hat{B} is simply $(1 + \lambda)B$, and if we assume either full optimization of \hat{M} , or partial optimization via a run of CG initialized from 0, this type of damping has the effect of simply rescaling the update δ by a factor of $1/(1 + \lambda)$. In section 20.8.8 we will discuss line-search methods which effectively accomplish the same type of rescaling.

Despite the nice scale invariance properties associated with these choices, there are good reasons not to use either of them in practice, or at least to use them only with certain modifications, and in conjunction with other approaches.

While the Tikhonov approach arguably makes too few assumptions about the local properties of f , damping approaches based on $D = B$ or its diagonal may make too many. In particular, they make the same modeling assumptions as the original undamped quadratic approximation M itself. For example, B may not even be full-rank, and in such a situation it may be the case that M will predict unbounded improvement along some direction in B 's nullspace, a problem which will not be handled by damping with $D = B$ for any value of λ , no matter how big. Even if B is full-rank, there may be directions of near-zero curvature which can cause a less extreme version of the same problem. Since the diagonal B will usually be full-rank even when B isn't, or just better conditioned in general, using it instead may give some limited immunity to these kinds of problems, but it is far from an ideal solution, as demonstrated in fig. 20.8.3.

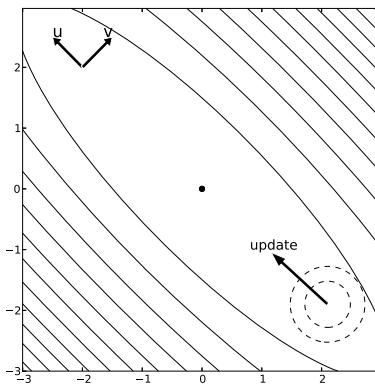


Fig. 20.3. A 2D toy example of how using $D = \text{diag}(B)$ results in an overly restricted update. Let $u = [-1, 1]^\top$ and $v = [1, 1]^\top$, and let B be $uu^\top + avv^\top$ where a is large (e.g. 10^4 , although we take $a = 15$ for display purposes). This matrix is full rank, and its diagonal entries are given by $[a+1, a+1]^\top$, representing the fact that the quadratic is highly sensitive to independent changes to the 2 parameters. The small circular region is where the update will be effectively restricted to when we make λ large enough.

In order to explain such degeneracies and understand why choices like $D = B$ can be bad, it is useful to more closely examine and critique our original choices for making them. The quadratic approximation breaks down due to higher-order effects (and even certain unmodelled 2nd-order effects in the case of the GGN matrix) and it is the goal of damping to help compensate for this. By taking $D = B$ we are penalizing directions according to their curvature, and so are in some sense assuming that the relative strength of the contribution to f from the high-order terms (and thus the untrustworthiness of M) along two different directions can be predicted reasonably well by looking at the ratio of their respective curvatures. And while there is a tendency for this to be true for certain objective functions, making this assumption too strongly may be dangerous.

Unfortunately, in the absence of any other information about the semi-local behavior of the function f , it may not always be clear what kind of assumption we should fall back on. To move towards the uniform scale assumption implied by the Tikhonov approach by choosing D to be some interpolation between the diagonal of B and a multiple of the identity (e.g. using the methods discussed in 20.11.2) seems like an arbitrary choice, since in general there may not be anything particularly special or natural about whatever default parameterization of f that we are given to work with. Despite this, such a strategy can be reasonably effective in some situations, and a reasonable compromise between Tikhonov damping and damping with B .

A conceivably better approach would be to collect information about higher-order derivatives, or to use information collected and aggregated from previous iterations of the optimization process to build a simple model of the coarse geometric structure of f . Or perhaps some useful information could be gleaned from examining the structure of the computational graph of f . Unfortunately we are unaware of the existence of general methods for building D based on such ideas, and so this remains a direction for future research.

In the next section we discuss a method called “structural damping” which constructs D using knowledge of the particular structure of deep and recurrent neural networks, in order to construct damping matrices which may be better at selectively penalizing directions for the purposes of damping.

20.8.4 Structural Damping

Recurrent Neural Networks are known to be difficult to train with gradient descent, so it is conceivable that problematic variations in scale and curvature are responsible. Indeed, a direct application of the implementation of HF presented by Martens [22] to RNNs can yield reasonable results, performing well on a family of synthetic pathological problems [19, 23] designed to have very long-range temporal dependencies of up to 100 time-steps. However Martens and Sutskever [23] found that performance could be made substantially better and more robust using an idea called “structural damping”.

Martens and Sutskever [23] found that a basic Tikhonov damping approach performed poorly when applied to training RNNs. In particular, in order to avoid very large and untrustworthy update proposals, they found λ needed to be very high, and this in turn would lead to much slower optimization. This need for a large λ can be explained by the extreme sensitivity of the RNN’s long sequence of hidden states to changes in the parameters and in particular the hidden dynamics matrix W_{hh} . While these sorts of problems exist with deep feed-forward neural networks like the autoencoders considered in Hinton and Salakhutdinov [17] and Martens [22], the situation with RNNs is much more extreme, since they have many more effective “layers”, and their parameters are applied repeatedly at every time-step and can thus have a dramatic effect on the entire hidden state sequence [4, 18]. Due to this extreme and highly non-linear sensitivity, local quadratic approximations are likely to be highly inaccurate in certain directions in parameter space, even over very small distances. A Tikhonov

damping approach can only compensate for this by imposing a strict penalty against changes in all directions, since it lacks any mechanism to be more selective.

Structural damping addresses this problem by imposing a quadratic penalty not just to changes in parameters, but also to certain intermediate quantities that appear in the evaluation of f , such as the hidden state activities of an RNN. This allows us to be more selective in the way we penalize directions of change in parameter space, focusing on those that are more likely to lead to the large changes in the hidden state sequence, which due to their highly nonlinear nature, tend to correspond to catastrophic breakdowns in the accuracy of the quadratic approximation.

Speculatively, structural damping may have another more subtle benefit for RNN learning. It is known [20, 23] that good random initializations give rise to nontrivial hidden state dynamics that can carry useful information about the past inputs even before any learning has taken place. So if an RNN is initialized carefully to contain such random dynamics, the inclusion of structural damping may encourage the updates to preserve them at least until the hidden-to-output weights have had some time to be adapted to the point where the long-range information contained in the hidden activities actually gets used to inform future predictions. After such a point, a locally greedy optimizer like HF will have more obvious reasons to preserve the dynamics.

To formalize structural damping we first re-express the nonlinear objective $f(\theta)$ as a composition of functions $L(z(h(\theta), \theta))$, where $h(\theta)$ computes the hidden states (whose change we wish to penalize), $z(h, \theta)$ computes the outputs, and $L(z)$ computes the loss.

Given the current parameter setting θ_{k-1} , the local (undamped) quadratic approximation is given by $M(\delta)$ and its curvature is B . We prevent large changes in the hidden state by penalizing the distance between $h(\theta_{k-1} + \delta)$ and $h(\theta_{k-1})$ according to the penalty function

$$S(\delta) = d(h(\theta_{k-1} + \delta); h(\theta_{k-1}))$$

where d is a distance function or loss function such as a squared error or the cross-entropy⁵.

Ideally, we would define the damped local objective as:

$$\hat{M}^\dagger(\delta) = M(\delta) + \mu S(\delta) + \lambda I$$

where μ is a strength constant similar to λ . But since we cannot minimize a non-quadratic objective with CG, we must resort to using a local quadratic approximation to $S(\delta)$. This will be given by $\delta^\top D_{k-1}\delta/2$ where D_{k-1} is the Gauss-Newton matrix of the penalty function $S(\delta)$ at $\delta = 0$. Note that the quadratic approximation to $S(\delta)$ does not have a linear term because $\delta = 0$ is a minimum of S .

Fortunately, it is straightforward to multiply by the generalized Gauss-Newton matrix of S using the techniques outlined in section 20.6. Thus we could compute

⁵ The cross-entropy is suitable when the hidden units use a logistic sigmoid nonlinearity

the products Bv and $\mu D_{k-1}v$ using two separate Gauss-Newton matrix-vector products, adding together the results, approximately doubling the computational burden. In order to avoid this, we can instead compute the sum $(B_{k-1} + \mu D_{k-1})v$ directly by exploiting the fact that $S(\delta)$ can be computed much more efficiently by reusing the $h(\theta_k + \delta)$ which gets computed as an intermediate quantity for $f(\theta_k + \delta)$. Indeed, consider a neural network whose output units include the hidden state as well as the predictions:

$$c(\theta) \equiv [h(\theta), z(h(\theta), \theta)]$$

and whose loss function is given by $L_\mu(h, y) = \mu d(h; h(\theta_{k-1}) + L(y)$, so that we have $f(\theta) + \mu S(\theta) = L_\mu(c(\theta))$. This “new” neural network includes structural damping in its loss, and any automatic routines computing the required Jacobian-vector products for c will be no more expensive than the analogous routines in the original network. Multiplication by the Hessian of the loss $L''_\mu = \begin{pmatrix} L''(y) & 0 \\ 0 & \mu d''(h; h(\theta_{k-1})) \end{pmatrix}$ is also easy and can be done block-wise.

20.8.5 The Levenberg-Marquardt Heuristic

For the penalty-based damping methods such as those described above to work well, λ (and perhaps also μ , as defined in the previous sub-section) must be constantly adapted to keep up with the changing local curvature properties of f .

Fortunately, we have found that the well-known Levenberg-Marquardt (LM) heuristic, which is usually used in the context of the LM method [25] to be effective at adapting λ in a sensible way even in the context of the truncated CG runs that are used in HF.

The key quantity behind the LM heuristic is the “reduction ratio”, denoted by ρ , which is given by

$$\rho \equiv \frac{f(\theta_{k-1} + \delta_k) - f(\theta_{k-1})}{M_{k-1}(\delta_k)} \quad (20.10)$$

The reduction ratio measures the ratio of the reduction in the objective $f(\theta_{k-1} + \delta_k) - f(\theta_{k-1})$ produced by the update δ_k , to the amount of reduction predicted by the quadratic model. When ρ is much smaller than 1, the quadratic model overestimates the amount of reduction and so λ should be increased, encouraging future updates to be more conservative and thus lie somewhere that the quadratic model more accurately predicts the reduction. In contrast, when ρ is close to 1, the quadratic approximation is likely to be fairly accurate near δ^* , and so we can afford to reduce λ , thus relaxing the constraints on δ_k and allowing for “larger” and more substantial updates.

The Levenberg-Marquardt heuristic is given by:

1. If $\rho > 3/4$ then $\lambda \leftarrow \lambda 2/3$
2. If $\rho < 1/4$ then $\lambda \leftarrow \lambda 3/2$

Although the constants in the above description of the LM are somewhat arbitrary, we found them to work well in our experiments.

Note that the above heuristic is valid in the situation where $\rho < 0$, which can arise in one of two ways. The first way is where $M_{k-1} < 0$ and $f(\theta_{k-1} + \delta_k) - f(\theta_{k-1}) > 0$, which means that the quadratic approximation is very inaccurate around δ^* and doesn't even get the sign of the change right. The other possibility is that $M_{k-1} > 0$ and $f(\theta_{k-1} + \delta_k) - f(\theta_{k-1}) < 0$, which can only occur if CG is initialized from a nonzero previous solution and doesn't make sufficient progress from that point to obtain a negative value M_{k-1} before being terminated/truncated. When this occurs one should either allow CG to use more iterations or possibly initialize the next run of CG from 0 (as this will guarantee that $M_{k-1} < 0$, since $M_{k-1}(0) = 0$ and CG decreases M_{k-1} monotonically).

While the definition of ρ in eqn. 20.10 uses the undamped quadratic in the denominator, the damped quadratic approximation \hat{M}_{k-1} can also be used, and in our experience will give similar results, favoring only slightly lower values of λ .

Because of this tendency for M to lose accuracy as CG iterates (see subsection 20.8.7), the value of ρ tends to decrease as well (sometimes after an initial up-swing caused by using a non-zero initialization as in section 20.10). If CG were to run to convergence, the Levenberg-Marquardt heuristic would work just as it does in the classical Levenberg-Marquardt algorithm, which is to say, very effectively and giving provable strong local convergence guarantees. But the situation becomes more complicated when this heuristic is used in conjunction with updates produced by unconverged runs of CG, because the “optimal” value of λ , which the LM heuristic is trying to find, will be a function of how much progress CG tends to make when optimizing M .

Fortunately, as long as the local properties of f change slowly enough, terminating CG according to a fixed maximum number of steps should result in a relatively stable and well-chosen value of λ .

But unfortunately, well intentioned methods which attempt to be smarter and terminate CG based on the value of f , for example, can run into problems caused by this dependency of the optimal λ on the performance of CG. In particular, this “smart” decision of when to stop CG will have an affect on ρ , which will affect the choice of λ via the LM heuristic, which will affect the damping and hence how the value of f evolves as CG iterates (at the next HF iteration), which will finally affect the decision of when to stop CG, bringing us full circle. It is this kind of feed-back which may result in unexpected and undesirable behaviors when using the LM heuristic, such as λ and the number of the length of the CG runs both going to zero as HF iterates, or both quantities creeping upwards to inappropriately large values.

20.8.6 Trust-Region Methods

In contrast to damping methods that are based on penalty terms designed to encourage updates to be “smaller” according to some measure, trust region methods impose an explicit constraint on the optimization of the quadratic model

M . Instead of performing unconstrained optimization on the (possibly damped) quadratic \hat{M} , a constrained optimization is performed over M . This is referred to as the trust-region sub-problem, and is defined by:

$$\delta_R^* = \operatorname{argmin}_{\delta: \delta \in R} M(\delta)$$

where $R \subseteq \mathbb{R}^n$ is some region localized around $\delta = 0$ called the “trust-region”. Ideally, R has the property that M remains a reasonable approximation to f for any $\delta \in R$, without being overly restrictive. Or perhaps more weakly (and practically), that whatever update δ_R^* is produced by solving the trust-region sub-problem will produce a significant reduction in f . Commonly, R is chosen to be a ball of radius r centered at 0, although it could just as easily be an ellipsoid or something more exotic, as long as the required constrained optimization can be performed efficiently enough.

There is a formal connection between trust region methods and penalty-based damping methods such as Tikhonov damping, which states that when R is an elliptical ball around 0 of radius r given $R = \{x : \|x\|_Q \leq r\}$ for some positive definite matrix Q , then for each quadratic function $M(\delta)$ there exists a λ such that

$$\operatorname{argmin}_{\delta: \delta \in R} M(\delta) = \operatorname{argmin}_\delta M(\delta) + \frac{\lambda}{2} \|\delta\|_Q^2$$

This result is valid for all quadratic functions M , even when B is indefinite, and can be proved using Lagrange multipliers.⁶

Trust-region methods have some appeal over the previously discussed penalty-based damping methods, because it may be easier to reason intuitively, and perhaps also mathematically, about the effect of an explicit trust-region (with a particular radius r) on the update than a quadratic penalty. Indeed, the trust region R is invariant to changes in the scale of the objective⁷, which may make it easier to tune, either manually or by some automatic method.

However, the trust region optimization problem is much more difficult than the unconstrained quadratic optimization of \hat{M} . It cannot be directly solved either by CG or by matrix inversion. Even in the case where a spherical trust-region with radius r is used, the previously discussed result is non-constructive and merely guarantees the existence of an appropriate λ that makes the exact minimizer of the Tikhonov damped \hat{M} equal to the solution of the trust region sub-problem. Finding such a λ is a hard problem, and while there are algorithms that do this [26], they involve expensive operations such as full decompositions of

⁶ For completeness, we present an outline of the proof here: Consider the Lagrangian $L(\delta, \lambda) = M(\delta) + (r - \frac{1}{2}\delta^\top Q\delta)\lambda$. It is known that there exists a λ^* such that the minimizer of the trust region problem δ^* satisfies $\partial L(\delta^*, \lambda^*)/\partial(\delta, \lambda) = 0$. For $M(\delta) = g^\top \delta - \delta^\top B\delta/2$, this is equivalent to $g - B\delta^* - \lambda^* Q\delta^* = 0$, and thus $(B + \lambda^* Q)\delta^* = g$. If the matrix $B + \lambda^* Q$ is positive definite, then δ^* can be expressed as the unconstrained minimization of $g^\top \delta - \delta^\top (B + \lambda^* I)\delta/2$.

⁷ If the objective is multiplied by 2, then λ also needs to be multiplied by 2 to achieve the same effect when using a penalty method. By contrast, the trust region R is unaffected by such a scale change.

the B matrix, or finding the solutions of multiple Tikhonov-damped quadratics of the form \hat{M} . When CG is used to perform partial optimization of the quadratic model, there are also good approximation algorithms [13] based on examining the tridiagonal decomposition of B (restricted to the Krylov subspace), but these require either storing a basis for the entire Krylov subspace (which may be impractical when n is large), or will require a separate run of CG once λ has been found via the tridiagonal decomposition, effectively doubling the amount of matrix-vector products that must be computed.

Even if we can easily solve (or partially optimize) the trust-region subproblem, we are still left with the problem of adjusting r . And while it is likely that the “optimal” r will remain a more stable quantity than the “optimal” λ over the parameter space, it still needs to be adjusted using some heuristic. Indeed, one heuristic which is advocated for adjusting r [30] is precisely the Levenberg-Marquardt heuristic discussed in section 20.8.5 which is already quite effective (in our experience) at adjusting λ directly. This leaves us with the question: why not just adjust λ directly and avoid the extra work required to compute λ by way of r ?

One method which is effective at finding reasonable *approximate* solutions of the trust-region sub-problem, for the case where R is a ball of radius r , is to run CG (initialized at zero) until the norm of the solution exceeds r (i.e. the solution leaves the trust-region) and truncate CG at that iteration, with a possible modification to the α multiplier for the final conjugate direction to ensure a solution of norm exactly r . This is known as the Steihaug-Toint method, and it can be shown [13] that, provided CG is initialized from a zero starting solution and no preconditioning is used, the norm of the solution will increase monotonically and that if CG is truncated in the manner described above, the resultant solution δ_k^\dagger will satisfy $M(\delta_k^\dagger) \leq \frac{1}{2}M(\delta_k^*)$ where δ_k^* is the optimal solution to the trust-region sub-problem. This seems like a good compromise, and it is more economical than approaches that try to solve the trust region problem exactly (or using better approximations, as discussed above). Unfortunately, the restriction that CG must be initialized from zero cannot be easily removed, and in our experience such initializations turn out to be very beneficial, as we will discuss in section 20.10.

Another argument against using the Steihaug-Toint method is that if the trust-region is left after only a few steps of CG, it will likely be the case that few, if any, of the low-curvature directions have converged to a significant degree (see section 20.9). And while we know that this will not affect the optimized value of M by more than a factor of 2, it will nonetheless produce a qualitatively different type of update than one produced using a penalty method, which will have a possibly negative effect on the overall optimization trajectory.

Another possible argument against using the Steihaug-Toint method is that it cannot be used with *preconditioned* CG. However, as long as we are willing to enforce an elliptical trust-region of the form $\{x : \|x\|_P < r\}$ where P is the preconditioning matrix, instead of a spherical one, the method still works and

its theory remains valid. And depending on the situation, this kind of elliptical trust-region may actually be a very natural choice.

20.8.7 CG Truncation as Damping

Within HF, the main reason for terminating CG before it has converged is one of a practical nature: the matrix-vector products are expensive and additional iterations will eventually provide diminishing returns as far as optimizing the quadratic model M . But there is another more subtle reason we may want to truncate early: CG truncation may be viewed as special type of damping which may be used in conjunction with (or as an alternative to) the various other damping methods discussed in this section.

As CG iterates, the accuracy of $M(\delta)$ tends to go down, even while $f(\theta_{k-1} + \delta)$ may still be improving. One way to explain this, assuming a zero initialization, is that the norm of δ will increase monotonically with each step, and thus be more likely to leave the implicit region around $\delta = 0$ where M is a reasonable approximation of f . There is also theory which suggests that CG will converge to δ^* first along directions of mostly higher curvature, and only later along directions of mostly low curvature (see section 20.9). While pursuing low curvature directions seems to be important for optimization of deep networks and RNNs, it also tends to be associated with large changes in θ which can lead to more serious breakdowns in the accuracy of the local quadratic model.

The Steihaug-Toint method described in section 20.8.6 already makes use of this phenomenon and relies exclusively on truncating the solution early to enforce trust-region constraints. And it is well-known that truncating CG, which effectively limits the size of the Krylov subspace, has certain regularizing properties in the context of solving noisy and ill-posed systems of linear equations [14].

Although it wasn't emphasize this in the paper, Martens [22] supplemented the progress-based CG termination criteria with a maximum limit of 250 steps. In practice, we have found that this maximum is consistently reached after the first 100 (or so) iterations of HF when the approach is applied to problems like the “CURVES” auto-encoder from Hinton and Salakhutdinov [17], and that it plays an important role which is not limited to saving computation time. Instead, we can observe that the improvement in the value of the objective f is non-monotonic in the number of CG steps, and may peak long before condition 20.2 is satisfied (see fig. 20.4).

Martens [22] proposed “CG-backtracking” as a method to take advantage of this tendency for earlier iterates to be more favorable as updates, by selecting the “best” iterate among some manageable subset, as measured by the associated reduction in f . One possible approach which is reasonably efficient is to compute the objective f only on a small subset of the current minibatch or training set, and only at every multiple of c steps for some fixed c , or every power of ν steps (rounding to the nearest integer) for some fixed ν , and then terminate once it appears that there will be no further possible improvement in the objective.

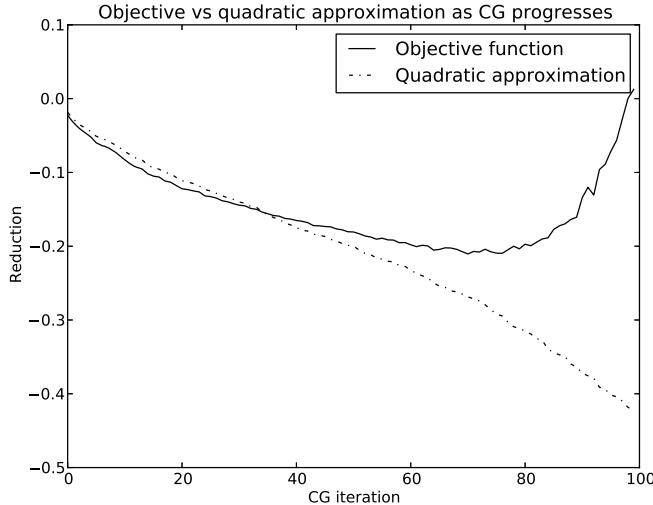


Fig. 20.4. A plot of the objective function (solid) versus the quadratic approximation (dotted) as a function of the number of CG steps. This was selected as a typical example of a single run of CG performed by HF on a typical deep auto-encoder training task.

An important thing to keep in mind is that the damping effect of CG truncation will depend on both the preconditioning scheme used within CG (as discussed in section 20.11), and on the particular manner in which CG is initialized (as discussed in sec. 20.10). In the extreme case where $P = B$, the eigenvalues will all be equal and CG will converge in one step, rendering CG truncation trivial/useless. And for “stronger” preconditioners that let CG converge faster it can be argued that truncation at a particular iteration i will have a smaller effect than with “weaker” preconditioners. But this perspective oversimplifies a very complex issue.

Preconditioning is effectively reparameterizing the quadratic optimization problem, which has the effect of creating new eigenvectors with new eigenvalues and of changing the Krylov subspace (as discussed further in section 20.11.1). This in turn affects the “order” in which CG tends to prioritize convergence along different directions (see section 20.9). Thus, when CG is terminated long before convergence, preconditioning will have an important effect on the nature of the implicit damping and thus on the quality of the update.

From the perspective of the Steihaug-Toint method and trust-regions, CG preconditioning can be thought of as determining the shape of the trust-region that is being implicitly enforced through the use of truncation. In particular, the trust region will be given by $R = \{x : \|x\|_P \leq r\}$, for some r , where P is the preconditioning matrix. Similarly, initializing CG from some non-zero point x_0 can be thought of as shifting the center of said trust-region away from 0. In both cases, the guarantee remains true that the solution found by truncated CG will be at least half as good (in terms of the value of M), subject to the trust-region

radius implied by the current iterate, as long as we modify definition of the trust region appropriately.

20.8.8 Line Searching

The most basic form of damping, which is present in almost every 2nd-order optimization algorithm, is standard line searching applied to the update proposal δ_k . In particular, we select a scalar α_k to multiply the update δ_k before adding it to θ so that $\theta_k = \theta_{k-1} + \alpha\delta_k$. Usually α is set to 1 as long as certain conditions hold (see Nocedal et al. [30]), and decreased only as necessary until they do. Doing this guarantees that certain local convergence theorems apply.

Provided that δ_k is a descent direction ($\delta_k^\top \nabla f(\theta_{k-1}) < 0$) we know that for a sufficiently small (but non-zero) value of α , we will have:

$$f(\theta_{k-1}) > f(\theta_k) = f(\theta_{k-1} + \alpha\delta_k) \quad (20.11)$$

δ_k will be descent direction as long as \hat{B}_{k-1} is positive definite, ∇f is computed on the entire training set, and \hat{M} is optimized either exactly, or partially with CG (provided that we achieve $M(\delta) < 0$). Thus, in many practical scenarios, a line search will ensure that the update results in a decrease in f , although it may be very small. And if we are content with the weaker condition that only the terms of f corresponding to the minibatches used to estimate $\nabla f(\theta_{k-1})$ must decrease, then we can drop the requirement that ∇f be computed using the whole training set.

One easy way to implement line searching is via the back-tracking approach, which starts at $\alpha = 1$ and repeatedly multiplies this by some constant $\beta \in (0, 1)$ until the “sufficient-decrease condition” applies. This is given by

$$f(\theta_{k-1} + \alpha\delta_k) \leq f(\theta_{k-1}) + c\alpha\nabla f(\theta_{k-1})^\top\delta_k$$

where c is some small constant like 10^{-2} . It can be shown that this following approach will produce an update which has strong convergence guarantees⁸.

Unlike 1st-order methods where the total number of updates to θ can often be on the order of $10^4 - 10^6$ for a large neural network, powerful approximate Newton methods like HF may only require on the order of $10^2 - 10^3$, so the expense of a line-search is much easier to justify.

Note also that it is also possible to view line searching as a special type of a penalty based damping method, where we use a penalty term of the form $\frac{1}{2\alpha}B$. In other words, we simply increase the scale of curvature matrix B by $1/\alpha$. This interpretation is valid as long as we solve \hat{M} exactly, or partially by CG as long as it is initialized from $\mathbf{0}$.

The line search is best thought of as a last line of defense to compensate for occasional and temporary maladjustment of the various non-static parameters of the other damping methods, such as λ or r , and not as a replacement for

⁸ But note that other possible ways of choosing an α that satisfies this condition may make α too small.

these methods. If the line search becomes very active (i.e., very often chooses an α strictly less than 1) there are two probable causes, which should be addressed directly instead of by relying on the line-search. The first is poorly designed/inappropriate damping methods and/or poor heuristics for adjusting their non-static meta-parameters. The second probable cause is that the data used to estimate the curvature and/or gradient is insufficient and the update has effectively “overfitted” the current minibatch.

The argument for not relying on line searches to fix whatever problems might exist with the updates produced by optimizing M is that they work by rescaling each eigen-component (or conjugate direction) of the update *equally* by the multiplicative factor α , which is a very limited and inflexible approach. It turns out that in many practical situations, the type of scaling modification performed by a penalty method like Tikhonov damping is highly preferable, such as when B is very ill-conditioned. In such a situation, minimizing M results in an update proposal δ_k which is scaled much too strongly in certain, possibly spurious, low-curvature eigen-directions, by a factor of possibly 10^3 or more, and a line-search will have to divide *all components* by this factor in order to make the update viable, which results in an extremely small update. Meanwhile, a Tikhonov approach, because it effectively modifies the curvatures by the additive constant λ (as discussed in section 20.8.1) can easily suppress these very low-curvature directions while leaving the higher curvature directions relatively untouched, which makes the update much bigger and more useful as a result.

20.9 Convergence of CG

In this section we will examine the theoretical convergence properties of CG and provide justifications for various statements regarding its convergence made throughout this report, such as how δ converges along different “directions” in parameter space, and how CG prioritizes these directions according to their “associated curvature”. This analysis has particularly important implications for preconditioning (see section 20.11) and CG truncation damping (see section 20.8.7).

Before we begin it should be noted that due to inexact computer arithmetic, in practice the conclusions of this analysis (which implicitly assume exact arithmetic) will only hold approximately. Indeed, CG, unlike many other numerical algorithms, is highly sensitive to numerical issues and after only 5–10 iterations on a high-precision machine will produce iterates that can differ significantly from those which would be produced by a hypothetical exact implementation.

We will analyze CG applied to a general quadratic objective of the form $q(x) = \frac{1}{2}x^\top Ax - b^\top x$ for a symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$. This can be related by to HF by taking $A = B$ (or $A = \hat{B}$ in the case of a damped quadratic approximation) and $b = -\nabla f$.

Note that x as defined here is an n -dimensional vector and should not be confused with its use elsewhere in this chapter as the input to a neuron.

Let $\{\lambda_j\}_{j=1}^n$ be the eigenvalues of A and $\{v_j\}_{j=1}^n$ the corresponding (unit) eigenvectors, x_0 be the initialization of CG, and x^* the minimizer of q . Since the

v_j 's are an orthonormal basis for \mathbb{R}^n (because A is symmetric and invertible) for any x we can express $x_0 - x^*$ in terms of the v_j 's, giving $x_0 - x^* = \sum_{j=1}^n \xi_j v_j$ for $\xi_j = v_j^\top (x_0 - x^*)$.

It can be shown [30, 32] that:

$$\|x_i - x^*\|_A^2 = \min_p \sum_{j=1}^n \lambda_j p(\lambda_j)^2 \xi_j^2 \quad (20.12)$$

where $\|z\|_A^2 \equiv \frac{1}{2} z^\top A z$ and where the minimum is taken over all polynomials of degree i with constant term 1. This result can be used to prove various convergence theorems for CG [30]. For example, CG will always converge to x^* after a number of iterations less than or equal to the number m of distinct eigenvalues of A , since it is easy to design a polynomial of degree m with constant term 1 that satisfies $p(\lambda_j) = 0$ for all j .

To gain more insight into eqn. 20.12 we will re-derive and re-express it in a way that implies an intuitive interpretation for each term. It is easy to show that:

$$q(z + x_0) - q(x_0) = \frac{1}{2} z^\top A z + r_0^\top z$$

where $r_0 = Ax_0 - b$ (i.e. the initial residual). And so $q(z + x_0) - q(x_0)$ is a quadratic in z with constant term equal to 0, and linear term r_0 .

Defining $\eta_j = r_0^\top v_j$ to be the size of eigenvector v_j in direction $r_0 = Ax_0 - b$ (which is the initial residual), and observing that $v_j^\top A v_j = \lambda_j v_j^\top v_j = \lambda_j$, we have for any $\alpha \in \mathbb{R}$:

$$q(\alpha v_j + x_0) - q(x_0) = \frac{1}{2} \alpha^2 v_j^\top A v_j + \alpha r_0^\top v_j = \frac{1}{2} \alpha^2 \lambda_j + \alpha \eta_j$$

Since the v_j 's are an orthonormal basis for \mathbb{R}^n (because A is symmetric and invertible), we can express $x - x_0$ (for any x) in terms of the v_j 's, giving $x = x_0 + \sum_j \beta_j v_j$ where $\beta_j = v_j^\top (x - x_0)$. Note that the v_j 's are also mutually conjugate (which follows from them being both orthonormal and eigenvectors) and that since $q(z + x_0) - q(x_0)$ is quadratic in z with constant term 0 we have that for any two conjugate vectors u and w :

$$q(u + w + x_0) - q(x_0) = (q(u + x_0) - q(x_0)) + (q(w + x_0) - q(x_0))$$

which is straightforward to show. Thus we have:

$$\begin{aligned} q(x) - q(x_0) &= q((x - x_0) + x_0) - q(x_0) \\ &= \sum_{j=1}^n (q(\beta_j v_j + x_0) - q(x_0)) = \sum_{j=1}^n \left(\frac{1}{2} \beta_j^2 \lambda_j + \beta_j \eta_j \right) \end{aligned}$$

What this says is that the size β_j of the contribution of each eigenvector/eigen-direction to x , have an independent influence on the value of $q(x) - q(x_0)$, and so

we can meaningfully talk about how each one of them independently “converges” as CG iterates.

In a sense, each β_j is being optimized by CG, with the ultimate goal of minimizing the corresponding 1-D quadratic $q(\beta_j v_j + x_0) - q(x_0)$, whose minimizer is $\beta_j^* = -\frac{\eta_j}{\lambda_j} = v_j^\top (x^* - x_0)$ with associated minimum value:

$$q(\beta_j^* v_j + x_0) - q(x_0) = -\frac{1}{2} \frac{\eta_j^2}{\lambda_j} = -\omega_j$$

where we define $\omega_j \equiv \frac{\eta_j^2}{\lambda_j}$. The difference between the current value of $q(\beta_j v_j + x_0)$ and its minimum has a particularly nice form:

$$q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0) = \frac{1}{2} \beta_j^2 \lambda_j + \beta_j \eta_j + \omega_j = \omega_j \left(\frac{\lambda_j}{\eta_j} \beta_j + 1 \right)^2$$

Now suppose that $x - x_0 \in K_i(A, r_0)$ so that there exists some $(i-1)$ -degree polynomial s s.t. $x - x_0 = s(A)r_0$ and note that $s(A)v_j = s(\lambda_j)v_j$. Then,

$$\beta_j = v_j^\top (x - x_0) = v_j^\top s(A)r_0 = (s(A)v_j)^\top r_0 = (s(\lambda_j)v_j)^\top r_0 = s(\lambda_j)\eta_j$$

so that:

$$\begin{aligned} q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0) &= \omega_j \left(\frac{\lambda_j}{\eta_j} s(\lambda_j)\eta_j + 1 \right)^2 \\ &= \omega_j (\lambda_j s(\lambda_j) + 1)^2 = \omega_j p(\lambda_j)^2 \end{aligned}$$

where we define $p(z) = z s(z) + 1$ (which is a general polynomial of degree i with constant coefficient 1).

Summarizing, we have:

$$\begin{aligned} q(x) - q(x^*) &= (q(x) - q(x_0)) - (q(x^*) - q(x_0)) \\ &= \sum_{j=1}^n (q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0)) = \sum_{j=1}^n \omega_j p(\lambda_j)^2 \end{aligned} \quad (20.13)$$

We now apply this results to CG. We know that after i steps, CG applied to q with initialization x_0 , finds the iterate x_i which minimizes $q(x_i)$ subject to restriction $x_i - x_0 \in K_i(A, r_0)$. This is equivalent to minimizing over all p with the requirement that p has degree i with constant coefficient 1, or in other words:

$$q(x_i) - q(x^*) = \min_p \sum_{j=1}^n \omega_j p(\lambda_j)^2 \quad (20.14)$$

Thus we see that CG is effectively picking a polynomial p to minimize the weighted sum of p^2 evaluated at the different eigenvalues.

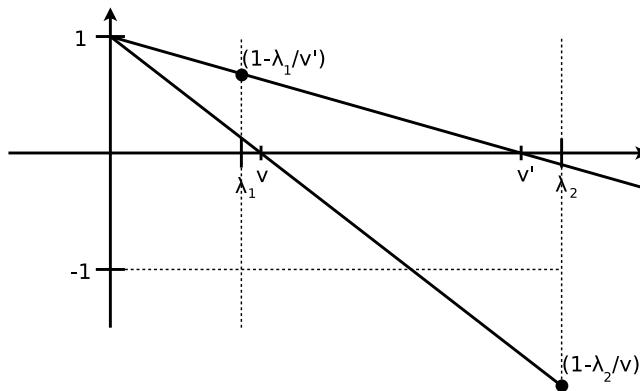


Fig. 20.5. This figure demonstrates geometrically how the contribution to the polynomial $p(z)$ of an additional root ν or ν' in the vicinity of a small eigenvalue λ_1 or a large eigenvalue λ_2 (resp.) affects the loss term associated with the other eigenvalue. In particular, the distance of the lines above or below the horizontal axis is equal to the factor whose square effectively multiplies the loss term associated with the given eigenvalue.

As mentioned before, there are various results that make use of expressions similar to this one in order to prove results about how the distribution of the eigenvalues of A determine how quickly CG can make progress optimizing q . One particularly interesting result states that if the eigenvalues cluster into m groups, then since we can easily design a degree- m polynomial p which is relatively small in the vicinity of each cluster by placing a root of p at each cluster center, the error will be quite low by the m -th iteration [30].

However, the particular form of eqn. 20.14 and its derivation allow us to paint a more intuitive picture of how CG operates. Each of the terms in the sum 20.13 correspond to a direction-restricted objective $q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0) = \omega_j p(\lambda_j)^2$ which are indirectly optimized by CG w.r.t. the β_j 's. The size each of these “loss” terms negatively correlates with how much progress CG has made in optimizing x along the corresponding eigen-directions, and by examining the form of these terms, we can talk about how CG will “prioritize” these different terms (and hence the optimization of their associated eigen-directions) through its choice of an optimal polynomial p .

Firstly, consider the “weights” $\omega_j = -(q(\beta_j^* v_j + x_0) - q(x_0)) = \frac{1}{2} \frac{\eta_j^2}{\lambda_j}$, which measure the total decrease in q that can be obtained by fully optimizing along the associated direction v_j . Their effect is thus to shift the focus of CG towards those eigen-directions which will give the most reduction. They are inversely proportional to the curvature λ_j , and proportional to $\eta_j^2 = (v_j^\top r_0)^2$, which is the square of the size of the contribution of the eigen-direction within the initial residual (which in HF will be the gradient of f when $x_0 = 0$), and this makes the ω_j 's “scale-invariant”, in the sense that any linear reparameterization which

preserves the eigenvectors of A , while possibly rescaling the eigenvalues, will have no effect on the ω_j 's.

Secondly, we note the effect of the size of the λ_j 's, or in other words, the curvatures associated with v_j 's. If it weren't for the requirement that p must have a constant term of 1, the λ_j 's probably wouldn't have any influence on CG's prioritizing of directions (beyond for how they modulate the weights ω_j). But note that general polynomials of degree i with constant coefficient 1 must have the form:

$$p(z) = \prod_{k=1}^i \left(1 - \frac{z}{\nu_k}\right)$$

for $\nu_k \in \mathbb{C}$. We will argue by illustrative example that this fact implies that CG will favor high-curvature directions, everything else being equal. Suppose there are two tight clusters of eigenvalues of A , a low-magnitude one located close to zero and a large-magnitude one located further away. Suppose also that they have equal total loss as measured by the sum of the associated $\omega_j p(\lambda_j)^2$'s (for the current p). Placing an additional root ν_k close to the large-magnitude cluster will greatly reduce the associated $\omega_j p(\lambda_j)^2$ loss terms in that cluster, by effectively multiplying each by $\left(1 - \frac{\lambda_j}{\nu_k}\right)^2$ which will be small due to the closeness of ν_k to each λ_j . Meanwhile, for the λ_j 's in the small-magnitude cluster, the associated loss terms will be multiplied by $\left(1 - \frac{\lambda_j}{\nu_k}\right)^2$ which won't be greater than 1 since $0 < \lambda_j < \nu_k$, implying that these loss terms will not increase (in fact, they will very slightly decrease).

Now contrast this with what would happen if we placed a root ν_k close to the small magnitude cluster. As before, the loss terms associated with that cluster will be greatly reduced. However, because $\lambda_j \gg \nu_k$ for λ_j 's in the large-magnitude cluster, we will have $\left(1 - \frac{\lambda_j}{\nu_k}\right)^2 \gg 1$ for such λ_j 's, and so the associated loss terms will greatly increase, possibly even resulting in a net increase in q . Thus CG, being optimal, will place the root near to the large-magnitude cluster in this situation, versus the small magnitude-one, despite convergence of either one yielding the same improvement in q .

20.10 Initializing CG

As in the previous section we will use the generic notation $q(x) = \frac{1}{2}x^\top Ax - b^\top x$ to refer to the quadratic objective being optimized by CG.

A useful property of CG is that it is able to make use of arbitrary initial guesses x_0 for x . This choice can have a strong effect on the performance of CG, which is not surprising since x_i depends strongly on the Krylov subspace $K_i(A, r_0)$ (where $r_0 = Ax_0 - b$), which in turn depends strongly on x_0 . From

the perspective of the previous section, an initial x_0 may be “more converged” than $x = 0$ along some eigen-directions and less converged along others, thus affecting the corresponding weights ω_j , which measure the total reduction that can be obtained by fully optimizing eigen-direction v_j (versus leaving it as it is in x_0). This “redistribution” of weights caused by taking a different x_0 may make the quadratic optimization easier or harder for CG to optimize, depending on how the eigenvalues and associated weights are distributed.

Since the local geometry of the error surface of f (and hence the local damped quadratic model $q = \hat{M}$) changes relatively slowly between updates (at least along some eigen-directions), this suggests using the previous update δ_{k-1} as the starting solution x_0 for CG, as was done by Martens [22].

In practice, this choice can result in an initial value of q which is higher than zero, and thus seemingly worse than just using $x_0 = 0$, which satisfies $q(x_0) = 0$. x_0 may not even be a descent direction, implying that $q(\epsilon x_0) > 0$ for all $\epsilon > 0$. But these objections are based on the naive notion that the value of q tells us everything there is to know about the quality of potential initialization. What we observe in practice is that while CG runs initialized with $x_0 = \delta_{k-1}$ “start slow” (as measured by the value of $q(x)$), they eventually catch up and then surpass runs started from $x_0 = 0$.

To make sense of this finding, we first note it is easy to design initializations which will have arbitrarily high values of q , but which require only one CG step to reach the minimum. To do this, we simply take the minimizer of q and add a large multiple of one of the eigenvectors of A to it. This corresponds to the situation where only one eigenvalue λ_j has non-zero weight ω_j , so that to make $q(x) - q(x^*) = 0$ CG can simply select the degree-1 polynomial which places a root at λ_j .

More generally, x_0 may be more converged than 0 along eigen-directions which are more numerous, or which have small and spread-out eigenvalues (i.e. curvatures), and meanwhile less converged than 0 (perhaps severely so) only along eigen-directions which are fewer in number, or have larger or more tightly clustered eigenvalues. If the later group has a larger total weight (given by the sum of the ω_j ’s as defined in the previous section) this will cause $q(x_0) > 0$. But since the former directions will be easier for CG to optimize than the latter, this implies that the given x_0 will still be a highly preferable initialization over “safer” choice of 0, as long as CG is given enough iterations to properly optimize the later group of badly initialized but “easy-to-fix” eigen-directions.

We surmise that the choice $x_0 = \delta_{k-1}$ fits into the situation described above, where the later group of eigen-directions correspond to the slowly optimized low-curvature directions that tend to remain descent-directions across many HF iterations. Consistent with this theory, is our observation that the number of CG steps required to achieve $q(x) < 0$ from the initialization $x_0 = \delta_{k-1}$ tends to grow linearly with the number of CG steps used at the previous HF iteration⁹.

⁹ This is, incidentally, one reason why it is good to use a fixed maximum number of CG iterations at each HF step.

Analogously to how the current update vector is “decayed” by a scalar constant when using gradient descent with momentum, we have found that it is helpful to slightly decay the initialization, taking $x_0 = \zeta \delta_{k-1}$ for some constant $0 \leq \zeta \leq 1$, such as 0.95.

Choosing this decay factor for HF carefully is not nearly as important as it can be for momentum methods. This is because while momentum methods modify their current update vectors by a single gradient-descent step, HF uses an entire run of CG, which can make much more significant changes. This allows HF to more easily scale back x_0 along eigen-directions, which while they may have been helpful at the previous θ , are no longer appropriate to follow from the current θ . In particular, x_0 will be quickly “corrected” along turbulent directions of high-curvature, reducing (but not completely eliminating) the need for a decay to help “clean up” these directions.

Our experience suggests that properly tuning the decay constant becomes more important as aggressive CG truncation, or other factors like weak preconditioning, limit CG’s ability either to modify the update from its initial value x_0 , or to make good progress along important low-curvature directions. While the former problem calls for lowering ζ , the later calls for raising it. The optimal value will likely depend on the amount of truncation, the type of preconditioning, and the local geometry of the objective being optimized. $\zeta = 0.95$ seems to be a good default value, but it may help to reduce it when using an approach which truncates CG very early. It may also be beneficial to increase it in the later stages of optimization where CG struggles much harder to optimize q along low curvature directions.

20.11 Preconditioning

As powerful as CG is, there are quadratics optimization problems which can be easily solved using more direct methods, that CG will struggle with. For example, if the curvature matrix is diagonal, CG will in general require i iterations to converge (where i is the number of distinct values on the diagonal) using a total of $O(in)$ time. Meanwhile, we could easily solve the entire system by straightforward inversion of the diagonal curvature matrix in time $O(n)$. CG is, in a sense, unaware of this special structure and unable to exploit it.

While the curvature matrix will in general not be diagonal or have any other special form that makes it easy to invert, there may nevertheless be cheap operations which can exploit information about the course structure of the curvature matrix to do some of the work in optimizing q , reducing the burden on CG.

In the context of HF, preconditioning refers to the reparameterization of \hat{M}^{10} according to some linear transformation relatively easy to invert, with the idea that CG will make more rapid progress per iteration optimizing w.r.t. the new parameterization.

¹⁰ We will use the $\hat{\cdot}$ notation for the damped quadratic and associated damped curvature matrix \hat{B} since this is what CG will actually optimize when used within HF.

Formally, given some invertible transformation defined by a matrix C , we transform the quadratic objective $\hat{M}(\delta)$ by a change of coordinates $\delta = C^{-1}\gamma$ and optimize w.r.t. γ instead of δ .

$$\hat{M}(C^{-1}\gamma) = \frac{1}{2}\gamma^\top C^{-\top} \hat{B} C^{-1} \gamma + \nabla f^\top C^{-1} \gamma$$

Applying preconditioning to CG is very easy and amounts to computing transformed residual vectors y_i at each iteration, by solving $Py_i = r_i$, where $P = C^\top C$ (see alg. 20.2). This can be accomplished, say, by multiplication of r_i by P^{-1} , which for many common choices of P (such as diagonal approximations of \hat{B}) is a cheap operation.

Preconditioning can be applied to other optimization methods, such as gradient descent, where it corresponds to a non-static linear reparameterization of the objective f that typically varies with each iteration, and amounts simply to multiplication of the gradient update by P^{-1} . In fact, one way to view 2nd-order optimization is as a particular non-static preconditioning approach for gradient descent, where P is given by the curvature matrix B (or some approximation or Krylov subspace restriction of it).

20.11.1 The Effects of Preconditioning

In section 20.9, we saw how the eigenvalues of the curvature matrix and the corresponding sizes of the contributions to the initial residual of each eigen-direction effectively determine the convergence characteristics of CG, in terms of the “order” in which the eigen-directions tend to converge, and how quickly. It was found that each eigen-direction has an effective “weight” ω_j (corresponding to the total decrease in q which can be obtained by completely optimizing it), and that CG prioritizes convergence along the eigen-directions both according to their weights and their associated curvature/eigenvalue, preferring larger values of both. Because CG is optimal, it will tend to make faster progress along directions whose the eigenvalues are close proximity to many other ones that are associated with directions of high-weight (due to its ability to make progress on many such directions at once when their eigenvalues are closely packed).

Thus to understand how a potential preconditioning scheme affects the convergence of CG we can look at the eigen-distribution of the transformed curvature matrix $C^{-\top} \hat{B} C^{-1}$, and the associated weights, which depend on the transformed initial residual $C^{-\top} (\hat{B}x_0 - \nabla f)$. Choices for C (or equivalently P) which yield tight clusters of eigenvalues should lead to overall faster convergence, at least along the directions which are contained in such clusters.

But as discussed in section 20.8.7, the eigenvectors and corresponding eigenvalue distribution will affect the order in which various directions converge, and this will interact in a non-trivial way with CG truncation damping. In particular, certain directions which would otherwise never be touched by CG in the original parameterization, either because their eigenvalues are located far away from any high-weight eigenvalue clusters, or because they have very low curvature (i.e., low eigenvalue), could, within the reparameterization, become part of

eigen-directions with the opposite properties, and thus be partially optimized by CG even when it is aggressively truncated.

This is a potential problem, since it is our experience that certain very low curvature directions tend to be highly non-trustworthy for neural network training objectives (in the default parameterization). In particular, they often tend to correspond to degeneracies in the quadratic model, such as those introduced by using different sets of data to compute the gradient and curvature-matrix vector products (see section 20.12.1), or to directions which yield small reductions in q for the current minibatch but large increases on other training data (an issue called “minibatch overfitting”, which is discussed in section 20.12.2).

20.11.2 Designing a Good Preconditioner

Designing a good preconditioner is an application specific art, especially for HF, and it is unlikely that any one preconditioning scheme will be the best in all situations. There will often be a trade-off between the computational efficiency of implementing the preconditioner and its effectiveness, both in terms of how it speeds of convergence of CG, and how it may reduce the effectiveness of CG truncation damping.

While the previous section describes how a preconditioner can help in theory, in practice it is not obvious how to design one based directly on insights about eigen-directions and their prioritization.

An approach which is popular and often very effective in various domains where CG is used is to design P to be some kind of easily inverted approximation of the curvature matrix (in our case, \hat{B}). While the ultimate purpose of preconditioning is to help CG optimize more effectively, which may conceivably be accomplished by less obvious choices for P , approximating \hat{B} may be an easier goal to approach directly. Justifying this idea is the fact that when $P = \hat{B}$, the preconditioned matrix is I , so CG will converge in one step.

Adopting the perspective that P should approximate \hat{B} , the task of designing a good preconditioner becomes one of balancing approximation quality with practical concerns, such as the cost of multiplying by P^{-1} .

Of course, “approximation quality” is a problematic concept, since the various ways we might want to define it precisely, such as via various matrix norms, may not correlate well with the effectiveness of P as a preconditioner. Indeed, CG is invariant to the overall scale of the preconditioner, and so while $\beta\hat{B}$ would be an optimal preconditioner for any $\beta > 0$, it could be considered an arbitrarily poor approximation to \hat{B} as β grows, depending on how we measure this.

Diagonal P 's are a very convenient choice due to the many nice properties they naturally possess, such as being full rank, easy to invert, and easy to store. They also tend to be quite effective for optimizing deep feed-forward neural networks, due to how the scale of the gradient and curvature w.r.t. the hidden activities grows or shrinks exponentially as we proceed backwards through the layers [4, 18], and how each parameter is associated with a single layer. Without compensating for this with diagonal preconditioning, the eigenvalues of the effective curvature matrix will likely be much more “spread out” and thus harder

for CG to deal with. By contrast, RNN optimization does not seem to benefit as much from diagonal preconditioning, as reported by Martens and Sutskever [23]. Despite how RNNs can possess per-timestep scale variations analogous to the per-layer scale variations sometimes seen with feed-forward nets, these won't manifest as differences in scales between any particular parameters (i.e. diagonal scale differences), due to the way each parameter is used at *every* time-step.

Many obvious ways of constructing non-diagonal preconditioners end up resulting in P 's which are expensive and cumbersome to use when n is large. For example, if P or P^{-1} is the sum of a k -rank matrix and a diagonal, it will require $O((k+1)n)$ storage, which for very large n will be a problem (unless k is very small).

A well-designed diagonal preconditioner P should represent a conservative estimate of the overall scale of each parameter, and while the diagonal of the curvature matrix is a natural choice in many situations, such as when the curvature matrix is diagonally dominant, it is seriously flawed for curvature matrices with a strong non-diagonal component. Nonetheless, building a diagonal preconditioner based on $d = \text{diag}(\hat{B})$ (or an approximation of this) is a sensible idea, and forms the basis of the approaches taken by Martens [22] and Chapelle and Erhan [10]. However, it may be beneficial, as Martens [22] found, not to use d directly, but to choose P to be somewhere between $\text{diag}(d)$ and a scalar multiple of the identity matrix. This has the effect of making it more gentle and conservative, and it works considerably better in practice. One way to accomplish this is by raising each entry of d (or equivalently, the whole matrix P) to some power $0 < \xi < 1$, which will make P tend to the identity as ξ approaches 0.

In situations where diagonal damping penalty terms like the Tikhonov term are weak or absent, it may also be beneficial to include an additional additive constant κ , which also has the effect of making P tend to a scalar multiple of the identity as κ grows so that we have:

$$P = (\text{diag}(d) + \kappa I)^\xi$$

If there is information available about the coarse relative scale of the parameters, in the form of some vector $s \in \mathbb{R}^n$, such as the reparameterized neural network example discussed in sec. 20.8.2, it may better to use $\kappa \text{diag}(s)$ instead of κI .

It is important to emphasize that d should approximate $\text{diag}(\hat{B})$ and not $\text{diag}(B)$, since it is the latter curvature matrix which is used in the quadratic which CG actually optimizes. When D is a diagonal matrix, one should take

$$d = \text{diag}(B) + \lambda D$$

where the latter contribution can be computed independently and exactly (and not via the methods for approximating $\text{diag}(B)$ which we will discuss next). Meanwhile, if the damping matrix D is non-diagonal, then one should take

$$d = \text{diag}(B + \lambda D)$$

where we might in fact use the aforementioned methods in order to approximate the diagonal of $B + \lambda D$ together.

So far the discussion ignored the cost of obtaining the diagonal of a curvature matrix. Although it is easy to compute Hessian-vector products of arbitrary functions, there exists no efficient exact algorithm for computing the diagonal of the Hessian of a general nonlinear function (Martens et al. [24, sec. 4]), so approximations must be used. Lecun et al. [2] report an efficient method for computing the diagonal of the Gauss-Newton matrix, but close examination reveals that it is mathematically unsound (although it can still be viewed as a heuristic approximation).

In case of the Gauss-Newton matrix, it is possible to obtain the exact diagonal at the cost of k runs of backpropagation, where k is the number of output units [6]. This approach can be generalized in the obvious way to compute the diagonal of the generalized Gauss-Newton matrix, and is feasible for classification problems with small numbers of classes, although not feasible for problems such as deep autoencoders or RNNs which have high-dimensional outputs. In the next sections, we describe two practical methods for approximating the diagonal of the GGN matrix regardless of the dimension of the output.

20.11.3 The Empirical Fisher Diagonal

One approach to approximating the diagonal of the GGN matrix G is to instead compute the diagonal of a related matrix for which exact computation of the diagonal is easier. For this purpose Martens [22] selected the Empirical Fisher Information matrix F , which is an approximation to the well-known Fisher information matrix [1] (which is itself related to the generalized Gauss-Newton matrix). The empirical Fisher Information matrix is given by

$$F \equiv \sum_i \nabla f_i \nabla f_i^\top$$

where ∇f_i is the gradient on case i . However, because of its special low-rank form, its diagonal is readily computable as:

$$\text{diag}(F) = \sum_i \text{sq}(\nabla f_i)$$

where $\text{sq}(x)$ denotes coordinate-wise square.

Because the ∇f_i 's are available from the gradient computation $\nabla f = \sum_i \nabla f_i$ over the minibatch, additionally computing $\text{diag}(F)$ over the same minibatch in parallel incurs no extra cost, save for the possible requirement of storing the ∇f_i 's, which can be avoided for feed-forward networks but not RNNs. Algorithm 20.11.3 computes the diagonal of the Empirical Fisher Information matrix without the extra storage. In the algorithm, each y_i is a matrix with B columns which represent the activations of a minibatch with B cases, and $\text{sq}(\cdot)$ is the coordinate-wise square. It differs from algorithm 20.2 only in lines 9 and 10.

In general, it is possible to compute the sum of squares of gradients in a minibatch in parallel without storing the squares of the individual gradients (which is often prohibitively expensive) whenever the computational graph of

Algorithm 20.6. An algorithm for computing the diagonal $\text{diag}(F)$ of the Empirical Fisher Information matrix of a feedforward neural network (includes the standard forward pass)

```

1: input:  $y_0, \theta$  mapped to  $(W_1, \dots, W_{\ell-1}, b_1, \dots, b_{\ell-1})$ 
2: for all  $i$  from 1 to  $\ell - 1$  do
3:    $x_{i+1} \leftarrow W_i y_i + b_i$ 
4:    $y_{i+1} \leftarrow s_{i+1}(x_{i+1})$ 
5: end for
6:  $dy_\ell \leftarrow \partial L(y_\ell; t_\ell) / \partial y_\ell$   $(t_\ell$  is the target)
7: for  $i$  from  $\ell - 1$  downto 1 do
8:    $dx_{i+1} \leftarrow dy_{i+1} s'_{i+1}(x_{i+1})$ 
9:   Set entries of  $\text{diag}([F])$  corresponding to  $W_i$  to be  $\text{sq}(dx_{i+1}) \text{sq}(y_i)^\top$ 
10:  Set entries of  $\text{diag}([F])$  corresponding to  $b_i$  to be  $\text{sq}(dx_{i+1}) \mathbf{1}_B^\top$ 
11:   $dy_i \leftarrow W_i^\top dx_{i+1}$ 
12: end for
13: output:  $\text{diag}(F)$ 
```

the gradient makes precisely one additive contribution to every parameter for each case. In this case, it is possible to add $[\nabla f_i]_j^2$ to the appropriate entry of $\text{diag}(F)$ as soon as it is computed, so we need not allocate temporary storage for $[\nabla f_i]_j$ for each i and j (rather, only each j).

However, when the computational graph of the derivative (for a given case i) makes multiple additive contributions to each $[\nabla f_i]_j$, it is necessary to allocate temporary storage for this quantity since we must square its total contribution *before* summing over the cases. Interestingly, this issue does not occur for the RNN's gradient computation, since without the per- i squaring, each contribution to $[\nabla f_i]_j$ can be stored in a single vector for all the i 's.

20.11.4 An Unbiased Estimator for the Diagonal of G

Chapelle and Erhan [10] give a randomized algorithm for computing an unbiased estimate of the diagonal of the generalized Gauss-Newton matrix, which requires the same amount of work as computing the gradient. And just as with any unbiased estimate, this approach can be repeatedly applied, and the results averaged, to achieve more precise estimates.

The method of Chapelle and Erhan is described in algorithm 20.7 below:

Algorithm 20.7. Computing an unbiased estimate of the diagonal of the GGN matrix

```

1: Sample  $v \in \mathbb{R}^m$  from a distribution satisfying  $\mathbb{E}[vv^\top] = I$ 
2: output  $\text{sq}\left(J^\top L''^{1/2} v\right)$ 
```

As discussed in section 20.6, multiplication of an arbitrary $v \in \mathbb{R}^m$ by the Jacobian J of F can be performed efficiently by the usual back-propagation algorithm. The correctness of algorithm 20.7 is easy to prove:

$$\begin{aligned}\mathbb{E} \left[\text{sq} \left(J^\top L''^{1/2} v \right) \right] &= \mathbb{E} \left[\text{diag} \left((J^\top L''^{1/2} v)(J^\top L''^{1/2} v)^\top \right) \right] \\ &= \text{diag} \left(J^\top L''^{1/2} \mathbb{E}[vv^\top] L''^{1/2} J \right) \\ &= \text{diag} \left(J^\top L''^{1/2} L''^{1/2} J \right) \quad (\text{as } \mathbb{E}[vv^\top] = I) \\ &= \text{diag}(J^\top L'' J) \\ &= \text{diag}(G)\end{aligned}$$

where we have used the identity $\text{sq}(x) = \text{diag}(xx^\top)$.

Martens et al. [24], following the work of Chapelle and Erhan [10], introduced an efficient unbiased approximation method for estimating the entire Hessian or GGN matrix of a given function (or just their diagonals, if this is desired) with a cost also comparable to computing the gradient. In the special case of estimating the diagonal of the GGN matrix, the two methods are equivalent. Of practical import, Martens et al. [24] also proved that sampling the components of v uniformly from $-1, 1$ will produce lower variance estimates than will be obtained by sampling them from $N(0, 1)$.

Computationally, the method of Chapelle and Erhan is very similar to the method for computing the diagonal of the Empirical Fisher Information that was described in the previous section. Indeed, while the latter compute $\text{sq}(J^\top \nabla L)$, this method computes $\text{sq}(J^\top L''^{1/2} v)$ for random v 's, and so the methods have similar implementations. In particular, both estimates can be computed in parallel over cases in the minibatch, and they share the issue with temporary stored discussed in the previous section, which can be overcome in for feed-forward networks but not RNNs.

In our experience, both methods tend to produce preconditioners with similar properties and performance characteristics, although Chapelle and Erhan [10] found that in certain situations this unbiased estimate gave better results. One clear advantage of this method is that it can correctly account for structural damping, which is not done by using the empirical Fisher matrix, as the gradients of the structural damping objective are equal to zero. The disadvantage of this method is that due to the stochastic nature of the curvature estimates, there could be parameters with non-zero gradients whose diagonal estimates could nonetheless be very small or even zero (which will never happen with the diagonal of the Fisher matrix). The diagonal of the Fisher matrix also has the additional advantage that it can be computed in tandem with the gradient at virtually no extra cost.

20.12 Minibatching

In modern machine learning applications, training sets can be very large, and a learning algorithm which processes all the examples in the training set to compute each parameter update (called “batch processing”) will likely be very slow or even totally impractical [7]. Some training datasets may even be infinite and so it may not even make any sense to talk about an algorithm operating in batch-mode at all.

“Online” or “stochastic” gradient algorithms like stochastic gradient descent (SGD) can theoretically use gradient information computed on arbitrarily small subsets of the training set, called “minibatches”, as long as the learning rate is sufficiently small. HF, on the other hand, uses minibatches to estimate the curvature matrix, which is used in a very strong way to produce large and sometimes aggressive parameter updates. These curvature matrix estimates may become increasingly low-rank and degenerate as the minibatch size shrinks (assuming no contribution from damping or weight-decay), which in some cases (see subsection 20.12.1) may lead to unbounded and nonsensical updates, although the damping mechanisms discussed in sec. 20.8 can compensate for this to some extent.

But even without these more obvious degeneracy issues, it can be argued that, intuitively, the matrix B captures “soft-constraints” about how far we can go in any one direction before making things worse, and if the constraints relevant to a particular training case are not well approximated in the curvature estimated from the minibatch, the update δ obtained from optimizing M could easily make the objective f worse on such a case, perhaps severely so.

Thus 2nd-order methods like HF which must estimate the curvature only from the current minibatch, may not work nearly as well with very small minibatches. And while there are several strategies to deal with minibatches that are “too small”, (as we will discuss in subsection 20.12.2), the benefits of using a 2nd-order method like HF may be diminished by their use.

Fortunately, in the case of neural networks, there are elegant and natural implementations which exploit data-parallelism and vectorization to efficiently compute gradients and curvature matrix-vector products over minibatches (see section 20.7). For highly parallel architectures like GPUs, these tend to give an increase in computational cost which remains sublinear (as a function of minibatch size) up to and beyond minibatch sizes which are useful in HF.

20.12.1 Higher Quality Gradient Estimates

Unlike 1st order optimization schemes like SGD where the number of iterations required to approach a good solution can reach as high as $10^5 - 10^7$, the number of iterations required by a strong 2nd-order optimizer like HF is in our experience orders of magnitude smaller, and usually around $10^2 - 10^3$. While the linear term $b = -\nabla f(\theta_{k-1})$ passed to CG needs to be computed only once for each update, CG may require on the order of $10^2 - 10^3$ matrix-vector products with

the curvature matrix $A = B$ to produce each update. These matrix-vector products are by far the most computationally costly part of any truncated Newton approach.

It may therefore be cost-effective to compute the gradient on a much larger minibatch than is used to compute the matrix-vector products. Martens [22] recommended using this technique (as does Byrd et al. [8]), and in our experience it can often improve optimization speed if used carefully and in the right contexts. But despite this, there are several good theoretical reasons why it might be better, at least in some situations, to use the same minibatch to compute gradient and curvature matrix-vector products. These have been corroborated by our practical experience.

We will refer to the minibatches used to compute the gradient and curvature matrix-vector products as the “gradient minibatch” and “curvature minibatch”, respectively.

When the gradient and curvature minibatches are equal, and the GGN curvature matrix is used, the quadratic model M maintains its interpretation as the local Taylor series approximation of a convex function, which will simply be the approximation of f obtained by linearizing F (see eqn. 20.9), but restricted to the data in the current minibatch. In such a situation, with some additional reasonable assumptions about the convex function L (strong convexity would be sufficient, but is more than what is needed), the quadratic model M can be written as:

$$\begin{aligned} M(\delta) &= \frac{1}{2}\delta^\top B\delta + \nabla f_{k-1}^\top \delta + f(\theta_{k-1}) = \frac{1}{2}\delta^\top J^\top L'' J\delta + \delta^\top (J^\top \nabla L) + f(\theta_{k-1}) \\ &= \frac{1}{2}(J\delta)^\top L''(J\delta) + (J\delta)^\top L''L''^{-1}\nabla L + \nabla L^\top L''^{-1}L''L''^{-1}\nabla L \\ &\quad - \nabla L^\top L''^{-1}L''L''^{-1}\nabla L + f(\theta_{k-1}) \\ &= \frac{1}{2}(J\delta + \nabla L)^\top L''(J\delta + \nabla L)^\top + c \\ &= \frac{1}{2}\|J\delta + L''^{-1}\nabla L\|_{L''}^2 + c \end{aligned}$$

where $c = f(\theta_{k-1}) - \nabla L^\top L''^{-1}\nabla L$ and all quantities are computed only on the current minibatch. Here we have used the fact that L'' is invertible, which follows from the fact that L is strongly convex.

This result is interesting because it applies only when B is the generalized Gauss-Newton matrix (instead of the Hessian), and it establishes a bound on the maximum improvement in f that the quadratic model M can ever predict: $\nabla L^\top L''^{-1}\nabla L$, a quantity which does not depend on the properties of the network, only on its current predictions and the associated convex loss function.

Such a boundedness result does not exist whenever M is estimated using different minibatches for the gradient and curvature. In this case, the estimated gradient may easily lie outside the column space of the estimated curvature matrix in the sense that there may exist directions d s.t. $g^\top d < 0$, $\|d\| = 1$, but $d^\top Bd = 0$. In such a case it is easy to see that the quadratic model is unbounded

and $M(\alpha d) \rightarrow -\infty$ as $\alpha \rightarrow \infty$. While boundedness can be guaranteed with the inclusion of damping penalty terms which ensure that the damped curvature matrix \hat{B}_k is positive definite, and will also be guaranteed when the curvature matrix B is full rank, it may be the case that the boundedness is “weak” in the sense that $d^\top Bd$ may be non-zero but extremely small, leading to a nearly degenerate update δ . For example, when using Tikhonov damping then we know that $d^\top \hat{B}_k d \geq \lambda$, but in order for this to sufficiently constrain the update along direction d , λ may have to be large enough that it would impose unreasonably high constraints on optimization in *all* directions.

More intuitively, the gradient represents a linear reward for movement in certain directions d (the strength of which is given by $g^\top d$) while the curvature matrix represents a quadratic penalty. If we include the linear rewards associated with a subset of cases without also including the corresponding quadratic penalties, then there is a chance that this will lead to a degenerate situation where some directions will have lots of reward (predicted linear reduction) without any corresponding penalty (curvature). This can result in an update which makes f worse even on cases contained in both the gradient and curvature minibatches, for reasons that have nothing directly to do with a breakdown in the reliability of the quadratic approximation to f . On the other hand, if the curvature and gradient minibatches are equal *and* the quadratic approximation to f is otherwise reliable (or properly damped), then using equal gradient and curvature minibatches provides a minimal guarantee that f will improve on the current minibatch after the proposed update is applied.

Another more subtle way in which using a smaller-sized curvature minibatch than gradient minibatch could be counterproductive, is that in addition to causing a dangerous underestimation of the curvature associated with the left-out cases, it may also lead to an *overestimation* of the curvature for the cases actually in the curvature-minibatch. This is because when we compute estimates of the curvature by averaging, we must divide by the number of cases in the minibatch, and since this number will be smaller for the curvature estimate than for the gradient, the gradient contributions from these cases will be smaller in proportion to the corresponding curvature terms.

Byrd et al. [8] showed that if the eigenvalues of the curvature matrices (estimated using any method) are uniformly bounded from below in the sense that there exists $\mu > 0$ s.t. $\hat{B} - \mu I$ is PSD for all possible \hat{B} 's which we might produce, then assuming the use of a basic line-search and other mild conditions, an truncated Newton algorithm like HF which estimates the gradient on the full training set will converge in the sense that the gradients will go to zero. But this result makes no use of the particular form of B and is as a result very weak, saying nothing about the rate of convergence, or how small the updates will have to be. As far as theorem is concerned, B can be any λ dependent matrix with the required boundedness property, and need not have anything to do with local quadratic models of f at all.

Despite all of these objections, the higher quality estimates of the gradient may nonetheless provide superior convergence properties in some situations. The

best trade-off between these various factors is likely to be highly dependent on the particular problem, the stage of the optimization (early versus late), and the damping mechanisms being used. Our experience is that penalty and CG-truncation damping become more active when there is a significant qualitative mismatch between the gradient and curvature estimates, which is more likely to happen when the training dataset, or the network’s responses to it, are particular “diverse”.

20.12.2 Minibatch Overfitting and Methods to Combat It

As mentioned in the previous section, the updates produced by HF may be effectively “overfit” to the current minibatch of training data. While a single update of SGD has the same problem, this is less of an issue because the updates are extremely cheap and numerous. HF, by contrast, performs a run of CG with anywhere between 10 to 300+ iterations, which is a long and expensive process, and must be performed using the same fixed estimates of the gradient and curvature from a single minibatch. Ideally, we could use a stochastic algorithm when optimizing the local quadratic models which would be able to see much more data at no extra cost. Unfortunately we are not aware of any batch methods which possess the same strongly optimal performance for optimizing quadratics as CG does, while also working well as a stochastic method.

The simplest solution to dealing with the minibatch overfitting problem is to increase the size of the minibatches, thus providing CG with more accurate estimates of the gradient and curvature. When optimizing neural networks, we have observed that the minibatch overfitting problem becomes gradually worse as optimization proceeds, and so implementing this solution will require continually growing the minibatches, possibly without bound. Fortunately, there are other ways of dealing with this problem.

The damping methods discussed in section 20.8 were developed to compensate for untrustworthiness of the local quadratic approximations M being made to f , which exists due to the simple fact that f is not actually a convex quadratic, and so M may fail to be a sensible approximation to f at its minimum δ^* . These methods work by imposing various soft or hard constraints on the update δ in order to keep it “closer” to 0 (where M trustworthy by construction), according to some metric.

Using minibatches to compute the gradient and curvature imposes a different kind of untrustworthiness on the quadratic model, arising from the fact that the function being approximated is not actually the true objective but rather just an unbiased sample-based approximation of it. But despite the differing nature of the source of this untrustworthiness, the previously developed damping methods turn out to be well suited to the task of compensating for it, in our experience.

Of these, decreasing the maximum number of CG steps, using larger minibatches for the gradient, and decreasing the default learning rate (which is equivalent to damping by adding multiples of B , as discussed in section 20.8.3) according to some SGD-like schedule, seem to be the most effective approaches in practice. If standard Tikhonov damping is used and its strength λ is increased to

compensate for minibatch overfitting, this will make HF behave asymptotically like SGD with a dynamically adjusted learning rate.

There is a compelling analogy between the minibatch overfitting which occurs when optimizing δ with CG, and the general overfitting of a non-linear optimizer applied to a conventional learning problem. And some of the potential solutions to both of these problems turn out to be analogous as well. Tikhonov damping, for example, is analogous to an L2 prior or “weight-decay” penalty (but centered at $\delta = 0$), and CG truncation is analogous to “early-stopping”.

Recall that damping approaches are justified as a method for dealing with quadratic approximation error because as the “size” of the update shrinks (according to any reasonable metric), it will eventually lie inside a region where this source of error must necessarily be negligible. This is due to the simple fact that any super-linear terms in the Taylor series expansion of f , which are unmodeled by M , will approach zero much more rapidly than the size of the update. It is important to keep in mind that a similar justification *does not* apply to the handling of minibatch-related estimation errors with update damping methods. Indeed, the negative gradient computed on a given minibatch may not even be a descent direction for the total objective (as computed on the complete training set), and even an infinitesimally small update computed from a given minibatch may actually make the total objective worse.

Thus, when tuning damping mechanisms to handle minibatch overfitting (either by hand, or dynamically using an automatic heuristic), one shouldn’t aim for obtaining a reduction in the total f that is a fixed multiple of that which is predicted by the minibatch-computed M (as is done in section 20.8.5), but rather to simply obtain a more modest reduction which is proportional to the relative contribution of the current minibatch to the entire training dataset.

It is also worthwhile noting that the practice of initializing CG from the update computed at the previous iteration of HF (as discussed in section 20.10) seems to bias CG towards finding solutions that generalize better to data outside of the current minibatch. We don’t have a good understanding for why this happens, but one possible explanation is that by carrying δ over to each new run of CG and performing an incomplete optimization on it using new data, δ is allowed to slowly grow (as HF iterates) along low-curvature directions¹¹ that by necessity, must generalize across lots of training data. The reasoning is that if such slowly optimized directions didn’t generalize well, then they would inevitably be detected as high-curvature ascent directions for some new minibatch and quickly zeroed out by CG before ever having a chance grow large in δ .

Finally, Byrd et al. [9] has developed methods to deal with the minibatch overfitting problem, which are based on heuristics that increase the minibatch size and also terminate CG early, according to estimates of the variance of the gradient and curvature-matrix vector products. While this is a potentially effective approach (which we don’t have experience with), there are several problems with it, in theory. First, variance is measured according to highly

¹¹ Which get optimized much more slowly by CG than high-curvature directions, as shown in section 20.9

parameterization-dependent metrics which are not particularly meaningful. Second, increasing the size of the minibatch, which is only one method to deal with minibatch overfitting, is not a strategy which will remain practical for very long. Thirdly, aggressive early termination heuristics for CG, similar to this one, tend to interact badly with non-zero CG initializations¹² and other forms of damping. And finally, there are other more direct ways of measuring how well updates will generalize, such as simply monitoring f on some training data outside of the current minibatch.

20.13 Tricks and Recipes

There are many things to keep in mind when designing an HF-style optimization algorithm for a particular application and it can be somewhat daunting even to those of us that have lots of experience with the method. So in order to make things easier in this regard, in this section we relate some of our experience in designing effective methods, and describe several particular setups that seem to work particularly well for certain deep neural network learning problems, assuming the use of a standard parameterization.

Common elements to all successful approaches we have tried are:

- use of the GGN matrix instead of the Hessian
- the CG initialization technique described in section 20.10
- a well-designed preconditioner. When using Tikhonov damping, a reasonable choice is an estimate of the diagonal of the GGN matrix, modified using the technique described in section 20.11.2 with $\kappa = 0$ and $\psi = 0.75$. When using one of the scale-sensitive methods described in section 20.8.3, it *may* be necessary to increase κ to something like 10^{-2} times the mean of the diagonal entries of D
- the use of one of diagonal damping methods, possibly in conjunction with structure damping for certain RNN learning problems. For feedforward network learning problems under the default parameterization, Tikhonov damping often works well, and usually so does using a modified estimate of the diagonal of the GGN matrix, provided that κ is large enough (as in the previous point)
- the use of the progress-based termination criterion for CG described in section 20.4 in addition to some other condition which may stop CG sooner, such as a fixed iteration limit
- dynamic adjustment of damping constants (e.g. λ) according to the LM heuristic or a similar method

Now, we describe the particulars of each of these successful methods.

First, there is the original algorithm described in [22], which works pretty well. Here, the “CG-backtracking” approach is used to select an iterate based on the objective function value (see section 20.8.7), the gradient is computed

¹² Because termination of CG may be forced before $\hat{M}(\delta) < 0$ is achieved.

on a larger subset of the training data than the curvature, and CG is always terminated before reaching a fixed maximum number of steps (around 50 – 250, depending on the problem).

Second, there is a subtle but powerful variation on the above method which differs only in how CG is terminated, how the iterate used for the update δ is selected, and how CG is initialized at the next iteration of HF. In particular, CG is terminated as soon as the objective function, as evaluated on the data in the “curvature minibatch” (see section 20.12.1) gets significantly worse than its value from some number of steps ago (e.g. 10). The iterate used as the parameter update δ is selected to minimize M (or perhaps f) as evaluated on some data which is *not* contained in the curvature minibatch. Lastly, CG is initialized at the next iteration $k + 1$, not from the previous update δ_k , but instead from the CG iterate which gave the highest objective function value on the curvature minibatch (which will be close to the last). In practice, the quantities used to determine when to terminate CG, and how to select the best iterate, do not need to be computed at every step of CG, and can also be computed on much smaller (but representative) subsets of data.

Finally, an approach which has emerged recently as perhaps the most efficient and effective, but also the most difficult to use, involves modifying HF to behaving more like a tradition momentum method, thus making stronger use of the CG initializations (see section 20.10) to better distribute work involved in optimizing the local quadratics across many iterations. To do this, we use a smaller maximum number of CG steps (around 25 to 50), smaller minibatches of training-data, and we also pay more attention to the CG initialization decay-constant ζ , which usually means increasing it towards the end of optimization. Using shorter runs of CG helps with minibatch overfitting, and makes it feasible to use smaller minibatches and also compute the gradient and curvature using the same data. And as discussed in section 20.12.1, computing the gradient on the same data as the curvature has numerous theoretical advantages, and in practice seems to result in a reduced need for extra damping, thus resulting in a λ that shrinks reliably towards 0 when adjusted by the LM heuristic. However, this method tends to produce “noisy” updates, which while arguably beneficial from the standpoint of global optimization, make it more difficult to obtain finer convergence on some problems. So when nearing the end of optimization, we adopt some of the methods described in section 20.12.2, such as lowering the learning rate, using shorter CG runs, increasing the minibatch size, and/or switching back to using larger minibatches to compute gradients (making sure to raise the damping constant λ to compensate for this) in order to achieve fine convergence.

One more piece of general advice we have is that using small amounts of weight-decay regularization can be highly beneficial from the standpoint of global optimization. In particular, to get the lowest training error possible, we have observed that it helps to use such regularization at the beginning of optimization only to disable it near the end. Also, using a good initialization is extremely important in regards to global optimization, and methods which work well for

deep networks include the sparse initialization scheme advocated in [22], and the method of Glorot & Bengio [12], and of course the pre-training techniques pioneered in [17].

20.14 Summary

We described various components of the Hessian-free optimization, how they can interact non-trivially, and how their effectiveness (or possibly harmfulness) is situation dependent. The main points to keep in mind are:

- for non-convex optimizations it is usually preferable to use the generalized Gauss-Newton matrix which is guaranteed to be PSD
- updates must be “damped” due to the untrustworthiness of the quadratic model
- there are various damping techniques that can be used, and their effectiveness depends highly on f and how it is parameterized
- truncating CG before convergence, in addition to making HF practical, can also provide a beneficial damping effect
- the strategy for terminating CG is usually a combination of progress-based heuristic and a hard-limit anywhere between 10 and 300+ (which should be considered an important meta-parameter)
- preconditioning can sometimes enable CG to make more rapid progress per step, but only if used correctly
- simple diagonal preconditioning methods tend to work well for feed-forward nets but not for RNNs
- preconditioning interacts non-trivially with certain forms of damping such as CG truncation, which must be kept in mind
- initializing CG from the update computed by the previous run of CG can have a beneficial “momentum-type effect”
- HF tends to require much larger minibatches than are used in SGD
- minibatch-overfitting can cause HF’s update proposals to be poor even for δ ’s where the quadratic model is trustworthy
- using more data to compute the gradients than the curvature matrix-vector products is a low-cost method of potentially increasing the quality of the updates, but it can sometimes do more harm than good
- minibatch-overfitting can also be combated using some of the standard damping methods, along with simply increasing the minibatch size
- structural damping works well for training RNNs, particularly on problems with pathological long-range dependencies
- exploiting data-parallelism is very important for producing an efficient implementation
- correctness of curvature matrix-vector products should be checked using finite difference methods
- the extra memory costs associated with the parallel computation of gradients and curvature matrix-vector products can be mitigated

The difficulty of customizing an HF approach for particular application will no doubt depend on the specifics of the model and the dataset. While in many cases a generic approach can be used to good effect, some more difficult problems like RNNs or feed-forward networks with non-standard parameterizations may require additional care. And even on “easier” problems, a better designed approach may allow one to surpass performance barriers that may have previously been mistaken for convergence.

This report has described many of the tricks and ideas, along with their theoretical justifications, which may be useful in this regard. While we can try to predict what combination of ideas will work best for a given problem, based on previous experience and/or mathematical/intuitive reasoning, the only way to be sure is with careful experimentation. Unfortunately, optimization theory has a long way to go before being able to predict the performance of a method like HF applied to the highly non-convex objectives functions associated with neural networks.

References

- [1] Amari, S.I.: Natural gradient works efficiently in learning. *Neural Computation* 10(2), 251–276 (1998)
- [2] Becker, S., Le Cun, Y.: Improving the convergence of back-propagation learning with second order methods. In: Proceedings of the 1988 Connectionist Models Summer School, pp. 29–37. Morgan Kaufmann, San Matteo (1988)
- [3] Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H.: Greedy layer-wise training of deep networks. In: Advances in Neural Information Processing Systems, vol. 19, p. 153 (2007)
- [4] Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks* 5(2), 157–166 (1994)
- [5] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., Bengio, Y.: Theano: a cpu and gpu math expression compiler. In: Proceedings of the Python for Scientific Computing Conference (SciPy), vol. 4 (2010)
- [6] Bishop, C.: Exact calculation of the hessian matrix for the multilayer perceptron. *Neural Computation* 4(4), 494–501 (1992)
- [7] Bottou, L., Bousquet, O.: The tradeoffs of large scale learning. In: Advances in Neural Information Processing Systems, vol. 20, pp. 161–168 (2008)
- [8] Byrd, R.H., Chin, G.M., Neveitt, W., Nocedal, J.: On the use of stochastic hessian information in optimization methods for machine learning. *SIAM Journal on Optimization* 21, 977 (2011)
- [9] Byrd, R.H., Chin, G.M., Nocedal, J., Wu, Y.: Sample size selection in optimization methods for machine learning. *Mathematical Programming*, 1–29 (2012)
- [10] Chapelle, O., Erhan, D.: Improved preconditioner for hessian free optimization. In: NIPS Workshop on Deep Learning and Unsupervised Feature Learning (2011)
- [11] Schraudolph, N.: Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent. *Neural Computation* 14 (2002)
- [12] Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS 2010). Society for Artificial Intelligence and Statistics (2010)

- [13] Gould, N.I.M., Lucidi, S., Roma, M., Toint, P.L.: Solving the trust-region subproblem using the lanczos method. *SIAM Journal on Optimization* 9(2), 504–525 (1999)
- [14] Hansen, P.C., O’Leary, D.P.: The use of the l-curve in the regularization of discrete ill-posed problems. *SIAM Journal on Scientific Computing* 14, 1487 (1993)
- [15] Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems (1952)
- [16] Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. *Neural Computation* 18(7), 1527–1554 (2006)
- [17] Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. *Science* 313(5786), 504–507 (2006)
- [18] Hochreiter, S.: Untersuchungen zu dynamischen neuronalen netzen. diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München (1991)
- [19] Hochreiter, S., Schmidhuber, J.: Long short-term memory. *Neural Computation* 9(8), 1735–1780 (1997)
- [20] Jaeger, H., Haas, H.: Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science* 304(5667), 78–80 (2004)
- [21] LeCun, Y., Bottou, L., Orr, G.B., Müller, K.-R.: Efficient BackProp. In: Orr, G.B., Müller, K.-R. (eds.) *NIPS-WS 1996*. LNCS, vol. 1524, pp. 9–50. Springer, Heidelberg (1998)
- [22] Martens, J.: Deep learning via hessian-free optimization. In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*, vol. 951 (2010)
- [23] Martens, J., Sutskever, I.: Learning recurrent neural networks with hessian-free optimization. In: *Proc. ICML* (2011)
- [24] Martens, J., Sutskever, I., Swersky, K.: Estimating the hessian by backpropagating curvature. In: *Proc. ICML* (2012)
- [25] Moré, J.J.: The levenberg-marquardt algorithm: implementation and theory. *Numerical Analysis*, 105–116 (1978)
- [26] Moré, J.J., Sorensen, D.C.: Computing a trust region step. *SIAM Journal on Scientific and Statistical Computing* 4, 553 (1983)
- [27] Nash, S.G.: Newton-type minimization via the lanczos method. *SIAM Journal on Numerical Analysis*, 770–788 (1984)
- [28] Nash, S.G.: A survey of truncated-newton methods. *Journal of Computational and Applied Mathematics* 124(1), 45–59 (2000)
- [29] Nesterov, Y.: A method for unconstrained convex minimization problem with the rate of convergence $\mathcal{O}(1/k^2)$. In: *Doklady AN SSSR*, vol. 269, pp. 543–547 (1983)
- [30] Nocedal, J., Wright, S.J.: *Numerical optimization*. Springer (1999)
- [31] Pearlmutter, B.A.: Fast exact multiplication by the hessian. *Neural Computation* 6(1), 147–160 (1994)
- [32] Shewchuk, J.R.: An introduction to the conjugate gradient method without the agonizing pain (1994)
- [33] Vinyals, O., Povey, D.: Krylov subspace descent for deep learning. *arXiv preprint arXiv:1111.4259* (2011)
- [34] Wengert, R.E.: A simple automatic derivative evaluation program. *Communications of the ACM* 7(8), 463–464 (1964)

Implementing Neural Networks Efficiently

Ronan Collobert¹, Koray Kavukcuoglu², and Clément Farabet^{3,4}

¹ Idiap Research Institute
Martigny, Switzerland

² NEC Laboratories America
Princeton, NJ, USA

³ Courant Institute of Mathematical Sciences
New York University, New York, NY, USA

⁴ Université Paris-Est
Équipe A3SI - ESIEE Paris, France

Abstract. Neural networks and machine learning algorithms in general require a flexible environment where new algorithm prototypes and experiments can be set up as quickly as possible with best possible computational performance. To that end, we provide a new framework called *Torch7*, that is especially suited to achieve both of these competing goals. *Torch7* is a versatile numeric computing framework and machine learning library that extends a very lightweight and powerful programming language Lua. Its goal is to provide a flexible environment to design, train and deploy learning machines. Flexibility is obtained via Lua, an extremely lightweight scripting language. High performance is obtained via efficient OpenMP/SSE and CUDA implementations of low-level numeric routines. *Torch7* can also easily be interfaced to third-party software thanks to Lua's light C interface.

Runtime efficiency is probably perceived as the most important topic when considering an efficient neural network implementation. One should however not under-estimate the time spent in designing the right neural network for a given task, or even the amount of work put into feeding data to the neural network properly. *Designing* the right network for a given task in a short amount of time requires a flexible development environment and a properly designed neural network toolbox.

Several efficient (in terms of runtime execution) neural network libraries for very specific needs are freely available. QuickNet¹ is a good example in the speech recognition community: it implements most commonly used algorithms, that is multi-layer perceptrons with few layers. However, flexible libraries are quite rare. It is not a trivial task to implement a library supporting a wide range of complex networks (such as convolutional networks for images, text or speech...), any type of connectivity (full connectivity, shared weights, order in layers...), or several type of training algorithms (stochastic gradient, batch, second order like

¹ <http://www.icsi.berkeley.edu/Speech/qn.html>

LBFGS...). It is even more difficult to find a unified environment where one can easily read, prepare, feed properly the data to the network, or debug the internals of the architecture (for example when the network is not training properly).

In Section 21.1, we will consider efficient neural network implementation in terms of *efficient environment*. We will then focus on the *runtime efficiency* and analyze different state-of-the-art approaches to speed-up the network training and testing phases in section 21.2. In this work, our analysis is built on the experience we acquired with our own neural network implementation, *Torch*², and more particularly the last version *Torch7*.

21.1 Efficient Environment

An efficient environment for implementing neural networks should not be only limited to neural networks themselves. It should *provide all necessary tools for efficient development of new numerical algorithms in general*. Often one needs various numerical algorithms to transform the data before feeding it to the neural network. Algorithms will strongly vary from one research domain to the other. Moreover, in the last few years, the research activity on neural networks started to intersect with many other domains like optimization, linear algebra, parallel processing to name a few. A successful framework should provide necessary tools to cope with the variability in the development process. Only in that case the framework would allow to not only easily investigate new types of models or new training algorithms, but also to easily compare or combine neural networks with other machine learning algorithms.

In order for a framework to provide necessary environment for development of new numerical algorithms, its *extension capabilities* should be very advanced. Machine learning researchers face many problems where there is need for using existing libraries. As we will see in Section 21.2, this includes interfacing efficient linear algebra libraries or even the neural network implementation itself. The ability to interface these existing libraries with as little runtime and code development overhead as possible is crucial for an efficient toolbox.

Finally, the *neural network toolbox implementation itself should be modular enough* to allow for the creation of any kind of new neural network models or implementation on different modalities of data, leaving the choice of the architecture as much as possible to the user.

In this section, we will cover the following three important points: efficiency of development environment, extension capabilities and modular neural network toolbox. The modular structure of *Torch7* that fuses advantages of high-level and low-level libraries is shown in Figure 21.1.

21.1.1 Scripting Language

A scripting (or interpreted) language is the most convenient solution for fast prototyping and development of new algorithms. At the same time, it is crucial

² <http://www.torch.ch>

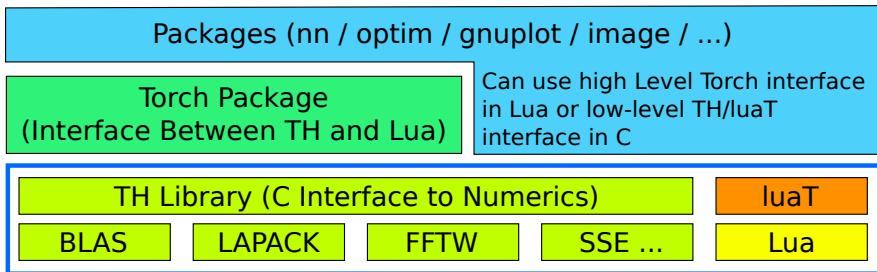


Fig. 21.1. Modular Structure of *Torch7*. Low level numerical libraries are interfaced with TH to provide a unified tensor library. luaT provides essential data structures for object/class manipulation in Lua. The core Torch package uses TH and luaT to provide a numerical computing environment purely in Lua. All other packages can use either Torch interface from inside Lua scripting environment or can interface low-level C interfaces for increased performance optimizations.

for the interpreted language to have a lightweight C API, both in terms of simplicity and efficiency. Simplicity in the C API encourages easier interfacing to existing external libraries and efficiency is the single most important criterion for large-scale applications.

In a complex development environment, the scripting language becomes the “glue” between heterogeneous components: different structures of the same concept (coming from different libraries) can be merged together using a high-level language, while keeping all the functionalities that are exposed from all the different libraries.

Lua. Among existing scripting languages³ finding the ones that satisfy *runtime efficiency* severely restricts the number of possibilities. In our machine learning framework *Torch7*, we chose Lua, the fastest interpreted language (with also the fastest Just In Time-JIT compiler⁴) we could find. Lua has also the advantage that it is designed to be easily *embedded* in a C application, and provides a *very clean* C API, based on a virtual stack to pass values and carry out function evaluation from C. This unifies the interface to C/C++ and minimizes the effort required for wrapping third party libraries.

Lua is intended to be used as a powerful, light-weight scripting language for any program that needs one. It is implemented as a library, written in pure C in the common subset of ANSI C and C++. Quoting Lua webpage⁵,

Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based

³ E.g. on <http://shootout.alioth.debian.org>

⁴ <http://luajit.org/>

⁵ <http://www.lua.org>

virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

Lua offers good support for object-oriented programming, functional programming, and data-driven programming. As shown in Figure 21.2, it handles numerical computations very efficiently (compared to C). This is a great asset for rapid implementation of new numerical algorithms. Lua’s main type is table, which implements associative arrays in a very efficient manner (see Figure 21.2). An associative array is an array that can be indexed not only with numbers, but also with strings or any other value of the language. Tables have no fixed size, can be resized dynamically, and can be used as “virtual tables” over another table, to simulate various object-oriented paradigms. Tables are the only data structuring mechanism in Lua, yet a powerful one. One can use tables to represent ordinary arrays, symbol tables, sets, records, queues, and other data structures, in a simple, uniform, and efficient way. Lua uses tables to represent packages as well. In addition, functions are first class citizens of the language. A function, just like any other variable can be passed as a variable to or returned from a function. Last, but not the least, Lua supports closures. Combined with tables, closures provide a very powerful and efficient syntax for data handling and programming complicated algorithms.

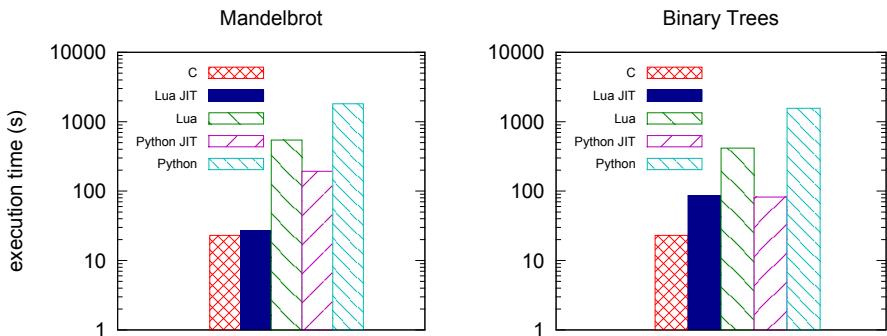


Fig. 21.2. Comparison of runtime efficiency of the C language (with gcc 4.4), Lua 5.1.4 and Python 2.7. Lua and Python JIT implementations were LuaJIT and PyPy, respectively. The Mandelbrot and Binary Trees benchmarks are taken from “The Computer Language Benchmarks Game”. All benchmarks were run using a single CPU on a high-end 12 cores Xeon server. The Mandelbrot benchmark makes a heavy use of numbers, while the Binary Trees benchmark makes a heavy use of data structures (struct in C, tables in Python and Lua). The execution time is reported (on a log-scale axis) for each language.

Why Not Python? It is hard to talk about a programming language without starting a flame war. While Lua is well known in the gaming programmer community, mostly due to its speed advantage and great embedding capabilities,

Python is more popular in more general public. With no doubt, Python ships with more libraries and one can find support about almost any problem easily in many different contexts. However, Lua offers at least two important advantages over Python:

- First and foremost, the simplicity of integrating existing C/C++ libraries is very important. Many efficient numerical algorithms are implemented in specialized packages in BLAS, LAPACK, FFTW and similar libraries. A lightweight interface to existing code is crucial for achieving a high performance environment. In section 21.2.8 we show quantitative results on efficiency of Lua compared to Python when wrapping BLAS function calls.
- Second, since Lua is embeddable in C/C++, any prototyped application can be turned into a final system/product with very little extra effort. Since Lua is written in pure C and does not have dependency to any external library, it can be easily used in embedded applications like, Android, iOS⁶, FPGAs⁷ and DSPs.

There are also alternatives to writing a custom interface between interpreted language and C/C++, like Simplified Wrapper and Interface Generator (SWIG)⁸. Although these might provide a simplified interface at first, writing a tensor library with several linear algebra backends requires a very fine-grained control and we found it is harder to manage this interface rather than using the native Lua API.

In addition to its performance advantage on number of operations (see Figure 21.2), Lua also provides other unique advantages – rarely found simultaneously in existing programming languages – for implementing a large-scale machine learning framework. In the following section we will show how we extended Lua’s basic numerical capabilities to a rather complete framework for developing complex numerical algorithms.

21.1.2 Multi-purpose Efficient N-Dimensional Tensor Object

Torch7 provides a generic Tensor library (called TH) that is written in pure C. This library is interfaced in Lua, providing new efficient multi-dimensional array types in the scripting language. Most packages in *Torch7* (or third-party packages that depend on *Torch7*) rely on this Tensor class to represent signals, images, videos..., allowing Lua to nicely “glue” most libraries together. Fast prototyping and creation of new packages is made possible, as the library is available directly from both Lua and C. Interfacing or extending existing libraries is very efficient. The following code demonstrates a few standard Tensor-based operations, from the Lua side:

⁶ <https://github.com/clementfarabet/torch-ios>

⁷ <http://www.neuflow.org>

⁸ www.swig.org

```

1 -- create a tensor of single-precision floats
2 t = torch.FloatTensor(100,100)
3
4 -- randomized: sampled from a normal distribution
5 l = torch.randn(100,100)
6
7 -- basic operators
8 r = t + l/2
9
10 -- in-place operators
11 r:add(0.5, t)
12
13 -- common operators
14 r = torch.log(torch.exp(-r)+10)

```

As in Matlab, multiple types can co-exist in *Torch7*, and it is easy to cast from one to the other:

```

1 -- a single-precision tensor
2 tfloat = torch.FloatTensor(100)
3
4 -- converted to double-precision
5 tdouble = tfloat:double()
6
7 r = torch.FloatTensor(tdouble:size())
8
9 -- automatically casts from double->float
10 r:copy(tdouble)

```

A sample matrix, matrix multiplication operation is done as in the following example.

```

1 x = torch.Tensor(1000,5000)
2 y = torch.Tensor(5000,3000)
3 z = torch.mm(x,y)
4 print(z:size())
5
6 1000
7 3000
8 [torch.LongStorage of size 2]

```

The *Torch7* Tensor library provides a lot of classic operations (including linear algebra operations), efficiently implemented in C, leveraging SSE instructions on Intel's platforms and optionally binding linear algebra operations to existing efficient BLAS/Lapack implementations (like Intel MKL, OpenBLAS or ATLAS). As we will see in the next section, we also support OpenMP instructions and CUDA GPU computing for certain subset of operations where these platforms offer unique performance advantages.

Related Approaches. Our Tensor library implementation got mostly inspired from SN [3] and Lush [5] toolboxes, which were one of the first to introduce

the concept (in a LISP language framework). Matlab also supports N-dimension arrays (even though early releases only supported 2D matrices). Compared to Matlab, we put big emphasis on *memory allocation control*, as we will see in Section 21.2.2. Numpy⁹ is another popular alternative, but only available for the Python language. As mentioned before, Lua offers unique advantages for a machine learning framework because of its speed and the simpler C interface.

21.1.3 Modular Neural Networks

Following [6], we view a neural network as a set of modules connected together in a particular graph. In *Torch7*, the “nn” package provides a set of standard neural network modules, as well as a set of container modules that can be used to define arbitrary directed (acyclic or not) graphs. By explicitly describing the graph’s architecture, using pluggable modules, we avoid the complexity of a graph parser, or any other middle-ware compiler. In practice, most networks are either sequential, or have simple branching patterns and recursions. The following example shows how to describe a multi-layer perceptron:

```

1 mlp = nn.Sequential()
2 mlp:add(nn.Linear(100,1000))
3 mlp:add(nn.Tanh())
4 mlp:add(nn.Linear(1000,10))
5 mlp:add(nn.SoftMax())

```

Each module (or container) provides standard functions to compute its output state, and back-propagate derivatives to its inputs, and to its internal parameters. Given the previous network, an input X , and the gradient of some error E with respect to the output Y — dE/dY —these three functions can be called like this:

```

1 -- compute the activations Y = f(X)
2 Y = mlp:updateOutput(X)
3
4 -- compute some loss E = l(Y,T)
5 E = loss:updateOutput(Y,T)
6
7 -- compute the gradient dE/dY = dl(Y,T)/dY
8 dE_dY = loss:updateGradInput(Y,T)
9
10 -- back-propagate the gradients, down to dE/dX
11 dE_dX = mlp:updateGradInput(X,dE_dY)
12
13 -- compute the gradients wrt the weights: dE/dW
14 mlp:accGradParameters(X,dE_dY)

```

The “nn” package in *Torch7* provides about 80 different neural network modules, allowing the user to implement most existing neural networks with minimal effort in pure Lua.

⁹ <http://numpy.scipy.org>

Leveraging the TH Library. Neural network modules in *Torch7* use Tensors provided by the TH library (see Section 21.1.2) to represent their own input data, output or internal states. Most modules are simply written in Lua, using the Torch package for intensive numerical operations. Only packages which require very specific operations have a dedicated C back-end. And, even in this case many of them use the TH library interface from C. In any case, Tensors are used as data containers to interact seamlessly with the rest of the library.

Training Algorithms. In *Torch7*, every neural network module, given the partial derivatives with respect to its outputs, is able to compute the partial derivative with respect to its parameters and its inputs. Thus, any complicated network structure can be trained using gradient-based optimization methods. Batch, mini-batch and stochastic gradient descent algorithms are supported. More advanced algorithms, such as second-order gradient descent algorithms like conjugate gradient or LBFGS are also possible, thanks to a numerical package called “optim”. While this optimization package is designed to be used stand-alone, it also provides second-order optimization capabilities for neural networks when used with the “nn” package.

21.1.4 Additional Torch7 Packages

Torch7 comes with many built-in and third-party packages. In order to encourage collaborations and redistribution of machine learning algorithms, a built-in package manager is provided. It can easily download, compile and install additional *Torch7* packages from any package repository, when needed. At this time, the most interesting packages related to numerical computation or numerical analysis are:

- **torch:** *Torch7*’s main package: provides Tensors, easy serialization and other basic functionalities. This package provides, Matlab-like functions to create, transform and use Tensors.
- **gnuplot:** This package provides plotting interface to Gnuplot using Tensors.
- **image:** An image processing package. It provides all the standard image processing functions such as loading and saving images, rescaling, rotating, converting color spaces, filtering operations, ...
- **optim:** A compact package providing steepest descent, conjugate gradient and limited memory BFGS implementations.
- **qt:** Full bindings between Qt and Lua¹⁰, with transparent conversion of *Torch7* Tensors from/to QImages. Great for quickly developing interactive demos with advanced GUIs (running natively on Linux, Mac or Windows platforms).

The list of available packages is constantly growing, as Lua makes it easy to interface any C library. Third-party packages include: *unsup*, which contains several unsupervised learning algorithms like sparse coding and auto encoders.

¹⁰ Thanks to Léon Bottou for this huge piece of work.

mattorch, which provides a two-way interface between Matlab’s matrix format and Torch’s tensor; *parallel*, which provides simple routines to fork and execute Lua code on local or remote machines, and exchange data using *Torch7*’s serialization mechanism; *camera*, a simple wrapper to camera/webcam drivers on Linux and Mac OSX; *imgraph*, a package that provides lots of routines to create edge-weighted graphs on images, and manipulate these graphs.

21.2 Efficient Runtime Execution

Torch7 has been designed with efficiency in mind, leveraging SSE when possible and supporting two ways of parallelization: OpenMP and CUDA. The Tensor library (which is interfaced to Lua using the “torch” package) makes heavy use of these techniques. From the user viewpoint, enabling CUDA and OpenMP can lead to great speedups in any “Lua” script, at zero implementation cost (because most packages rely on the Tensor library). Other packages (like the “nn” package) also leverage OpenMP and CUDA for more specific usages not covered by the Tensor library. In the following we explain specific advantages of *Torch7* for achieving an excellent runtime performance.

21.2.1 Float or Double Representations

One of the major computational bottlenecks of modern computers is their memory bandwidth. When implementing any numerical algorithm, the number of memory accesses should be always reduced by all means. This has an impact not only on the coding style, but also on the floating point type we will choose when implementing neural networks. A C “double” takes usually 8 bytes in memory, while C “float” takes only 4. Given that high precision is rarely required in neural networks, one might consider using floating point precision in most cases. On a simple matrix-matrix operation, speedups of $\times 2$ are common when using floats instead of doubles. In practice similar speedups are also observed in neural networks using floating point precision. In that respect, in *Torch7*, the user can easily choose (at runtime) the default floating point type.

21.2.2 Memory Allocation Control

One of the main complaints about using high level interfaces (such as Matlab) for numerical programming is the loss of control over memory allocation. The high-level abstraction makes it very hard for the researcher to know when a copy of a tensor is created. Although this is not a major problem for small-scale applications, as the data size grows, repeated copy and memory allocation might become problematic and even a bottleneck for the algorithm. To avoid such problems, *Torch7* tensor library is designed to support complete control over new memory allocation only when the user wants to use it. To better demonstrate this point, we repeat the matrix multiplication example.

```

1 x = torch.Tensor(1000,5000)
2 y = torch.Tensor(5000,3000)
3 z = torch.mm(x,y) print(z.size())
4
5 1000
6 3000
7 [torch.LongStorage of size 2]

```

One can see that the tensor z , which did not exist before, is newly allocated in this context. One can imagine that these series of operations are done repeatedly inside a loop. In this case, *Torch7* allows the following intuitive syntax.

```

1 x = torch.Tensor(1000,5000)
2 y = torch.Tensor(5000,3000)
3 z = torch.Tensor(1000,3000)
4 torch.mm(z,x,y)

```

As it can be seen from the example, the *torch.mm* function also can take three arguments, in which case the first argument becomes the result of the operation. This syntax is implemented for all operations in the Tensor library consistently, so that for every single operation, the user has the choice of passing in the target Tensor or allocating a new one without any overhead and heavy syntax. For example the following element-wise *Sin* operation can be represented in two different ways.

```

1 x = torch.rand(1000)
2
3 -- a new tensor is created
4 tsin = torch.sin(x)
5
6 -- a scalar one is added to tensor x (x is reused)
7 x:add(1)
8
9 -- tsin is reused
10 torch.sin(tsin,x)

```

In this example, both scalar addition to tensor x and calculating the *Sin* of resulting tensor did not allocate any new memory. In the above example, we also hinted another use of tensor library, where one can make method calls on a tensor object, as in any object oriented language. This syntax makes it explicit that the operation is directly applied on the object that the method call is done.

21.2.3 BLAS/LAPACK Interfaces

The key to a successful numerical computation framework is to have efficient implementations of linear algebra operations. This requires highly sophisticated algorithms with very precise implementations. In order to be able to provide the best experience, the C tensor library (TH) that is included in *Torch7* interfaces BLAS and LAPACK libraries.¹¹ All the major Level 1, 2 and 3 BLAS

¹¹ <http://www.netlib.org>.

operations like matrix-vector products, matrix-matrix products and most major linear algebra routines like singular value decomposition, matrix inverse, least square solutions are interfaced to BLAS and LAPACK libraries respectively. This interface provides the user with a rich experience of building block operations where higher level algorithms can easily be implemented.

21.2.4 SIMD Instructions

Most computations involved in a neural network consist in applying the same type of operations over (possibly large) vectors or matrices. Several CPU architectures, such as PowerPC, Intel or ARM support SIMD (Single Instruction, Multiple Data) operations which are perfectly suited for this kind of task: for example with SSE (Streaming SIMD Extensions) on Intel processors one might perform 4 additions over a vector with a unique instruction. Calling these instructions instead of regular CPU instructions might lead to great speedup. This type of optimization is unfortunately CPU-specific. Fortunately, in many cases one can rely on BLAS/LAPACK implementations specialized for a given platform, which leverage SIMD instructions. For other neural network specific cases, such as convolutions, one must implement specialized routines for each architecture of choice. In *Torch7*, we try to leverage SSE (on Intel architectures) and NEON (on ARM architectures) whenever possible. Compared to a non-SSE implementation $1.5 \times$ speedup are common, as shown in Figure 21.3 in the case of convolutional neural networks.

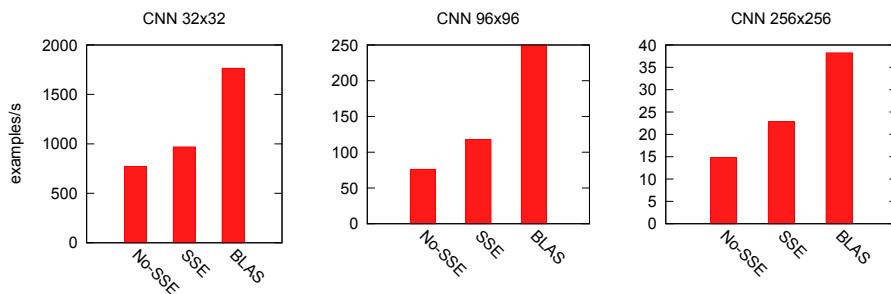


Fig. 21.3. Comparison of several convolutional neural network implementations (without SSE, with SSE or with BLAS). Tests were conducted using one core on a Intel bi-Xeon X5690 server. Performance is given in number of examples processed by second (higher is better). Three different architectures were tested, with input image sizes of 32x32, 96x96 and 256x256 respectively.

21.2.5 Ordering Memory Accesses

As already mentioned in Section 21.2.1 and Section 21.2.2, memory accesses are one of the main bottleneck on today's computers. In fact, not only the *number*

of accesses is important, but also the *order* of these accesses. For example, operations with tensors not contiguous in memory (say, with extra jumps between each element of the tensor) should always be avoided. In many cases, it is better to organize the tensor in a contiguous memory block (possibly at the cost of the copy), before performing any intensive computations. A striking example for neural networks is convolutions. When performing a convolution over an image, successive dot products are done between the kernel and all possible patches of the image. One can create a copy of all these patches beforehand (the drawback being a huge memory cost for large convolutions or large images) and then apply a matrix-matrix operation (using BLAS) to compute all dot products. The memory consumption increases proportional to the number of pixels of convolutional kernel. As shown in Figure 21.3, this leads to unbeatable runtime performance, even though the initial memory copy is quite large. *Torch7* provides this implementation as part of the neural net package too. Whenever there is sufficient memory available, it is advantageous to use this implementation which uses an innovative design that takes advantage of multi-core CPU architectures.

21.2.6 OpenMP Support

Open Multi-Processing (OpenMP) provides a shared memory CPU parallelization framework on C/C++ and Fortran languages on almost every operating system and compiler toolset. It generally requires minimal modification for integrating into an existing project. *Torch7* is designed and developed to use OpenMP directives for various operations in its tensor library and neural network package. Although the details of the OpenMP specification is beyond the scope of this work, below we show one of the most commonly used OpenMP directive, parallelization over for-loops:

```

1 // private makes a copy for each thread
2 #pragma omp parallel for private(i)
3 for (i=0; i<N; i++)
4 {
5     a[i] = i*i;
6 }
```

Without the **omp parallel for** directive at line 2, this piece of code will run to completion using a single thread. However, since each loop iteration is independent from each other, it becomes a trivial single line addition to existing code that parallelizes this computation over many cores.

Torch7 automatically detects if the compiler supports OpenMP directives and compiles a high level package that adds multi-threaded tensor operations, convolutions and several neural network classes. The switch from single threaded code to multi-threaded code is completely transparent to the user and it only requires *-l openmp* argument to be passed to torch executable. With this option, *Torch7* by default uses the OpenMP enabled function calls when available. The number of threads to be used can be specified by either setting the “OMP_NUM_THREADS” environment variable to desired number:

```
1 bash# export OMP_NUM_THREADS=4
```

or from inside lua by

```
1 torch.setNumThread(4)
```

function. Moreover, openmp can even be temporarily enabled or disabled using the following function calls.

```
1 torch.setNumThreads(1)
2 torch.setNumThreads(N)
```

Multi-threading of BLAS operations rely on the specific BLAS library that *Torch7* is linked against. For example Intel's MKL library also uses OpenMP for parallelizing Level 3 BLAS operations. In the neural network package *nn*, the convolutional layers, most common non-linearity functions like tanh and sigmoid, pooling operations like average, sum and max pooling and various other primitive operations like sum, square modules are all parallelized. For all the models that apply element-wise operations, the parallelization is almost as trivial as shown in the example above. For more complicated modules like convolutional layers with multiple input output feature maps, the function evaluation pass is parallelized over output feature maps so that every output feature is calculated in parallel. For calculating the gradient wrt kernels, operations are parallelized over kernels and over input feature maps for gradient wrt inputs. Using this strategy the convolutional network architecture can be sped up almost linearly.

21.2.7 CUDA Support

CUDA (Compute Unified Device Architecture) is nVidia's framework for programming their graphics processors to perform general purpose computations. CUDA exposes the hierarchy of memories available to the graphics processor, the two main ones being the external (large, high-latency) DRAM and the internal shared memory (a couple of kB, low-latency). It also exposes the hierarchy of compute cores, and how they interact with each other, and with the different types of memory.

Contrary to common belief, we found that writing CUDA code (kernels) can be significantly simplified. It is very easy to obtain decent performance, and the simplest kernels already yield satisfying speedups over regular C. The only three things to know, and carefully handle are: understanding the interaction between shared memory and threads; understand memory coalescing, to maximize bandwidth to/from external DRAM; understand the hierarchy of processing units, to efficiently divide the workload between blocks and threads. Once understood, these concepts were sufficient to allow us to write our own 2D convolutions, which are computed at about 200GFLOP/s on a GTX580, for large enough inputs. For smaller inputs, our OpenMP+SSE implementation remains more efficient. It is worth mentioning that *Torch7* employs an efficient, yet general method for implementing a wide variety of CUDA kernels. As it is shown in the upcoming

sections, this strategy results in the best performance in most cases. However, it is also possible to achieve superior performance by developing CUDA kernels under specific assumptions, like particular input or operator shape and sizes. Despite the performance advantage, these cases generally require significant development effort and produce modules that can not be reused, thus they are not suitable for a general machine learning library.

Once built with CUDA, *software!**Torch7* provides a new Tensor type: `torch.CudaTensor`. Tensors with this particular type lives in the GPU's DRAM memory. All operators defined on standard Tensors are also defined on CudaTensors, which completely abstracts the use of the graphics processor. Here is a small illustrative example, that demonstrates the simplicity of the interface:

```

1  -- lives in the CPU's DRAM
2  tf = torch.FloatTensor(4,100,100)
3
4  -- lives in the GPU's DRAM
5  tc = tf.cuda()
6
7  -- performed by the GPU
8  tc:mul(3)
9
10 -- res lives in the CPU's DRAM
11 res = tc.float()
```

On top of the Tensors' main operators, all the matrix-based operators are available, as well as most standard convolution routines.

21.2.8 Benchmarks

In this section we analyze the efficiency of *Torch7* in two different setups: first in the framework of a matrix-matrix multiplication benchmark, then when training various neural networks. To that effect, we compare *Torch7* with *Numpy* and *Theano*. We chose *Numpy* as a reference because it is a widely-used numerical library for Python, the latter being itself a widely-used scripting language. *Theano* [1] is a recent compiler for mathematical expressions, built upon Python and *Numpy*, and which has been shown as over-performing many neural network implementations, which makes it a very relevant baseline. In our experiments we chose the latest version of each software, that is *Theano* 0.5, *Numpy* 1.6.1 and *Scipy* 0.10.1.

Measuring the Overhead of Interpreted Languages. The majority of the computation for neural networks and many numerical algorithms is spent in BLAS calls for performing linear algebra operations. To that end, we demonstrate the efficiency of the *Torch7* and also the underlying C library TH.

The *Torch7* numerical routines follow a simple design that contains layers. The first layer is an efficient C library that provides a high level tensor package (TH). TH library provides a templated design that enables the choice of different precisions. Available types are, Byte (unsigned char), Char (char), Short

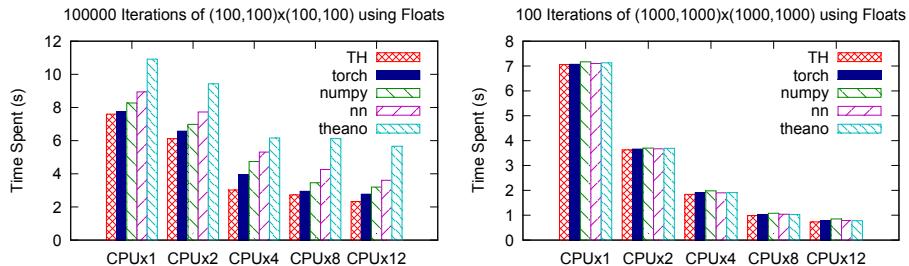


Fig. 21.4. Benchmarks of matrix multiplication performance using C, *Torch7* torch package, nn package in *Torch7*, Numpy and Theano. Tests were conducted on a machine with two Intel Xeon X5690 CPUs with 6 computational cores in each CPU. Hyper-threading was disabled. We considered multi-thread computation using 1, 2, 4, 8 and 12 CPU cores. Performance is given in seconds of time spent for processing, therefore smaller is better.

(16 bit integer), Integer (32 bit integer), Long (64 bit integer), Float (32 bit floating point) and Double (64 bit floating precision). TH library does not have any dependencies to Lua or any other language other than standard C libraries, therefore it is also suitable to be used by other projects that rely on efficient numerical routines. The choice of C language was a careful choice as with Lua. Since TH uses only C, it can be compiled in almost any programming environment like cellphones, DSP, embedded systems, etc. TH provides interface to many BLAS operations, but also contains hand-coded operations for all functions in case no BLAS library is available. It also provides and interface to several most widely used LAPACK routines for linear algebra. The second layer on top of TH is the *torch* package that integrates TH into Lua. All of the TH mathematical operations are interfaced from the Lua language in the *torch* packages. Finally, the *nn* package uses the *torch* package to provide a modular, yet fast and efficient, neural network library.

One might argue that such a layered approach would introduce quite a bit of overhead. In order to quantify the overhead coming from each layer, we selected matrix-matrix multiplication as our test case since it is one of the most widely used operations in linear algebra and ran tests using different sizes of matrices and different layers of programming. We used 100×100 and 1000×1000 , matrices and benchmarked using a C only program that directly uses TH library, using torch library, using linear layer (with no bias) from nn package in Torch7. We also included tests using Numpy package and finally Theano. We compiled all packages using Intel MKL library to be able achieve the best possible performance and maximize the advantages of using CPU threading. As it can be seen from the results given in Figure 21.4, the overhead coming from TH, *Torch7* or *nn* libraries is minimal, even for small size matrices. Even though Python gets a bit more overhead, for larger matrices the overhead is minimal in all configurations.

Comparing Machine Learning Packages. In a recent paper [1], the authors introduced a new compiler for mathematical expressions, built upon Python and Numpy. As for *Torch7*, Theano is (at this time) mainly used in a neural network framework. Theano can be either run on a CPU or a GPU. The authors of Theano showed benchmarks (involving the training of various neural networks architectures) comparing with other alternative implementations (when running Theano over a GPU), including Torch5, Matlab with GPUmat (running over a GPU) or EBLearn¹². Below, we reproduce these exact benchmarks, limiting ourselves to *Torch7* versus Theano, as Theano appears already faster than any existing implementation.

Table 21.1. Convolutional Network Architectures used in the benchmark study

	32×32	96×96	256×256	# F. Maps
1.c Convolution	5×5	7×7	7×7	6
1.p Max-pooling	2×2	3×3	5×5	6
2.c Convolution	5×5	7×7	7×7	16
2.p Max-pooling	2×2	3×3	4×4	16
3.l Linear			120 output features	
4.o Linear			10 output features	

For a fair comparison, we compiled both Numpy and SciPy (on which Theano relies) and *Torch7* against MKL Intel library. Latest versions of Theano also support direct link against MKL for certain operations (without passing by Numpy), which we setup carefully. We ran the experiments on a Intel Xeon X5690 with 12 cores. We optionally used a nVidia Tesla M2090 GPU. Following [1] benchmark suite, we considered the training of three kinds of multi-layer Perceptrons. **1.** 784 inputs, 10 classes, cross-entropy cost, and respectively no-hidden layer. **2.** One hidden layer of size 500. **3.** Three hidden layers of size 1000. We also considered the training of three kinds of convolutional neural networks (as shown in Table 21.1) on 32×32 , 96×96 , and 256×256 input images, following exactly the architectures given in [1]. The optimization algorithms we used were pure stochastic gradient descent (SGD) and SGD with a mini-batch of 60 examples. We compare all architectures running on a single CPU core, over multiple cores using OpenMP, or on the GPU. Note that Theano does not support OpenMP. However, it gets a speedup (on the multi-layer Perceptron benchmarks), since the Intel MKL library (called through Numpy) supports multiple threads using OpenMP.

As shown in Figure 21.5, *Torch7* is faster than Theano on most benchmarks. Interestingly, Theano underperforms for small architectures using pure SGD training (left column in Figure 21.5), which might be explained by a Python overhead, as mentioned in the previous section. Another interesting comment is

¹² <http://www.eblearn.sf.net>

the surprising performance of OpenMP implementations compared to the GPU implementation. As it can be seen from the graphs only largest network architectures will benefit from using the GPU. It is also worth mentioning that for CNN with 32×32 inputs using batch training, Theano's GPU implementation is superior than *Torch 7*. Under certain conditions, GPU optimizations might pay off by providing significant speed-ups, however they also require significant development effort for covering a small input domain. For CNN experiments a second *Torch7* benchmark, TorchMM is included. In this case matrix-matrix product operations for performing convolutions as explained in section 21.2.5 are used. It can be seen that this implementation significantly outperforms other models from *Theano* and *Torch7*, including GPU implementations.

21.3 Efficient Optimization Heuristics

As pointed in Chapter 18, the size of datasets have grown faster than the speed of processors in the last couple of years. When estimating the parameters of a neural network, it is then crucial to use an optimization procedure that can scale accordingly. Recently, research on optimization methods for neural networks has become an important topic [2, 4, 8, 7]. *Torch 7* provides a flexible framework designed particularly to make it easy for developing optimization algorithms on neural networks.

Let us consider the case of supervised learning, when one has a training set of N examples (x_n, y_n) , with x_n an observed input vector and y_n an output target vector that we wish to predict. We consider a loss function $l(\hat{y}_n, y_n)$ that measures the cost of predicting \hat{y}_n when the actual answer is y_n . We also consider a predictor $f_{\mathbf{w}}(x_n)$, with trainable parameters \mathbf{w} . The task of learning can be defined as finding the vector \mathbf{w} that minimizes the loss function L over the entire training set:

$$L(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N l(f_{\mathbf{w}}(x_n), y_n), \quad (21.1)$$

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} L(\mathbf{w}). \quad (21.2)$$

This general form of loss minimization can be easily carried out using one of a variety of optimization methods like Conjugate Gradient Descent (CG), BFGS or Limited Memory BFGS, Levenberg-Marquardt methods or simple SGD. In *Torch7*, these heuristics and methods can be carried out simply, using one unifying idea: decoupling the form of the function $f_{\mathbf{w}}$ from the optimization procedure. By grouping all the trainable parameters into a single parameter vector and using a vector of gradients of the same size for gradients, the type and shape of the neural network is completely abstracted from the developer. Combined with powerful closure mechanism of Lua, one can develop optimization algorithms

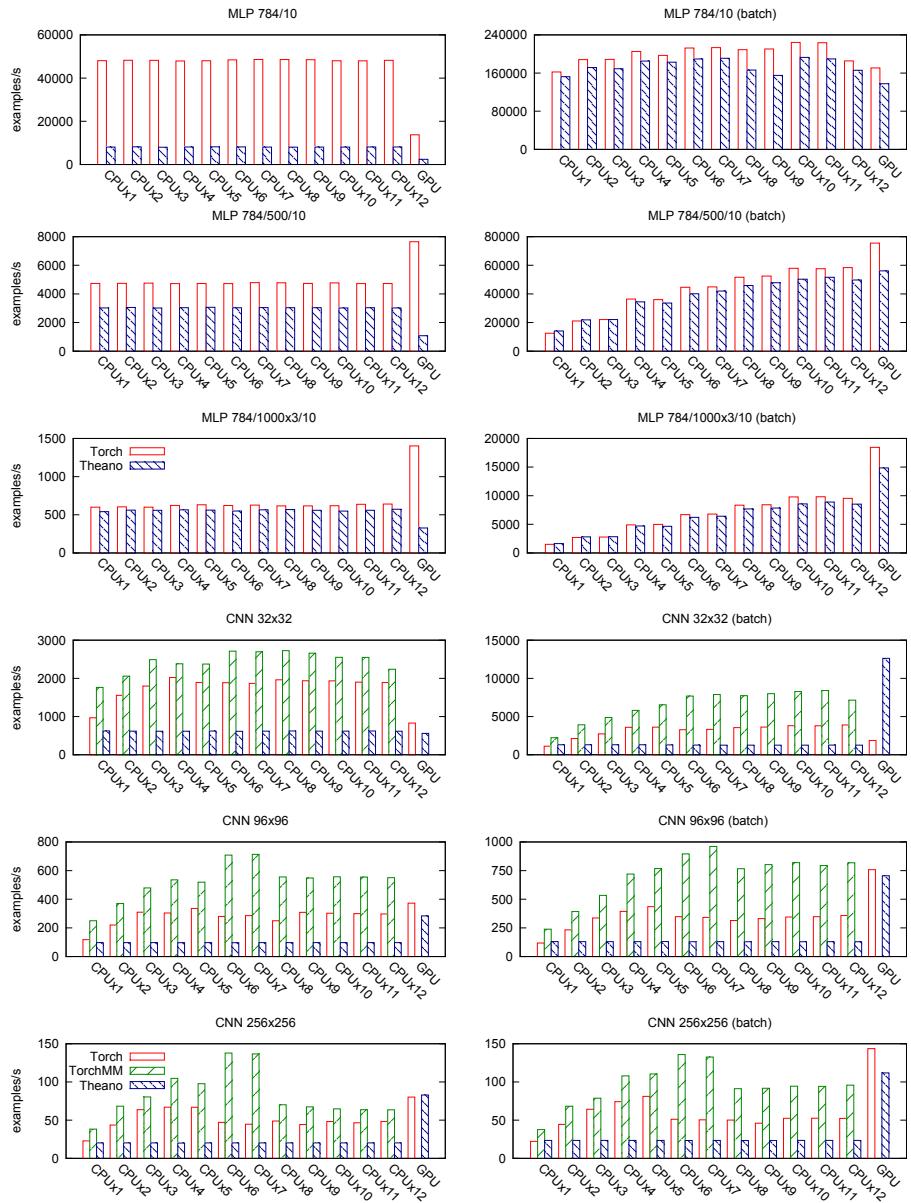


Fig. 21.5. Benchmarks of *Torch7* versus *Theano*, while training various neural networks architectures with SGD algorithm. Tests were conducted on a machine with two Intel Xeon X5690 CPUs and Nvidia M2090 GPU. We considered multi-thread computation using 1 to 12 CPU cores using OpenMP and GPU with Nvidia CUDA interface. Performance is given in number of examples processed by second (higher is better). “batch” means 60 examples at a time were fed when training with SGD. *TorchMM* uses the convolutional neural network layer implementation introduced in Section 21.2.5.

for most complicated neural network models as easy for the simplest ones. The following code shows how this is done:

```

1  -- create an arbitrary model:
2  model = nn.Sequential()
3  model:add( nn.Linear(100,1000) )
4  model:add( nn.Tanh() )
5  model:add( nn.Linear(1000,10) )
6
7  -- and a loss function:
8  loss = nn.MSECriterion()
9
10 -- extract the parameters, and the gradient holder
11 w,dloss_dw = model:getParameters()
12
13 -- w and dl_dw are two vectors of the same size

```

Once the trainable parameter vector has been extracted, arbitrary, external optimization procedures can be used. *Torch7* provides a few standard methods (LBFGS, CG, SGD, ASGD) which simply require: (1) a function that computes L_w and $\frac{dL}{dw}$ and (2) the parameter vectors w and dL/dw . Of course, L_w can be either the true loss, or any approximation of it. The function that is defined is responsible for sampling from the training dataset, and estimating these approximations.

With these two concepts in mind, one can easily define a loop over a training dataset, and define a closure at each iteration, which computes L_w and $\frac{dL}{dw}$. The following listing shows an example of such a loop, assuming a pre-shuffled training dataset in which each entry is a tuple (x_n, y_n) :

```

1  -- assuming a training dataset 'trainset', and the model
2  -- defined above: 'model', 'w' and 'dL_dw':
3  for e = 1,nepochs do
4      for i,sample in ipairs(trainset) do
5          -- next training pair:
6          x_n = sample[1]
7          y_n = sample[2]
8
9          -- create closure that estimates y_n_hat = f_w(x_n),
10         -- stochastically
11         feval = function()
12             -- estimate loss:
13             y_n_hat = model:forward(x_n)
14             f = loss:forward(y_n_hat, y_n)
15
16             -- estimate gradients:
17             dloss_dw:zero()
18             dloss_dy_n_hat = loss:backward(y_n_hat, y_n)
19             model:backward(x_n, dloss_dy_n_hat)
20
21             -- return loss, and gradients

```

```

22         return f,dloss_dw
23     end
24
25     -- now that the closure is defined, pass it to an
26     -- optimization algorithm:
27     w,fs = optim.sgd(feval,w)
28
29     -- + the new w is returned, but as computations are
30     -- done in place, it is typically not necessary to
31     -- store it (the old w contains the new value)
32     -- + fs is a list of all the function (loss)
33     -- evaluations that were done during optimization.
34     -- SGD only returns one value, as it does not
35     -- perform any line search.
36 end

```

In the listing above, one can see that the loss and gradient estimation can be easily changed at runtime, and estimated over arbitrary batch sizes. To use a batch size different than 1 (as done above), one simply needs to create a list of training pairs, and the *feval* function needs to loop over these training pairs to estimate the approximate loss and gradients.

21.4 Conclusion

Compared to the early days of neural network training, the challenges towards an efficient implementation did not change a lot, however the means changed slightly. Already in the late 80's, the SN [3] toolbox was providing a scripting language (LISP) for building neural networks in a modular way. At the time, memory bandwidth and processor speed were about the same order of magnitude. Nowadays, we have to pay much more attention on memory accesses, counting number of instructions for optimizing the code is not sufficient anymore. Specific vectorized instructions can be easily integrated, but will not give order of magnitude speedups. In the end, what brings most advantage is parallelization. As computers become more and more parallel, it becomes crucial to leverage parallelization frameworks properly, such as OpenMP. On a more extreme side, GPUs (e.g. running with CUDA) are not as attractive as some could have expected: GPU-specific implementations require heavy extra work for a speedup (see Figure 21.5) which can be quite disappointing compared to what one can get with few extra lines of code with OpenMP.

Acknowledgments. *Torch7* is the official successor to *Torch3*¹³ and *Torch5*.¹⁴ Over the years many distinguished machine learning researchers have contributed to *Torch*, including Léon Bottou (we thank him in particular for providing the Qt library interface), Jason Weston, Iain Melvin, Samy Bengio and Johnny Mariéthoz.

¹³ <http://www.torch.ch/torch3>

¹⁴ <http://torch5.sourceforge.net/>

We would also like to thank Yann LeCun and Léon Bottou for sharing their work in SN, Lush, and extending their support and advices.

Finally, we thank James Bergstra for making his benchmark code available.¹⁵

References

- [1] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Bengio, Y.: Theano: a CPU and GPU math expression compiler. In: Proceedings of the Python for Scientific Computing Conference, SciPy (2010)
- [2] Bottou, L.: Large-scale machine learning with stochastic gradient descent. In: Lechevallier, Y., Saporta, G. (eds.) Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT 2010), Paris, France, pp. 177–187. Springer (August 2010)
- [3] Bottou, L., LeCun, Y.: SN: A simulator for connectionist models. In: Proceedings of NeuroNimes 1988, Nîmes, France (1988)
- [4] Le, Q.V., Coates, A., Prochnow, B., Ng, A.Y.: On optimization methods for deep learning. Learning, 265–272 (2011)
- [5] LeCun, Y., Bottou, L.: Lush reference manual. Technical report (2002), code, <http://lush.sourceforge.net>
- [6] LeCun, Y., Bottou, L., Orr, G.B., Müller, K.-R.: Efficient BackProp. In: Orr, G.B., Müller, K.-R. (eds.) NIPS-WS 1996. LNCS, vol. 1524, pp. 9–50. Springer, Heidelberg (1998)
- [7] Martens, J.: Deep learning via hessian-free optimization. In: Proceedings of the 27th International Conference on Machine Learning (ICML), vol. 951 (2010)
- [8] Vinyals, O., Povey, D.: Krylov subspace descent for deep learning. Arxiv preprint arXiv:1111.4259 (2011)

¹⁵ <http://www.github.com/jaberg/DeepLearningBenchmarks>

Better Representations: Invariant, Disentangled and Reusable

Preface

In many cases, the amount of labeled data is limited and does not allow for fully identifying the function that needs to be learned. When labeled data is scarce, the learning algorithm is exposed to simultaneous underfitting and overfitting. The learning algorithm starts to “invent” nonexistent regularities (overfitting) while at the same time not being able to model the true ones (underfitting). In the extreme case, this amounts to perfectly memorizing training data and not being able to generalize at all to new data.

The following five chapters present various tricks to solve the underfitting/overfitting problem. These include approaches to force invariance into the model in order to increase the signal-to-noise ratio, pretraining methods to disentangle the factors of variation, and how to use auxiliary tasks to learn a shared representation.

In image recognition, we know a priori that a good classifier should be translation invariant, rotation invariant, scale invariant, etc. Convolutional neural networks [7] are special networks composed of *convolution* and *pooling* layers that explicitly implement the translation invariance at multiple scales. They can be trained using standard backpropagation. One can also construct the convolutional network [9] one layer at the time in an unsupervised fashion. Chapter 22 [3] shows that, in this context, k-means is a particularly efficient method to learn the convolution filters.

Alternatively, the desired invariance can be injected in the model with artificial training samples that are translated, rotated or distorted versions of the original samples. The idea is presented in Chapter 23 [2]. A well-engineered set of transformations can produce a very large number of artificial samples and considerably improve the generalization of the neural network. Things can become complicated as invariance is not always absolute: For example, the handwritten digit “1” is rotation-invariant only up to a certain angle, beyond which it can be confused with the handwritten digit “7”.

Both convolutional networks and generating artificial samples yield excellent results on problems such as handwritten digit recognition and classification of small images. Combination of both techniques brings even higher performance. Unfortunately, these methods are only applicable when invariance is known. In the most general case, the invariance is unknown and other tricks are necessary in order to improve the learned representation.

The representation can be improved by learning an unsupervised model as a first step [5], using large amounts of unlabeled data. Unsupervised pretraining aims to construct a network of disentangled factors of variation. As a second step, the supervised learner only needs to choose in the set of disentangled factors, those that best predict the task of interest. Unsupervised pretraining transfers the burden of complex nonlinear modeling to unlabeled data which is generally available in much larger amounts. This two-steps approach was shown

to significantly reduce the underfitting/overfitting problem [5, 4] and improve handwritten digit and phoneme recognition performance.

A well-established algorithm for learning the unsupervised representation is the restricted Boltzmann machine. Chapter 24 [6] explains step by step how to train it successfully and how to choose the multiple hyperparameters. Chapter 25 [8] shows how keeping the model centered throughout training facilitates learning in the more sophisticated deep Boltzmann machine.

The series of five chapters terminates with the question of using auxiliary tasks to improve the solution learned by a neural network. As it was shown in Chapter 8 [1], this can be achieved by sharing an internal representation (and its associated parameters) across multiple related tasks and training the resulting multi-task network with backpropagation. A variant of multitask learning is introduced in Chapter 26 [10] where the auxiliary task is not defined by a set of labels but by a set of pairwise similarities between samples. This extension considerably widens the domain of applicability of multi-task learning.

Grégoire & Klaus

References

- [1] Caruana, R.: A Dozen Tricks with Multitask Learning. In: Orr, G.B., Müller, K.-R. (eds.) NIPS-WS 1996. LNCS, vol. 1524, pp. 163–189. Springer, Heidelberg (1998)
- [2] Ciresan, D.C., Meier, U., Gambardella, L.M., Schmidhuber, J.: Deep Big Multilayer Perceptrons for Digit Recognition. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 581–598. Springer, Heidelberg (2012)
- [3] Coates, A., Ng, A.Y.: Learning Feature Representations with k-means. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 561–580. Springer, Heidelberg (2012)
- [4] Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., Bengio, S.: Why does unsupervised pre-training help deep learning? *J. Machine Learning Res.* 11, 625–660 (2010)
- [5] Hinton, G.E., Osindero, S., Teh, Y.-W.: A fast learning algorithm for deep belief nets. *Neural Computation* 18, 1527–1554 (2006)
- [6] Hinton, G.E.: A Practical Guide to Training Restricted Boltzmann Machines. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 437–478. Springer, Heidelberg (2012)
- [7] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient based learning applied to document recognition. *IEEE* 86(11), 2278–2324 (1998)
- [8] Montavon, G., Müller, K.-R.: Deep Boltzmann Machines and the Centering Trick. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 621–637. Springer, Heidelberg (2012)
- [9] Serre, T., Wolf, L., Poggio, T.: Object recognition with features inspired by visual cortex. In: Computer Vision and Pattern Recognition Conference, pp. 994–1000. IEEE Press (2005)
- [10] Weston, J., Ratle, F., Collobert, R.: Deep Learning via Semi-Supervised Embedding. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 639–655. Springer, Heidelberg (2012)

Learning Feature Representations with K-Means

Adam Coates and Andrew Y. Ng

Stanford University, Stanford CA 94306, USA

Abstract. Many algorithms are available to learn deep hierarchies of features from unlabeled data, especially images. In many cases, these algorithms involve multi-layered networks of features (e.g., neural networks) that are sometimes tricky to train and tune and are difficult to scale up to many machines effectively. Recently, it has been found that K-means clustering can be used as a fast alternative training method. The main advantage of this approach is that it is very fast and easily implemented at large scale. On the other hand, employing this method in practice is not completely trivial: K-means has several limitations, and care must be taken to combine the right ingredients to get the system to work well. This chapter will summarize recent results and technical tricks that are needed to make effective use of K-means clustering for learning large-scale representations of images. We will also connect these results to other well-known algorithms to make clear when K-means can be most useful and convey intuitions about its behavior that are useful for debugging and engineering new systems.

22.1 Introduction

A major goal in machine learning is to learn deep hierarchies of features for other tasks. For instance, given a set of unlabeled images, many current algorithms seek to greedily learn successive layers of features that will make subsequent classification tasks (e.g., object recognition) easier to accomplish. A typical approach taken in the literature is to use an unsupervised learning algorithm to train a model of the unlabeled data and then use the results to extract interesting features from the data [35, 21, 31]. Depending on the choice of unsupervised learning scheme, it is sometimes difficult to make these systems work well. There can be many hyper-parameters and not much intuition for how to tune them. More recently, we have found that using K-means clustering as the unsupervised learning module in these types of “feature learning” pipelines can lead to excellent results, often rivaling state-of-the-art systems [11]. In this chapter, we will review some of this work with added notes on useful tricks and observations that are helpful for building large-scale feature learning systems.

K-means has already been identified as a successful method to learn features from images by computer vision researchers. The popular “bag of features” model [13, 28] from the computer vision community is very similar to the pipeline that we will use in this chapter, and many conclusions here are similar to those

identified by vision researchers [18, 1]. In this chapter, however, we will focus on the ingredients needed to make K-means work well in a setting similar to that used by other deep learning and feature learning systems: learning directly from raw inputs (pixel intensities) and building multi-layered hierarchies, as well as connecting K-means to other well-known feature learning systems.

The classic K-means clustering algorithm finds cluster centroids that minimize the distance between data points and the nearest centroid. Also called “vector quantization”, K-means can be viewed as a way of constructing a “dictionary” $\mathcal{D} \in \mathbb{R}^{n \times k}$ of k vectors so that a data vector $x^{(i)} \in \mathbb{R}^n$, $i = 1, \dots, m$ can be mapped to a code vector that minimizes the error in reconstruction. In this chapter, we will use a modified version of K-means (sometimes called “gain shape” vector quantization [41], or “spherical K-means” [14]) that finds \mathcal{D} according to:

$$\begin{aligned} & \underset{\mathcal{D}, s}{\text{minimize}} \sum_i \|\mathcal{D}s^{(i)} - x^{(i)}\|_2^2 \\ & \text{subject to } \|s^{(i)}\|_0 \leq 1, \forall i \\ & \quad \text{and } \|\mathcal{D}^{(j)}\|_2 = 1, \forall j \end{aligned}$$

where $s^{(i)}$ is a “code vector” associated with the input $x^{(i)}$, and $\mathcal{D}^{(j)}$ is the j 'th column of the dictionary \mathcal{D} . The goal here is to find a dictionary \mathcal{D} and a new representation, $s^{(i)}$, of each example $x^{(i)}$ that satisfies several criteria. First, given $s^{(i)}$ and \mathcal{D} , we should be able to reconstruct the original $x^{(i)}$ well; in particular, we aim to minimize the squared difference between $x^{(i)}$ and its corresponding reconstruction $\mathcal{D}s^{(i)}$. This goal is optimized under two constraints. The first constraint, $\|s^{(i)}\|_0 \leq 1$, means that each $s^{(i)}$ is constrained to have at most one non-zero entry. Thus we are searching not only for a new representation of $x^{(i)}$ that preserves it as well as possible, but also for a very simple or parsimonious representation. The second constraint requires that each dictionary column have unit length, preventing them from becoming arbitrarily large or small. Otherwise we could arbitrarily rescale $\mathcal{D}^{(j)}$ and the corresponding $s_j^{(i)}$ without effect.

This algorithm is very similar in spirit to other algorithms for learning efficient coding schemes, such as sparse coding [34, 17]:

$$\begin{aligned} & \underset{\mathcal{D}, s}{\text{minimize}} \sum_i \|\mathcal{D}s^{(i)} - x^{(i)}\|_2^2 + \lambda \|s^{(i)}\|_1 \\ & \text{subject to } \|\mathcal{D}^{(j)}\|_2 = 1, \forall j. \end{aligned}$$

Sparse coding optimizes the same type of reconstruction objective, but constrains the complexity of $s^{(i)}$ by adding a penalty $\lambda \|s^{(i)}\|_1$ that encourages $s^{(i)}$ to be sparse. This is similar to the constraint used by K-means ($\|s^{(i)}\|_0 \leq 1$), but allows more than one non-zero entry in each $s^{(i)}$, enabling a much more accurate representation of each $x^{(i)}$ while still requiring each $s^{(i)}$ to be simple.

From their descriptions above, it is no surprise that K-means and more sophisticated dictionary-learning schemes like sparse coding are often

interchangeable—differing in their optimization objectives, but producing code vectors $s^{(i)}$ and dictionaries \mathcal{D} that accomplish similar goals. Empirically though, sparse coding appears to be a better performer in many applications. For instance, replacing K-means with sparse coding in the classic bag-of-features model has been shown to significantly improve image recognition results [39]. Despite its simplicity, however, K-means is still a very useful algorithm for learning features due to its speed and scalability. Sparse coding requires us to solve a convex optimization problem [36, 15, 32] for every $s^{(i)}$ repeatedly during the learning procedure and thus is very expensive to deploy at large scale. For K-means, by contrast, the optimal $s^{(i)}$ used in the algorithm above is simply:

$$s_j^{(i)} = \begin{cases} \mathcal{D}^{(j)\top} x^{(i)} & \text{if } j == \arg \max_l |\mathcal{D}^{(l)\top} x^{(i)}| \\ 0 & \text{otherwise.} \end{cases} \quad (22.1)$$

Because this can be done very quickly (and solving for \mathcal{D} given s is also easy), we can train very large dictionaries rapidly by alternating optimization of \mathcal{D} and s . As well, K-means does not have any parameters requiring tuning other than k , the number of centroids, making it relatively easy to get working. The surprise is that large dictionaries learned by K-means often work very well in practice provided we mix in a few other ingredients that are less commonly documented in other works. This chapter is about what these ingredients are as well as some intuition about why they are needed and how they affect results. For most of this work, we will use images (or image patches) as input data to the algorithm, but the basic principles are applicable to other types of data as well.

22.2 Data, Pre-processing and Initialization

We will begin with a dataset composed of small image patches. For concreteness, we will work with 16-by-16 pixel grayscale patches represented as a vector of 256 pixel intensities (i.e., $x^{(i)} \in \mathcal{R}^{256}$), but color patches can also be used similarly. These patches can be collected from unlabeled imagery by cropping out random 16-by-16 chunks. In order to build a “complete” dictionary (i.e., a dictionary with at least 256 centroids), we should ensure that there will be enough patches so that each cluster can claim a reasonable number of inputs. For 16-by-16 gray patches, $m = 100,000$ patches is enough. In practice, we will often need *more* data to train a K-means dictionary than is necessary for other algorithms (e.g., sparse coding), since each data point contributes to just 1 centroid in the end. Usually the added expense is easily offset by the speed of training. For notational convenience, we will assume that our data points are packed into the columns of a matrix $X \in \mathcal{R}^{n \times m}$. (Similarly, we will denote by S the matrix whose columns are the code vectors $s^{(i)}$ from Eq. (22.1).)

22.2.1 Pre-processing

Before running a learning algorithm on our input data points $x^{(i)}$, it is useful to normalize the brightness and contrast of the patches. That is, for each $x^{(i)}$ we

subtract out the mean of the intensities and divide by the standard deviation. A small value is added to the variance before division to avoid divide by zero and also suppress noise. For pixel intensities in the range [0, 255], adding 10 to the variance is often a good starting point:

$$x^{(i)} = \frac{\tilde{x}^{(i)} - \text{mean}(\tilde{x}^{(i)})}{\sqrt{\text{var}(\tilde{x}^{(i)}) + 10}}$$

where $\tilde{x}^{(i)}$ are unnormalized patches and “mean” and “var” are the mean and variance of the elements of $\tilde{x}^{(i)}$.

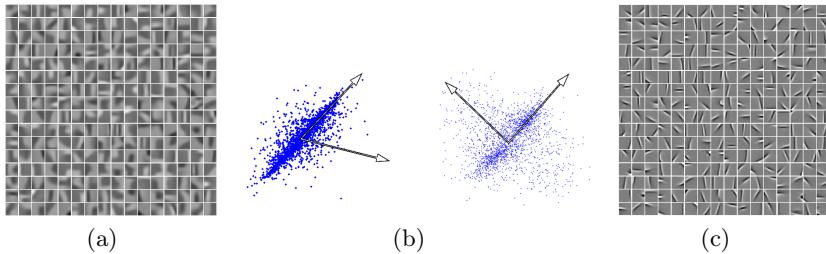


Fig. 22.1. (a) Centroids learned by K-means from natural images without whitening. (b) A cartoon depicting the effect of whitening on the K-means solution. Left: unwhitened data, where the centroids tend to be biased by the correlated data. Right: whitened data, where centroids are more orthogonal. (c) Centroids learned from whitened image patches.

After normalization, we can try to run K-means on the new input patches. The centroids that are obtained (i.e., the columns of the dictionary \mathcal{D}) are visualized as patches in Figure 22.1a. It can be seen that K-means tends to learn low-frequency edge-like centroids. This result has been reproduced many times in the past [16, 37, 2]. Unfortunately, it turns out that these centroids tend to work poorly in recognition tasks [11]. One explanation for this result is that the correlations between nearby pixels (i.e., low-frequency variations in the images) tend to be very strong even after brightness and contrast normalization. In the presence of these correlations, K-means tends to generate many highly correlated centroids rather than spreading the centroids out to span the data more evenly. A cartoon depicting this problem is shown on the left of Figure 22.1b. To remedy this situation, one should use whitening (also called “sphering”) to rescale the input data to remove these correlations [22]. This tends to cause K-means to allocate more centroids in the orthogonal directions, as shown on the right of Figure 22.1b.

A simple choice of whitening transform is the ZCA whitening transform. If $VDV^\top = \text{cov}(x)$ is the eigenvalue decomposition of the covariance of the data points x , then the whitened points are computed as $V(D + \epsilon_{zca}I)^{-1/2}V^\top x$, where

ϵ_{zca} is a small constant. For contrast-normalized data, setting ϵ_{zca} to 0.01 for 16-by-16 pixel patches, or 0.1 for 8-by-8 pixel patches is a good starting point. Note that setting this number too small can cause high-frequency noise to be amplified and make learning more difficult. Since rotating the data does not alter the behavior of K-means, one can also use other whitening transforms such as PCA whitening (which differ from ZCA only by a rotation).

Running K-means on whitened image patches yields sharper edge features similar to those discovered by sparse coding, ICA, and others as seen in Figure 22.1c. This procedure of normalization, whitening, and K-means clustering is an effective “off the shelf” unsupervised learning module that can serve in many feature-learning roles. From this point forward, we will assume that whenever we apply K-means to new data that they are normalized and whitened as described here. But keep in mind that proper choices of the ϵ parameters for normalization and whitening can sometimes require adjustment for new data sources. Though these are likely best set by cross validation, they can often be tuned visually (e.g., to yield image patches with high contrast, not too much noise, and not too much low-frequency undulation).

22.2.2 Initialization

The usual K-means clustering algorithm is known to require a number of small tweaks to avoid common problems like empty clusters. One important consideration is the initialization of the centroids. Though it is common to initialize K-means to randomly-chosen examples drawn from the data, this has been found to be a poor choice. It is possible that images tend to group too densely in some areas, and thus initializing K-means with randomly chosen patches leads to a large number of centroids starting close together. Many of these centroids ultimately end up becoming near-empty clusters, as a single cluster accumulates all of the data points located within a dense area. Instead, it is better to randomly initialize the centroids from a Normal distribution and then normalize them to unit length. Note that because of the whitening stage, we expect that the important components of our data have already been rescaled to a more or less spherical distribution, so initializing to random vectors on a sphere is not a terrible starting point.

Other well-known heuristics for improving the behavior of K-means can be useful. For instance, heuristics for reinitializing empty clusters are commonly used in other implementations. In practice, the initialization scheme above works relatively well for image data. When empty clusters do occur, reinitializing the centroids with random examples is usually sufficient, but this is rarely necessary.¹ In fact, for a sufficiently scalable implementation, we can often train many centroids and simply throw away clusters that have too few data points.

¹ Often, a large number of empty clusters indicates that the whitening or normalization parameters are improperly tuned, or the data is too high-dimensional for K-means to be successful.

Another minor tweak that improves behavior is to use damped updates of the centroids. Specifically, at each iteration we compute new centroids according to:

$$\begin{aligned}\mathcal{D}_{\text{new}} &:= \arg \min_{\mathcal{D}} \|\mathcal{D}S - X\|_2^2 + \|\mathcal{D} - \mathcal{D}_{\text{old}}\|_2^2 \\ &= (SS^\top + I)^{-1}(XS^\top + \mathcal{D}_{\text{old}}) \\ &\propto XS^\top + \mathcal{D}_{\text{old}} \\ \mathcal{D}_{\text{new}} &:= \text{normalize}(\mathcal{D}_{\text{new}}).\end{aligned}$$

Note that this form of damping does not affect “big” clusters very much (the j ’th column of XS^\top will be large compared to $\mathcal{D}_{\text{old}}^{(j)}$) and only serves to prevent small clusters from being pulled too far in a single iteration.

Including the initialization and pre-processing, the full K-means training routine presented above is summarized here:

1. Normalize inputs:

$$x^{(i)} := \frac{x^{(i)} - \text{mean}(x^{(i)})}{\sqrt{\text{var}(x^{(i)}) + \epsilon_{\text{norm}}}}, \forall i$$

2. Whiten inputs:

$$\begin{aligned}[V, D] &:= \text{eig}(\text{cov}(x)); // \text{ So } VDV^\top = \text{cov}(x) \\ x^{(i)} &:= V(D + \epsilon_{\text{zca}}I)^{-1/2}V^\top x^{(i)}, \forall i\end{aligned}$$

3. Loop until convergence (typically 10 iterations is enough):

$$\begin{aligned}s_j^{(i)} &:= \begin{cases} \mathcal{D}^{(j)\top} x^{(i)} & \text{if } j == \arg \max_l |\mathcal{D}^{(l)\top} x^{(i)}| \\ 0 & \text{otherwise.} \end{cases} \quad \forall j, i \\ \mathcal{D} &:= XS^\top + \mathcal{D} \\ \mathcal{D}^{(j)} &:= \mathcal{D}^{(j)} / \|\mathcal{D}^{(j)}\|_2 \quad \forall j\end{aligned}$$

22.3 Comparison to Sparse Feature Learning

As was shown above, K-means learns oriented edge-like features when applied to natural images, much like ICA [23] or sparse coding [34]. An important practical question is whether this is accidental (e.g., because edges are so common that learning “exemplars” from images is enough to find them) or whether this implies that K-means can perform a type of sparse decomposition similar to ICA. When we attempt to apply K-means to other types of data such as audio or features

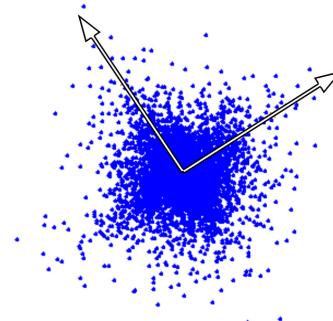


Fig. 22.2. The result of running spherical K-means on points sampled from a heavy-tailed distribution. The K-means “centroids” tend to point in the directions of the tails.

computed by lower layers in a deep network, it is important to understand to what extent this clustering algorithm mimics well-known projection methods like ICA and what the limitations are. It is clear that because each $s^{(i)}$ is allowed to contain only a single non-zero entry, K-means tries to learn centroids that single-handedly explain an entire input image. It is thus not guaranteed that the learned centroids will always be like the filters produced by sparse coding or ICA. These algorithms learn genuine “distributed” representations where a single image can be explained jointly by multiple columns of the dictionary instead of just one. Nevertheless, empirically it turns out that K-means *does* tend to discover sparse projections of the data under the right conditions. Because of this property, we can often use the learned dictionary in a manner similar to the dictionaries or filters learned by other algorithms that explicitly search for sparse, distributed representations.

One intuition for why this tends to occur can be seen in a simple low-dimensional example. Consider the case where our data is sampled from two independent, heavy-tailed (sparse) random variables. After normalization, the data will be most dense near the coordinate axes, and less dense in the quadrants between them. As a result, while K-means will tend to represent many points very badly, training 2 centroids on this distribution will tend to yield a dictionary of basis vectors pointing in the direction of the tails (i.e., in the sparse directions). This result is illustrated in Figure 22.2.

If the data dimensionality is not too high (e.g., a hundred or so) this “tail-seeking” phenomenon also shows up in more complicated scenarios. Figure 22.3 shows examples of three sets of images generated from sparse sources. On the left (at top), are 16-by-16 images with pixels sampled from independent Laplace distributions (sparse pixel intensities). In the middle (top) are images composed of a sparse combination of gratings and at right (top) a sparse combination of non-orthogonal gabor filters. The bottom row shows the result of learning 256 centroids with K-means from 500000 examples drawn from each distribution. It can be seen that K-means does, in fact, roughly recover the sparse projections we

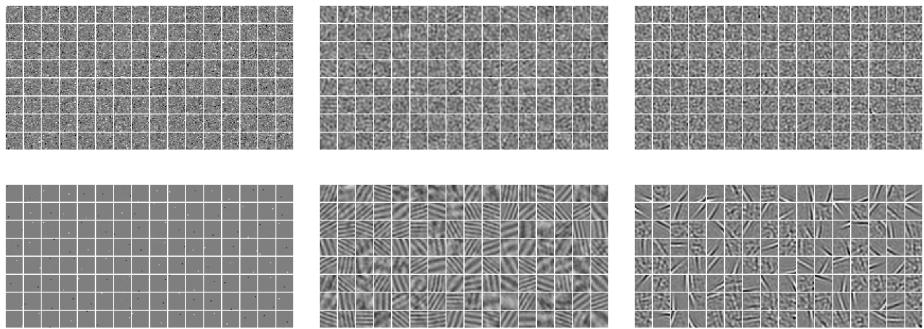


Fig. 22.3. Top: Three different sparse distributions of images. Bottom: Filters (centroids) identified by K-means with a complete (256-centroid) dictionary.

would expect. A similar experiment appears in [34] to demonstrate the source-separation ability of sparse coding—yet K-means tends to recover the same filters even though these filters are clearly not especially similar to individual images. That is, K-means can potentially do more than merely recover “exemplar” images from the input distribution.

When applied to natural images, it is evident that the learned centroids $\mathcal{D}^{(j)}$ (as in Figure 22.1c) are relatively sparse projections of the data. Figure 22.4a shows a histogram of responses resulting from projecting randomly chosen (whitened) image patches onto 4 different filters. The 4 filters used are: (i) a centroid learned by K-means, (ii) a basis vector trained via sparse coding, (iii) a randomly selected image patch, and (iv) a randomly initialized filter. In the figure, it can be seen that using the K-means-trained centroid as a linear filter gives us a very sparse projection of the data. Thus, it appears that relative to other simple choices K-means does tend to seek out very sparse projections of the data, even though its formulation as a clustering algorithm does not aim to do this explicitly.

Despite this empirical similarity to ICA and sparse coding, K-means does have a major drawback: it turns out that its ability to discover sparse directions in the data depends heavily on the dimensionality of the input and the quantity of data. In particular, as the dimensionality increases, we need increasingly large quantities of data to get clean results. For instance, to obtain the results above we had to use a very large number of examples. We can obtain similar results easily with sparse coding with just 10000 examples. For very large patches, we must use even more. Figure 22.4b shows the results of running K-means on 500000 64-by-64 patches—note that while we can capture a few edges, most of the clusters are composed of a single patch from the dataset. At this scale, empty or near-empty clusters become far more common and extremely large amounts of data are needed to make K-means work well. This tradeoff is the main driver in deciding when and how to employ K-means: depending on the dimensionality of the input, a certain amount of data will be required (typically much more than is needed for similar results from sparse coding). For modest dimensionalities (e.g.,

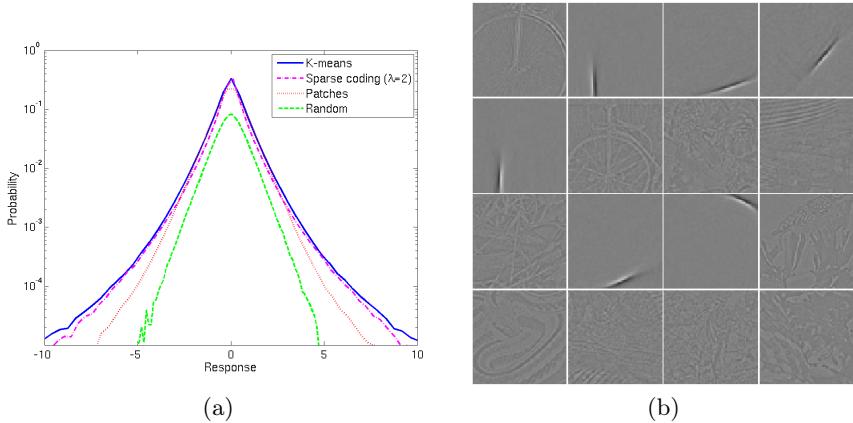


Fig. 22.4. (a) A comparison of the distribution of linear filter responses when filters obtained by 4 different methods are applied to natural image patches. K-means tends to learn filters with very sparse response characteristics similar to sparse coding—much more sparse than randomly chosen patches or randomly initialized filters. (b) “Centroids” learned from 64-by-64 pixel patches. At this scale, K-means becomes difficult to apply as many clusters become empty or singletons.

hundreds of inputs), this tradeoff can be advantageous because the additional data requirements do not outweigh the very large constant-factor speedup that is gained by training with K-means. For very high dimensionalities, however, it may well be the case that another algorithm like sparse coding works better or even faster.

22.4 Application to Image Recognition

The above discussion has provided the basic ingredients needed to turn K-means into a simple feature learning method. Given a batch of unlabeled data, we can now learn a dictionary \mathcal{D} whose columns yield more or less sparse projections of the data points. Just as with sparse coding, we can use the learned dictionary (centroids) to define features for a supervised learning task [35]. A typical pipeline used for image recognition applications (that is easy to use with learned features) is based on the classic spatial pyramid model developed in the computer vision literature [13, 28, 39, 11]. It is similar in many ways to a single-layered convolutional neural network [29, 30]. The main components of this pipeline are: (i) the unsupervised training algorithm (in this case, K-means), which generates a bank of filters \mathcal{D} , (ii) a function $f : \mathcal{R}^n \rightarrow \mathcal{R}^k$ that maps an input image patch $x \in \mathcal{R}^n$ to a feature vector $y \in \mathcal{R}^k$ given the dictionary of k filters. For instance, we could choose $f(x; \mathcal{D}) = g(\mathcal{D}^\top x)$ for an element-wise nonlinear function $g(\cdot)$.

Using the learned feature extractor $f(x; \mathcal{D})$, given any p -by- p pixel image patch, we can now compute a representation $y \in \mathcal{R}^k$ for that patch. We can thus

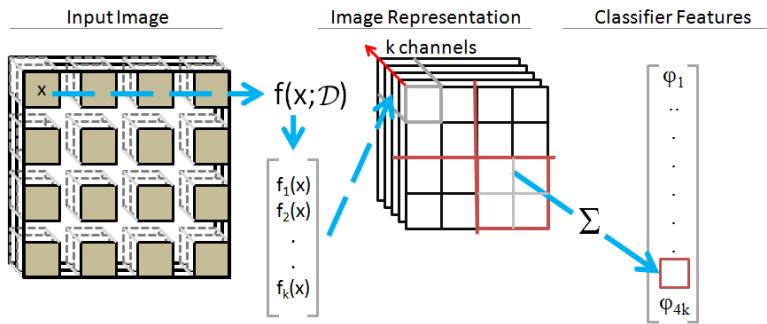


Fig. 22.5. A standard image recognition pipeline used in conjunction with K-means dictionary learning

define a (single layer) representation of the entire image by applying the function f to many sub-patches. Specifically, given an image of w -by- w pixels, we define a $(w - p + 1)$ -by- $(w - p + 1)$ -by- k array of features by computing the representation y for every p -by- p “sub-patch” of the input image. For computational efficiency, we may also “step” our p -by- p feature extractor across the image with some step-size (or “stride”) greater than 1. This is illustrated in Figure 22.5.

Before classification, we typically reduce the dimensionality of the image representation by pooling. For a step size of 1 pixel, our feature extractor produces a $(w - p + 1)$ -by- $(w - p + 1)$ -by- k representation. We can reduce this by summing (or applying some other reduction, e.g., max) over local regions of the feature responses. Once we have the pooled responses, we could attempt to learn higher-level features by applying K-means again (this is the approach pursued by [1]), or just use all of the features directly with a standard linear classification algorithm (e.g., SVM).

22.4.1 Parameters

The processing pipeline above has a large number of tunable parameters, such as the patch size p , the choice of $f(x; \mathcal{D})$, and the step size. It turns out that getting these parameters set correctly can make a major difference in performance for practical applications. In fact, these parameters often have a bigger influence on classification performance than the training algorithm itself. When we are unhappy with performance, searching for better choices of these parameters can often be more beneficial than trying to modify our learning algorithm [11, 18]. Here we will briefly summarize current advice on how to choose these parameters.

First, when we use K-means to train the filter bank \mathcal{D} , we noted previously that the input dimensionality can significantly influence the data requirements and success of training. Thus, in addition to other effects it may have on classification performance, it is important to choose the patch size p wisely. If p is too large (e.g., 64 pixels, as in Figure 22.4b), then K-means may yield poor results. Though this situation can be debugged visually for applications to image data,

it is much more difficult to know when K-means is doing well when it is applied to other kinds of data such as when training a multi-layered network where the higher-level features are hard to visualize. For this reason, it is recommended that p be chosen by cross validation or it should be set so that the dimensionality of the data passed to K-means is kept small (typically not more than a few hundred, depending on the amount of data used). For image patches, 6-by-6 or 8-by-8 pixel (color or gray) patches work consistently well in the pipeline outlined above.

Depending on the choice of pooling method, the step size and pooling regions may need to be chosen differently. There is a significant body of work covering these areas [6, 5, 4, 12, 24, 18]. In our own experience, for single layers of features, average pooling over a few equally-sized regions (e.g., a 2-by-2 or 3-by-3 grid) can work very well in practice and is a good “first try” for image recognition.

Finally, the number of features k learned by the algorithm has a significant influence on the results. It has been observed several times [18, 11] that learning large numbers of features can substantially improve supervised classification results. Indeed, it is frequently best to set k as large as compute resources will allow, considering data requirements. Though performance typically asymptotes as k becomes extremely large, increasing the size of k is a very effective way to squeeze out a bit of extra performance from an already-built system. This trend, combined with the fact that K-means is especially effective for building very large dictionaries, is the main advantage of the system presented above.

22.4.2 Encoders

The choice of the feature “encoding” function $f(x; \mathcal{D})$ also has a major impact on recognition performance. For instance, consider the “hard assignment” encoding where we take $f(x; \mathcal{D}) = s$, with s the standard “one-hot” code used during K-means training (Eq. (22.1)). It is well-known that this scheme performs very poorly compared to other choices [18, 1]. Thus, once we have run K-means to train our filters, one should certainly make an effort to choose a better encoder f . There are many encoding schemes present in the literature [34, 38, 40, 42, 19, 20, 7] and, while they often include their own training algorithms, one can choose to use K-means-trained dictionaries in conjunction with many of them.

Unfortunately, many encoding schemes proposed for computer vision tasks are potentially very expensive to compute. Many require us to solve a difficult optimization problem in order to compute $f(x; \mathcal{D})$ [34, 40, 20, 7]. On the other hand, some encoders are simple nonlinear functions of the filter responses $D^\top x$ that can be computed extremely quickly. In previous work it has appeared that algorithms like sparse coding are generally the best performers in benchmarks [39, 4]. However, in some cases we can manage to get away with much simpler encodings. Specifically, when we use the single-layer architecture outlined above, it turns out that algorithms like sparse coding and more sophisticated variants (e.g., spike-slab sparse coding [20]) are difficult to top when we have very little labeled data. But as can be seen in Figure 22.6, with much more labeled data the disparity in performance between sparse coding and a very simple nonlinear

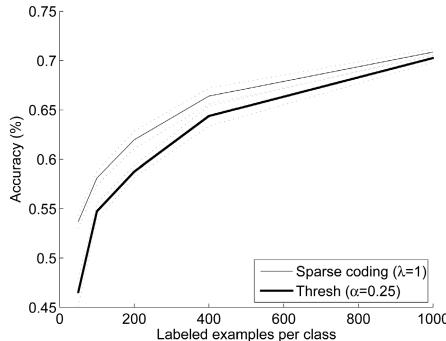


Fig. 22.6. A comparison of the performance between two encoders on the CIFAR-10 [25] benchmark as a function of the number of labeled examples. When labeled data is scarce, an expensive encoder can be worthwhile. If labeled data is plentiful, a fast, simple encoder such as a soft-threshold nonlinearity is sufficient.

encoder decreases significantly. We have found, not surprisingly, that as we use increasing quantities of labeled data the supervised learning stage takes over and works equally well with most reasonable encoders.

As a result of this observation, application designers should consider the quantity of available labeled data. If labeled data is abundant, a fast feed-forward encoder works well (and is the easiest to use on large datasets). If labeled data is scarce, however, it can be worthwhile to use a more expensive encoding scheme. In the large-scale case we have found that soft-threshold nonlinearities ($f(x; \mathcal{D}, \alpha) = \max\{0, \mathcal{D}^\top x - \alpha\}$, where α is a tunable constant) work very well. By contrast, sigmoid nonlinearities (e.g., $f(x; \mathcal{D}, b) = (1 + \exp(-\mathcal{D}^\top x + b))^{-1}$) appear to work significantly less well [33, 12] in similar instances.

22.5 Local Receptive Fields and Multiple Layers

Several times we have referred to the difficulties involved in applying K-means to high-dimensional data. In Section 22.4.1 it was explained that we should choose the image patch size p (“receptive field” size) carefully to avoid exceeding the modeling capacity of K-means (as in Figure 22.4b). If we have a very large image it is generally not effective to apply K-means to the entire input at once.² Instead, applying K-means to p -by- p pixel sub-patches is a reasonable substitute, since we expect that most learned features will be localized to a small region. This “trick” allows us to keep the input size to K-means relatively small (e.g., just p^2 input dimensions for grayscale patches), but still use the resulting filters

² Note that for very large inputs it becomes impractical to perform whitening, which requires solving a very large optimization problem (e.g., eigenvalue decomposition). In the authors’ experience K-means starts giving poor results before this computational bottleneck is reached.

on a much larger image by either reusing the filters for every p -by- p pixel sub-patch of the image, or even by re-running K-means independently on each p -by- p region (if, for some reason, the features present in other parts of the image differ significantly). Though this approach is well-known from computer vision applications the same trick works more generally and in some cases is indispensable for building working systems.

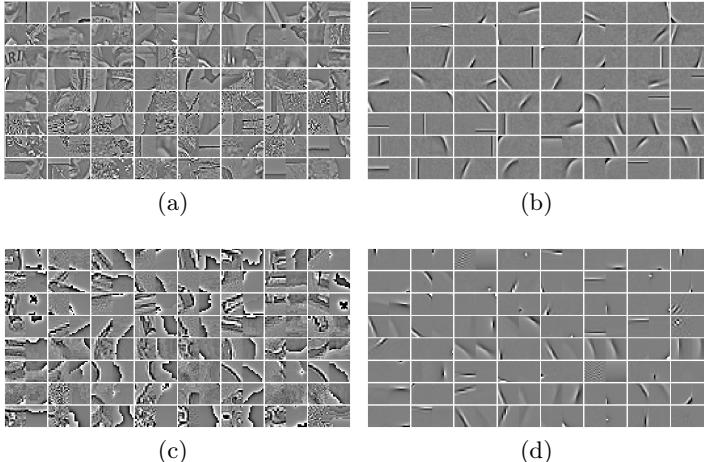


Fig. 22.7. (a) A dataset composed of concatenated pairs of independently sampled image patches. (b) Centroids trained from pairs of image patches. Note that, as expected, K-means learns filters that only involve half of the input at a time. Due to the increased dimensionality, significantly more data is needed compared to training on each image separately. (c) A dataset composed of image-depth pairs. The left half of each example is a whitened grayscale patch. The right half is a “depth image” [27] where pixel intensity represents the distance of a surface from the camera. (d) Centroids learned jointly from image-depth pairs learn only a few weak features that use both modalities for similar reasons as in (b).

Consider a situation where our input is, in fact, a concatenation of two independent signals. Concretely, let us take two image patches drawn at random from a larger dataset and concatenate them side-by-side as in Figure 22.7a. When we run K-means on this type of data we end up with the centroids in Figure 22.7b where individual centroids tend to model just one of the two independent components, much like we would expect from ICA. Unfortunately, as observed previously, to achieve this result we actually need more data than if we had tried to model the two patches separately. Hence, whenever we can determine *a priori* that our input variables can be split into independent chunks, we should try to split them up immediately and run K-means on each chunk separately. Note that the contrived example here occurs in real applications, such as learning features from RGB-Depth data [27]: Figure 22.7c shows examples of image intensity concatenated with depth patches and Figure 22.7d shows centroids learned from them. Since at this scale

depth tends to be only weakly related to raw pixel intensity, it might be better to run K-means separately on each modality of the data.

22.5.1 Deep Networks

In Section 22.4 we presented a simple pipeline that enabled us to extract a single layer of features from an image by taking a dictionary learned from small patches and using it to extract features over a larger image (see Figure 22.5). We then used a pooling stage to reduce the number of features before applying a supervised learning algorithm. It would be nice if we could learn higher layers of features by taking the resulting single-layer representation and passing it back through our feature learning pipeline. For instance, one simple way we might try to accomplish this is to compute the pooled feature values for all of the examples in our unlabeled dataset X to give us a new dataset Z , then apply the exact same learning pipeline to Z to learn new features. This simple approach has been applied in [1], but it turns out that this straight-forward strategy can hit a major snag: the inputs Z used for the second layer of features will often be very high dimensional if, as is common, we use very large dictionaries of features (e.g., $k = 10000$ or more).

Concretely, let's consider a simple example.³ Suppose that our goal is to learn features for a 20-by-20 pixel image patch. With the approach of Section 22.4 we train $k = 10000$ 16-by-16 pixel filters with K-means from 16-by-16 patches cropped out of our dataset. We then take the learned dictionary \mathcal{D} and extract feature responses with a step size of 4 pixels $f(x; \mathcal{D})$ from the 20-by-20 pixel images, yielding a 2-by-2-by-10000 image representation. Finally, we sum up the responses over each 2-by-2 region (i.e., all responses produced by each filter) to yield a 1-by-1-by-10000 “pooled representation” which we will take as Z . We can think of each feature value z_j as being a slightly translation-invariant version of the feature detector associated with the filter $\mathcal{D}^{(j)}$. Note that each vector in Z now has 10000 dimensions to represent the original 400-dimensional patch. At this scale, even learning a “complete” representation of 10000 features from the 10000-dimensional inputs Z becomes challenging. Regrettably, there is no obvious choice of local receptive field that can be made here: the 10000 features are unorganized and we have no way to split them up by hand.

One proposed solution to this problem is to use a simple form of pair-wise “dependency test” to help identify groups of dependent input variables in an automated way. If we can do this, then we can break up the input vector coordinates into small groups suitable for input to K-means instead of picking groups by hand. This tool is most valuable for building multiple layers of features with K-means.

As an example, we can use a type of dependency called “energy correlation”. Given two whitened inputs (i.e., two inputs z_j and z_k that have no linear correlation) their energy correlation is just the correlation between their squared

³ The numbers used in this example are chosen to illustrate the problem and its solution. For a more detailed and realistic setup, see [10].

responses. In particular, if we have $\mathbb{E}[z] = 0$ and $\mathbb{E}[zz^\top] = I$, then we will define the dependency between inputs z_j and z_k as:

$$d[z_j, z_k] = \text{corr}(z_j^2, z_k^2) = \mathbb{E}[z_j^2 z_k^2 - 1] / \sqrt{\mathbb{E}[z_j^4 - 1] \mathbb{E}[z_k^4 - 1]}.$$

This metric is easy to compute by first whitening the input data Z with ZCA whitening [3], then computing the pairwise similarities between all of the features:

$$d(j, k; Z) \equiv d[z_j, z_k] \equiv \frac{\sum_i z_j^{(i)2} z_k^{(i)2} - 1}{\sqrt{\sum_i (z_j^{(i)4} - 1) \sum_i (z_k^{(i)4} - 1)}}.$$

This computation is practical for fewer than 10000 input features. It can still be computed approximately for hundreds of thousands of features if necessary [10]. Thus, we now have a function $d(j, k; Z)$ that can provide a measure of the dependency between features z_j and z_k observed in a given dataset Z .

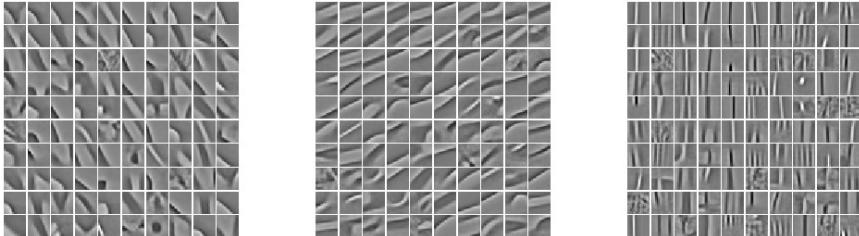


Fig. 22.8. Groups of features selected by an automated dependency test. The features corresponding to each group of filters would be processed separately by K-means to build a second layer of features.

Now we would like to try to learn some “higher level” features on top of the Z representation. Using our dependency test we can find reasonable choices of receptive fields in a relatively simple way: we pick one feature, say z_0 , and then use the dependency test to find the R features with the strongest dependence on z_0 according to the test (i.e., find the indices j so that $d(0, j; Z)$ is large). We then run K-means using only these R features as input. If we pick R small enough (e.g., 100 or 200) the usual normalization, whitening and K-means training steps can be applied easily and require virtually no tuning to work well. Because of the smaller input dimension we only need to train a few hundred centroids and thus we can use much less data than would be required to run K-means on the original 10000-dimensional dataset. This procedure can be repeated for many choices of the “seed” feature (z_0 above) until we have trained dictionaries from receptive fields covering all of the input variables in z . Figure 22.8 shows the first-layer filters from \mathcal{D} corresponding to some of these groups of features (i.e., these are the $\mathcal{D}^{(j)}$ whose pooled responses z_j have high dependence according to the energy correlation test).

Table 22.1. Results on CIFAR-10 (full)

Architecture	Accuracy (%)
1 Layer	78.3%
1 Layer (4800 maps)	80.6%
2 Layers (Single RF)	77.4%
2 Layers (Random RF)	77.6%
2 Layers (Learned RF)	81.2%
3 Layers (Learned RF)	82.0%
VQ (6000 maps) [12]	81.5%
Conv. DBN [26]	78.9%
Deep NN [8]	80.49%
Multi-column Deep NN [9]	88.79%

Table 22.2. Results on CIFAR-10 (400 ex. per class)

Architecture	Accuracy (%)
1 Layer	64.6% ($\pm 0.8\%$)
1 Layer (4800 maps)	63.7% ($\pm 0.7\%$)
2 Layers (Single RF)	65.8% ($\pm 0.3\%$)
2 Layers (Random RF)	65.8% ($\pm 0.9\%$)
2 Layers (Learned RF)	69.2% ($\pm 0.7\%$)
3 Layers (Learned RF)	70.7% ($\pm 0.7\%$)
Sparse coding (1 layer) [12]	66.4% ($\pm 0.8\%$)
VQ (1 layer) [12]	64.4% ($\pm 1.0\%$)

Table 22.3. Classification Results on STL-10

Architecture	Accuracy (%)
1 Layer	54.5% ($\pm 0.8\%$)
1 Layer (4800 maps)	53.8% ($\pm 1.6\%$)
2 Layers (Single RF)	55.0% ($\pm 0.8\%$)
2 Layers (Random RF)	54.4% ($\pm 1.2\%$)
2 Layers (Learned RF)	58.9% ($\pm 1.1\%$)
3 Layers (Learned RF)	60.1% ($\pm 1.0\%$)
Sparse coding (1 layer) [12]	59.0% ($\pm 0.8\%$)
VQ (1 layer) [12]	54.9% ($\pm 0.4\%$)

The effectiveness of this sort of approach combined with K-means has been shown in previous work [10]. Table 22.1 details results obtained on the full CIFAR dataset with various settings and comparisons to other contemporary methods. First, we can see in these results that learning 2 layers of features is essentially fruitless when using naive choices of receptive fields: a single receptive field that includes all of the inputs, or receptive fields that connect to random inputs. Indeed, the results for 2 layer networks are *worse* (77.4% and 77.6%) than obtained using a single layer alone (78.3%). This should be expected: using a single receptive field, K-means is unable to build good features due to the high-dimensional input (like Figure 22.4b), yet using a random receptive field wastes representational power modeling unrelated inputs (like Figure 22.7). By contrast, results obtained with the receptive field learning scheme above (with 2nd-order dependency measure) are significantly better: achieving a significant improvement over the baseline single-layer results, and even out-performing a much larger single-layer network. With 3 layers, this system improves further to 82.0% accuracy. Achieving the best possible results (as reported in [9]) may require supervised training of the entire network, but this result demonstrates very clearly the importance of controlling the *connectivity* of features in order for

K-means to work well in deep networks (where we typically use only unlabeled data to construct features).

Training only from unlabeled data is much more useful in a scenario where we have limited labeled training data. Tables 22.2 and 22.3 show results obtained from similar experiments on the CIFAR-10 dataset when using only 400 labeled examples per class, and the STL-10 [11] dataset (where only 100 labels are available per class). The results are very similar, even though we have less supervision: poor choices of receptive fields almost entirely negate the benefits of training multiple layers of features, but using the simple receptive field selection technique above allows us to successfully build up to 3 layers of useful features with K-means.

22.6 Conclusion

In this chapter we have reviewed many results, observations and tricks that are useful for building feature-learning systems with K-means as a scalable unsupervised learning module. The major considerations that we have covered, which practitioners should keep in mind before embarking on a new application, are summarized as:

1. Mean and contrast normalize inputs.
2. Use whitening to “sphere” the data, taking care to set the ϵ parameter appropriately. If whitening cannot be performed due to input dimensionality, one should split up the input variables.
3. Initialize K-means centroids randomly from Gaussian noise and normalize.
4. Use damped updates to help avoid empty clusters and improve stability.
5. Be mindful of the impact of dimensionality and sparsity on K-means. K-means tends to find sparse projections of the input data by seeking out “heavy-tailed” directions. Yet when the data is not properly whitened, the input dimensionality is very high, or there is insufficient data, it may perform poorly.
6. With higher dimensionalities, K-means will require significantly increased amounts of data, possibly negating its speed advantage.
7. Exogenous parameters in the system (pooling, encoding methods, etc.) can have a bigger impact on final performance than the learning algorithm itself. Consider spending compute resources on more cross-validation for parameters before concluding that a more expensive learning scheme is required.
8. Using more centroids almost always helps when using the image recognition pipeline described in this chapter, provided we have enough training data. Indeed, whenever more compute resources become available, this is the first thing to try.
9. When labeled data is abundant, find a cheap encoder and let a supervised learning system do most of the work. If labeled data is limited (e.g., hundreds of examples per class), an expensive encoder may work better.

10. Use local receptive fields wherever possible. Input data dimensionality is the main bottleneck to the success of K-means and should be kept as low as possible. If local receptive fields cannot be chosen by hand, try an automated dependency test to help cut up your data into (overlapping) groups of inputs with lower dimensionality. This is likely a necessity for deep networks!

The above recommendations cover essentially all of the tools, tricks and insights that underlie recent feature-learning results based on K-means. Though it is unclear how far K-means can be pushed in comparison to more expressive algorithms, the tips above are enough to know when K-means is appropriate and to get it working in many challenging scenarios.

References

- [1] Agarwal, A., Triggs, B.: Hyperfeatures – Multilevel Local Coding for Visual Recognition. In: Leonardis, A., Bischof, H., Pinz, A. (eds.) ECCV 2006, Part I. LNCS, vol. 3951, pp. 30–43. Springer, Heidelberg (2006)
- [2] Aharon, M., Elad, M., Bruckstein, A.: K-SVD: An algorithm for designing over-complete dictionaries for sparse representation. *IEEE Transactions on Signal Processing* 54(11), 4311–4322 (2006)
- [3] Bell, A., Sejnowski, T.J.: The ‘independent components’ of natural scenes are edge filters. *Vision Research* 37(23), 3327–3338 (1997)
- [4] Boureau, Y., Bach, F., LeCun, Y., Ponce, J.: Learning mid-level features for recognition. In: 23rd Conference on Computer Vision and Pattern Recognition, pp. 2559–2566 (2010)
- [5] Boureau, Y., Ponce, J., LeCun, Y.: A theoretical analysis of feature pooling in visual recognition. In: 27th International Conference on Machine Learning, pp. 111–118 (2010)
- [6] Boureau, Y., Roux, N.L., Bach, F., Ponce, J., LeCun, Y.: Ask the locals: multi-way local pooling for image recognition. In: 13th International Conference on Computer Vision, pp. 2651–2658 (2011)
- [7] Bradley, D.M., Bagnell, J.A.: Differentiable sparse coding. In: Advances in Neural Information Processing Systems, vol. 22, pp. 113–120 (2008)
- [8] Ciresan, D.C., Meier, U., Masci, J., Gambardella, L.M., Schmidhuber, J.: Flexible, high performance convolutional neural networks for image classification. In: International Joint Conference on Artificial Intelligence, pp. 1237–1242 (2011)
- [9] Ciresan, D.C., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. In: Computer Vision and Pattern Recognition, pp. 3642–3649 (2012)
- [10] Coates, A., Ng, A.Y.: Selecting receptive fields in deep networks. In: Advances in Neural Information Processing Systems, vol. 24, pp. 2528–2536 (2011)
- [11] Coates, A., Lee, H., Ng, A.Y.: An analysis of single-layer networks in unsupervised feature learning. In: 14th International Conference on AI and Statistics, pp. 215–223 (2011)
- [12] Coates, A., Ng, A.Y.: The importance of encoding versus training with sparse coding and vector quantization. In: 28th International Conference on Machine Learning, pp. 921–928 (2011)
- [13] Csurka, G., Dance, C., Fan, L., Willamowski, J., Bray, C.: Visual categorization with bags of keypoints. In: ECCV Workshop on Statistical Learning in Computer Vision, pp. 59–74 (2004)

- [14] Dhillon, I.S., Modha, D.M.: Concept decompositions for large sparse text data using clustering. *Machine Learning* 42(1), 143–175 (2001)
- [15] Efron, B., Hastie, T., Johnstone, I., Tibshirani, R.: Least angle regression. *The Annals of statistics* 32(2), 407–499 (2004)
- [16] Fei-Fei, L., Perona, P.: A Bayesian hierarchical model for learning natural scene categories. In: *Computer Vision and Pattern Recognition*, vol. 2, pp. 524–531 (2005)
- [17] Garrigues, P., Olshausen, B.: Group sparse coding with a laplacian scale mixture prior. In: *Advances in Neural Information Processing Systems*, vol. 23, pp. 676–684 (2010)
- [18] van Gemert, J.C., Geusebroek, J.-M., Veenman, C.J., Smeulders, A.W.M.: Kernel Codebooks for Scene Categorization. In: Forsyth, D., Torr, P., Zisserman, A. (eds.) *ECCV 2008, Part III. LNCS*, vol. 5304, pp. 696–709. Springer, Heidelberg (2008)
- [19] Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: *14th International Conference on Artificial Intelligence and Statistics*, pp. 315–323 (2011)
- [20] Goodfellow, I., Courville, A., Bengio, Y.: Spike-and-slab sparse coding for unsupervised feature discovery. In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning* (2011)
- [21] Hinton, G., Osindero, S., Teh, Y.: A fast learning algorithm for deep belief nets. *Neural Computation* 18(7), 1527–1554 (2006)
- [22] Hyvärinen, A., Hurri, J., Hoyer, P.: *Natural Image Statistics*. Springer (2009)
- [23] Hyvärinen, A., Oja, E.: Independent component analysis: algorithms and applications. *Neural Networks* 13(4-5), 411–430 (2000)
- [24] Jarrett, K., Kavukcuoglu, K., Ranzato, M., LeCun, Y.: What is the best multi-stage architecture for object recognition? In: *12th International Conference on Computer Vision*, pp. 2146–2153 (2009)
- [25] Krizhevsky, A.: Learning multiple layers of features from Tiny Images. Master's thesis, Dept. of Comp. Sci., University of Toronto (2009)
- [26] Krizhevsky, A.: Convolutional Deep Belief Networks on CIFAR-10 (2010) (unpublished manuscript)
- [27] Lai, K., Bo, L., Ren, X., Fox, D.: A large-scale hierarchical multi-view RGB-D object dataset. In: *International Conference on Robotics and Automation*, pp. 1817–1824 (2011)
- [28] Lazebnik, S., Schmid, C., Ponce, J.: Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In: *Computer Vision and Pattern Recognition* (2006)
- [29] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Backpropagation applied to handwritten zip code recognition. *Neural Computation* 1, 541–551 (1989)
- [30] LeCun, Y., Huang, F.J., Bottou, L.: Learning methods for generic object recognition with invariance to pose and lighting. In: *Computer Vision and Pattern Recognition*, vol. 2, pp. 97–104 (2004)
- [31] Lee, H., Grosse, R., Ranganath, R., Ng, A.Y.: Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In: *26th International Conference on Machine Learning*, pp. 609–616 (2009)
- [32] Mairal, J., Bach, F., Ponce, J., Sapiro, G.: Online learning for matrix factorization and sparse coding. *Journal of Machine Learning Research* 11, 19–60 (2010)
- [33] Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: *27th International Conference on Machine Learning*, pp. 807–814 (2010)
- [34] Olshausen, B.A., Field, D.J.: Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature* 381(6583), 607–609 (1996)

- [35] Raina, R., Battle, A., Lee, H., Packer, B., Ng, A.: Self-taught learning: transfer learning from unlabeled data. In: 24th International Conference on Machine learning, pp. 759–766 (2007)
- [36] Tibshirani, R.: Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 267–288 (1996)
- [37] Varma, M., Zisserman, A.: A statistical approach to material classification using image patch exemplars. *Transactions on Pattern Analysis and Machine Intelligence* 31(11), 2032–2047 (2009)
- [38] Wang, J., Yang, J., Yu, K., Lv, F., Huang, T., Gong, Y.: Locality-constrained linear coding for image classification. In: Computer Vision and Pattern Recognition, pp. 3360–3367 (2010)
- [39] Yang, J., Yu, K., Gong, Y., Huang, T.S.: Linear spatial pyramid matching using sparse coding for image classification. In: Computer Vision and Pattern Recognition, pp. 1794–1801 (2009)
- [40] Yu, K., Zhang, T., Gong, Y.: Nonlinear learning using local coordinate coding. In: Advances in Neural Information Processing Systems, vol. 22, pp. 2223–2231 (2009)
- [41] Zetzsche, C., Krieger, G., Wegmann, B.: The atoms of vision: Cartesian or polar? *Journal of the Optical Society of America* 16(7), 1554–1565 (1999)
- [42] Zhou, X., Yu, K., Zhang, T., Huang, T.S.: Image Classification Using Super-Vector Coding of Local Image Descriptors. In: Daniilidis, K., Maragos, P., Paragios, N. (eds.) *ECCV 2010, Part V. LNCS*, vol. 6315, pp. 141–154. Springer, Heidelberg (2010)

Deep Big Multilayer Perceptrons for Digit Recognition

Dan Claudiu Ciresan^{1,2}, Ueli Meier^{1,2},
 Luca Maria Gambardella^{1,2}, and Jürgen Schmidhuber^{1,2}

¹ IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland

² University of Lugano & SUPSI, Switzerland

Abstract. The competitive MNIST handwritten digit recognition benchmark has a long history of broken records since 1998. The most recent advancement by others dates back 8 years (error rate 0.4%). Good old on-line back-propagation for plain multi-layer perceptrons yields a very low 0.35% error rate on the MNIST handwritten digits benchmark with a single MLP and 0.31% with a committee of seven MLP. All we need to achieve this until 2011 best result are many hidden layers, many neurons per layer, numerous deformed training images to avoid overfitting, and graphics cards to greatly speed up learning.

Keywords: NN (Neural Network), MLP (Multilayer Perceptron), GPU (Graphics Processing Unit), training set deformations, MNIST¹, committee, BP (back-propagation).

Note: This work combines three previously published papers [6, 7, 22].

23.1 Introduction

Automatic handwriting recognition is of academic and commercial interest. Current algorithms are already pretty good at learning to recognize handwritten digits. Post offices use them to sort letters; banks use them to read personal checks. MNIST [21] is the most widely used benchmark for isolated handwritten digit recognition. More than a decade ago, artificial neural networks called Multilayer Perceptrons or MLPs [35, 20, 29] were among the first classifiers tested on MNIST. Most had few layers or few artificial neurons (units) per layer [21], but apparently back then they were the biggest feasible MLPs, trained when CPU cores were at least 20 times slower than today. A more recent MLP with a single hidden layer of 800 units achieved 0.70% error [33]. However, more complex methods listed on the MNIST web page always seemed to outperform MLPs, and the general trend went towards more and more complex variants of Support Vector

¹ <http://yann.lecun.com/exdb/mnist/>

Machines or SVMs [13] and combinations of NNs and SVMs [19] etc. Convolutional neural networks (CNNs) achieved a record-breaking 0.40% error rate [33], using novel elastic training image deformations. Recent methods pre-train each hidden CNN layer one by one in an unsupervised fashion (this seems promising especially for small training sets), then use supervised learning to achieve 0.39% error rate [26, 27]. The biggest MLP so far [31] also was pre-trained without supervision, then piped its output into another classifier to achieve an error of 1% without domain-specific knowledge. Deep MLPs initialized by unsupervised pretraining were also successfully applied to speech recognition [23].

Are all these complexifications of plain MLPs really necessary? Can't one simply train really big plain MLPs on MNIST? One reason is that at first glance deep MLPs do not seem to work better than shallow networks [1]. Training them is hard as back-propagated gradients quickly vanish exponentially in the number of layers [16, 18, 15], just like in the first recurrent neural networks [17]. Indeed, previous deep networks successfully trained with back-propagation (BP) either had few free parameters due to weight-sharing [21, 33] or used unsupervised, layer-wise pre-training [14, 1, 26]. But is it really true that deep BP-MLPs do not work at all, or do they just need more training time? How to test this? Unfortunately, on-line BP for hundreds/thousands of epochs on large MLPs may take weeks or months on standard serial computers. But can't one parallelize it? Well, on computer clusters this is hard due to communication latencies between individual computers. Parallelization across training cases and weight updates for mini-batches [24] might alleviate this problem, but still leaves the task of parallelizing fully online-BP. Only GPUs are capable of such fine grained parallelism. Multi-threading on a multi-core processor is not easy either. We may speed up BP using SSE (Streaming Single Instruction, Multiple Data Extensions), either manually, or by setting appropriate compiler flags. The maximum theoretical speedup under single precision floating-point, however, is four, which is not enough. And MNIST is large - its 60,000 images take almost 50MB, too much to fit in the L2/L3 cache of any current processor. This requires to continually access data in considerably slower RAM. To summarize, currently it is next to impossible to train big MLPs on CPUs.

We showed how to overcome all these problems by training large, deep MLPs on graphics cards [6] and obtained an error rate of 0.35% with a deep MLP. Deformations proved essential to prevent MLPs with up to 12 million free parameters from overfitting. To let the deformation process keep up with network training speed we had to port it onto the GPU as well.

At some stage in the classifier design process one usually has collected a set of possible classifiers. Often one of them yields best performance. Intriguingly, however, the sets of patterns misclassified by the different classifiers do not necessarily overlap. This information could be harnessed in a committee. In the context of handwritten recognition it was already shown [4] how a combination of various classifiers can be trained more quickly than a single classifier yielding the same error rate. Here we focus on improving recognition rate using a committee of MLP. Our goal is to produce a group of classifiers whose errors on

various parts of the training set differ (are uncorrelated) as much as possible [2]. We show that for handwritten digit recognition this can be achieved by training identical classifiers on data normalized in different ways prior to training.

23.2 Data

MNIST consists of two datasets, one for training (60,000 images) and one for testing (10,000 images). Many studies divide the training set into two sets consisting of 50,000 images for training and 10,000 for validation. Our network is trained on slightly deformed images, continually generated in on-line fashion; hence we may use the whole un-deformed training set for validation, without wasting training images. Pixel intensities of the original gray scale images range from 0 (background) to 255 (max foreground intensity). $28 \times 28 = 784$ pixels per image get mapped to real values $\frac{\text{pixel intensity}}{127.5} - 1.0$ in $[-1.0, 1.0]$, and are fed into the NN input layer.

23.3 Architectures

We train 5 MLPs with 2 to 9 hidden layers and varying numbers of hidden units. Mostly but not always the number of hidden units per layer decreases towards the output layer (Table 23.3). There are 1.34 to 12.11 million free parameters (or weights, or synapses).

We use standard on-line BP [30], without momentum, but with a variable learning rate that shrinks by a multiplicative constant after each epoch, from 10^{-3} down to 10^{-6} . Weights are initialized with a uniform random distribution in $[-0.05, 0.05]$. Each neuron's activation function is a scaled hyperbolic tangent: $y(a) = A \tanh Ba$, where $A = 1.7159$ and $B = 0.6666$ [21], and a softmax output layer is used. Weight initialization and annealing rate are not overly important as long as sensible choices are made.

23.4 Deforming Images to Get More Training Instances

So far, the best results on MNIST were obtained by deforming training images [33], thus greatly increasing their number. This allows for training networks with many weights without overfitting. We combine affine (rotation, scaling and horizontal shearing) and elastic deformations (Figure 23.1), characterized by the following real-valued parameters:

- σ and α : for elastic distortions emulating uncontrolled oscillations of hand muscles [33];
- β : a random angle from $[-\beta, +\beta]$ describes either rotation or horizontal shearing. In case of shearing, $\tan \beta$ defines the ratio between horizontal displacement and image height;
- γ_x, γ_y : for horizontal and vertical scaling, randomly selected from $[1 - \gamma/100, 1 + \gamma/100]$.

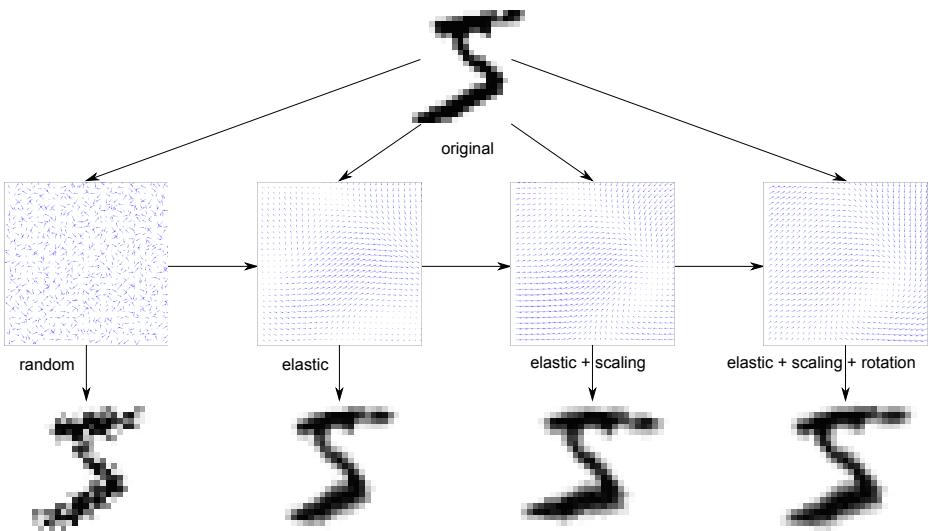


Fig. 23.1. Original digit (top) and distorted digits (bottom). The digit was distorted with four different displacement fields shown in the middle.

Each affine deformation is fully defined by the corresponding real-valued parameter that is randomly drawn from a uniform interval. Building the elastic deformation field on the other hand consists of three parts: 1) create an initial random distortion vector field, 2) smooth the random distortion field by convolving it with a Gaussian kernel defined by a standard deviation σ , and 3) scale the smoothed deformation field with α , the elastic scaling parameter.

At the beginning of every epoch the entire original MNIST training set gets deformed. Initial experiments with small networks suggested the following deformation parameters: $\sigma = 5.0 - 6.0$, $\alpha = 36.0 - 38.0$, $\gamma = 15 - 20$. Since digits 1 and 7 are similar they get rotated/sheared less ($\beta = 7.5^\circ$) than other digits ($\beta = 15.0^\circ$).

It takes 83 CPU seconds to deform the 60,000 MNIST training images, most of them (75 seconds) for elastic distortions. Only the most time-consuming part of the latter—convolution with a Gaussian kernel—is ported to the GPU. The MNIST training set is split into 600 sequentially processed minibatches of 100 samples each. MNIST digits are scaled from the original 28×28 pixels to 29×29 pixels, to get a proper center, which simplifies convolution. Each batch grid (10×10 images) has 290×290 cells, zero-padded to 310×310 , thus avoiding margin effects when applying a Gaussian convolution kernel of size 21×21 . The GPU program groups many threads into a block, where they share the same

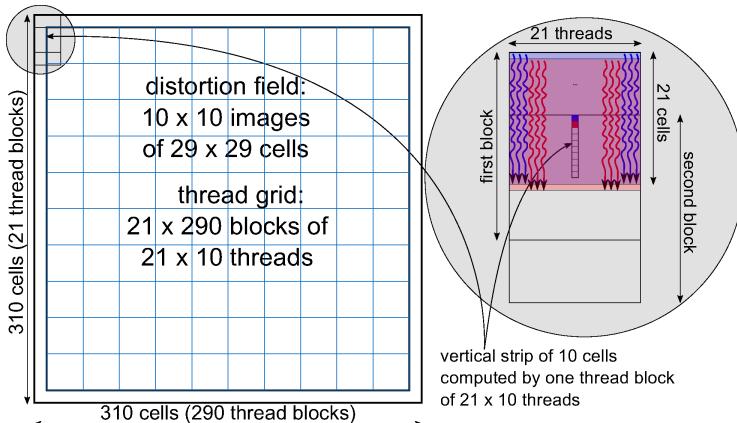


Fig. 23.2. Mapping the thread grid of convolution onto the distortion field

Gaussian kernel and parts of the random field. All 29×290 blocks contain 21 (the kernel size) \times 10 threads, each computing a vertical strip of the convolution (Figure 23.2). Generating the elastic displacement field takes only 1.5 seconds. Deforming the whole training set is more than 10 times faster, taking 6.5 instead of the original 83 seconds. Further optimizations would be possible by porting all deformations onto GPU, and by using the hardware's interpolation capabilities to perform the final bilinear interpolation. We omitted these since deformations are already pretty fast (deforming all images of one epoch takes only 3-10 % of total computation time, depending on MLP size).

23.5 Forming a Committee

The training procedure of a single network of the committee is summarized in Figure 23.3. Each network is trained separately on normalized or original data. The normalization is done for all digits in the training set prior to training (normalization stage). For the network trained on original MNIST data the normalization step is omitted. Normalization of the original MNIST data is mainly motivated by practical experience. MNIST digits are already normalized such that the width or height of the bounding box equals 20 pixels. The variation of the aspect ratio for various digits is quite large, and we normalize the width of the bounding box to range from 10 to 20 pixels with a step-size of 2 pixels prior to training for all digits except ones. Normalizing the original MNIST training data results in 6 normalized training sets. In total we perform experiments with seven different data sets (6 normalized and the original MNIST).

We perform six experiments to analyze performance improvements due to committees. Each committee consists of seven randomly initialized one-hidden-layer MLPs with 800 hidden units, trained with the same algorithm on randomly

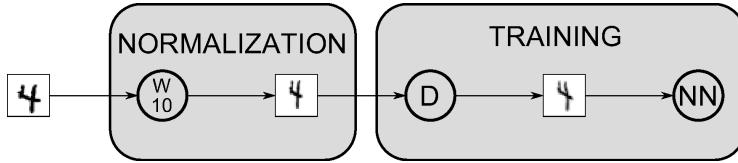


Fig. 23.3. Training a committee member. Original MNIST training data (left digit) is normalized (W_{10}) prior to training (middle digit). The normalized data is distorted (D) for each training epoch and used as input (right digit) to the network (NN). Each depicted digit represents the whole training set.

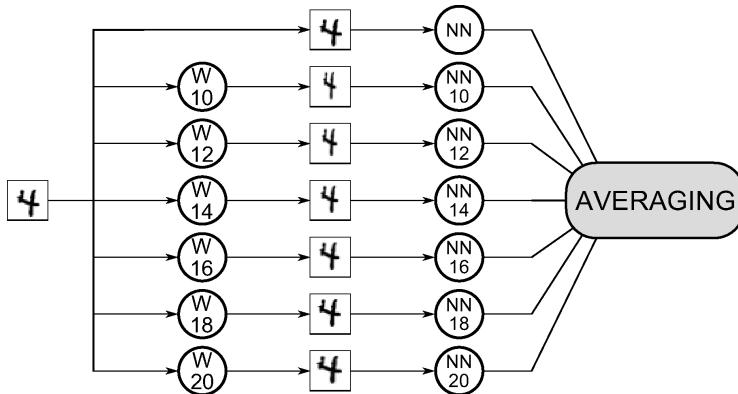


Fig. 23.4. Testing with a committee. If required, the input digits are width-normalized (W blocks) and then processed by the corresponding MLP. The committee is formed by averaging the outputs of all MLPs.

selected batches. The six committees differ only in how the data are normalized (or not) prior to training and on how the data are deformed during training. The committees are formed by simply averaging the corresponding outputs as shown in Figure 23.4.

The first two experiments are performed on undeformed original MNIST images. We train a committee of seven MLPs on original MNIST and we also form a committee of MLPs trained on normalized data. In Table 23.1 the error rates are listed for each of the individual nets and the committees. The improvement of the committee with respect to the individual nets is marginal for the first experiment. Adding normalization, the individual experts as well as the corresponding committee of the second experiment achieve substantially better recognition rates.

To study the combined effect of normalization and deformation, we perform four additional experiments on deformed MNIST (Tab. 23.2). Unless stated

Table 23.1. Error rates of individual nets and of the two resulting committees. For experiment 1 seven randomly initialized nets are trained on the original MNIST, whereas for experiment 2 seven randomly initialized nets are trained on width-normalized data: WN x - Width Normalization of the bounding box to be x pixels wide; ORIG - original MNIST.

	Error rate [%]	
	Exp. 1	Exp. 2
Net 1:	init 1: 1.79	WN 10: 1.62
Net 2:	init 2: 1.80	WN 12: 1.37
Net 3:	init 3: 1.77	WN 14: 1.48
Net 4:	init 4: 1.72	WN 16: 1.53
Net 5:	init 5: 1.91	WN 18: 1.56
Net 6:	init 6: 1.86	WN 20: 1.49
Net 7:	init 7: 1.75	ORIG: 1.79
Average:	1.70	1.31

otherwise, default elastic deformation parameters $\sigma = 6$ and $\alpha = 36$ are used. All experiments with deformed images use independent horizontal and vertical scaling of maximum 12.5% and a maximum rotation of $\pm 12.5^\circ$. Experiment 3 is similar to Experiment 1, except that the data are continually deformed. Error rates of the individual experts are much lower than without deformation (Tab. 23.1). In experiment 4 we randomly reselect training and validation sets for each of the individual experts, simulating in this way the bootstrap aggregation technique [3]. The resulting committee performs slightly better than that of experiment 3. In experiment 5 we vary deformations for each individual network. Error rates of some of the nets are bigger than in experiments 3 and 4, but the resulting committee has a lower error rate. In the last experiment we train seven MLPs on width-normalized images that are also continually deformed during training. The error rate of the committee (0.43 %) is the best result ever reported for such a simple architecture. We conclude that width-normalization is essential for good committee performance, i.e. it is not enough to form a committee from trained nets with different initializations (experiment 3) or trained on subsets of the original dataset (experiment 4).

23.6 Using the GPU to Train Deep MLPs

Using simple tricks, such as creating a virtually infinite amount of training data through random distortions at the beginning of every epoch and forming a committee of experts trained on differently preprocessed data, state-of-the art results are obtained on MNIST with a relatively small (800 hidden units) single hidden layer MLP. Here we report results using deep MLPs, with as many as 5 hidden layers and up to 12 millions of free parameters, that are prohibitive to train on

Table 23.2. Error rates of the individual nets and of the resulting committees. In experiments 3 and 4 seven randomly initialized nets are trained on deformed ($\sigma = 6$, $\alpha = 36$) MNIST, whereas in experiment 4 training and validation sets are reselected. In experiment 5 seven randomly initialized nets are trained on deformed (different σ , α) MNIST, and in experiment 6 seven randomly initialized nets are trained on width-normalized, deformed ($\sigma = 6$, $\alpha = 36$) MNIST. WN x - Width Normalization of the bounding box to be x pixels wide; ORIG - original MNIST.

	Error rate [%]			
	Exp. 3	Exp. 4	Exp. 5	Exp. 6
Net 1:	init 1:	0.72	0.68	$\sigma = 4.5 \alpha = 36$: 0.69
Net 2:	init 2:	0.71	0.82	$\sigma = 4.5 \alpha = 42$: 0.94
Net 3:	init 3:	0.72	0.73	$\sigma = 6.0 \alpha = 30$: 0.55
Net 4:	init 4:	0.71	0.69	$\sigma = 6.0 \alpha = 36$: 0.72
Net 5:	init 5:	0.62	0.71	$\sigma = 6.0 \alpha = 42$: 0.60
Net 6:	init 6:	0.65	0.70	$\sigma = 7.5 \alpha = 30$: 0.86
Net 7:	init 7:	0.69	0.75	$\sigma = 7.5 \alpha = 36$: 0.79
Average:		0.56	0.53	0.49
				WN 10: 0.64
				WN 12: 0.78
				WN 14: 0.70
				WN 16: 0.60
				WN 18: 0.59
				WN 20: 0.70
				ORIG: 0.71

current CPUs but can successfully be trained on GPUs in a few days. All simulations were performed on a computer with a Core i7 920 2.66GHz processor, 12GB of RAM, and a GTX 480 graphics card. The GPU accelerates the deformation routine by a factor of 10 (only elastic deformations are GPU-optimized); the forward propagation (FP) and BP routines are sped up by a factor of 50. We pick the trained MLP with the lowest validation error, and evaluate it on the MNIST test set.

23.6.1 Single MLP

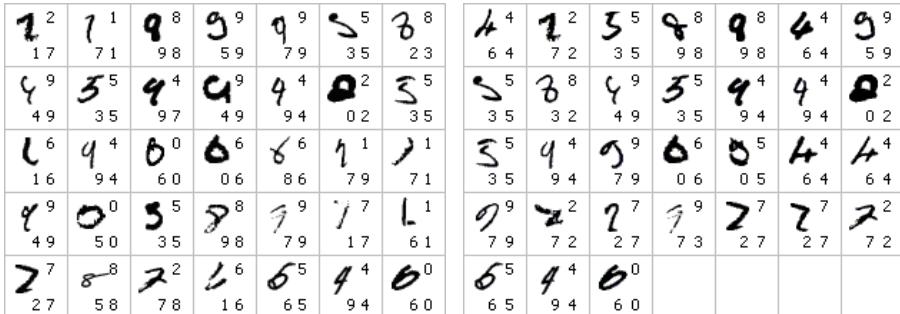
We train various MLP and summarize the results in Table 23.3. Training starts with a learning rate of 10^{-3} multiplied with 0.997 after every epoch until it reaches 10^{-6} , thus resulting in more than 2000 epochs, which can be computed in a few days even for the biggest net. The best network has an error rate of only 0.35% (35 out of 10,000 digits). This is better than the best previously published results, namely, 0.39% [26] and 0.40% [33], both obtained by more complex methods. The 35 misclassified digits are shown in Figure 23.5a. Many of them are ambiguous and/or uncharacteristic, with obviously missing parts or strange strokes etc. Interestingly, the second guess of the network is correct for 30 out of the 35 misclassified digits. The best test error of this MLP is even lower (0.32%) and may be viewed as the maximum capacity of the network, i.e. what it can learn if we do not get the result for the lowest error on validation set. Performance clearly profits from adding hidden layers and more units per

Table 23.3. Error rates on MNIST test set. Architecture: 841 input neurons, hidden layers containing 2500, 2000, 1500, 1000 and 500 neurons, and 10 outputs. TEfBV - test error for best validation, BTE - best test error.

ID	architecture (number of neurons in each layer)	TEfBV [%]	BTE [%]	simulation time [h]	weights [millions]	test error [%] no distortion
1	1000, 500, 10	0.49	0.44	23.4	1.34	1.78
2	1500, 1000, 500, 10	0.46	0.40	44.2	3.26	1.85
3	2000, 1500, 1000, 500, 10	0.41	0.39	66.7	6.69	1.73
4	2500, 2000, 1500, 1000, 500, 10	0.35	0.32	114.5	12.11	1.71
5	$9 \times 1000, 10$	0.44	0.43	107.7	8.86	1.81

layer. For example, network 5 has more but smaller hidden layers than network 4 (Table 23.3).

Networks with up to 12 million weights can successfully be trained by plain gradient descent to achieve test errors below 1% after 20-30 epochs in less than 2 hours of training. How can networks with so many parameters generalize well on the unseen test set? Answer: the continual deformations of the training set generate a virtually infinite supply of training examples, and the network rarely sees any training image twice. Without any distortions, the error for all networks is around 1.7-1.8% (last column in Table 23.3).



(a)

(b)

Fig. 23.5. The misclassified digits, together with the two most likely predictions (bottom, from left to right) and the correct label according to MNIST (top, right): (a) the best network from Table 23.3. (b) the committee from Table 23.4.

23.6.2 Committee of MLP

Here we list results of a committee of MLP with the architecture that obtained 0.35% error rate on MNIST (841 neurons in the input layer, five hidden layers

Table 23.4. Error rates of the individual nets and of the resulting committee. Architecture: 841 input neurons, hidden layers containing 2500, 2000, 1500, 1000 and 500 neurons, and 10 outputs. WN x—Width Normalization of the bounding box to be x pixels wide.

WN	10	12	14	16	18	20	ORIGINAL MNIST
test error [%]	0.52	0.45	0.44	0.49	0.36	0.38	0.35
committee error [%]						0.31	

containing 2500, 2000, 1500, 1000 and 500 neurons, and 10 outputs). We train six additional nets with the same architecture on normalized data (the width of the digits is normalized prior to training) and form a committee by averaging the predictions of the individual nets (Table 23.4). The width-normalization is essential for good committee performance as shown in Section 23.5. All committee members distort their width-normalized training dataset before each epoch.

Interestingly, the error of the extremely simple committee (0.31%) is lower than those of the individual nets. This is the best result ever reported on MNIST using MLP. Many of the 31 misclassified digits (Figure 23.5b) are ambiguous and/or uncharacteristic, with obviously missing parts or strange strokes etc. Remarkably, the committee’s second guess is correct for 29 of the 31.

23.7 Discussion

In recent decades the amount of raw computing power per Euro has grown by a factor of 100-1000 per decade. Our results show that this ongoing hardware progress may be more important than advances in algorithms and software (although the future will belong to methods combining the best of both worlds). Current graphics cards (GPUs) are already more than 50 times faster than standard microprocessors when it comes to training big and deep neural networks by the ancient algorithm, on-line back-propagation (weight update rate up to $7.5 \times 10^9 / s$, and more than 10^{15} per trained network). On the competitive MNIST handwriting benchmark, single precision floating-point GPU-based neural nets surpass all previously reported results, including those obtained by much more complex methods involving specialized architectures, unsupervised pre-training, combinations of machine learning classifiers etc. Training sets of sufficient size to avoid overfitting are obtained by appropriately deforming images. Of course, the approach is not limited to handwriting, and obviously holds great promise for many visual and other pattern recognition problems.

Although big deep MLP are very powerful general classifiers when combined with an appropriate distortion algorithm to enhance the training set, they cannot compete with dedicated architectures such as max-pooling convolutional neural networks on complex image classification problems. For tasks more difficult than handwritten digit recognition MLP are not competitive anymore, both in classification performance and required training time. We have recently shown [11]

that large convolutional neural networks combined with max-pooling [32] improve the state-of-the-art by 30-80% for a plethora of benchmarks like Latin letters [8], Chinese characters [11], traffic signs [9, 12], stereo projection of 3D models [10, 11] and even small natural images [11].

Acknowledgments. Part of this work got started when Dan Cireşan was a PhD student at University "Politehnica" of Timişoara. He would like to thank his PhD advisor, Ştefan Holban, for his guidance, and Răzvan Moşincat for providing a CPU framework for MNIST. This work was partially supported by the Swiss Commission for Technology and Innovation (CTI), Project n. 9688.1 IFF: Intelligent Fill in Form., and by a FP7-ICT-2009-6 EU Grant, Project Code 270247: A Neuro-dynamic Framework for Cognitive Robotics: Scene Representations, Behavioral Sequences, and Learning.

Appendix - GPU Implementation

Graphics Processing Unit

Until 2007 the only way to program a GPU was to translate the problem-solving algorithm into a set of graphical operations. Despite being hard to code and difficult to debug, several GPU-based NN implementations were developed when GPUs became faster than CPUs. Two layer MLPs [34] and CNNs [5] have been previously implemented on GPUs. Although speedups were relatively modest, these studies showed how GPUs can be used for machine learning. More recent GPU-based CNNs trained in batch mode are two orders of magnitude faster than CPU-based CNNs [32].

The GPU code is written using CUDA (Compute Unified Device Architecture), a C-like general programming language. GPU speed and memory bandwidth are vastly superior to those of CPUs, and crucial for fast MLP implementations. To fully understand our algorithm in terms of GPU / CUDA, please visit the NVIDIA website [25]. According to CUDA terminology, the CPU is called *host* and the graphics card *device* or *GPU*.

Deformations

Only the most time-consuming part of the latter – convolution with a gaussian kernel [33] – is ported to the GPU. The MNIST training set is split into 600 sequentially processed batches. MNIST digits are scaled from the original 28×28 pixels to 29×29 pixels, to get a proper center, which simplifies convolution. An image grid has 290×290 cells, zero-padded to 300×300 , thus avoiding margin effects when applying a Gaussian convolution kernel of size 21×21 .

Our GPU program groups many threads into a block, where they share the same gaussian kernel and parts of the random field. The blocks contain 21 (the kernel size) $\times 10$ threads, each computing a vertical strip of the convolution operation (Listing 23.1).

Listing 23.1. Convolution Kernel for elastic distortion

```

1  __global__ void ConvolveField(float *randomfield, int width, int height,
2      float *kernel, float *outputfield, float elasticScale){
3      float sum=0;
4      const int stride_k=GET_STRIDE(GAUSSIAN_FIELD_SIZE,pitch_x
5          >>2); //stride for gaussian kernel
6      __shared__ float K[GAUSSIAN_FIELD_SIZE][stride_k]; //kernel (21 x
7          32 values)
8      __shared__ float R[GAUSSIAN_FIELD_SIZE+9][
9          GAUSSIAN_FIELD_SIZE]; //random field (30 x 21 values)
10     __shared__ float s[10][GAUSSIAN_FIELD_SIZE]; //partial sums (10 x
11         21 values)
12     int stride_in=GET_STRIDE(width,pitch_x>>2); //random field stride as
13         a multiple of 32
14     int stride_out=GET_STRIDE(width-GAUSSIAN_FIELD_SIZE+1,
15         pitch_x>>2); //output stride as a multiple of 32
16
17     //loading gaussian kernel into K (21 x 21 values)
18     K[ 0+threadIdx.y][threadIdx.x] = kernel[( 0+threadIdx.y)*stride_k +
19         threadIdx.x];//rows 0..9
20     K[10+threadIdx.y][threadIdx.x] = kernel[(10+threadIdx.y)*stride_k +
21         threadIdx.x];//rows 10..19
22     if(threadIdx.y==0)
23         K[20+threadIdx.y][threadIdx.x] = kernel[(20+threadIdx.y)*stride_k +
24             threadIdx.x];//row 20
25
26     //loading randomfield into R
27     //0..9 x 21 values
28     R[ 0+threadIdx.y][threadIdx.x] = randomfield[(10*blockIdx.y+ 0+
29         threadIdx.y)*stride_in + blockIdx.x + threadIdx.x];
30     //10..19 x 21 values
31     R[10+threadIdx.y][threadIdx.x] = randomfield[(10*blockIdx.y+10+
32         threadIdx.y)*stride_in + blockIdx.x + threadIdx.x];
33     //20..29 x 21 values
34     R[20+threadIdx.y][threadIdx.x] = randomfield[(10*blockIdx.y+20+
35         threadIdx.y)*stride_in + blockIdx.x + threadIdx.x];
36     __syncthreads(); //wait until everything is read into shared memory
37
38     //computing partial sums
39     #pragma unroll 21 //GAUSSIAN_FIELD_SIZE
40     for(int i=0;i<GAUSSIAN_FIELD_SIZE;i++)
41         sum += R[threadIdx.y + i][threadIdx.x] * K[i][threadIdx.x];
42     s[threadIdx.y][threadIdx.x]=sum;
43     __syncthreads();
44
45     if(threadIdx.x==0){ //the first column of threads computes the final values
46         of the convolutions
47         #pragma unroll 20//GAUSSIAN_FIELD_SIZE-1
48         for(int i=1;i<GAUSSIAN_FIELD_SIZE;i++) sum+=s[threadIdx.y][i
49             ];
50         outputfield[(blockIdx.y*10+threadIdx.y)*stride_out + blockIdx.x] =
51             sum * elasticScale;
52     }
53 }
```

Training Algorithm

We closely follow the standard BP algorithm [30], except that BP of deltas and weight updates are disentangled and performed sequentially. This allows for more parallelism within each routine.

Forward Propagation

The algorithm is divided into two kernels. The weight matrix W is partitioned as illustrated in Figure 23.6.

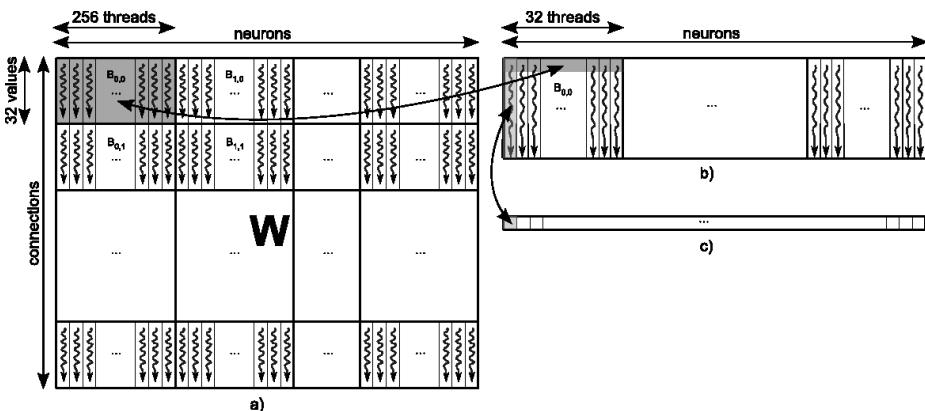


Fig. 23.6. Forward propagation: a) mapping of kernel 1 grid onto the padded weight matrix; b) mapping the kernel 2 grid onto the partial dot products matrix; c) output of forward propagation

Kernel 1. Each block has 256 threads (Figure 23.6a), each computing a partial dot product of 32 component vectors. The dot products are stored in a temporary matrix (Figure 23.6b). This kernel has a very high throughput: average memory bandwidth is 115GB/s. This is possible because many relatively small blocks keep the GPU busy. Each block uses shared memory for storing the previous layer activations, which are simultaneously read by the first 32 threads of each block and then used by all 256 threads. After thread synchronization, the partial dot products are computed in parallel (Listing 23.2). The number of instructions is kept to a minimum by pre-computing all common index parts.

Kernel 2. The thread grid (Figure 23.6b) has only one row of blocks consisting of *warp* threads, since each thread has to compute a complete dot product (Figure 23.6c) and then pipe it into the activation function. This kernel (Listing 23.2) is inefficient for layers with fewer than 1024 incoming connections per neuron, especially for the last layer which has only ten neurons, one for each digit. That is, its grid will have only one block, occupying only 6% of the GTX 480 GPU.

Listing 23.2. Forward propagation kernels

```

1  __global__ void MLP_FP_reduction_Kernel1(float *prevLN, float *W,
2      float *partialsum, unsigned int neurons, unsigned int prevneurons){
3      const int threads=256;
4      const int stride=GET_STRIDE(neurons,pitch_x>>2); //horizontal stride
5          of W matrix
6      int X=blockIdx.x*threads + threadIdx.x; //precomputing expressions
7      int Y=X+stride*blockIdx.y;
8      int Z=blockIdx.y*pitch_y*stride + X;
9      float sum=0.0f;
10     shared__float output[pitch_y];
11     if(blockIdx.y==0)
12         if(threadIdx.x==0) output[0]=1.0f;
13     else if(threadIdx.x<pitch_y) //there are only 32 values to read and
14         128 threads
15         output[threadIdx.x] = threadIdx.x-1<prevneurons ? prevLN[
16             threadIdx.x-1] : 0.0f;
17     else;
18     else if(threadIdx.x<pitch_y) //there are only 32 values to read and 128
19         threads
20         output[threadIdx.x] = blockIdx.y*pitch_y+threadIdx.x-1<
21             prevneurons ?
22                 prevLN[blockIdx.y*pitch_y+threadIdx.x-1] : 0.0f;
23     else;
24     syncthreads();
25     if(X<neurons){//compute partial sums
26         //pragma unroll 32
27         int size=0;
28         if((blockIdx.y+1)*pitch_y>=prevneurons+1)
29             size = prevneurons + 1 - blockIdx.y*pitch_y;
30         else size=pitch_y;
31         for (int ic=0; ic<size; ic++){
32             sum += output[ic] * W[Z];
33             Z+=stride;
34         }
35         partialsum[Y]=sum;
36     }
37 }
38 __global__ void MLP_FP_reduction_Kernel2(float *currLN, float *
39     partialsum, unsigned int neurons, unsigned int size){
40     float sum=0.0f;
41     int idx = blockIdx.x*(pitch_x>>2) + threadIdx.x; //precomputed index
42     unsigned int stride = GET_STRIDE(neurons,pitch_x>>2); //stride for
43         partialsum matrix
44
45     if(idx>=neurons) return; //is this thread computing a true neuron?
46     for (int i=0; i<size; i++) sum += partialsum[i*stride+idx]; //computing
47         the final dot product
48     currLN[idx] = SIGMOIDF(sum); //applying activation
49 }
```

Backward Propagation

This is similar to FP, but we need W^T for coalesced access. Instead of transposing the matrix, the computations are performed on patches of data read from device memory into shared memory, similar to the optimized matrix transposition algorithm of [28]. Shared memory access is much faster, without coalescing restrictions. Because we have to cope with layers of thousands of neurons, back-propagating deltas uses a reduction method implemented in two kernels communicating partial results via global memory (Listing 23.3).

Listing 23.3. Backpropagating deltas kernels

```

1  __global__ void backPropagateDeltasFC_A(float *indelta, float *weights,
2      unsigned int ncon, unsigned int nrneur, float *partial){
3      const int px = pitch_x>>2;
4      unsigned int stride_x = GET_STRIDE(nrneur,px);
5      unsigned int stride_y = GET_STRIDE(ncon,pitch_y);
6      float outd = 0.0;
7      int idx = blockIdx.x*px+threadIdx.x;
8      int X = blockIdx.y*pitch_y*stride_x + idx;
9      int Y = threadIdx.x;
10     __shared__ float w[32*33]; //pitch_y and px should be equal ! +1 to
11         // avoid bank conflict!
12     __shared__ float id[px]; //input delta
13     #pragma unroll 32 //read the weight patch in shared memory
14     for(int i=0;i<pitch_y;i++){w[Y]=weights[X]; X+=stride_x; Y+=33;}
15     //read the input delta patch in shared memory
16     if(idx>=nrneur) id[threadIdx.x]=0; //a fake input delta for nonexistent
17         //indelta
18     else id[threadIdx.x]=indelta[idx];
19     __syncthreads(); //not needed for block with warp number of threads:
12         //implicit synchronization
20     #pragma unroll 32 //compute partial results
21     for(int i=0;i<px;i++) outd+=w[threadIdx.x*33+i]*id[i];
22         //write out the partial results
23     partial[blockIdx.x*stride_y + blockIdx.y*pitch_y + threadIdx.x] = outd;
24 }
25 __global__ void backPropagateDeltasFC_B(float *outdelta, float *instates,
26     unsigned int ncon, unsigned int nrneur, float *partial){
27     int px=pitch_x>>2;
28     unsigned int stride_x = GET_STRIDE(nrneur,px);
29     unsigned int stride_y = GET_STRIDE(ncon,pitch_y);
30     float outd = 0.0;
31     int size=stride_x/px;
32     int idx=blockIdx.x*pitch_y+threadIdx.x;
33     if(idx==0); //true only for block and thread 0
34     else{
35         for(int i=0;i<size;i++)
36             outd+=partial[i*stride_y + idx];
37         outdelta[idx-1] = outd * DSIGMOIDF(instates[idx-1]); // -1 BIAS ...
38     }
39 }
```

Kernel 1. The bi-dimensional grid is divided into blocks of *warp* (32) threads. The kernel starts by reading a patch of 32×32 values from W. The stride of the shared memory block is 33 (*warp* + 1), thus avoiding all bank conflicts and significantly improving speed. Next, 32 input delta values are read and all memory locations that do not correspond to real neurons (because of vertical striding) are zero-padded to avoid branching in subsequent computations. The number of elements is fixed to *warp* size, and the computing loop is unrolled for further speedups. Before finishing, each thread writes its own partial dot product to global memory.

Kernel 2. This kernel completes BP of deltas by summing up partial deltas computed by the previous kernel. It multiplies the final result by the derivative of the activation function applied to the current neuron's state, and writes the new delta to global memory.

Weight Updating

The algorithm (Listing 23.4) starts by reading the appropriate delta, and pre-computes all repetitive expressions. Then the first 16 threads read the states from global memory into shared memory. The “bias neuron” with constant activation 1.0 is dealt with by conditional statements, which could be avoided through expressions containing the conditions. Once threads are synchronized, each single thread updates 16 weights in a fixed unrolled loop.

Listing 23.4. Weights adjustment kernel

```

1  __global__ void adjustWeightsFC(float *states, float *deltas, float *weights,
2      float eta, unsigned int ncon, unsigned int nrneur){
3          const int pitch_y=16;
4          const int threads=256;
5          unsigned int px = pitch_x >> 2;
6          unsigned int stride_x = GET_STRIDE(nrneur,px);
7          float etadeltak = eta*deltas[blockIdx.x*threads+threadIdx.x];
8          int b=blockIdx.y*stride_x*pitch_y + threads*blockIdx.x + threadIdx.x;
9          __shared__ float st[pitch_y]; //for states
10         int cond1 = blockIdx.y || threadIdx.x;
11         int cond2 = (blockIdx.y+1)*pitch_y <= ncon;
12         int size = cond2 * pitch_y + !cond2 * (ncon%pitch_y);
13         if(threadIdx.x < pitch_y) st[threadIdx.x] = cond1 * states[blockIdx.y*
14             pitch_y + threadIdx.x - 1] + !cond1;
15         __syncthreads();
16
17         if (blockIdx.x*threads + threadIdx.x < nrneur){
18             #pragma unroll 16
19             for (int j=0; j<16; j++){
20                 t=weights[b];
21                 t -= etadeltak * st[j];
22                 weights[b]=t;
23                 b+=stride_x;}}
```

References

- [1] Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H.: Greedy layer-wise training of deep networks. In: Neural Information Processing Systems (2006)
- [2] Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2006)
- [3] Breiman, L.: Bagging predictors. *Machine Learning* 24, 123–140 (1996)
- [4] Chellapilla, K., Shilman, M., Simard, P.: Combining Multiple Classifiers for Faster Optical Character Recognition. In: Bunke, H., Spitz, A.L. (eds.) DAS 2006. LNCS, vol. 3872, pp. 358–367. Springer, Heidelberg (2006)
- [5] Chellapilla, K., Puri, S., Simard, P.: High performance convolutional neural networks for document processing. In: International Workshop on Frontiers in Handwriting Recognition (2006)
- [6] Ciresan, D.C., Meier, U., Gambardella, L.M., Schmidhuber, J.: Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation* 22(12), 3207–3220 (2010)
- [7] Ciresan, D.C., Meier, U., Gambardella, L.M., Schmidhuber, J.: Handwritten Digit Recognition with a Committee of Deep Neural Nets on GPUs. Technical Report IDSIA-03-11, Istituto Dalle Molle di Studi sull’Intelligenza Artificiale, IDSIA (2011)
- [8] Ciresan, D.C., Meier, U., Gambardella, L.M., Schmidhuber, J.: Convolutional neural network committees for handwritten character recognition. In: International Conference on Document Analysis and Recognition, pp. 1135–1139 (2011)
- [9] Ciresan, D.C., Meier, U., Masci, J., Schmidhuber, J.: A committee of neural networks for traffic sign classification. In: International Joint Conference on Neural Networks, pp. 1918–1921 (2011)
- [10] Ciresan, D.C., Meier, U., Masci, J., Gambardella, L.M., Schmidhuber, J.: Flexible, high performance convolutional neural networks for image classification. In: International Joint Conference on Artificial Intelligence, pp. 1237–1242 (2011)
- [11] Ciresan, D.C., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. In: Computer Vision and Pattern Recognition, pp. 3642–3649 (2012)
- [12] Ciresan, D.C., Meier, U., Masci, J., Schmidhuber, J.: Multi-column deep neural network for traffic sign classification. *Neural Networks* 32, 333–338 (2012)
- [13] Decoste, D., Scholkopf, B.: Training invariant support vector machines. *Machine Learning* (46), 161–190 (2002)
- [14] Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. *Science* 313 (2006)
- [15] Hinton, G.E.: To recognize shapes, first learn to generate images. Computational Neuroscience: Theoretical Insights into Brain Function (2007)
- [16] Hochreiter, S.: Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München (1991), <http://www7.informatik.tu-muenchen.de/~hochreit/>; advisor: J. Schmidhuber
- [17] Hochreiter, S., Schmidhuber, J.: Long short-term memory. *Neural Computation* 9, 1735–1780 (1997)
- [18] Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J.: Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In: Kremer, S.C., Kolen, J.F. (eds.) A Field Guide to Dynamical Recurrent Neural Networks. IEEE Press (2001)
- [19] Lauer, F., Suen, C., Bloch, G.: A trainable feature extractor for handwritten digit recognition. *Pattern Recognition* (40), 1816–1824 (2007)

- [20] LeCun, Y.: Une procédure d'apprentissage pour réseau à seuil asymétrique (a learning scheme for asymmetric threshold networks). In: Proceedings of Cognitiva 1985, Paris, France, pp. 599–604 (1985)
- [21] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86(11), 2278–2324 (1998)
- [22] Meier, U., Ciresan, D.C., Gambardella, L.M., Schmidhuber, J.: Better digit recognition with a committee of simple neural nets. In: ICDAR, pp. 1135–1139 (2011)
- [23] Mohamed, A., Dahl, G., Hinton, G.E.: Deep belief networks for phone recognition. In: Proc. of NIPS 2009 Workshop on Deep Learning for Speech Recognition and Related Applications (2009)
- [24] Nair, V., Hinton, G.E.: 3D object recognition with deep belief nets. In: Advances in Neural Information Processing Systems (2009)
- [25] NVIDIA: NVIDIA CUDA. Reference Manual, vol. 2.3. NVIDIA (2009)
- [26] Ranzato, M., Poultney, C., Chopra, S., LeCun, Y.: Efficient learning of sparse representations with an energy-based model. In: Platt, J., et al. (eds.) *Advances in Neural Information Processing Systems (NIPS 2006)*. MIT Press (2006)
- [27] Ranzato, M.: Fu Jie Huang, Y.L.B., LeCun, Y.: Unsupervised learning of invariant feature hierarchies with applications to object recognition. In: Proc. of Computer Vision and Pattern Recognition Conference (2007)
- [28] Ruetsch, G., Micikevicius, P.: Optimizing matrix transpose in cuda. In: NVIDIA GPU Computing SDK, pp. 1–2 (2009)
- [29] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In: Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1: Foundations, pp. 318–362. MIT Press, Cambridge (1986)
- [30] Russell, S., Norvig, P.: *Artificial Intelligence: A Modern Approach*, 2nd edn. Prentice-Hall, Englewood Cliffs (2003)
- [31] Salakhutdinov, R., Hinton, G.: Learning a nonlinear embedding by preserving class neighborhood structure. In: Proc. of the International Conference on Artificial Intelligence and Statistics, vol. 11 (2007)
- [32] Scherer, D., Behnke, S.: Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors. In: Proc. of NIPS 2009 Workshop on Large-Scale Machine Learning: Parallelism and Massive Datasets (2009)
- [33] Simard, P., Steinkraus, D., Platt, J.C.: Best practices for convolutional neural networks applied to visual document analysis. In: Seventh International Conference on Document Analysis and Recognition, pp. 958–963 (2003)
- [34] Steinkraus, D., Simard, P.Y.: Gpus for machine learning algorithms. In: International Conference on Document Analysis and Recognition, pp. 1115–1120 (2005)
- [35] Werbos, P.J.: *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University (1974)

A Practical Guide to Training Restricted Boltzmann Machines

Geoffrey E. Hinton

University of Toronto, Toronto, Ontario M5S 3G4
 Department of Computer Science
hinton@cs.toronto.edu

Abstract. Restricted Boltzmann machines (RBMs) have been used as generative models of many different types of data. RBMs are usually trained using the contrastive divergence learning procedure. This requires a certain amount of practical experience to decide how to set the values of numerical meta-parameters. Over the last few years, the machine learning group at the University of Toronto has acquired considerable expertise at training RBMs and this guide is an attempt to share this expertise with other machine learning researchers.

24.1 Introduction

Restricted Boltzmann machines (RBMs) have been used as generative models of many different types of data including labeled or unlabeled images [7], windows of mel-cepstral coefficients that represent speech [12], bags of words that represent documents [15], and user ratings of movies [17]. In their conditional form they can be used to model high-dimensional temporal sequences such as video or motion capture data [20] or speech [11]. Their most important use is as learning modules that are composed to form deep belief nets [7].

RBM s are usually trained using the contrastive divergence learning procedure [5]. This requires a certain amount of practical experience to decide how to set the values of numerical meta-parameters such as the learning rate, the momentum, the weight-cost, the sparsity target, the initial values of the weights, the number of hidden units and the size of each mini-batch. There are also decisions to be made about what types of units to use, whether to update their states stochastically or deterministically, how many times to update the states of the hidden units for each training case, and whether to start each sequence of state updates at a data-vector. In addition, it is useful to know how to monitor the progress of learning and when to terminate the training.

For any particular application, the code that was used gives a complete specification of all of these decisions, but it does not explain why the decisions were made or how minor changes will affect performance. More significantly, it does not provide a novice user with any guidance about how to make good decisions for a new application. This requires some sensible heuristics and the ability to relate failures of the learning to the decisions that caused those failures.

Over the last few years, the machine learning group at the University of Toronto has acquired considerable expertise at training RBMs and this guide is an attempt to share this expertise with other machine learning researchers.

24.2 An Overview of Restricted Boltzmann Machines and Contrastive Divergence

Skip this section if you already know about RBMs

Consider a training set of binary vectors which we will assume are binary images for the purposes of explanation. The training set can be modeled using a two-layer network called a “Restricted Boltzmann Machine” [18, 2, 5] in which stochastic, binary pixels are connected to stochastic, binary feature detectors using symmetrically weighted connections. The pixels correspond to “visible” units of the RBM because their states are observed; the feature detectors correspond to “hidden” units. A joint configuration, (\mathbf{v}, \mathbf{h}) of the visible and hidden units has an energy [9] given by:

$$E(\mathbf{v}, \mathbf{h}) = - \sum_{i \in \text{visible}} a_i v_i - \sum_{j \in \text{hidden}} b_j h_j - \sum_{i,j} v_i h_j w_{ij} \quad (24.1)$$

where v_i, h_j are the binary states of visible unit i and hidden unit j , a_i, b_j are their biases and w_{ij} is the weight between them. The network assigns a probability to every possible pair of a visible and a hidden vector via this energy function:

$$p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} e^{-E(\mathbf{v}, \mathbf{h})} \quad (24.2)$$

where the “partition function”, Z , is given by summing over all possible pairs of visible and hidden vectors:

$$Z = \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \quad (24.3)$$

The probability that the network assigns to a visible vector, \mathbf{v} , is given by summing over all possible hidden vectors:

$$p(\mathbf{v}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \quad (24.4)$$

The probability that the network assigns to a training image can be raised by adjusting the weights and biases to lower the energy of that image and to raise the energy of other images, especially those that have low energies and therefore make a big contribution to the partition function. The derivative of the log probability of a training vector with respect to a weight is surprisingly simple.

$$\frac{\partial \log p(\mathbf{v})}{\partial w_{ij}} = \langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}} \quad (24.5)$$

where the angle brackets are used to denote expectations under the distribution specified by the subscript that follows. This leads to a very simple learning rule for performing stochastic steepest ascent in the log probability of the training data:

$$\Delta w_{ij} = \epsilon(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}}) \quad (24.6)$$

where ϵ is a learning rate.

Because there are no direct connections between hidden units in an RBM, it is very easy to get an unbiased sample of $\langle v_i h_j \rangle_{\text{data}}$. Given a randomly selected training image, \mathbf{v} , the binary state, h_j , of each hidden unit, j , is set to 1 with probability

$$p(h_j = 1 \mid \mathbf{v}) = \sigma(b_j + \sum_i v_i w_{ij}) \quad (24.7)$$

where $\sigma(x)$ is the logistic sigmoid function $1/(1 + \exp(-x))$. $v_i h_j$ is then an unbiased sample.

Because there are no direct connections between visible units in an RBM, it is also very easy to get an unbiased sample of the state of a visible unit, *given a hidden vector*

$$p(v_i = 1 \mid \mathbf{h}) = \sigma(a_i + \sum_j h_j w_{ij}) \quad (24.8)$$

Getting an unbiased sample of $\langle v_i h_j \rangle_{\text{model}}$, however, is much more difficult. It can be done by starting at any random state of the visible units and performing alternating Gibbs sampling for a very long time. One iteration of alternating Gibbs sampling consists of updating all of the hidden units in parallel using equation 24.7 followed by updating all of the visible units in parallel using equation 24.8.

A much faster learning procedure was proposed in [5]. This starts by setting the states of the visible units to a training vector. Then the binary states of the hidden units are all computed in parallel using equation 24.7. Once binary states have been chosen for the hidden units, a “reconstruction” is produced by setting each v_i to 1 with a probability given by equation 24.8. The change in a weight is then given by

$$\Delta w_{ij} = \epsilon(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{recon}}) \quad (24.9)$$

A simplified version of the same learning rule that uses the states of individual units instead of pairwise products is used for the biases.

The learning works well even though it is only crudely approximating the gradient of the log probability of the training data [5]. The learning rule is much more closely approximating the gradient of another objective function called the Contrastive Divergence [5] which is the difference between two Kullback-Leibler divergences, but it ignores one tricky term in this objective function so it is not even following that gradient. Indeed, Sutskever and Tieleman have shown that it is not following the gradient of any function [19]. Nevertheless, it works well enough to achieve success in many significant applications.

RBM s typically learn better models if more steps of alternating Gibbs sampling are used before collecting the statistics for the second term in the learning

rule, which will be called the negative statistics. CD_n will be used to denote learning using n full steps of alternating Gibbs sampling.

24.3 How to Collect Statistics When Using Contrastive Divergence

To begin with, we shall assume that all of the visible and hidden units are binary. Other types of units will be discussed in section 24.13. We shall also assume that the purpose of the learning is to create a good generative model of the set of training vectors. When using RBMs to learn Deep Belief Nets (see the article on Deep Belief Networks at www.scholarpedia.org) that will subsequently be fine-tuned using backpropagation, the generative model is not the ultimate objective and it may be possible to save time by underfitting it, but we will ignore that here.

24.3.1 Updating the Hidden States

Assuming that the hidden units are binary and that you are using CD_1 , the hidden units should have stochastic binary states when they are being driven by a data-vector. The probability of turning on a hidden unit, j , is computed by applying the logistic function $\sigma(x) = 1/(1 + \exp(-x))$ to its “total input”:

$$p(h_j = 1) = \sigma(b_j + \sum_i v_i w_{ij}) \quad (24.10)$$

and the hidden unit turns on if this probability is greater than a random number uniformly distributed between 0 and 1.

It is very important to make these hidden states binary, rather than using the probabilities themselves. If the probabilities are used, each hidden unit can communicate a real-value to the visible units during the reconstruction. This seriously violates the information bottleneck created by the fact that a hidden unit can convey at most one bit (on average). This information bottleneck acts as a strong regularizer.

For the last update of the hidden units, it is silly to use stochastic binary states because nothing depends on which state is chosen. So use the probability itself to avoid unnecessary sampling noise. When using CD_n , only the final update of the hidden units should use the probability.

24.3.2 Updating the Visible States

Assuming that the visible units are binary, the correct way to update the visible states when generating a reconstruction is to stochastically pick a 1 or 0 with a probability determined by the total top-down input:

$$p_i = p(v_i = 1) = \sigma(a_i + \sum_j h_j w_{ij}) \quad (24.11)$$

However, it is common to use the probability, p_i , instead of sampling a binary value. This is not nearly as problematic as using probabilities for the data-driven hidden states and it reduces sampling noise thus allowing faster learning. There is some evidence that it leads to slightly worse density models (Tijmen Tieleman, personal communication, 2008). This probably does not matter when using an RBM to pretrain a layer of hidden features for use in a deep belief net.

24.3.3 Collecting the Statistics Needed for Learning

Assuming that the visible units are using real-valued probabilities instead of stochastic binary values, there are two sensible ways to collect the positive statistics for the connection between visible unit i and hidden unit j :

$$\langle p_i h_j \rangle_{\text{data}} \quad \text{or} \quad \langle p_i p_j \rangle_{\text{data}}$$

where p_j is a probability and h_j is a binary state that takes value 1 with probability p_j . Using h_j is closer to the mathematical model of an RBM, but using p_j usually has less sampling noise which allows slightly faster learning¹.

24.3.4 A Recipe for Getting the Learning Signal for CD₁

When the hidden units are being driven by data, *always* use stochastic binary states. When they are being driven by reconstructions, always use probabilities without sampling.

Assuming the visible units use the logistic function, use real-valued probabilities for both the data and the reconstructions².

When collecting the pairwise statistics for learning weights or the individual statistics for learning biases, use the probabilities, not the binary states, and make sure the weights have random initial values to break symmetry.

24.4 The Size of a Mini-batch

It is possible to update the weights after estimating the gradient on a single training case, but it is often more efficient to divide the training set into small “mini-batches” of 10 to 100 cases³. This allows matrix-matrix multiplies to be used which is very advantageous on GPU boards or in Matlab.

¹ Using h_j always creates more noise in the positive statistics than using p_j but it can actually create less noise in the *difference* of the positive and negative statistics because the negative statistics depend on the binary decision for the state of j that is used for creating the reconstruction. The probability of j when driven by the reconstruction is highly correlated with the binary decision that was made for j when it was driven by the data.

² So there is nothing random about the generation of the reconstructions given the binary states of the hidden units.

³ The word “batch” is confusing and will be avoided because when it is used to contrast with “on-line” it usually means the entire training set.

To avoid having to change the learning rate when the size of a mini-batch is changed, it is helpful to divide the total gradient computed on a mini-batch by the size of the mini-batch, so when talking about learning rates we will assume that they multiply the average, per-case gradient computed on a mini-batch, not the total gradient for the mini-batch.

It is a serious mistake to make the mini-batches too large when using stochastic gradient descent (see chapter 1 and 18 for more details). Increasing the mini-batch size by a factor of N leads to a more reliable gradient estimate but it does not increase the maximum stable learning rate by a factor of N , so the net effect is that the weight updates are smaller *per gradient evaluation*⁴.

24.4.1 A Recipe for Dividing the Training Set into Mini-batches

For datasets that contain a small number of equiprobable classes, the ideal mini-batch size is often equal to the number of classes and each mini-batch should contain one example of each class to reduce the sampling error when estimating the gradient for the whole training set from a single mini-batch. For other datasets, first randomize the order of the training examples then use minibatches of size about 10.

24.5 Monitoring the Progress of Learning

It is easy to compute the squared error between the data and the reconstructions, so this quantity is often printed out during learning. The reconstruction error on the entire training set should fall rapidly and consistently at the start of learning and then more slowly. Due to the noise in the gradient estimates, the reconstruction error on the individual mini-batches will fluctuate gently after the initial rapid descent. It may also oscillate gently with a period of a few mini-batches when using high momentum (see section 24.9).

Although it is convenient, the reconstruction error is actually a very poor measure of the progress of learning. It is not the function that CD_n learning is approximately optimizing, especially for $n \gg 1$, and it systematically confounds two different quantities that are changing during the learning. The first is the difference between the empirical distribution of the training data and the equilibrium distribution of the RBM. The second is the mixing rate of the alternating Gibbs Markov chain. If the mixing rate is very low, the reconstruction error will be very small even when the distributions of the data and the model are very different. As the weights increase the mixing rate falls, so decreases

⁴ The easy way to parallelize the learning on a cluster is to divide each mini-batch into sub-mini-batches and to use different cores to compute the gradients on each sub-mini-batch. The gradients computed by different cores must then be combined. To minimize the ratio of communication to computation, it is tempting to make the sub-mini-batches large. This usually makes the learning much less efficient, thus wiping out much of the gain achieved by using multiple cores (Vinod Nair, personal communication, 2007).

in reconstruction error do not necessarily mean that the model is improving and, conversely, small increases do not necessarily mean the model is getting worse. Large increases, however, are a bad sign except when they are temporary and caused by changes in the learning rate, momentum, weight-cost or sparsity meta-parameters.

24.5.1 A Recipe for Using the Reconstruction Error

Use it but don't trust it. If you really want to know what is going on during the learning, use multiple histograms and graphic displays as described in section 24.15. Also consider using Annealed Importance Sampling [16] to estimate the density on held out data. If you are learning a joint density model of labelled data (see section 24.16), consider monitoring the discriminative performance on the training data and on a held out validation set.

24.6 Monitoring the Overfitting

When learning a generative model, the obvious quantity to monitor is the probability that the current model assigns to a datapoint. When this probability starts to decrease for held out validation data, it is time to stop learning. Unfortunately, for large RBMs, it is very difficult to compute this probability because it requires knowledge of the partition function. Nevertheless, it is possible to directly monitor the overfitting by comparing the free energies of training data and held out validation data. In this comparison, the partition function cancels out. The free energy of a data vector can be computed in a time that is linear in the number of hidden units (see section 24.16.1). If the model is not overfitting at all, the average free energy should be about the same on training and validation data. As the model starts to overfit the average free energy of the validation data will rise relative to the average free energy of the training data and this gap represents the amount of overfitting⁵.

24.6.1 A Recipe for Monitoring the Overfitting

After every few epochs, compute the average free energy of a representative subset of the training data and compare it with the average free energy of a validation set. Always use the same subset of the training data. If the gap starts growing, the model is overfitting, though the probability of the training data may be growing even faster than the gap, so the probability of the validation data may still be improving. Make sure that the same weights are used when computing the two averages that you wish to compare.

⁵ The average free energies often change by large amounts during learning and this means very little because the log partition function also changes by large amounts. It is only *differences* in free energies that are easy to interpret without knowing the partition function.

24.7 The Learning Rate

If the learning rate is much too large, the reconstruction error usually increases dramatically and the weights may explode.

If the learning rate is reduced while the network is learning normally, the reconstruction error will usually fall significantly. This is not necessarily a good thing. It is due, in part, to the smaller noise level in the stochastic weight updates and it is generally accompanied by slower learning in the long term. Towards the end of learning, however, it typically pays to decrease the learning rate. Averaging the weights across several updates is an alternative way to remove some of the noise from the final weights.

24.7.1 A Recipe for Setting the Learning Rates for Weights and Biases

A good rule of thumb for setting the learning rate (Max Welling, personal communication, 2002) is to look at a histogram of the weight updates and a histogram of the weights. The updates should be about 10^{-3} times the weights (to within about an order of magnitude). When a unit has a very large fan-in, the updates should be smaller since many small changes in the same direction can easily reverse the sign of the gradient. Conversely, for biases, the updates can be bigger.

24.8 The Initial Values of the Weights and Biases

The weights are typically initialized to small random values chosen from a zero-mean Gaussian with a standard deviation of about 0.01. Using larger random values can speed the initial learning, but it may lead to a slightly worse final model. Care should be taken to ensure that the initial weight values do not allow typical visible vectors to drive the hidden unit probabilities very close to 1 or 0 as this significantly slows the learning. If the statistics used for learning are stochastic, the initial weights can all be zero since the noise in the statistics will make the hidden units become different from one another even if they all have identical connectivities.

It is usually helpful to initialize the bias of visible unit i to $\log[p_i/(1 - p_i)]$ where p_i is the proportion of training vectors in which unit i is on. If this is not done, the early stage of learning will use the hidden units to make i turn on with a probability of approximately p_i .

When using a sparsity target probability of t (see section 24.11), it makes sense to initialize the hidden biases to be $\log[t/(1 - t)]$. Otherwise, initial hidden biases of 0 are usually fine. It is also possible to start the hidden units with quite large negative biases of about -4 as a crude way of encouraging sparsity.

24.8.1 A Recipe for Setting the Initial Values of the Weights and Biases

Use small random values for the weights chosen from a zero-mean Gaussian with a standard deviation of 0.01. Set the hidden biases to 0. Set the visible biases to $\log[p_i/(1 - p_i)]$ where p_i is the proportion of training vectors in which unit i is on. Look at the activities of the hidden units occasionally to check that they are not always on or off.

24.9 Momentum

Momentum is a simple method for increasing the speed of learning when the objective function contains long, narrow and fairly straight ravines with a gentle but consistent gradient along the floor of the ravine and much steeper gradients up the sides of the ravine. The momentum method simulates a heavy ball rolling down a surface. The ball builds up velocity along the floor of the ravine, but not across the ravine because the opposing gradients on opposite sides of the ravine cancel each other out over time. Instead of using the estimated gradient times the learning rate to increment the *values* of the parameters, the momentum method uses this quantity to increment the *velocity*, \mathbf{v} , of the parameters and the current velocity is then used as the parameter increment.

The velocity of the ball is assumed to decay with time and the “momentum” meta-parameter, α is the fraction of the previous velocity that remains after computing the gradient on a new mini-batch:

$$\Delta\theta_i(t) = v_i(t) = \alpha v_i(t-1) - \epsilon \frac{dE}{d\theta_i}(t) \quad (24.12)$$

If the gradient remains constant, the terminal velocity will exceed $\epsilon dE/d\theta_i$ by a factor of $1/(1 - \alpha)$. This is a factor of 10 for a momentum of 0.9 which is a typical setting of this meta-parameter. The temporal smoothing in the momentum method avoids the divergent oscillations across the ravine that would be caused by simply increasing the learning rate by a factor of $1/(1 - \alpha)$.

The momentum method causes the parameters to move in a direction that is not the direction of steepest descent, so it bears some resemblance to methods like conjugate gradient, but the way it uses the previous gradients is much simpler. Unlike methods that use different learning rates for each parameter, momentum works just as well when the ravines are not aligned with the parameter axes.

An alternative way of viewing the momentum method (Tijmen Tieleman, personal communication, 2008) is as follows: It is equivalent to increasing the learning rate by a factor of $1/(1 - \alpha)$ but delaying the full effect of each gradient estimate by dividing the full increment into a series of exponentially decaying installments. This gives the system time to respond to the early installments by moving to a region of parameter space that has opposing gradients before it feels the full effect of the increment. This, in turn, allows the learning rate to be larger without causing unstable oscillations.

At the start of learning, the random initial parameter values may create very large gradients and the system is unlikely to be in the floor of a ravine, so it is usually best to start with a low momentum of 0.5 for a number of parameter updates. This very conservative momentum typically makes the learning more stable than no momentum at all by damping oscillations across ravines [4].

24.9.1 A Recipe for Using Momentum

Start with a momentum of 0.5. Once the large initial progress in the reduction of the reconstruction error has settled down to gentle progress, increase the momentum to 0.9. This shock may cause a transient increase in the reconstruction error. If this causes a more lasting instability, keep reducing the learning rate by factors of 2 until the instability disappears.

24.10 Weight-Decay

Weight-decay works by adding an extra term to the normal gradient. The extra term is the derivative of a function that penalizes large weights. The simplest penalty function, called “L2”, is half of the sum of the squared weights times a coefficient which will be called the weight-cost.

It is important to multiply the derivative of the penalty term by the learning rate. Otherwise, changes in the learning rate change the function that is being optimized rather than just changing the optimization procedure.

There are four different reasons for using weight-decay in an RBM. The first is to improve generalization to new data by reducing overfitting to the training data⁶. The second is to make the receptive fields of the hidden units smoother and more interpretable by shrinking useless weights. The third is to “unstick” hidden units that have developed very large weights early in the training and are either always firmly on or always firmly off. A better way to allow such units to become useful again is to use a “sparsity” target as described in section 24.11.

The fourth reason is to improve the mixing rate of the alternating Gibbs Markov chain. With small weights, the Markov chain mixes more rapidly⁷. The CD learning procedure is based on ignoring derivatives that come from later steps in the Markov chain (Hinton, Osindero and Teh, 2006), so it tends to approximate maximum likelihood learning better when the mixing is fast. The ignored derivatives are then small for the following reason: When a Markov chain is very close to its stationary distribution, the best parameters for modeling samples from the chain are very close to its current parameters.

⁶ Since the penalty is applied on every mini-batch, Bayesians really ought to divide the weight-cost by the size of the training set. They can then interpret weight-decay as the effect of a Gaussian weight prior whose variance is independent of the size of the training set. This division is typically not done. Instead, larger weight-costs are used for smaller training sets.

⁷ With all zero weights, it reaches its rather boring stationary distribution in a single full step.

A different form of weight-decay called “L1” is to use the derivative of the sum of the absolute values of the weights. This often causes many of the weights to become exactly zero whilst allowing a few of the weights to grow quite large. This can make it easier to interpret the weights. When learning features for images, for example, L1 weight-decay often leads to strongly localized receptive fields.

An alternative way to control the size of the weights is to impose a maximum allowed value on the sum of the squares or absolute values of the incoming weights for each unit. After each weight update, the weights are rescaled if they exceed this maximum value. This helps to avoid hidden units getting stuck with extremely small weights, but a sparsity target is probably a better way to avoid this problem.

24.10.1 A Recipe for Using Weight-Decay

For an RBM, sensible values for the weight-cost coefficient for L2 weight-decay typically range from 0.01 to 0.00001. Weight-cost is typically not applied to the hidden and visible biases because there are far fewer of these so they are less likely to cause overfitting. Also, the biases sometimes need to be quite large.

Try an initial weight-cost of 0.0001. If you are using Annealed Importance Sampling [16] to estimate the density on a held-out validation set, try adjusting the weight-cost by factors of 2 to optimize density. Small differences in weight-cost are unlikely to cause big differences in performance. If you are training a joint density model that allows you to test discriminative performance on a validation set this can be used in place of the density for optimizing the weight-cost. However, in either case, remember that weight-decay does more than just preventing overfitting. It also increases the mixing rate which makes CD learning a better approximation to maximum likelihood. So even if overfitting is not a problem because the supply of training data is infinite, weight-decay can still be helpful.

24.11 Encouraging Sparse Hidden Activities

Hidden units that are only rarely active are typically easier to interpret than those that are active about half of the time. Also, discriminative performance is sometimes improved by using features that are only rarely active [13].

Sparse activities of the binary hidden units can be achieved by specifying a “sparsity target” which is the desired probability of being active, $p \ll 1$. An additional penalty term is then used to encourage the actual probability of being active, q , to be close to p . q is estimated by using an exponentially decaying average of the mean probability that a unit is active in each mini-batch:

$$q_{new} = \lambda q_{old} + (1 - \lambda)q_{current} \quad (24.13)$$

where $q_{current}$ is the mean activation probability of the hidden unit on the current mini-batch.

The natural penalty measure to use is the cross entropy between the desired and actual distributions:

$$\text{Sparsity penalty} \propto -p \log q - (1-p) \log(1-q) \quad (24.14)$$

For logistic units this has a simple derivative of $q - p$ with respect to the total input to a unit. This derivative, scaled by a meta-parameter called “sparsity-cost”, is used to adjust both the bias and the incoming weights of each hidden unit. It is important to apply the same derivative to both. If the derivative is only applied to the bias, for example, the bias will typically keep becoming more negative to ensure the hidden unit is rarely on, but the weights will keep becoming more positive to make the unit more useful.

24.11.1 A Recipe for Sparsity

Set the sparsity target to between 0.01 and 0.1⁸. Set the decay-rate, λ , of the estimated value of q to be between 0.9 and 0.99. Histogram the mean activities of the hidden units and set the sparsity-cost so that the hidden units have mean probabilities in the vicinity of the target. If the probabilities are tightly clustered around the target value, reduce the sparsity-cost so that it interferes less with the main objective of the learning.

24.12 The Number of Hidden Units

Intuitions derived from discriminative machine learning are a bad guide for determining a sensible number of hidden units. In discriminative learning, the amount of constraint that a training case imposes on the parameters is equal to the number of bits that it takes to specify the label. Labels usually contain very few bits of information, so using more parameters than training cases will typically cause severe overfitting. When learning generative models of high-dimensional data, however, it is the number of bits that it takes to specify a data vector that determines how much constraint each training case imposes on the parameters of the model. This can be several orders of magnitude greater than number of bits required to specify a label. So it may be quite reasonable to fit a million parameters to 10,000 training images if each image contains 1,000 pixels. This would allow 1000 globally connected hidden units. If the hidden units are locally connected or if they use weight-sharing, many more can be used.

24.12.1 A Recipe for Choosing the Number of Hidden Units

Assuming that the main issue is overfitting rather than the amount of computation at training or test time, estimate how many bits it would take to describe

⁸ If you are only using the sparsity target to revive hidden units that are never active and suppress hidden units that are always active, a target value of 0.5 makes sense (even though it makes nonsense of the name).

each data-vector if you were using a good model (*i.e.* estimate the typical negative \log_2 probability of a datavector under a good model). Then multiply that estimate by the number of training cases and use a number of parameters that is about an order of magnitude smaller. If you are using a sparsity target that is very small, you may be able to use more hidden units. If the training cases are highly redundant, as they typically will be for very big training sets, you need to use fewer parameters.

24.13 Different Types of Unit

RBM's were developed using binary visible and hidden units, but many other types of unit can also be used. A general treatment for units in the exponential family is given in [24]. The main use of other types of unit is for dealing with data that is not well-modeled by binary (or logistic) visible units.

24.13.1 Softmax and Multinomial Units

For a binary unit, the probability of turning on is given by the logistic sigmoid function of its total input, x .

$$p = \sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + e^0} \quad (24.15)$$

The energy contributed by the unit is $-x$ if it is on and 0 if it is off. Equation 24.15 makes it clear that the probability of each of the two possible states is proportional to the negative exponential of its energy. This can be generalized to K alternative states.

$$p_j = \frac{e^{x_j}}{\sum_{i=1}^K e^{x_i}} \quad (24.16)$$

This is often called a “softmax” unit. It is the appropriate way to deal with a quantity that has K alternative values which are not ordered in any way. A softmax can be viewed as a set of binary units whose states are mutually constrained so that exactly one of the K states has value 1 and the rest have value 0. When viewed in this way, the learning rule for the binary units in a softmax is identical to the rule for standard binary units. The only difference is in the way the probabilities of the states are computed and the samples are taken.

A further generalization of the softmax unit is to sample N times (with replacement) from the probability distribution instead of just sampling once. The K different states can then have integer values bigger than 1, but the values must add to N . This is called a multinomial unit and, again, the learning rule is unchanged.

24.13.2 Gaussian Visible Units

For data such as patches of natural images or the Mel-Cepstrum coefficients used to represent speech, logistic units are a very poor representation. One solution is to replace the binary visible units by linear units with independent Gaussian noise. The energy function then becomes:

$$E(\mathbf{v}, \mathbf{h}) = \sum_{i \in \text{vis}} \frac{(v_i - a_i)^2}{2\sigma_i^2} - \sum_{j \in \text{hid}} b_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} h_j w_{ij} \quad (24.17)$$

where σ_i is the standard deviation of the Gaussian noise for visible unit i .

It is possible to learn the variance of the noise for each visible unit but this is difficult using CD₁. In many applications, it is much easier to first normalise each component of the data to have zero mean and unit variance and then to use noise free reconstructions, with the variance in equation 24.17 set to 1. The reconstructed value of a Gaussian visible unit is then equal to its top-down input from the binary hidden units plus its bias.

The learning rate needs to be about one or two orders of magnitude smaller than when using binary visible units and some of the failures reported in the literature are probably due to using a learning rate that is much too big. A smaller learning rate is required because there is no upper bound to the size of a component in the reconstruction and if one component becomes very large, the weights emanating from it will get a very big learning signal. With binary hidden and visible units, the learning signal for each training case must lie between -1 and 1 , so binary-binary nets are much more stable.

24.13.3 Gaussian Visible and Hidden Units

If both the visible and the hidden units are Gaussian, the instability problems become much worse. The individual activities are held close to their means by quadratic “containment” terms with coefficients determined by the standard deviations of the assumed noise levels:

$$E(\mathbf{v}, \mathbf{h}) = \sum_{i \in \text{vis}} \frac{(v_i - a_i)^2}{2\sigma_i^2} + \sum_{j \in \text{hid}} \frac{(h_j - b_j)^2}{2\sigma_j^2} - \sum_{i,j} \frac{v_i}{\sigma_i} \frac{h_j}{\sigma_j} w_{ij} \quad (24.18)$$

If any of the eigenvalues of the weight matrix become sufficiently large, the quadratic interaction terms can dominate the containment terms and there is then no lower bound to the energy that can be achieved by scaling up the activities in the direction of the corresponding eigenvector. With a sufficiently small learning rate, CD₁ can detect and correct these directions so it is possible to learn an undirected version of a factor analysis model [10] using all Gaussian units, but this is harder than using EM [3] to learn a directed model.

24.13.4 Binomial Units

A simple way to get a unit with noisy integer values in the range 0 to N is to make N separate copies of a binary unit and give them all the same weights and bias [21]. Since all copies receive the same total input, they all have the same probability, p , of turning on and this only has to be computed once. The expected number that are on is Np and the variance in this number is $Np(1 - p)$. For small p , this acts like a Poisson unit, but as p approaches 1 the variance becomes small again which may not be desirable. Also, for small values of p the growth in p is exponential in the total input. This makes learning much less stable than for the rectified linear units described in section 24.13.5.

One nice thing about using weight-sharing to synthesize a new type of unit out of binary units is that the mathematics underlying binary-binary RBM's remains unchanged.

24.13.5 Rectified Linear Units

A small modification to binomial units makes them far more interesting as models of real neurons and also more useful for practical applications. All copies still have the same learned weight vector \mathbf{w} and the same learned bias, b , but each copy has a different, fixed offset to the bias. If the offsets are $-0.5, -1.5, -2.5, \dots, -(N - 0.5)$ the sum of the probabilities of the copies is extremely close to having a closed form:

$$\sum_{i=1}^{\infty} \sigma(x - i + 0.5) \approx \log(1 + e^x) \quad (24.19)$$

where $x = \mathbf{v}\mathbf{w}^T + b$. So the total activity of all of the copies behaves like a smoothed version of a rectified linear unit that saturates for sufficiently large input. Even though $\log(1 + e^x)$ is not in the exponential family, we can model it accurately using a set of binary units with shared weights and fixed bias offsets. This set has no more parameters than an ordinary binary unit, but it provides a much more expressive variable. The variance is $\sigma(x)$ so units that are firmly off do not create noise and the noise does not become large when x is large.

A drawback of giving each copy a bias that differs by a fixed offset is that the logistic function needs to be used many times to get the probabilities required for sampling an integer value correctly. It is possible, however, to use a fast approximation in which the sampled value of the rectified linear unit is not constrained to be an integer. Instead it is approximated by $\max(0, x + N(0, 1))$ where $N(0, 1)$ is Gaussian noise with zero mean and unit variance. This type of rectified linear unit seems to work fine for either visible units or hidden units when training with CD1 [14].

If both visible and hidden units are rectified linear, a much smaller learning rate may be needed to avoid unstable dynamics in the activity or weight updates. If the weight between two rectified linear units is greater than 1 there is no lower bound to the energy that can be achieved by giving both units very high

activities so there is no proper probability distribution. Nevertheless, contrastive divergence learning may still work provided the learning rate is low enough to give the learning time to detect and correct directions in which the Markov chain would blow up if allowed to run for many iterations. RBM's composed of rectified linear units are more stable than RBM's composed of Gaussian units because the rectification prevents biphasic oscillations of the weight dynamics in which units alternate between very high positive activity for one mini-batch followed by very high negative activity for the next mini-batch.

24.14 Varieties of Contrastive Divergence

Although CD_1 is not a very good approximation to maximum likelihood learning, this does not seem to matter when an RBM is being learned in order to provide hidden features for training a higher-level RBM. CD_1 ensures that the hidden features retain most of the information in the data vector and it is not necessarily a good idea to use a form of CD that is a closer approximation to maximum likelihood but is worse at retaining the information in the data vector. If, however, the aim is to learn an RBM that is a good density or joint-density model, CD_1 is far from optimal.

At the beginning of learning, the weights are small and mixing is fast so CD_1 provides a good approximation to maximum likelihood. As the weights grow, the mixing gets worse and it makes sense to gradually increase the n in CD_n [1, 17]. When n is increased, the difference of pairwise statistics that is used for learning will increase so it may be necessary to reduce the learning rate.

A more radical departure from CD_1 is called “persistent contrastive divergence” [22]. Instead of initializing each alternating Gibbs Markov chain at a datavector, which is the essence of CD learning, we keep track of the states of a number of persistent chains or “fantasy particles”. Each persistent chain has its hidden and visible states updated one (or a few) times after each weight update. The learning signal is then the difference between the pairwise statistics measured on a mini-batch of data and the pairwise statistics measured on the persistent chains. Typically the number of persistent chains is the same as the size of a mini-batch, but there is no good reason for this. The persistent chains mix surprisingly fast because the weight-updates repel each chain from its current state by raising the energy of that state [23].

When using persistent CD, the learning rate typically needs to be quite a lot smaller and the early phase of the learning is much slower in reducing the reconstruction error. In the early phase of learning the persistent chains often have very correlated states, but this goes away with time. The final reconstruction error is also typically larger than with CD_1 because persistent CD is, asymptotically, performing maximum likelihood learning rather than trying to make the distribution of the one-step reconstructions resemble the distribution of the data. Persistent CD learns significantly better models than CD_1 or even CD_{10} [22] and is the recommended method if the aim is to build the best density model of the data.

Persistent CD can be improved by adding to the standard parameters an overlay of “fast weights” which learn very rapidly but also decay very rapidly [23]. These fast weights improve the mixing of the persistent chains. However, the use of fast weights introduces yet more meta-parameters and will not be discussed further here.

24.15 Displaying What Is Happening during Learning

There are many ways in which learning can go wrong and most of the common problems are easy to diagnose with the right graphical displays. The three types of display described below give much more insight into what is happening than simply monitoring the reconstruction error.

Histograms of the weights, the visible biases and the hidden biases are very useful. In addition, it is useful to examine histograms of the increments to these parameters when they are updated, though it is wasteful to make these histograms after every update.

For domains in which the visible units have spatial or temporal structure (*e.g.* images or speech) it is very helpful to display, for each hidden unit, the weights connecting that hidden unit to the visible units. These “receptive” fields are a good way of visualizing what features the hidden units have learned. When displaying the receptive fields of many hidden units it can be very misleading to use different scales for different hidden units. Gray-scale displays of receptive fields are usually less pretty but much more informative than false colour displays.

For a single minibatch, it is very useful to see a two-dimensional, gray-scale display with a range of [0,1] that shows the probability of each binary hidden unit on each training case in a mini-batch⁹. This immediately allows you to see if some hidden units are never used or if some training cases activate an unusually large or small number of hidden units. It also shows how certain the hidden units are. When learning is working properly, this display should look thoroughly random without any obvious vertical or horizontal lines. Histograms can be used instead of this display, but it takes quite a few histograms to convey the same information.

24.16 Using RBM’s for Discrimination

There are three obvious ways of using RBMs for discrimination. The first is to use the hidden features learned by the RBM as the inputs for some standard discriminative method. This will not be discussed further here, though it is probably the most important way of using RBM’s, especially when many layers of hidden features are learned unsupervised before starting on the discriminative training.

The second method is to train a separate RBM on each class. After training, the free energy of a test vector, \mathbf{t} , is computed (see subsection 24.16.1) for each

⁹ If there are more than a few hundred hidden units, just use a subset of them.

class-specific RBM. The log probability that the RBM trained on class c assigns to the test vector is given by:

$$\log p(\mathbf{t}|c) = -F_c(\mathbf{t}) - \log Z_c \quad (24.20)$$

where Z_c is the partition function of that RBM. Since each class-specific RBM will have a different, unknown partition function, the free energies cannot be used directly for discrimination. However, if the number of classes is small it is easy to deal with the unknown log partition functions by simply training a “softmax” model (on a separate training set) to predict the class from the free energies of all of the class specific RBMs:

$$\log p(\text{class} = c|\mathbf{t}) = \frac{e^{-F_c(\mathbf{t}) - \log \hat{Z}_c}}{\sum_d e^{-F_d(\mathbf{t}) - \log \hat{Z}_d}} \quad (24.21)$$

where the \hat{Z} are parameters that are learned by maximum likelihood training of the softmax. Of course, equation 24.21 can also be used to learn the weights and biases of each RBM but this requires a lot of data to avoid overfitting. Combining the discriminative gradients for the weights and biases that come from equation 24.21 with the approximate gradients that come from contrastive divergence will often do better than either method alone. The approximate gradient produced by contrastive divergence acts as a strong regularizer to prevent overfitting and the discriminative gradient ensures that there is some pressure to use the weights and biases in a way that helps discrimination.

The third method is to train a joint density model using a single RBM that has two sets of visible units. In addition to the units that represent a data vector, there is a “softmax” label unit that represents the class. After training, each possible label is tried in turn with a test vector and the one that gives lowest free energy is chosen as the most likely class. The partition function is not a problem here, since it is the same for all classes. Again, it is possible to combine discriminative and generative training of the joint RBM by using discriminative gradients that are the derivatives of the log probability of the correct class [6]:

$$\log p(\text{class} = c|\mathbf{t}) = \frac{e^{-F_c(\mathbf{t})}}{\sum_d e^{-F_d(\mathbf{t})}} \quad (24.22)$$

24.16.1 Computing the Free Energy of a Visible Vector

The free energy of visible vector \mathbf{v} is the energy that a single configuration would need to have in order to have the same probability as all of the configurations that contain \mathbf{v} :

$$e^{-F(\mathbf{v})} = \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \quad (24.23)$$

It is also given by the expected energy minus the entropy:

$$F(\mathbf{v}) = - \sum_i v_i a_i - \sum_j p_j x_j + \sum_j (p_j \log p_j + (1 - p_j) \log(1 - p_j)) \quad (24.24)$$

where $x_j = b_j + \sum_i v_i w_{ij}$ is the total input to hidden unit j and $p_j = \sigma(x_j)$ is the probability that $h_j = 1$ given \mathbf{v} . A good way to compute $F(\mathbf{v})$ is to use yet another expression for the free energy:

$$F(\mathbf{v}) = - \sum_i v_i a_i - \sum_j \log(1 + e^{x_j}) \quad (24.25)$$

24.17 Dealing with Missing Values

In a directed belief net it is very easy to deal with missing values for visible variables. When performing inference, a missing value that is at the receiving end of a directed connection has no effect on the units that send connections to it. This is not true for the undirected connections used in an RBM. To perform inference in the standard way, the missing value must first be filled in and there are at least two ways to do this.

A particularly simple type of missing value occurs when learning a joint density for data in which each training case is composed of a vector \mathbf{v} such as an image plus a single discrete label. If the label is missing from a subset of the cases, it can be Gibbs sampled from its exact conditional distribution. This is done by computing the free energy (see section 24.16.1) for each possible value of the label and then picking label l with probability proportional to $\exp(-F(l, \mathbf{v}))$. After this, the training case is treated just like a complete training case.

For real-valued visible units, there is a different way to impute missing values that still works well even if several values are missing from the same training case [8]. If the learning cycles through the training set many times, the missing values can be treated in just the same way as the other parameters. Starting with a sensible initial guess, a missing value is updated each time the weights are updated, but possibly using a different learning rate. The update for the missing value for visible unit i on training case c is:

$$\Delta v_i^c = \epsilon \left(\frac{\partial F}{\partial \hat{v}_i^c} - \frac{\partial F}{\partial v_i^c} \right) \quad (24.26)$$

where v_i^c is the imputed value and \hat{v}_i^c is the reconstruction of the imputed value. Momentum can be used for imputed values in just the same way as it is used for the usual parameters.

There is a more radical way of dealing with missing values that can be used when the number of missing values is very large. This occurs, for example, with user preference data where most users do not express their preference for most objects [17]. Instead of trying to impute the missing values, we pretend they do not exist by using RBMs with different numbers of visible units for different training cases. The different RBMs form a family of different models with shared

weights. Each RBM in the family can now do correct inference for its hidden states, but the tying of the weights means that they may not be ideal for any particular RBM. Adding a visible unit for a missing value and then performing correct inference that integrates out this missing value does not give the same distribution for the hidden units as simply omitting the visible unit which is why this is a family of models rather than just one model. When using a family of models to deal with missing values, it can be very helpful to scale the hidden biases by the number of visible units in the RBM [15].

Acknowledgements. This research was supported by NSERC and the Canadian Institute for Advanced Research. Many of my past and present graduate students and postdocs have made valuable contributions to the body of practical knowledge described in this chapter. I have tried to acknowledge particularly valuable contributions in the chapter, but I cannot always recall who suggested what.

References

- [1] Carreira-Perpignan, M.A., Hinton, G.E.: On contrastive divergence learning. In: Artificial Intelligence and Statistics (2005)
- [2] Freund, Y., Haussler, D.: Unsupervised learning of distributions on binary vectors using two layer networks. In: Advances in Neural Information Processing Systems 4, pp. 912–919. Morgan Kaufmann, San Mateo (1992)
- [3] Ghahramani, Z., Hinton, G.: The EM algorithm for mixtures of factor analyzers. Technical Report CRG-TR-96-1, University of Toronto (May 1996)
- [4] Hinton, G.E.: Relaxation and its role in vision. PhD Thesis (1978)
- [5] Hinton, G.E.: Training products of experts by minimizing contrastive divergence. Neural Computation 14(8), 1711–1800 (2002)
- [6] Hinton, G.E.: To recognize shapes, first learn to generate images. In: Computational Neuroscience: Theoretical Insights into Brain Function (2007)
- [7] Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Computation 18(7), 1527–1554 (2006)
- [8] Hinton, G.E., Osindero, S., Welling, M., Teh, Y.: Unsupervised discovery of non-linear structure using contrastive backpropagation. Cognitive Science 30, 725–731 (2006b)
- [9] Hopfield, J.J.: Neural networks and physical systems with emergent collective computational abilities. Proceedings of the National Academy of Sciences 79, 2554–2558 (1982)
- [10] Marks, T.K., Movellan, J.R.: Diffusion networks, product of experts, and factor analysis. In: Proc. Int. Conf. on Independent Component Analysis, pp. 481–485 (2001)
- [11] Mohamed, A.R., Hinton, G.E.: Phone recognition using restricted boltzmann machines. In: ICASSP 2010 (2010)
- [12] Mohamed, A.R., Dahl, G., Hinton, G.E.: Deep belief networks for phone recognition. In: NIPS 22 Workshop on Deep Learning for Speech Recognition (2009)
- [13] Nair, V., Hinton, G.E.: 3-d object recognition with deep belief nets. In: Advances in Neural Information Processing Systems, vol. 22, pp. 1339–1347 (2009)
- [14] Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: Proc. 27th International Conference on Machine Learning (2010)

- [15] Salakhutdinov, R.R., Hinton, G.E.: Replicated softmax: An undirected topic model. In: Advances in Neural Information Processing Systems, vol. 22 (2009)
- [16] Salakhutdinov, R.R., Murray, I.: On the quantitative analysis of deep belief networks. In: Proceedings of the International Conference on Machine Learning, vol. 25, pp. 872–879 (2008)
- [17] Salakhutdinov, R.R., Mnih, A., Hinton, G.E.: Restricted Boltzmann machines for collaborative filtering. In: Ghahramani, Z. (ed.) Proceedings of the International Conference on Machine Learning, vol. 24, pp. 791–798. ACM (2007)
- [18] Smolensky, P.: Information processing in dynamical systems: Foundations of harmony theory. In: Rumelhart, D.E., McClelland, J.L. (eds.) Parallel Distributed Processing, vol. 1, ch. 6, pp. 194–281. MIT Press, Cambridge (1986)
- [19] Sutskever, I., Tieleman: On the convergence properties of contrastive divergence. In: Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS), Sardinia, Italy (2010)
- [20] Taylor, G., Hinton, G.E., Roweis, S.T.: Modeling human motion using binary latent variables. In: Advances in Neural Information Processing Systems. MIT Press (2006)
- [21] Teh, Y.W., Hinton, G.E.: Rate-coded restricted Boltzmann machines for face recognition. In: Advances in Neural Information Processing Systems, vol. 13, pp. 908–914 (2001)
- [22] Tieleman, T.: Training restricted Boltzmann machines using approximations to the likelihood gradient. In: Proceedings of the Twenty-first International Conference on Machine Learning (ICML 2008). ACM (2008)
- [23] Tieleman, T., Hinton, G.E.: Using fast weights to improve persistent contrastive divergence. In: Proceedings of the 26th International Conference on Machine Learning, pp. 1033–1040. ACM, New York (2009)
- [24] Welling, M., Rosen-Zvi, M., Hinton, G.E.: Exponential family harmoniums with an application to information retrieval. In: Advances in Neural Information Processing Systems, pp. 1481–1488. MIT Press, Cambridge (2005)

25

Deep Boltzmann Machines and the Centering Trick

Grégoire Montavon¹ and Klaus-Robert Müller^{1,2}

¹ Technische Universität Berlin, 10587 Berlin, Germany,
Machine Learning Group

² Korea University, Anam-dong, Seongbuk-gu, Seoul 136-713, Korea,
Department of Brain and Cognitive Engineering
`{gregoire.montavon,klaus-robert.mueller}@tu-berlin.de`

Abstract. Deep Boltzmann machines are in theory capable of learning efficient representations of seemingly complex data. Designing an algorithm that effectively learns the data representation can be subject to multiple difficulties. In this chapter, we present the “centering trick” that consists of rewriting the energy of the system as a function of centered states. The centering trick improves the conditioning of the underlying optimization problem and makes learning more stable, leading to models with better generative and discriminative properties.

Keywords: Deep Boltzmann machine, centering, reparameterization, unsupervised learning, optimization, representations.

25.1 Introduction

Deep Boltzmann machines are undirected networks of interconnected units that learn a joint probability density over these units by adapting connections between them. They are in theory capable of learning statistically and computationally efficient representations of seemingly complex data distributions.

Designing an algorithm that effectively learns the data representation can be subject to multiple difficulties. Deep Boltzmann machines are sensitive to the parameterization of their energy function. In addition, the gradient of the optimization problem is not directly accessible and must instead be approximated stochastically by continuously querying the model throughout training.

In this chapter, we present the “centering trick” that consists of rewriting the energy function of the deep Boltzmann machine as a function of centered states. We argue that centering improves the conditioning of the optimization problem and facilitates the emergence of complex structures in the deep Boltzmann machine.

We demonstrate on the MNIST dataset that the centering trick allows mid-sized deep Boltzmann machines to be trained faster and to produce a solution which is a good generative model of data but also distills interesting discriminative features in the top layer.

25.2 Boltzmann Machines

In this section, we give some background on the Boltzmann machine [6]. We will use the following notation: The sigmoid function is defined as $\text{sigm}(x) = \frac{e^x}{e^x + 1}$, $x \sim \mathcal{B}(p)$ denotes that the variable x is drawn randomly from a Bernoulli distribution of parameter p and $\langle \cdot \rangle_P$ denotes the expectation operator with respect to a probability distribution P . All these operations apply element-wise to the input if the latter is a vector.

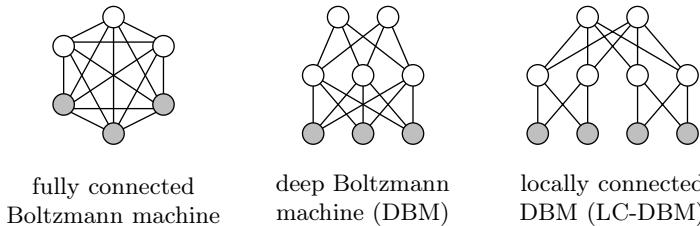


Fig. 25.1. Example of Boltzmann machines used in practice with visible units depicted in gray and hidden units depicted in white. The layered structure of a DBM is interesting because a particular representation of data forms at each layer, possibly enabling the emergence of interesting statistics.

A Boltzmann machine is a network of M_x interconnected binary units that associates to each state $x \in \{0, 1\}^{M_x}$ the probability

$$p(x; \theta) = \frac{e^{-E(x; \theta)}}{\sum_{\xi} e^{-E(\xi; \theta)}}.$$

The term in the denominator is the called the partition function and makes probabilities sum to one. The function

$$E(x; \theta) = -\frac{1}{2}x^\top Wx - x^\top b.$$

is the energy of the state x given the model parameters $\theta = (W, b)$. From these equations, we can interpret a good model of data as a model θ that has low energy in regions of high data density and high energy elsewhere. The matrix W of size $M_x \times M_x$ is symmetric and contains the connection strengths between units. The vector b of size M_x contains the biases associated to each unit. The diagonal of W is constrained to be zero. Units are either visible units (representing the sensory input) or hidden units (representing latent variables that are not directly observable but contribute to explaining data).

From the equations above, we can derive the conditional probability of each unit being activated given the other units

$$p(x_i = 1 | x_{-i}; \theta) = \text{sigm}(b_i + \sum_{j \neq i} W_{ij} x_j)$$

where x_{-i} denotes the set of all units but x_i . The gradient of the data log-likelihood with respect to model parameters W and b takes the form

$$\frac{\partial}{\partial W} \langle \log p(x_{\text{vis}}; \theta) \rangle_{\text{data}} = \langle xx^\top \rangle_{\text{data}} - \langle xx^\top \rangle_{\text{model}} \quad (25.1)$$

$$\frac{\partial}{\partial b} \langle \log p(x_{\text{vis}}; \theta) \rangle_{\text{data}} = \langle x \rangle_{\text{data}} - \langle x \rangle_{\text{model}} \quad (25.2)$$

where x_{vis} are the visible units (i.e. the subset of units that represent the sensory input). The terms $\langle \cdot \rangle_{\text{data}}$ and $\langle \cdot \rangle_{\text{model}}$ are respectively the data-dependent expectations (obtained by conditioning the joint distribution on the observed state of the visible units) and the data-independent expectations obtained by sampling freely from the joint probability distribution.

25.2.1 Deep Boltzmann Machines

It is often desirable to incorporate some predefined structure to the Boltzmann machine. The most common way to achieve this is to remove certain connections in the network, that is, forcing parts of the matrix W to zero. Examples of Boltzmann machines with different structures are shown in Figure 25.1. For example, in the deep Boltzmann machine (DBM) [12], units are organized in a deep layered manner where only adjacent layers communicate and where units within the same layer are disconnected. Locally connected deep Boltzmann machines add further constraints to the model by forcing the modeling of the interaction between remote parts of the input to take place in the top layer.

The special layered structure of the DBM and its multiple variants has two advantages: First, particular statistics can emerge at each layer that may capture interesting features of data. Second, the layered structure of the DBM can be folded into a bipartite graph (one side containing odd layers and the other side containing even layers) where each side of the graph is conditionally independent given the other side.

In the case of the deep Boltzmann machine shown in Figure 25.2 (left) with M_x , M_y and M_z units at each layer, the energy associated to each state $(x, y, z) \in \{0, 1\}^{M_x + M_y + M_z}$ is

$$E(x, y, z; \theta) = -y^\top Wx - z^\top Vy - x^\top a - y^\top b - z^\top c$$

where $\theta = \{W, V, a, b, c\}$ groups the model parameters. The bipartite graph structure of the deep Boltzmann machine implies that an efficient alternating Gibbs sampler can be derived:

$$\begin{aligned} x &\sim \mathcal{B}(\text{sigm}(W^\top y + a)) \\ z &\sim \mathcal{B}(\text{sigm}(V^\top y + c)) \\ y &\sim \mathcal{B}(\text{sigm}(Wx + V^\top z + b)). \end{aligned} \quad (25.3)$$

A similar alternating Gibbs sampler can be used for sampling states when the input units x are clamped to the data:

$$\begin{aligned} z &\sim \mathcal{B}(\text{sigm}(Vy + c)) \\ y &\sim \mathcal{B}(\text{sigm}(Wx_{\text{data}} + V^\top z + b)). \end{aligned} \quad (25.4)$$

These alternating Gibbs samplers are illustrated in Figure 25.2 (right) and allow us to collect the data-independent and data-dependent statistics that intervene in the computation of the gradient (see Equation 25.1 and 25.2).

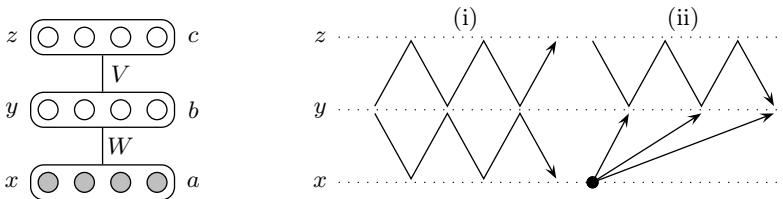


Fig. 25.2. On the left, diagram of a two-layer deep Boltzmann machine along with its parameters. On the right, different sampling methods: (i) the path followed by the alternating Gibbs sampler and (ii) the path followed by the alternating Gibbs sampler when the input is clamped to data.

25.3 Training Boltzmann Machines

While Equation 25.1 and 25.2 provide an exact gradient for minimizing the log-likelihood of data, keeping track of data statistics and model statistics is computationally demanding. The mixing rate of the model (i.e. the speed at which the alternating Gibbs sampler converges to the model's true distribution) is typically slow and implies that we need to resort to some approximation.

Collecting *data-dependent statistics* is relatively easy as the complexity of the distribution is reduced by the clamping of visible units to the data. In the case where hidden units are independent when conditioned on the visible units, sampling can be achieved exactly in only one pass of the Gibbs sampler. This is the case of the restricted Boltzmann machine [4] presented in Chapter 24 [5]. In practice, when the number of hidden-to-hidden connections is relatively low or the connections are not particularly strong, reasonable approximations can be obtained by running a few steps of the alternating Gibbs sampler.

Collecting *data-independent statistics* is much harder and typically requires hundreds or thousands of iterations before converging to the true probability distribution. A workaround to the problem is to approximate these statistics by a small set of persistent chains (or “free particles”) $\{x_1, \dots, x_n\}$ that are continuously updated throughout training. This idea called *persistent contrastive divergence* has been proposed by Tieleman [19].

The intuition behind persistent contrastive divergence is the following: let's first remember that the minima of the energy function correspond to high probability states and that the free particles are therefore inclined to descend the energy function. As the model is trained, the energy of the free particles is raised under the effect of the gradient update and free particles are encouraged to slide down the "bump" created by the gradient update. The higher the learning rate, the higher the bumps, and the faster the particles are descending the energy function. This implies that free particles are mixing much faster under the effect of training than in a static setting.

When training a deep Boltzmann machine, at least two sources of instability can be identified: (1) *Approximation instability*: The stochastic and approximate nature of the learning algorithms described above implies that the estimation of the gradient is noisy. The noise comes in part from the stochastic gradient descent procedure, but principally from the approximate sampling procedure that may cause systematically biased estimates of the gradient. (2) *Structural instability*: As it has been identified by Cho et al. [3], in standard Boltzmann machines, the weight matrix W tends to model in the first steps of the learning algorithm a global bias instead of co-dependencies between each units as we would expect. This is particularly problematic in the case of a Boltzmann machine with hidden-to-hidden connections such as the DBM, because hidden units tend to conglomerate and form a bias that may speed up learning initially but that eventually destroys the learning signal between pairs of hidden units.

25.3.1 The Centering Trick

The centering trick attempts to mitigate these sources of instability by ensuring that units activations intervening in the computation of the gradient are centered. Centering aims to produce a better conditioned optimization problem that is more robust to the noise of the learning procedure and to avoid the use of units as a global bias. Centering was already advocated in Chapter 1 [7] and 10 [17] in the context of backpropagation networks. Centering can be achieved by rewriting the energy of the Boltzmann machine as a function of centered states:

Center the Boltzmann machine

$$E(x; \theta) = -\frac{1}{2}(x - \beta)^\top W(x - \beta) - (x - \beta)^\top b$$

The new variable β represents the *offset* associated to each unit of the network and must be set to the mean activation of x . Setting $\beta = \text{sigm}(b_0)$ where b_0 is the initial bias ensures that units are initially centered. A similar parameterization of the energy function has been proposed by Tang and Sutskever [18] where the offset parameters were restricted to visible units. As we will see later, the Boltzmann machine also benefits from centering the hidden units. From this

new energy function, we can derive the conditional probability of each unit in the centered Boltzmann machine:

$$p(x_i = 1|x_{-i}; \theta) = \text{sigm}(b_i + \sum_{j \neq i} W_{ij}(x - \beta)_j).$$

Similarly, the gradient of the model log-likelihood with respect to W and b now takes the form:

$$\frac{\partial}{\partial W} \langle \log p(x_{\text{vis}}; \theta) \rangle_{\text{data}} = \langle (x - \beta)(x - \beta)^T \rangle_{\text{data}} - \langle (x - \beta)(x - \beta)^T \rangle_{\text{model}} \quad (25.5)$$

$$\frac{\partial}{\partial b} \langle \log p(x_{\text{vis}}; \theta) \rangle_{\text{data}} = \langle x - \beta \rangle_{\text{data}} - \langle x - \beta \rangle_{\text{model}}. \quad (25.6)$$

These gradients are similar to the *enhanced gradients* proposed by Cho et al. [3] and to those arising from the parameterization proposed by Arnold et al. [1] at the difference that our gradients do not account for the possibility that offsets β deviate from the mean activations of units throughout training. If the latter effect is problematic, it is possible to reparameterize the network continuously or at regular intervals so that the offsets correspond to the new expected means $\langle x \rangle_{\text{data}}$. The reparameterization $\theta \rightarrow \theta'$ must leave the energy function invariant up to a constant, that is, $E(x; \theta) = E(x; \theta') + \text{const}$. Solving the equation under the new centering constraints leads to the update equations $W' = W$, $b' = b + W(\langle x \rangle_{\text{data}} - \beta)$ and $\beta' = \langle x \rangle_{\text{data}}$.

Update biases and offsets at regular intervals

$$\begin{aligned} b' &= b + W(\langle x \rangle_{\text{data}} - \beta) \\ \beta' &= \langle x \rangle_{\text{data}} \end{aligned}$$

Similar derivations can be made for the deep Boltzmann machine. The energy function of the deep Boltzmann machine becomes $E(x, y, z; \theta) = -(y - \beta)^T W(x - \alpha) - (z - \gamma)^T V(y - \beta) - (x - \alpha)^T a - (y - \beta)^T b - (z - \gamma)^T c$ where α , β and γ are the offsets associated to the units in each layer. A basic algorithm for training a centered deep Boltzmann machine is given in Figure 25.3.

25.3.2 Understanding the Centering Trick

We look at the effect of centering on the stability of learning in a Boltzmann machine. We argue that when the Boltzmann machine is centered, the optimization problem is better conditioned (see Figure 25.5), more precisely, the ratio between the highest and the lowest eigenvalue of the Hessian \mathbf{H} is smaller. We define ξ as the centered state $\xi = x - \beta$. Substituting $x - \beta$ by ξ in Equation 25.5, the derivative of the data log-likelihood with respect to the weight parameter becomes

$$\frac{\partial}{\partial W} \langle \log p(x; \theta) \rangle_{\text{data}} = \langle \xi \xi^T \rangle_{W, \text{data}} - \langle \xi \xi^T \rangle_W$$

Training a centered deep Boltzmann machine

$W, V = 0, 0$
 $a, b, c = \text{sigm}^{-1}(x_{\text{data}}), b_0, c_0$
 $\alpha, \beta, \gamma = \text{sigm}(a), \text{sigm}(b), \text{sigm}(c)$
 initialize free particle $(x_m, y_m, z_m) = (\alpha, \beta, \gamma)$
loop
 initialize data particle $(x_d, y_d, z_d) = (\text{pick(data}), \beta, \gamma)$
loop
 $y_d \leftarrow \text{B}(\text{sigm}(W(x_d - \alpha) + V^\top(z_d - \gamma) + b))$
 $z_d \leftarrow \text{B}(\text{sigm}(V(y_d - \beta) + c))$
end loop
 $y_m \leftarrow \text{B}(\text{sigm}(W(x_m - \alpha) + V^\top(z_m - \gamma) + b))$
 $x_m \leftarrow \text{B}(\text{sigm}(W^\top(y_m - \beta) + a))$
 $z_m \leftarrow \text{B}(\text{sigm}(V(y_m - \beta) + c))$
 $W = W + \eta \cdot [(y_d - \beta)(x_d - \alpha)^\top - (y_m - \beta)(x_m - \alpha)^\top]$
 $V = V + \eta \cdot [(z_d - \gamma)(y_d - \beta)^\top - (z_m - \gamma)(y_m - \beta)^\top]$
 $a = a + \eta \cdot (x_d - x_m) + \nu \cdot W^\top(y_d - \beta)$
 $b = b + \eta \cdot (y_d - y_m) + \nu \cdot W(x_d - \alpha) + \nu \cdot V^\top(z_d - \gamma)$
 $c = c + \eta \cdot (z_d - z_m) + \nu \cdot V(y_d - \beta)$
 $\alpha = (1 - \nu) \cdot \alpha + \nu \cdot x_d$
 $\beta = (1 - \nu) \cdot \beta + \nu \cdot y_d$
 $\gamma = (1 - \nu) \cdot \gamma + \nu \cdot z_d$
end loop

Fig. 25.3. Basic algorithm for training a two-layer centered deep Boltzmann machine. The algorithm is based on persistent contrastive divergence and is kept minimal for the sake of simplicity. The variable η is the learning rate and the variable ν is the rate of the moving average necessary for the reparameterization. A Python implementation of the algorithm is available at <http://gregoire.montavon.name/code/dbm.py>.

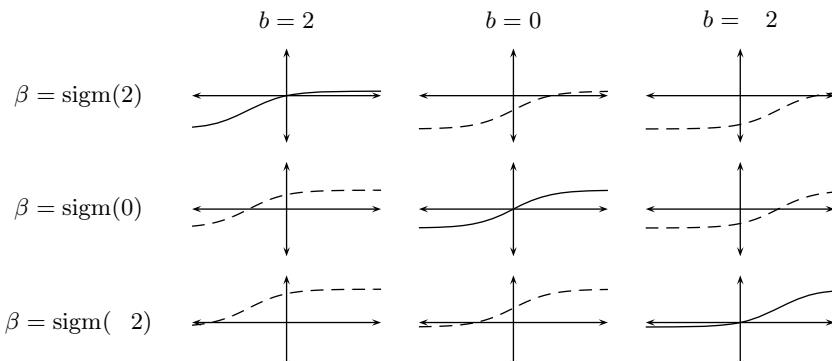


Fig. 25.4. Example of sigmoids $f(x) = \text{sigm}(x + b) - \beta$ with different biases b and offsets β . This figure illustrates how setting $\beta_0 = \text{sigm}(b_0)$ ensures that the sigmoid crosses the origin initially and do not contribute to modeling a bias component.

where $\langle \cdot \rangle_W$ denotes the expectation with respect to the probability distribution associated to a model of weight parameter W and $\langle \cdot \rangle_{W,\text{data}}$ denotes the expectation of the same model with visible units clamped to data. Using the definition of the directional derivative, the second derivative with respect to a random direction V (which is equal to the projected Hessian $\mathbf{H}V$) can be expressed as:

$$\begin{aligned}\mathbf{H}V &= \frac{\partial}{\partial V} \left(\frac{\partial}{\partial W} \langle \log p(x; W) \rangle_{\text{data}} \right) \\ &= \lim_{h \rightarrow 0} \frac{1}{h} \left(\frac{\partial}{\partial W} \langle \log p(x; W + hV) \rangle_{\text{data}} - \frac{\partial}{\partial W} \langle \log p(x; W) \rangle_{\text{data}} \right) \\ &= \lim_{h \rightarrow 0} \frac{1}{h} \left((\langle \xi \xi^\top \rangle_{W+hV, \text{data}} - \langle \xi \xi^\top \rangle_{W+hV}) - (\langle \xi \xi^\top \rangle_{W, \text{data}} - \langle \xi \xi^\top \rangle_W) \right) \\ &= \lim_{h \rightarrow 0} \frac{1}{h} (\langle \xi \xi^\top \rangle_{W+hV, \text{data}} - \langle \xi \xi^\top \rangle_{W, \text{data}}) - \lim_{h \rightarrow 0} \frac{1}{h} (\langle \xi \xi^\top \rangle_{W+hV} - \langle \xi \xi^\top \rangle_W)\end{aligned}$$

From the last line, we can see that the Hessian can be decomposed into a data-dependent term and a data-independent term. A remarkable fact is that in absence of hidden units, the data-dependent part of the Hessian is zero, because the model—and therefore, the perturbation of the model—have no influence on the states. The conditioning of the optimization problem can therefore be analyzed exclusively from a model perspective. The data-dependent term is likely to be small even in the presence of hidden variables due to the sharp reduction of entropy caused by the clamping of visible units to data.

We can think of a well-conditioned model as a model for which a perturbation of the model parameter W in any direction V causes a well-behaved perturbation of state expectations $\langle \xi \xi^\top \rangle_W$. Pearlmutter [11] showed that in a Boltzmann machine with no hidden units, the projected Hessian can be further reduced to

$$\mathbf{H}V = \langle \xi \xi^\top \rangle_W \cdot \langle D \rangle_W - \langle \xi \xi^\top D \rangle_W \quad \text{where } D = \frac{1}{2} \xi^\top V \xi \quad (25.7)$$

thus, getting rid of the limit and leading to numerically more accurate estimates. Chapter 1 [7] shows that the stability of the optimization problem can be quantified by the *condition number* defined as the ratio between the largest eigenvalue λ_1 and the smallest eigenvalue λ_n of \mathbf{H} . A geometrical interpretation of the condition number is given in Figure 25.5 (left). A low rank approximation of the Hessian can be obtained as

$$\hat{\mathbf{H}} = \mathbf{H}(V_0 | \dots | V_n) = (\mathbf{H}V_0 | \dots | \mathbf{H}V_n) \quad (25.8)$$

where the columns of $(V_0 | \dots | V_n)$ form a basis of independent unit vectors that projects the Hessian on a low-dimensional random subspace. The condition number can then be estimated by performing a singular value decomposition of the projected Hessian $\hat{\mathbf{H}}$ and taking the ratio between the largest and smallest resulting eigenvalues.

We estimate the condition number λ_1/λ_n of a fully connected Boltzmann machine of 50 units at initial state ($W = 0$) for different bias and offset pa-

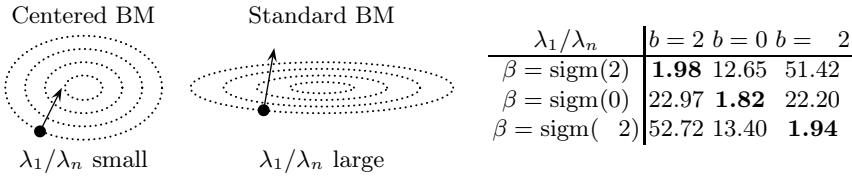


Fig. 25.5. Left: relation between the condition number λ_1/λ_n and the shape of the optimization problem. Gradient descent is easier to achieve when the condition number is small. Right: condition number obtained for centered Boltzmann machines (shown in bold on the diagonal) and for non-centered deep Boltzmann machines (off-diagonal elements). It can clearly be seen that the condition number is much smaller when the Boltzmann machine is centered.

parameters b and β using Equation 25.7 and 25.8. The condition numbers are obtained by drawing 100 random unit vectors for the projection matrix V and for each of them, estimating the statistics by sampling 1000 independent states ξ . Numerical estimates are given in Figure 25.5 (right) and clearly exhibit the better conditioning occurring when the Boltzmann machine is centered (i.e. when $\beta = \text{sigm}(b)$).

25.4 Evaluating Boltzmann Machines

In this section, we present two complementary approaches to evaluating a Boltzmann machine. The first method consists of looking at the discriminative components built in different portions of the Boltzmann machine (e.g. layers of a DBM) using kernels. The analysis is based on the work of Montavon et al. [8, 9] that characterizes the representation that emerges from the learning algorithm at each layer of a neural network. Second, we present a method introduced by Salakhutdinov and Hinton [13] that measures the generative performance of the Boltzmann machine in terms of log-likelihood of test data.

25.4.1 Discriminative Analysis

When Boltzmann machines incorporate a special structure, for example, via restricted connectivity, it can be useful to measure the discriminative capability of the representations emerging in specific portions of the network. Measuring the discriminative capability of a group of units is non-trivial, because (1) the meaningful information is not carried in a symbolic form but instead, distributed across the multiple units of the group and (2) the mapping of each data point onto these units is not deterministic but instead corresponds to the conditional distribution over the units in the group given the input x . We present here a method that exploits the insight that the projection of the input distribution onto the group of units forms a kernel (with a well-defined feature space [16]) that can be analyzed with respect to certain properties (or labels) t . The method was first introduced by Montavon et al. [8] in the context of backpropagation

networks. The method is based on the observation that leading kernel principal components can be approximated up to high accuracy from a finite, typically small set of samples [2]. We propose a family of kernels for representing the top layer of a DBM:

A family of kernels for deep Boltzmann machines:

$$k(x, x') = \mathbb{E}_{z, z'}[f(z|x, z'|x')]$$

where $z|x$ and $z'|x'$ denote the random top-layer activities conditioned respectively on data points x and x' and where the function $f(z, z')$ is a similarity metric between z and z' . Typical choices for f are:

$$\text{linear: } f(z, z') = \langle z, z' \rangle$$

$$\text{radial basis function: } f(z, z') = \exp\left(\frac{-\|z - z'\|}{\sigma}\right)$$

$$\text{equality: } f(z, z') = 1_{\{z=z'\}}$$

These kernels are able to gracefully deal with multimodal posteriors in the top-level distribution as the expectation operator lies outside the “detection function” f and therefore, accounts for all possible modalities of $z|x$ and $z'|x'$. Note that since the units of the Boltzmann machine are binary, norms $\|\cdot\|_1^1, \dots, \|\cdot\|_p^p$ are equivalent. Also, the linear and equality functions correspond up to some normalization to the extremal cases of the radial basis function kernel (with σ very large or very small). Once a kernel has been chosen, the analysis proceeds in four steps:

1. Collect a small test set X of size $n \times m$ and its associated label matrix T of size $n \times c$ and compute the empirical kernel K of size $n \times n$. The kernel can be built iteratively by running a Gibbs sampler on each data point and taking the average of all kernels. Alternatively a moving average of the kernel matrix can be maintained throughout training in order to keep track of the discriminative performance.
2. Perform an eigendecomposition of the kernel matrix $K = U \Lambda U^\top$ where Λ is a diagonal matrix representing the eigenvalues of K sorted by decreasing order, and where the columns of U represent the eigenvectors associated to each eigenvalue. These eigenvectors are the kernel principal components of X with respect to the kernel k and form a non-linear subspace that spans the main directions of variation in the data [15].
3. The representation is then evaluated by looking at how many kernel principal components are capturing the task T . Let $U_{1..d}$ and $\Lambda_{1..d}$ be the matrices containing respectively the d leading eigenvectors and eigenvalues. Compute the projected outputs $Y_d = U_{1..d} U_{1..d}^\top T$. These predictions are the optimal fit in the least square sense based on the d kernel principal components¹.

¹ Alternatively, if we would like to focus on the discrimination boundary between classes, a logistic model of type $\max_\beta \text{trace}(\log(\text{softmax}(\Phi_{1..d} \cdot \beta) \cdot T^\top))$ can be fitted, where $\Phi_{1..d} = U_{1..d} \Lambda_{1..d}^{0.5}$ is the empirical feature space and β of size $d \times c$ contains the regression parameters.

4. Compute the residuals curve $e(d)$ and the AUC:

$$e(d) = \|T - Y_d\|_F^2 \quad \text{AUC} = \frac{1}{n} \sum_{d=1}^n e(d) \quad (25.9)$$

These two quantities serve as metrics for evaluating how well the task is represented by the kernel. An interpretation of residuals curves $e(d)$ is given in Figure 25.6.

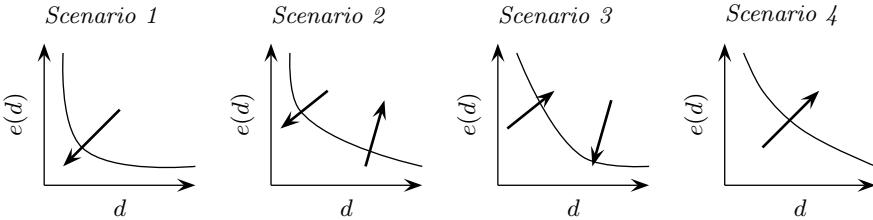


Fig. 25.6. Cartoon showing how to interpret residuals curves yield by various kernels on a certain task. *Scenario 1*: the kernel contains all label-relevant information in its principal components. This is the optimal case. *Scenario 2*: a large amount of label-relevant information is contained in the leading components, but remaining information is missing. *Scenario 3*: the task relevant information is contained in a large number of principal components but can be predicted accurately. *Scenario 4*: the kernel is not suited for the task of interest. Note that although Scenario 2 and 3 have similar AUC, their residuals curves are qualitatively very different.

25.4.2 Generative Analysis

The generative performance, measured in terms of data likelihood is what the Boltzmann machine is optimized for. Unfortunately, data likelihood can not be measured easily as it involves the estimation of the partition function that is generally intractable. As a consequence, we must recourse to sophisticated approximation schemes. We present an analysis introduced by Salakhutdinov and Murray [14] that estimates the likelihood of the learned Boltzmann machine based on annealed importance sampling (AIS) [10]. We describe here the basic analysis. Salakhutdinov and Hinton [13] introduced more elaborate procedures for particular types of Boltzmann machines such as restricted, semi-restricted and deep Boltzmann machines.

As we have seen in Section 25.2, a deep Boltzmann machine associates to each input x a probability

$$p(x; \theta) = \frac{\Psi(\theta, x)}{Z(\theta)}$$

$$\text{where } \Psi(\theta, x) = \sum_{y,z} p^*(x, y, z; \theta) \quad \text{and} \quad Z(\theta) = \sum_{x,y,z} p^*(x, y, z; \theta)$$

and where $p^*(x, y, z; \theta) = e^{-E(x, y, z; \theta)}$ is the unnormalized probability of state (x, y, z) . Computing $\Psi(\theta, x)$ and $Z(\theta)$ analytically is intractable because of the exponential number of elements involved in the sum. We must therefore resort to approximation procedures. Let us first rewrite the ratio of partition functions as follows:

$$p(x; \theta) = \frac{\Psi(\theta, x)}{Z(\theta)} = \frac{\frac{\Psi(\theta, x)}{\Psi(0, x)}}{\frac{Z(\theta)}{Z(0)}} \cdot \frac{\Psi(0, x)}{Z(0)} \quad (25.10)$$

It can be noticed that the ratio of base-rate partition functions ($\theta = 0$) is easy to compute as $\theta = 0$ makes all units independent. It has the analytical form

$$\frac{\Psi(0, x)}{Z(0)} = \frac{1}{2^{M_x}}. \quad (25.11)$$

The two other ratios in Equation 25.10 can be estimated using annealed importance sampling. The annealed importance sampling method proceeds as follows:

Annealed importance sampling (AIS) [10]:

1. Generate a sequence of states ξ_1, \dots, ξ_T using a sequence of transition operators $\mathcal{T}(\xi, \xi'; \theta_0), \dots, \mathcal{T}(\xi, \xi'; \theta_K)$ that leave $p(\xi)$ invariant, that is,
 - Draw ξ_0 from the base model (e.g. a random vector of zero and ones)
 - Draw ξ_1 given ξ_0 using $\mathcal{T}(\xi, \xi'; \theta_1)$
 - ...
 - Draw ξ_K given ξ_{K-1} using $\mathcal{T}(\xi, \xi'; \theta_K)$
2. Compute the importance weight

$$\omega_{\text{AIS}} = \frac{p^*(\xi_1; \theta_1)}{p^*(\xi_1; \theta_0)} \cdot \frac{p^*(\xi_2; \theta_2)}{p^*(\xi_2; \theta_1)} \cdot \dots \cdot \frac{p^*(\xi_K; \theta_K)}{p^*(\xi_K; \theta_{K-1})}$$

It can be shown that if the sequence of models $\theta_0, \theta_1, \dots, \theta_K$ where $\theta_0 = 0$ and $\theta_K = \theta$ evolves slowly enough, the importance weight obtained with the annealed importance sampling procedure is an estimate for the ratio between the partition function of the model θ and the partition function of the base rate model.

In our case, ξ denotes the state (x, y, z) of the DBM and the transition operator $\mathcal{T}(\xi, \xi'; \theta)$ corresponds to the alternating Gibbs samplers defined in Equation 25.3 and 25.4. The sequence of parameters $\{\theta_0, \dots, \theta_K\}$ can, for example, lie on the line between 0 and θ , that is, $\theta_k = \alpha_k \cdot \theta_K$ where $\alpha_0 < \dots < \alpha_K$. Alternatively, the sequence of parameters can be those that are observed throughout training. In that case, maintaining a moving average of the parameter throughout

training is necessary as the learning noise creates unnecessarily large variations between two adjacent parameters.

We can now compute the two ratios of partition functions of Equation 25.10 as

$$\frac{Z(\theta)}{Z(0)} \approx E[\omega_{\text{AIS}}] \quad \text{and} \quad \frac{\Psi(\theta, x)}{\Psi(0, x)} \approx E[\nu_{\text{AIS}}(x)] \quad (25.12)$$

where ω_{AIS} is the importance weight resulting from the annealing process with the freely running Gibbs sampler and ν_{AIS} is the importance weight resulting from the annealing with input units clamped to the data point. Substituting Equation 25.11 and 25.12 into Equation 25.10, we obtain

$$p(x; \theta) \approx \frac{E[\nu_{\text{AIS}}(x)]}{E[\omega_{\text{AIS}}]} \cdot \frac{1}{2^{M_x}}$$

and therefore, the log-likelihood of the model is estimated as:

$$E_X[\log(p(x; \theta))] \approx E_X[\log E[\nu_{\text{AIS}}(x)]] - \log E[\omega_{\text{AIS}}] - M_x \log(2). \quad (25.13)$$

Generally, computing an average of the importance weight ν_{AIS} for each data point x can take a long time. In practice, we can approximate it with a single AIS run for each data point. In that case, it follows from Jensen's inequality that

$$E_X[\log \nu_{\text{AIS}}(x)] - \log E[\omega_{\text{AIS}}] \leq E_X[\log E[\nu_{\text{AIS}}(x)]] - \log E[\omega_{\text{AIS}}]. \quad (25.14)$$

Consequently, this approximation tends to produce slightly pessimistic estimates of the model log-likelihood, however the variance of ν_{AIS} is low compared to the variance of ω_{AIS} because the clamping of visible units to data points reduces the diversity of AIS runs.

25.5 Experiments

In this section, we present a few experiments that demonstrate the effectiveness of the centering trick for learning deep Boltzmann machines. We use the MNIST handwritten digits recognition dataset that consists of 60000 training samples and 10000 test samples. Each sample is a 28×28 grayscale image representing a handwritten digit along with its label. Grayscale values (between 0 and 1) are treated as probabilities.

Architectures: We consider a deep Boltzmann machine (DBM) made of 28×28 input units, 200 intermediate units and 25 top units and a locally-connected DBM (LC-DBM) made of 28×28 input units, 400 intermediate units that connect to random input patches of size 6×6 and 100 top units. These architectures are illustrated in Figure 25.7 (left). In the DBM, the modeling load is concentrated in the first layer with the top layer serving merely to model global digit-like features. On the other hand, in the LC-DBM, most of the modeling load is postponed to the second layer and the first layer serves essentially as a low-level local

preprocessor. The initial offsets and biases for visible units are set to $\alpha = \langle x \rangle_{\text{data}}$ and $a_0 = \text{sigm}^{-1}(\alpha)$. We consider different initial biases ($b_0, c_0 = -2$, $b_0, c_0 = 0$ and $b_0, c_0 = 2$) and offsets ($\beta, \gamma = \text{sigm}(-2)$, $\beta, \gamma = \text{sigm}(0)$ and $\beta, \gamma = \text{sigm}(2)$) for the hidden units. These offsets and initial biases correspond to the sigmoids plotted in Figure 25.4.

Learning: We use persistent contrastive divergence [19] to train the network and keep track of 25 free particles in background of the learning procedure. We use a Gibbs sampler to collect *both* the data-independent and data-dependent statistics. At each iteration of the learning procedure, we run 3 iterations of the alternating Gibbs sampler for collecting the data-dependent statistics (from a minibatch of 25 data points) and one iteration for updating the data-independent statistics. We use a learning rate $\eta = 0.0004$ per sample for each layer.

Evaluation: Evaluating the DBM in an online fashion requires to keep track of the model parameters throughout training. We reduce the learning noise by maintaining a moving average of the sequence of parameters observed during learning. The moving average is tuned to remember approximately 10% of the training history. We keep track of 500 data-dependent chains running on the smoothed sequence of parameters and from which top-layer statistics $k(z, z')$ and ratios $\Psi(\theta, x)/\Psi(0, x)$ are estimated. We also keep track of 100 data-independent chains on the same sequence of parameters and from which the ratio $Z(\theta)/Z(0)$ is estimated. Discriminative performance is measured as the projection residuals of the labels on the kernel principal components and the area under the error curve (see Equation 25.9) using an exponential RBF kernel with σ set to the mean of pairwise distances between z and z' . Generative performance is measured in terms of data log-likelihood (see Equation 25.13).

Results: Figure 25.7 summarizes the results of our analysis and corroborates the importance of centering for obtaining a better discriminative and generative model of data. The centered DBM systematically produces better top-layer AUC errors and has higher log-likelihood. The importance of centering for improving generative models is particularly marked for the locally-connected DBM (LC-DBM) where the top-layer is crucial for modeling long-range dependencies. These results suggest that the centering trick is particularly useful when dealing with hierarchical architectures where global statistics are handled only in the deep layers of the network. Figure 25.8 (left) shows that the centered DBM yields a kernel that contains most of the label information in its leading components and has a low level of noise in the remaining components. This corresponds to Scenario 1 of Figure 25.6. On the other hand, in the case of the non-centered DBM, the labels span a few leading components of the kernel, but the remaining components have a high level of noise. This corresponds to Scenario 2 of Figure 25.6. As a comparison, the simple input-layer RBF kernel exhibits high dimensionality and low noise and thus, corresponds to Scenario 3 of Figure 25.6. The importance of centering for producing good top-layer kernels is further confirmed by looking at the second layer filters, visualized in Figure 25.8 (right) using a

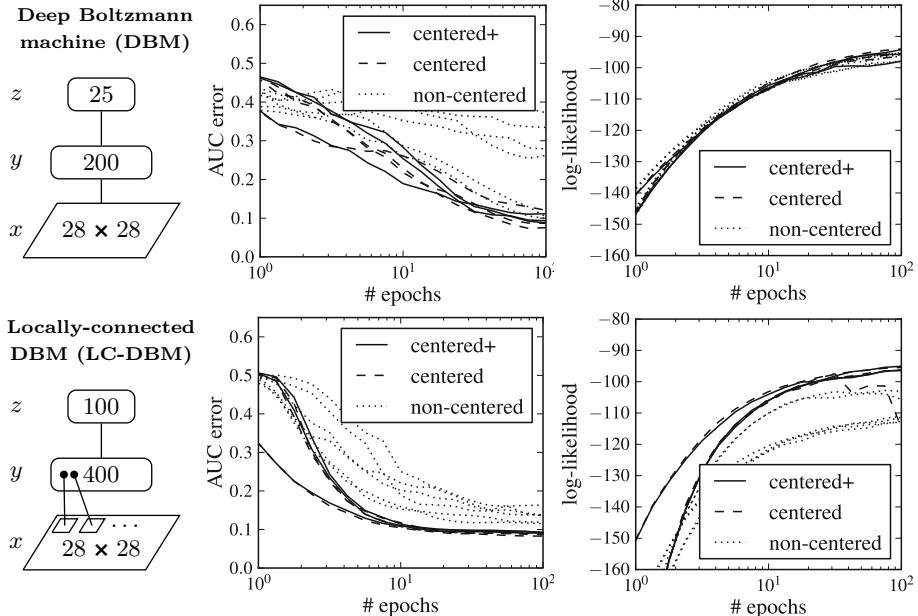


Fig. 25.7. Evolution of the AUC error and log-likelihood throughout training. “Centered+” designates deep Boltzmann machines that are continuously recentered throughout training. In the DBM, reasonable generative performance can be achieved without centering as the top layer is simply ignored by the rest of the model. In the LC-DBM, centering is important for both generative and discriminative performance as the top layer is required for modeling long-range dependencies. Continuously recentering yields the most robust performance.

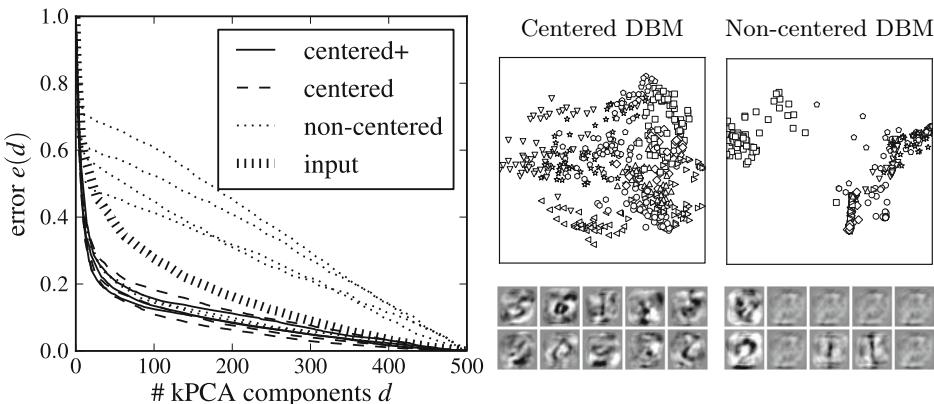


Fig. 25.8. Comparison of the top-layer representation produced by centered and non-centered DBMs. Left: error residuals produced by centered DBMs and non-centered DBMs. Right: 2D-PCA (with a linear kernel) and second-layer filters. Results suggest richer top-layer representations for centered DBMs than for non-centered DBMs.

linear back-projection, and observing that they are much richer for the centered DBM than for the non-centered one.

25.6 Conclusion

Learning deep Boltzmann machines is a difficult optimization problem that can be highly sensitive to the parameterization of its energy function. In this chapter, we propose the *centering trick* that consists of rewriting the energy as a function of centered states. The centering trick improves the stability of deep Boltzmann machines and allows to learn models that exhibit both advantageous discriminative and generative properties.

Our experiments have been most successful on mid-scale models (in the range of a few hundred hidden units). The high representational power of deep Boltzmann machines makes it hard to extend our experiments to larger scale models (of thousands of units) without using an explicit regularizer such as layer-wise pretraining or limited connectivity. We believe that applying the centering trick to large-scale models should be made in conjunction with a strong regularizer that limits the effective dimensionality of the model.

Acknowledgements. The authors thank Mikio Braun and the multiple reviewers for their useful comments. This work was supported by the World Class University Program through the National Research Foundation of Korea funded by the Ministry of Education, Science, and Technology, under Grant R31-10008. The authors also acknowledge partial support by DFG (MU 987/17-1).

References

- [1] Arnold, L., Auger, A., Hansen, N., Ollivier, Y.: Information-geometric optimization algorithms: A unifying picture via invariance principles, arXiv:1106.3708 (2011)
- [2] Braun, M.L., Buhmann, J., Müller, K.-R.: On relevant dimensions in kernel feature spaces. *Journal of Machine Learning Research* 9, 1875–1908 (2008)
- [3] Cho, K., Raiko, T., Ilin, A.: Enhanced gradient and adaptive learning rate for training restricted Boltzmann machines. In: *Proceedings of the 28th International Conference on Machine Learning*, pp. 105–112 (2011)
- [4] Hinton, G.E.: Training products of experts by minimizing contrastive divergence. *Neural Computation* 14(8), 1771–1800 (2002)
- [5] Hinton, G.E.: A Practical Guide to Training Restricted Boltzmann Machines. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) *NN: Tricks of the Trade*, 2nd edn. LNCS, vol. 7700, pp. 599–619. Springer, Heidelberg (2012)
- [6] Hinton, G.E., Sejnowski, T.J.: Learning and relearning in Boltzmann machines. In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, pp. 282–317. MIT Press (1986)
- [7] LeCun, Y., Bottou, L., Orr, G.B., Müller, K.-R.: Efficient BackProp. In: Orr, G.B., Müller, K.-R. (eds.) *NIPS-WS 1996*. LNCS, vol. 1524, pp. 9–50. Springer, Heidelberg (1998)

- [8] Montavon, G., Braun, M.L., Müller, K.-R.: Kernel analysis of deep networks. *Journal of Machine Learning Research* 12, 2563–2581 (2011)
- [9] Montavon, G., Braun, M.L., Müller, K.-R.: Deep Boltzmann machines as feed-forward hierarchies. *Journal of Machine Learning Research - Proceedings Track* 22, 789–804 (2012)
- [10] Neal, R.M.: Annealed importance sampling. *Statistics and Computing* 11(2), 125–139 (2001)
- [11] Pearlmutter, B.A.: Fast exact multiplication by the Hessian. *Neural Computation* 6(1), 147–160 (1994)
- [12] Salakhutdinov, R., Hinton, G.E.: Deep Boltzmann machines. In: *Proceedings of the International Conference on Artificial Intelligence and Statistics*, vol. 5, pp. 448–455 (2009)
- [13] Salakhutdinov, R.: Learning and Evaluating Boltzmann Machines. Technical Report UTML TR 2008-002, Dept. of Computer Science, University of Toronto (2008)
- [14] Salakhutdinov, R., Murray, I.: On the quantitative analysis of deep belief networks. In: *Proceedings of the 25th International Conference on Machine Learning, ICML 2008*, pp. 872–879 (2008)
- [15] Schölkopf, B., Smola, A., Müller, K.-R.: Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation* 10(5), 1299–1319 (1998)
- [16] Schölkopf, B., Mika, S., Burges, C.J.C., Knirsch, P., Müller, K.-R., Rätsch, G., Smola, A.J.: Input space versus feature space in kernel-based methods. *IEEE Transactions on Neural Networks* 10(5), 1000–1017 (1999)
- [17] Schraudolph, N.N.: Centering Neural Network Gradient Factors. In: Orr, G.B., Müller, K.-R. (eds.) *NIPS-WS 1996. LNCS*, vol. 1524, pp. 207–226. Springer, Heidelberg (1998)
- [18] Tang, Y., Sutskever, I.: Data normalization in the learning of restricted Boltzmann machines. Technical Report UTML-TR-11-2, Department of Computer Science, University of Toronto (2011)
- [19] Tieleman, T.: Training restricted Boltzmann machines using approximations to the likelihood gradient. In: *Proceedings of the 25th International Conference on Machine Learning*, pp. 1064–1071 (2008)

Deep Learning via Semi-supervised Embedding

Jason Weston¹, Frédéric Ratle², Hossein Mobahi³, and Ronan Collobert^{4,*}

¹ Google, New York, USA
jweston@google.com

² Nuance Communications, Montreal, Canada
frederic.ratle@gmail.com

³ Department of Computer Science, University of Illinois Urbana-Champaign, USA
hmobahi2@illinois.edu

⁴ IDIAP Research Institute, Martigny, Switzerland
ronan@collobert.com

Abstract. We show how nonlinear semi-supervised embedding algorithms popular for use with “shallow” learning techniques such as kernel methods can be easily applied to deep multi-layer architectures, either as a regularizer at the output layer, or on each layer of the architecture. Compared to standard supervised backpropagation this can give significant gains. This trick provides a simple alternative to existing approaches to semi-supervised deep learning whilst yielding competitive error rates compared to those methods, and existing shallow semi-supervised techniques.

26.1 Introduction

In this chapter we describe a trick for improving the generalization ability of neural networks by utilizing unlabeled *pairs* of examples for semi-supervised learning. The field of semi-supervised learning [7] has the goal of improving generalization on supervised tasks using unlabeled data. One of the tricks they use is the so-called embedding of data into a lower dimensional space (or the related task of clustering) which are unsupervised dimensionality reduction techniques that have been intensively studied. For example, researchers have used nonlinear embedding or cluster representations as features for a supervised classifier, with improved results. Many of those proposed architectures are *disjoint* and *shallow*, by which we mean the unsupervised dimensionality reduction algorithm is trained on unlabeled data separately as a first step, and then its results are fed to a supervised classifier which has a shallow architecture such as a (kernelized) linear model. For example, several methods learn a clustering or a distance measure based on a nonlinear manifold embedding as a first step [8, 9]. Transductive

* Much of the work in this chapter was completed while Jason Weston and Ronan Collobert were working at NEC labs, Princeton, USA, and while Frédéric Ratle was affiliated with the University of Lausanne, Switzerland. See [28] and [18] for conference papers on this subject.

Support Vector Machines (TSVMs) [26] (which employs a kind of clustering) and LapSVM [2] (which employs a kind of embedding) are examples of methods that are *joint* in their use of unlabeled data and labeled data, while their architecture is shallow. In this work we use the same embedding trick as those researchers, but apply it to (deep) neural networks.

Deep architectures seem a natural choice in hard AI tasks which involve several *sub-tasks* which can be coded into the layers of the architecture. As argued by several researchers [14, 3] semi-supervised learning is also natural in such a setting as otherwise one is not likely to ever have enough data to perform well. This is both because of the dearth of label data, and because of the difficulty of training the architectures. Secondly, intuitively one would think that training on labeled and unlabeled data *jointly* should help guide the best use of the unlabeled data for the labeled task compared to a two-stage disjoint approach. (However, to our knowledge there is no systematic evidence of the latter, and there might be reasons to train disjointly, for example label prediction tends to overfit faster than the embedding because you have less data to fit them. Doing unsupervised pretraining first and supervised fine-tuning afterwards might naturally solve this problem. On the other hand, it is only because the problem is non-convex that a two-stage approach does anything at all – all the learning from the first stage may be “forgotten”).

Several authors have recently proposed methods for using unlabeled data in deep neural network-based architectures. These methods either perform a greedy layer-wise pre-training of weights using unlabeled data alone followed by supervised fine-tuning (which can be compared to the *disjoint* shallow techniques for semi-supervised learning described before), or learn unsupervised encodings at multiple levels of the architecture jointly with a supervised signal. Only considering the latter, the basic setup we advocate is simple:

1. Choose an unsupervised learning algorithm.
2. Choose a model with a deep architecture.
3. The unsupervised learning is plugged into any (or all) layers of the architecture as an *auxiliary task*.
4. Train supervised and unsupervised tasks using the same architecture *simultaneously* (with a joint objective function).

The aim is that the unsupervised method will improve accuracy on the task at hand. In this chapter we advocate a simple way of performing deep learning by leveraging *existing* ideas from semi-supervised algorithms developed in *shallow* architectures. In particular, we focus on the idea of combining an *embedding*-based regularizer with a supervised learner to perform semi-supervised learning, such as is used in Laplacian SVMs [2]. We show that this method can be: (i) generalized to multi-layer networks and trained by stochastic gradient descent; and (ii) is valid in the *deep* learning framework given above. Experimentally, we also show that it seems to work quite well. We expect this is due to several effects: firstly, the extra embedding objective acts both as a data-dependent regularizer but secondly also as a weakly-supervised task that is correlated well with the supervised task of interest. Finally, adding this training objective at multiple layers of the network helps to train all

the layers rather than just backpropagating from the final layer as in supervised learning.

Although the core of this chapter focuses on a particular algorithm (embedding) in a joint setup, we expect the approach would also work in a disjoint setup too, and with other unsupervised algorithms, for example the approach of Transductive SVM has also been generalized to the deep learning case [15].

26.2 Semi-supervised Embedding

Our method will adapt existing semi-supervised embedding techniques for shallow methods to neural networks. Hence, before we describe the method, let us first review existing semi-supervised approaches. A key assumption in many semi-supervised algorithms is the structure assumption¹: points within the same structure (such as a cluster or a manifold) are likely to have the same label. Given this assumption, the aim is to use unlabeled data to uncover this structure. In order to do this many algorithms such as cluster kernels [8], LDS [9], label propagation [30] and LapSVM [2], to name a few, make use of regularizers that are directly related to unsupervised embedding algorithms. To understand these methods we will first review some relevant approaches to linear and nonlinear embedding.

26.2.1 Embedding Algorithms

We will focus on a rather general class of embedding algorithms that can be described by the following type of optimization problem: given the data x_1, \dots, x_U find an embedding $f(x_i)$ of each point x_i by minimizing

$$\sum_{i,j=1}^U L(f(x_i, \alpha), f(x_j, \alpha), W_{ij})$$

w.r.t. the learning parameters α , subject to

Balancing constraint.

This type of optimization problem has the following main ingredients:

- $f(x) \in \mathbb{R}^n$ is the embedding one is trying to learn for a given example $x \in \mathbb{R}^d$. It is parametrized by α . In many techniques $f(x_i) = f_i$ is a lookup table where each example i is assigned an independent vector f_i .
- L is a loss function between pairs of examples.
- The matrix W of weights W_{ij} specifies the similarity or dissimilarity between examples x_i and x_j . This is supplied in advance and serves as a kind of label for the loss function.
- A balancing constraint is often required for certain objective functions so that a trivial solution is not reached.

¹ This is often referred to as the cluster assumption or the manifold assumption [7].

As is usually the case for such machine learning setups, one can specify the model type (family of functions) and the loss to get different algorithmic variants. Many well known methods fit into this framework, we describe some pertinent ones below.

Multidimensional scaling (MDS) [16] is a classical algorithm that attempts to preserve the distance between points, whilst embedding them in a lower dimensional space, e.g. by using the loss function

$$L(f_i, f_j, W_{ij}) = (\|f_i - f_j\| - W_{ij})^2$$

MDS is equivalent to PCA if the metric is Euclidean [29].

ISOMAP [25] is a nonlinear embedding technique that attempts to capture manifold structure in the original data. It works by defining a similarity metric that measures distances along the manifold, e.g. W_{ij} is defined by the shortest path on the neighborhood graph. One then uses those distances to embed using conventional MDS.

Laplacian Eigenmaps [1] learn manifold structure by emphasizing the preservation of *local distances*. One defines the distance metric between the examples by encoding them in the Laplacian $\tilde{L} = W - D$, where $D_{ii} = \sum_j W_{ij}$ is diagonal. Then, the following optimization is used:

$$\sum_{ij} L(f_i, f_j, W_{ij}) = \sum_{ij} W_{ij} \|f_i - f_j\|^2 = f^\top \tilde{L} f \quad (26.1)$$

subject to the balancing constraint:

$$f^\top D f = I \text{ and } f^\top D 1 = 0. \quad (26.2)$$

Siamese Networks [4] are also a classical method for nonlinear embedding. Neural networks researchers think of such models as a network with two identical copies of the same function, with the same weights, fed into a “distance measuring” layer to compute whether the two examples are similar or not, given labeled data. In fact, this is exactly the same as the formulation given at the beginning of this section.

Several loss functions have been proposed for *siamese networks*, here we describe a margin-based loss proposed by the authors of [13]:

$$L(f_i, f_j, W_{ij}) = \begin{cases} \|f_i - f_j\|_2 & \text{if } W_{ij} = 1, \\ \max(0, m - \|f_i - f_j\|_2)^2 & \text{if } W_{ij} = 0 \end{cases} \quad (26.3)$$

which encourages similar examples to be close, and dissimilar ones to have a distance of at least m from each other. Note that no balancing constraint is needed with such a choice of loss as the margin constraint inhibits a trivial solution. Compared to using constraints like (26.2) this is much easier to optimize by gradient descent.

26.2.2 Semi-supervised Algorithms

Several *semi-supervised* classification algorithms have been proposed which take advantage of the algorithms described in the last section. Here we assume the setting where one is given $M+U$ examples x_i , but only the first M have a known label y_i .

Label Propagation [30] adds a Laplacian Eigenmap type regularization to a nearest-neighbor type classifier:

$$\min_f \sum_{i=1}^M \|f_i - y_i\|^2 + \lambda \sum_{i,j=1}^{M+U} W_{ij} \|f_i - f_j\|^2 \quad (26.4)$$

The algorithm tries to give two examples with large weighted edge W_{ij} the same label. The neighbors of neighbors tend to also get the same label as each other by transitivity, hence the name *label propagation*.

LapSVM [2] uses the Laplacian Eigenmaps type regularizer with an SVM:

$$\min_{w,b} \|w\|^2 + \gamma \sum_{i=1}^M H(y_i f(x_i)) + \lambda \sum_{i,j=1}^{M+U} W_{ij} \|f(x_i) - f(x_j)\|^2 \quad (26.5)$$

where $H(x) = \max(0, 1 - x)$ is the hinge loss, and the final classifier will be $f(x) = w \cdot x + b$.

Other Methods In [9] a method called *graph* is suggested which combines a modified version of ISOMAP with an SVM. The authors also suggest to combine modified ISOMAP with TSVMs rather than SVMs, and call it *Low Density Separation* (LDS).

26.3 Semi-supervised Embedding for Deep Learning

We would like to use the ideas developed in semi-supervised learning for *deep learning*. Deep learning consists of learning a model with several layers of non-linear mapping. In this chapter we will consider multi-layer networks with N layers of hidden units that give a C -dimensional output vector:

$$f_i(x) = \sum_{j=1}^d w_j^{O,i} h_j^N(x) + b^{O,i}, \quad i = 1, \dots, C \quad (26.6)$$

where w^O are the weights for the output layer, and typically the k^{th} layer is defined as

$$h_i^k(x) = S \left(\sum_j w_j^{k,i} h_j^{k-1}(x) + b^{k,i} \right), \quad k > 1 \quad (26.7)$$

$$h_i^1(x) = S \left(\sum_j w_j^{1,i} x_j + b^{1,i} \right) \quad (26.8)$$

and S is a non-linear squashing function such as tanh. Here, we describe a standard *fully connected* multi-layer network but prior knowledge about a particular problem could lead one to other network designs. For example in sequence and image recognition time delay and convolutional networks (TDNNs and CNNs) [17] have been very successful. In those approaches one introduces layers that apply convolutions on their input which take into account locality information in the data, i.e. they learn features from image patches or windows within a sequence.

The general method we propose for *deep learning via semi-supervised embedding* is to add a semi-supervised regularizer in deep architectures in one of three different modes, as shown in Figure 26.1:

- (a) Add a semi-supervised loss (regularizer) to the supervised loss on the entire network's output (26.6):

$$\sum_{i=1}^M \ell(f(x_i), y_i) + \lambda \sum_{i,j=1}^{M+U} L(f(x_i), f(x_j), W_{ij}) \quad (26.9)$$

This is most similar to the *shallow* techniques described before, e.g. equation (26.5).

- (b) Regularize the k^{th} hidden layer (26.7) directly:

$$\sum_{i=1}^M \ell(f(x_i), y_i) + \lambda \sum_{i,j=1}^{M+U} L(f^k(x_i), f^k(x_j), W_{ij}) \quad (26.10)$$

where $f^k(x) = (h_1^k(x), \dots, h_{HU_k}^k(x))$ is the output of the network up to the k^{th} hidden layer (HU_k is the number of hidden units on layer k).

- (c) Create an auxiliary network which shares the first k layers of the original network but has a new final set of weights:

$$g_i(x) = \sum_j w_j^{AUX,i} h_j^k(x) + b^{AUX,i} \quad (26.11)$$

We train this network to *embed* unlabeled data simultaneously as we train the original network on *labeled* data.

One can use the loss function (26.3) for embedding, and the hinge loss

$$\ell(f(x), y) = \sum_{c=1}^C H(y(c)f_c(x)),$$

for labeled examples, where $y(c) = 1$ if $y = c$ and -1 otherwise. For neighboring points, this is the same regularizer as used in LapSVM and Laplacian Eigenmaps.

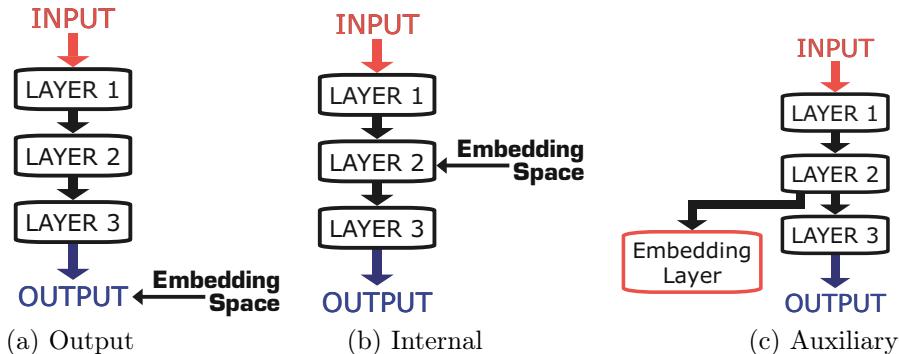


Fig. 26.1. Three modes of embedding in deep architectures

Algorithm 26.1 *EmbedNN*

Input: labeled data (x_i, y_i) , $i = 1, \dots, M$, unlabeled data x_i , $i = M+1, \dots, U$, set of functions $f(\cdot)$, and embedding functions $g^k(\cdot)$, see Figure 26.1 and equations (26.9), (26.10) and (26.11).

repeat

Pick a random *labeled* example (x_i, y_i)

Make a gradient step to optimize $\ell(f(x_i), y_i)$

for each embedding function $g^k(\cdot)$ **do**

Pick a random pair of neighbors x_i, x_j .

Make a gradient step for $\lambda L(g^k(x_i), g^k(x_j), 1)$

Pick a random unlabeled example x_n .

Make a gradient step for $\lambda L(g^k(x_i), g^k(x_n), 0)$

end for

until stopping criteria is met.

For non-neighbors, where $W_{ij} = 0$, this loss ‘‘pulls’’ points apart, thus inhibiting trivial solutions without requiring difficult constraints such as (26.2). To achieve an embedding *without* labeled data the latter is necessary or all examples would collapse to a single point in the embedding space. This regularizer is therefore preferable to using (26.1) alone. Pseudocode of the overall approach is given in Algorithm 26.1.

Some possible tricks to take into consideration are:

- The hyperparameter λ : in most of our experiments we simply set this to $\lambda = 1$ and it worked well due to the alternating updates in Algorithm 26.1. Note however if you are using many embedding loss functions they will dominate the objective in that case.
- We note that near the end of optimization it may be advantageous to reduce the learning rate of the regularizer more than the learning rate for the

term that is minimizing the training error so that the training error can be as low as possible on noiseless tasks (however we did not try this in our experiments).

- If you use an internal embedding on the first layer of your network, it is likely that this embedding problem is harder than an internal embedding on a later layer, so you might not want to give them all the same learning rate or margin, but that complicates the hyperparameter choices. An alternative idea would be to use auxiliary layers on earlier layers, or even go through two auxiliary layers, rather than one to make the embedding task easier. Auxiliary layers are thrown away at test time.
- Embedding on the last output layer may not always be a good idea, depending on the type of network. For example if you are using a softmax last layer the 2-norm type embedding loss may not be appropriate for the log probability representation in the last layer. In that case we suggest to do the embedding on the last-but-one layer instead.
- Finally, although we did not try it, training in a disjoint fashion, i.e. doing the embedding training first, and then continuing training with a fine tuning step with only the labeled data, might simplify these hyperparameter choices above.

26.3.1 Labeling Unlabeled Data as Neighbors (Building the Graph)

Training neural networks online using stochastic gradient descent is fast and can scale to millions of examples. A possible bottleneck with the described approach is computation of the matrix W , that is, computing which unlabeled examples are neighbors and have value $W_{ij} = 1$. Embedding algorithms often use k -nearest neighbor for this task. Many methods for its fast computation do exist, for example hashing and tree-based methods.

However, there are also many other ways of collecting neighboring unlabeled data that do not involve computing k -nn. For example, if one has access to unlabeled sequence data the following tricks can be used:

- For image tasks one can make use of the temporal coherence of unlabeled video: two successive frames are very likely to contain similar content and represent the same concept classes. Each object in the video is also likely to be subject to small transformations, such as translation, rotation or deformation over neighboring frames. Hence, using this with semi-supervised embedding could learn classes that are invariant to those changes. For example, one can take images from two consecutive (or close) frames of video as a neighboring pair with $W_{ij} = 1$. Such pairs are likely to have the same label, and are collected cheaply. Frames that are far apart are assigned $W_{ij} = 0$.
- For text tasks one can use documents to collect unsupervised pairs. For example, one could consider sentences (or paragraphs) of a document as neighbors that contain semantically similar information (they are probably about the same topic).
- Similarly, for speech tasks it might be possible to use audio streams in the same way.

26.3.2 When Do We Expect This Approach to Work?

One can see the described approach as an instance of multi-task learning [6] using unsupervised auxiliary tasks. In common with other semi-supervised learning approaches, and indeed other deep learning approaches, given a k -nn type approach to building unlabeled pairs we only expect this to work if $p(x)$ is useful for the supervised task $p(y|x)$, i.e. if the structure assumption is true. That is, if the decision rule lies in a region of low density with respect to the distance metric chosen for k -nearest neighbors. We believe many natural tasks have this property.

However, if the graph is built using sequence data as described in the previous section, it is then possible that the method does not rely on the low density assumption at all. To see this, consider uniform two-dimensional data where the class label is positive if it is above the y-axis, and negative if it is below. A nearest-neighbor graph gives no information about the class label, or equivalently there is no margin to optimize for TSVMs. However, if sequence data (analogous to a video) only has data points with the same class label in consecutive frames then this would carry information. Further, no computational cost is associated with collecting video data for computing the embedding loss, in contrast to building neighbor graphs. Finally, note that in high dimensional spaces nearest neighbors might also perform poorly, e.g. in the pixel space of images.

26.3.3 Why Is This Approach Good?

There are a number of reasons why the deep semi-supervised embedding trick might be useful compared to competing approaches:

- Deep embedding is very easy to optimize by gradient descent as it has a very simple loss function. This means it can be applied to any kind of neural network architecture cheaply and efficiently. As well as being generally applicable, it is also quite easy to implement.
- Compared to a reconstruction based loss function, such as used in an autoencoder, our approach can be much cheaper to do the gradient updates. In our approach there is an encoding step, but no decoding step. That is, the loss is measured in the usually relatively low-dimensional embedding space. For high-dimensional input data (even if that data is sparse) e.g. text data, the reconstruction can be very slow, e.g. a bag-of-words representation with a dictionary of tens of thousands of words. Further, in a convolutional-pooling network architecture it might be hard to reconstruct the original data, so again an encoder-decoder system might be hard to do, but our method only requires an encoder.
- Our approach does not necessarily require the so called low density assumption which most other approaches depend upon. Many methods only work on data when that assumption is true (which we do not know in advance in general). Our method may still work, depending on how the pair-data is collected. This point was elaborated in the previous subsection.

Table 26.1. Datasets used in our experiments. The first three are small scale datasets used in the same experimental setup as found in [9, 24, 10]. The following six datasets are large scale. The Mnist 1h, 6h, 1k, 3k and 60k variants are MNIST with a labeled subset of data, following the experimental setup in [10]. SRL is a Semantic Role Labeling task [20] with one million labeled training examples and 631 million unlabeled examples. COIL100 is an object detection dataset [19].

data set	classes	dims	points	labeled
g50c	2	50	500	50
Text	2	7511	1946	50
Uspst	10	256	2007	50
Mnist1h	10	784	70k	100
Mnist6h	10	784	70k	600
Mnist1k	10	784	70k	1000
Mnist3k	10	784	70k	3000
Mnist60k	10	784	70k	60000
SRL	16	-	631M	1M
COIL100 (30 objects)	30	72x72 pixels	7200	120
COIL100 (100 objects)	100	72x72 pixels	7200	400

26.4 Experimental Evaluation

We test the semi-supervised embedding approach on several datasets summarized in Table 26.1.

26.4.1 Small-Scale Experiments

g50c, Text and Uspst are small-scale datasets often used for semi-supervised learning experiments [9, 24, 10]. We followed the same experimental setup, averaging results of ten splits of 50 labeled examples where the rest of the data is unlabeled. In these experiments we test the embedding regularizer on the output of a neural network (see equation (26.9) and Figure 26.1(a)). We define a two-layer neural network (NN) with hu hidden units. We define W so that the 10 nearest neighbors of i have $W_{ij} = 1$, and $W_{ij} = 0$ otherwise. We train for 50 epochs of stochastic gradient descent and fixed $\lambda = 1$, but for the first 5 we optimized the supervised target alone (without the embedding regularizer). This gives two free hyperparameters: the number of hidden units $hu = \{0, 5, 10, 20, 30, 40, 50\}$ and the learning rate $lr = \{0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001\}$.

We report the optimum choices of these values optimized both by 5-fold cross validation and by optimizing on the test set in Table 26.2. Note the datasets are very small, so cross validation is unreliable. Several methods from the literature optimized their hyperparameters using the test set (those that are not marked with (cv)). Our *EmbedNN* is competitive with state-of-the-art semi-supervised methods based on SVMs, even outperforming them in some cases.

Table 26.2. Results on Small-Scale Datasets. We report the best test error over the hyperparameters of our method, *EmbedNN*, as in the methodology of [9] as well as the error when optimizing the parameters by cross-validation, $\text{EmbedNN}^{(cv)}$. LDS^(cv) and LapSVM^(cv) also use cross-validation.

	g50c	Text	Uspst
SVM	8.32	18.86	23.18
TSVM	5.80	5.71	17.61
LapSVM ^(cv)	5.4	10.4	12.7
LDS ^(cv)	5.4	5.1	15.8
Label propagation	17.30	11.71	21.30
Graph SVM	8.32	10.48	16.92
NN	10.62	15.74	25.13
<i>EmbedNN</i>	5.66	5.82	15.49
<i>EmbedNN</i> ^(cv)	6.78	6.19	15.84

Table 26.3. Results on MNIST with 100, 600, 1000 and 3000 labels. A two-layer Neural Network (NN) is compared to an NN with Embedding regularizer (*EmbedNN*) on the output (O), i^{th} layer (I_i) or auxiliary embedding from the i^{th} layer (A_i) (see Figure 26.1). A convolutional network (CNN) is also tested in the same way. We compare to SVMs and TSVMs. RBM, SESM, DBN-NCA and DBN-rNCA (marked with (*)) taken from [21, 23] are trained on a different data split.

	Mnist1h	Mnist6h	Mnist1k	Mnist3k
SVM	23.44	8.85	7.77	4.21
TSVM	16.81	6.16	5.38	3.45
RBM ^(*)	21.5	-	8.8	-
SESM ^(*)	20.6	-	9.6	-
DBN-NCA ^(*)	-	10.0	-	3.8
DBN-rNCA ^(*)	-	8.7	-	3.3
NN	25.81	11.44	10.70	6.04
<i>Embed</i> ^{O} NN	17.05	5.97	5.73	3.59
<i>Embed</i> ^{I^1} NN	16.86	9.44	8.52	6.02
<i>Embed</i> ^{A^1} NN	17.17	7.56	7.89	4.93
CNN	22.98	7.68	6.45	3.35
<i>Embed</i> ^{O} CNN	11.73	3.42	3.34	2.28
<i>Embed</i> ^{I^5} CNN	7.75	3.82	2.73	1.83
<i>Embed</i> ^{A^5} CNN	7.87	3.82	2.76	2.07

Table 26.4. Mnist1h dataset with deep networks of 2, 6, 8, 10 and 15 layers; each hidden layer has 50 hidden units. We compare classical NN training with *EmbedNN* where we either learn an embedding at the output layer (O) or an auxiliary embedding on all layers at the same time (ALL).

	2	4	6	8	10	15
NN	26.0	26.1	27.2	28.3	34.2	47.7
<i>Embed</i> ^{O} NN	19.7	15.1	15.1	15.0	13.7	11.8
<i>Embed</i> ^{ALL} NN	18.2	12.6	7.9	8.5	6.3	9.3

Table 26.5. Full Mnist60k dataset with deep networks of 2, 6, 8, 10 and 15 layers, using either 50 or 100 hidden units. We compare classical NN training with $Embed^{ALL}$ NN where we learn an auxiliary embedding on all layers at the same time.

	2	4	6	8	10	15
NN (HUs=50)	2.9	2.6	2.8	3.1	3.1	4.2
$Embed^{ALL}$ NN	2.8	1.9	2.0	2.2	2.4	2.6
NN (HUs=100)	2.0	1.9	2.0	2.2	2.3	3.0
$Embed^{ALL}$ NN	1.9	1.5	1.6	1.7	1.8	2.4

26.4.2 MNIST Experiments

We compare our method in all three different modes (Figure 26.1) with conventional semi-supervised learning (TSVM) using the same data split and validation set as in [10]. We also compare to several deep learning methods: RBMs (Restricted Boltzmann Machines), SESM (Sparse Encoding Symmetric Machine), DBN-NCA and DBN-rNCA (Deep Belief Nets - (regularized) Neighbourhood Components Analysis). (Note, however the latter were trained on a different data split). In these experiments we consider 2-layer networks (NN) and 6-layer convolutional neural nets (CNN) for embedding. We optimize the parameters of NN ($hu = \{50, 100, 150, 200, 400\}$ hidden units and learning rates as before) on the validation set. The CNN architecture is fixed: 5 layers of image patch-type convolutions, followed by a linear layer of 50 hidden units, similar to [17]. The results given in Table 26.3 show the effectiveness of embedding in all three modes, with both NNs and CNNs.

26.4.3 Deeper MNIST Experiments

We then conducted a similar set of experiments but with very deep architectures – up to 15 layers, where each hidden layer has 50 hidden units. Using Mnist1h, we first compare conventional NNs to $Embed^{ALL}$ NN where we learn an auxiliary nonlinear embedding (50 hidden units and a 10 dimensional embedding space) on each layer, as well as $Embed^O$ NN where we only embed the outputs. Results are given in Table 26.4. When we increase the number of layers, NNs trained with conventional backpropagation overfit and yield steadily *worse* test error (although they are easily capable of achieving zero training error). In contrast, $Embed^{ALL}$ NN *improves* with increasing depth due to the semi-supervised “regularization”. Embedding on *all* layers of the network has made *deep learning* possible. $Embed^O$ NN (embedding on the outputs) also helps, but not as much.

We also conducted some experiments using the full MNIST dataset, Mnist60k. Again using deep networks of up to 15 layers using either 50 or 100 hidden units $Embed^{ALL}$ NN outperforms standard NN. Results are given in Table 26.5. Despite the lack of availability of extra unlabeled data, we still see the same effect as in the semi-supervised case that NN overfits with increasing capacity, whereas $EmbedNN$ is more robust (even if it exhibits some overfitting compared to the optimal depth, it is nowhere near as pronounced.) Increasing the number of

hidden units is likely to improve these results further, e.g. using 4 layers and 500 hidden units on each layer, one obtains 1.27% using $\text{Embed}^{ALL}\text{NN}$. Overall, these results show that the regularization in Embed^{ALL} is useful in settings outside of a semi-supervised learning.

Table 26.6. A deep architecture for Semantic Role Labeling with no prior knowledge outperforms state-of-the-art systems ASSERT and SENNA that incorporate knowledge about parts-of-speech and parse trees. A convolutional network (CNN) is improved by learning an auxiliary embedding ($\text{Embed}^{A1}\text{CNN}$) for words represented as 100-dimensional vectors using the entire Wikipedia website as unlabeled data.

Method	Test Error
ASSERT [20]	16.54%
SENNNA [11]	16.36%
CNN [no prior knowledge]	18.40%
$\text{Embed}^{A1}\text{CNN}$ [no prior knowledge]	14.55%

26.4.4 Semantic Role Labeling

The goal of semantic role labeling (SRL) is, given a sentence and a relation of interest, to label each word with one of 16 tags that indicate that word’s semantic role with respect to the action of the relation. For example the sentence “*The cat eats the fish in the pond*” is labeled in the following way: “*The_{ARG0} cat_{ARG0} eats_{REL} the_{ARG1} fish_{ARG1} in_{ARGM-LOC} the_{ARGM-LOC} pond_{ARGM-LOC}*” where ARG0 and ARG1 effectively indicate the subject and object of the relation “eats” and ARGM-LOC indicates a locational modifier. The PropBank dataset includes around 1 million labeled words from the Wall Street Journal. We follow the experimental setup of [11] and train a 5-layer convolutional neural network for this task, where the first layer represents the input sentence words as 50-dimensional vectors. Unlike [11], we do not give any prior knowledge to our classifier. In that work words were stemmed and clustered using their parts-of-speech. Our classifier is trained using only the original input words.

We attempt to improve this system by, as before, learning an *auxiliary embedding* task. Our embedding is learnt using unlabeled sentences from the Wikipedia web site, consisting of 631 million words in total using the scheme described in Section 26.3. The same lookup table of word vectors as in the supervised task is used as input to an 11 word window around a given word, yielding 550 features. Then a linear layer projects these features into a 100 dimensional embedding space. All windows of text from Wikipedia are considered neighbors, and non-neighbors are constructed by replacing the middle word in a sentence window with a random word. Our lookup table indexes the most frequently used 30,000 words, and all other words are assigned index 30,001.

The results in Table 26.6 indicate a clear improvement when learning an auxiliary embedding. ASSERT [20] is an SVM parser-based system with many hand-coded features, and SENNA is a NN which uses part-of-speech information to build its word vectors. In contrast, our system is the only state-of-the-art

Table 26.7. Test Accuracy on COIL100 in various settings. Both 30 and 100 objects were used following [27]. The semi-supervised embedding algorithm using temporal coherence of video (*Embed CNN*) on the last but one layer of an 8 layer CNN, with various choices of video, outperforms a standard (otherwise identical) 8-layer CNN and other baselines. (Note that with 100 objects this is a transductive approach, as we use the test set as unlabeled data during training, whereas with 30 objects a semi-supervised approach is used.)

Method	30 objects	100 objects
Nearest Neighbor	81.8	70.1
SVM	84.9	74.6
SpinGlass MRF	82.79	69.41
Eigen Spline	84.6	77.0
VTU	89.9	79.1
Standard CNN	84.88	71.49
<i>Embed CNN</i>	95.03	92.25

method that does not use prior knowledge in the form of features derived from parts-of-speech or parse tree data. The use of neural network techniques for this application is explored in much more detail in [12], although a different semi-supervised technique is used in that work.

26.4.5 Object Recognition Using Unlabeled Video

Finally, we detail some experiments using unlabeled video for semi-supervised embedding, more details of these experiments can be found in [18]. We used the COIL100 image dataset [19] which contains color pictures of 100 objects, each 72x72 pixels. There are 72 different views for every object, i.e. there are 7200 images in total. The images were obtained by placing the objects on a turntable and taking a shot for each 5 degree turn. Note that the rotation of the objects can be viewed as an unlabeled video which we can use in our semi-supervised embedding approach.

The setup of our experiments is as follows. First, we use a standard convolutional neural network (CNN) without utilizing any temporal information to establish a baseline. We used an 8-layer network consisting of three sets of convolution followed by subsampling layers, a final convolution layer and a fully connected layer that predicts the outputs.

For comparability with the settings available from other studies on COIL100, we choose two experimental setups. These are (i) when all 100 objects of COIL are considered in the experiment and (ii) when only 30 labeled objects out of 100 are studied (for both training and testing). In either case, 4 out of 72 views (at 0, 90, 180, and 270 degrees) per object are used for training, and the rest of the 68 views are used for testing. The results are given in Table 26.7 compared to some existing methods [22, 27, 5]. Note that using 100 objects is a harder task than using 30 objects (classes).

To use the semi-supervised embedding trick on our CNN for video, we treat COIL100 as a continuous unlabeled video sequence of rotating objects with 72 consecutive frames per each object (after 72 frames the continuous video switches object). We perform the embedding on the last but one layer of our 8 layer CNN, i.e. on the representation yielded by the successive layers of the network just before the final softmax. For the 100 object result, the test set is hence part of the unlabeled video (a so-called “transductive” setting). Here we obtained 92.25% accuracy (*Embed CNN*) which is much higher than the best alternative method (VTU) and the standard CNN that we trained.

A natural question is what happens if we do not have access to test data during training, i.e. the setting is a typical semi-supervised situation rather than a “transductive” setting. To explore this, we used 30 objects as the primary task, i.e. 4 views of each object in this set were used for training, and the rest for test. The other 70 objects only were treated as an unlabeled video sequence (again, images of each object were put in consecutive frames of a video sequence). Training with 4 views of 30 objects (labeled data) and 72 views of 70 objects (unlabeled video sequence) resulted in an accuracy of 95.03% on recognizing 68 views of the 30 objects. This is about 10% above the standard CNN performance.

26.5 Conclusion

In this chapter, we showed how one can improve supervised learning for deep architectures if one jointly learns an embedding task using unlabeled data. Researchers using *shallow* architectures already showed two ways of using embedding to improve generalization: (i) embedding unlabeled data as a *separate* pre-processing step (i.e., first layer training) and; (ii) using embedding as a regularizer (i.e., at the output layer). It appears similar techniques can also be used for multi-layer neural networks as well, using the tricks described in this chapter.

References

- [1] Belkin, M., Niyogi, P.: Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation* 15(6), 1373–1396 (2003)
- [2] Belkin, M., Niyogi, P., Sindhwani, V.: Manifold regularization: a geometric framework for learning from Labeled and Unlabeled Examples. *Journal of Machine Learning Research* 7, 2399–2434 (2006)
- [3] Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H.: Greedy layer-wise training of deep networks. In: *Advances in Neural Information Processing Systems*, NIPS 19 (2007)
- [4] Bromley, J., Bentz, J.W., Bottou, L., Guyon, I., LeCun, Y., Moore, C., Sackinger, E., Shah, R.: Signature verification using a siamese time delay neural network. *International Journal of Pattern Recognition and Artificial Intelligence* 7(4) (August 1993)
- [5] Caputo, B., Hornegger, J., Paulus, D., Niemann, H.: A spin-glass markov random field for 3-d object recognition. Technical Report LME-TR-2002-01, Institut fur Informatik, Universitat Erlangen Nurnberg (2002)

- [6] Caruana, R.: Multitask Learning. *Machine Learning* 28(1), 41–75 (1997)
- [7] Chapelle, O., Schölkopf, B., Zien, A.: *Semi-Supervised Learning*. Adaptive computation and machine learning. MIT Press, Cambridge (2006)
- [8] Chapelle, O., Weston, J., Schölkopf, B.: Cluster kernels for semi-supervised learning. In: Becker, S., Thrun, S., Obermayer, K. (eds.) NIPS, vol. 15, pp. 585–592. MIT Press, Cambridge (2003)
- [9] Chapelle, O., Zien, A.: Semi-supervised classification by low density separation. In: International Conference on Artificial Intelligence and Statistics (AISTATS), pp. 57–64 (January 2005)
- [10] Collobert, R., Sinz, F., Weston, J., Bottou, L.: Large scale transductive svms. *Journal of Machine Learning Research* 7, 1687–1712 (2006)
- [11] Collobert, R., Weston, J.: Fast semantic extraction using a novel neural network architecture. In: Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics, pp. 25–32 (2007)
- [12] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. *The Journal of Machine Learning Research* 12, 2493–2537 (2011)
- [13] Hadsell, R., Chopra, S., LeCun, Y.: Dimensionality reduction by learning an invariant mapping. In: Proc. Computer Vision and Pattern Recognition Conference (CVPR 2006). IEEE Press (2006)
- [14] Hinton, G.E., Osindero, S., Teh, Y.: A fast learning algorithm for deep belief nets. *Neural Comp.* 18(7), 1527–1554 (2006)
- [15] Karlen, M., Weston, J., Erkan, A., Collobert, R.: Large scale manifold transduction. In: Proceedings of the 25th International Conference on Machine Learning, pp. 448–455. ACM (2008)
- [16] Kruskal, J.B.: Multidimensional scaling by optimizing goodness of fit to a non-metric hypothesis. *Psychometrika* 29(1), 1–27 (1964)
- [17] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86(11) (1998)
- [18] Mobahi, H., Collobert, R., Weston, J.: Deep learning from temporal coherence in video. In: Proceedings of the 26th Annual International Conference on Machine Learning, pp. 737–744. ACM (2009)
- [19] Nene, S.A., Nayar, S.K., Murase, H.: Columbia object image library (coil-100). Technical Report CU-CS-006-96 (February 1996)
- [20] Pradhan, S., Ward, W., Hacioglu, K., Martin, J., Jurafsky, D.: Shallow semantic parsing using support vector machines. In: Proceedings of HLT/NAACL 2004 (2004)
- [21] Ranzato, M., Huang, F., Boureau, Y., LeCun, Y.: Unsupervised learning of invariant feature hierarchies with applications to object recognition. In: Proc. Computer Vision and Pattern Recognition Conference (CVPR 2007). IEEE Press (2007)
- [22] Roobaert, D., Van Hulle, M.: View-based 3d object recognition with support vector machines. In: IEEE International Workshop on Neural Networks for Signal Processing, pp. 77–84 (1999)
- [23] Salakhutdinov, R., Hinton, G.: Learning a Nonlinear Embedding by Preserving Class Neighbourhood Structure. In: International Conference on Artificial Intelligence and Statistics, AISTATS (2007)
- [24] Sindhwani, V., Niyogi, P., Belkin, M.: Beyond the point cloud: from transductive to semi-supervised learning. In: International Conference on Machine Learning, ICML (2005)
- [25] Tenenbaum, J.B., de Silva, V., Langford, J.C.: A global geometric framework for nonlinear dimensionality reduction. *Science* 290(5500), 2319–2323 (2000)

- [26] Vapnik, V.: Statistical Learning Theory. John Wiley and Sons, New York (1998)
- [27] Wersing, H., Körner, E.: Learning optimized features for hierarchical models of invariant recognition. *Neural Computation* 15(7), 1559–1599 (2003)
- [28] Weston, J., Ratle, F., Collobert, R.: Deep learning via semi-supervised embedding. In: Proceedings of the 25th International Conference on Machine Learning, pp. 1168–1175. ACM (2008)
- [29] Williams, C.K.I.: On a connection between kernel PCA and metric multidimensional scaling. In: Advances in Neural Information Processing Systems, NIPS 13 (2001)
- [30] Zhu, X., Ghahramani, Z.: Learning from labeled and unlabeled data with label propagation. Technical Report CMU-CALD-02-107, Carnegie Mellon University (2002)

Identifying Dynamical Systems for Forecasting and Control

Preface

Identifying dynamical systems from data is a promising approach to data forecasting and optimal control. Data forecasting is an essential component of rational decision making in quantitative finance, marketing and planning. Optimal control systems, that is, systems that can sense the environment and react appropriately, enable the design of cost efficient gas turbines, smart grids and human-machine interfaces.

A successful architecture for the task of modeling a dynamical system is the recurrent neural network (RNN). The state of the dynamical system is represented by the set of units that compose the network and the transition between two consecutive states is determined by the recurrent connection between these units. The network can be trained with backpropagation through time, that is, standard backpropagation on the RNN unfolded in time. Recurrent neural networks are notoriously difficult to train, specially, when the neural network has to model long-term dependencies. Indeed, the local learning signal is of little use when local variations (high frequency components of the time series) do not reflect global trends (low frequency components).

This state of affairs has led many to seek alternatives to backpropagation through time. A radical departure from backpropagation is the *Echo State Network* [2]. The idea behind echo state networks is simple: (1) Create a huge neural network with *random* recurrent connections and (2) fit a linear model between the activations of the network and the time series to predict. The huge random RNN is called a “reservoir” and implements an overcomplete set of nonlinear primitives, only a subset of which, are useful in order to model the time series to predict. Tuning the reservoir to produce the most task-relevant primitives requires some practical experience. Best practices for tuning echo state networks are described in Chapter 27 [3].

An alternative approach for overcoming the inherent difficulties of backpropagation is to pay particular attention to the structure of the RNN. A carefully designed RNN helps error derivatives to flow on larger time scales. Tricks such as overshooting, error correction neural networks or variant-invariant separation are introduced in Chapter 28 [6]. The same type of networks can be applied to the identification of state-action representation for control systems. Chapter 29 [1] shows how to apply recurrent neural networks to the identification of a full-fledged Markov decision process from the observation of an existing control system. This so-called Markov decision process extraction network (MPEN) encourages the emergence of a joint state-action representation that best captures the relevant information for the control task.

Q-learning [5] is a popular reinforcement learning algorithm for control systems. It associates to each state-action pairs a *Q*-value that indicates how close to the goal the action at a given state brings us. *Q*-values are determined dynamically by the *Q*-learning algorithm, as a result of the exploration of the state-action space by the controller. The question remains how can *Q*-values generalize in a continuous state space. Chapter 30 [4] answers this question and provides a practical guide to set up step-by-step neural reinforcement controllers.

Grégoire & Klaus

References

- [1] Duell, S., Udluft, S., Sterzing, V.: Solving Partially Observable Reinforcement Learning Problems with Recurrent Neural Networks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 687–707. Springer, Heidelberg (2012)
- [2] Jaeger, H., Haas, H.: Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication. *Science* 304(5667), 78–80 (2004)
- [3] Lukoševičius, M.: A Practical Guide to Applying Echo State Networks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 659–686. Springer, Heidelberg (2012)
- [4] Riedmiller, M.: 10 Steps and Some Tricks to Set Up Neural Reinforcement Controllers. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 735–757. Springer, Heidelberg (2012)
- [5] Watkins, C.J.C.H.: Learning from Delayed Rewards. Ph.D. thesis, Cambridge University (1989)
- [6] Zimmermann, H.-G., Tietz, C., Grothmann, R.: Forecasting with Recurrent Neural Networks: 12 Tricks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 687–707. Springer, Heidelberg (2012)

A Practical Guide to Applying Echo State Networks

Mantas Lukoševičius

Jacobs University Bremen, Campus Ring 1,
28759 Bremen, Germany
 m.lukosevicius@jacobs-university.de

Abstract. Reservoir computing has emerged in the last decade as an alternative to gradient descent methods for training recurrent neural networks. Echo State Network (ESN) is one of the key reservoir computing “flavors”. While being practical, conceptually simple, and easy to implement, ESNs require some experience and insight to achieve the hailed good performance in many tasks. Here we present practical techniques and recommendations for successfully applying ESNs, as well as some more advanced application-specific modifications.

27.1 Introduction

Training Recurrent Neural Networks (RNNs) is inherently difficult. This (de-) motivates many to avoid them altogether. RNNs, however, represent a very powerful generic tool, integrating both large dynamical memory and highly adaptable computational capabilities. They are the Machine Learning (ML) model most closely resembling biological brains, the substrate of natural intelligence.

Error backpropagation (BP) [40] is to this date one of the most important achievements in artificial neural network training. It has become the standard method to train especially Feed-Forward Neural Networks (FFNNs). Many useful practical aspects of BP are discussed in other chapters of this book and in its previous edition, e.g., [26]. BP methods have also been extended to RNNs [51, 52], but only with a partial success. One of the conceptual limitations of BP methods for RNNs is that bifurcations can make training non-converging [8]. Even when they do converge, this convergence is slow, computationally expensive, and can lead to poor local minima.

Ten years ago an alternative trend of understanding, training, and using RNNs has been proposed with Echo State Networks (ESNs) [16, 21] in ML, and Liquid State Machines (LSMs) [32] in computational neuroscience. It was shown that RNNs often work well enough even without full adaptation of all network weights. In the classical ESN approach the RNN (called *reservoir*) is generated randomly, and only the readout from the reservoir is trained. It should be noted that this basic idea was first clearly spelled out in a neuroscientific model of the corticostriatal processing loop [7]. Perhaps surprisingly this approach yielded excellent performance in many benchmark tasks, e.g., [16, 15, 19, 22, 47, 48].

The trend started by ESNs and LSMs became lately known as Reservoir Computing (RC) [49]. RC is currently a prolific research area, giving important insights into RNNs, procuring practical machine learning tools, as well as enabling computation with non-conventional hardware [31]. RC today subsumes a number of related methods and extensions of the original idea [29], but the original ESN approach still holds its ground for its simplicity and power.

The latest developments in BP for RNNs, second-order gradient descent methods called Hessian-free optimization, presented in [34] and discussed in a chapter [35] of this book, alleviate some of the mentioned shortcomings. In particular, they perform better on problems which require long memory [34]. These are known to be hard for BP RNN training [1], unless networks are specifically designed to deal with them [13]. Structural damping of Hessian-free optimization [34], an online adaptation of the learning process which penalizes big changes in RNN activations, likely tends to drive the learning process away from passing through many bifurcations (that are exactly big changes in activations and can probably be anticipated and avoided to some extent). On a benchmark suite designed to challenge long short-term memory acquisition, ESNs however still outperform Hessian-free trained RNNs [23].

ESNs from their beginning proved to be a highly practical approach to RNN training. It is conceptually simple and computationally inexpensive. It reinvigorated interest in RNNs, by making them accessible to wider audiences. However, the apparent simplicity of ESNs can sometimes be deceptive. Successfully applying ESNs needs some experience. There is a number of things that can be done wrong. In particular, the initial generation of the raw reservoir network is influenced by a handful of global parameters, and these have to be set judiciously. The same, however, can be said about virtually every ML technique. Techniques and recommendations on successfully applying ESNs will be addressed in this work.

We will try to organize the “best practices” of ESNs into a logical order despite the fact that they are often non-sequentially interconnected. We will start with defining the ESN model and the basic learning procedure in Section 27.2. Then we will detail out guidelines on producing good reservoirs in Section 27.3, various aspects of training different types of readouts in Section 27.4, and dealing with output feedback in Section 27.5. We will end with a short summary in Section 27.6.

27.2 The Basic Model

ESNs are applied to supervised temporal ML tasks where for a given training input signal $\mathbf{u}(n) \in \mathbb{R}^{N_u}$ a desired target output signal $\mathbf{y}^{\text{target}}(n) \in \mathbb{R}^{N_y}$ is known. Here $n = 1, \dots, T$ is the discrete time and T is the number of data points in the training dataset. In practice the dataset can consist of multiple sequences of varying lengths, but this does not change the principles. The task is to learn a model with output $\mathbf{y}(n) \in \mathbb{R}^{N_y}$, where $\mathbf{y}(n)$ matches $\mathbf{y}^{\text{target}}(n)$ as well as possible, minimizing an error measure $E(\mathbf{y}, \mathbf{y}^{\text{target}})$, and, more importantly, generalizes well to unseen data. The error measure E is typically a Mean-Square Error (MSE), for example Root-Mean-Square Error (RMSE)

$$E(\mathbf{y}, \mathbf{y}^{\text{target}}) = \frac{1}{N_y} \sum_{i=1}^{N_y} \sqrt{\frac{1}{T} \sum_{n=1}^T (y_i(n) - y_i^{\text{target}}(n))^2}, \quad (27.1)$$

which is also averaged over the N_y dimensions i of the output here.

The RMSE can also be dimension-wise normalized (divided) by the variance of the target $\mathbf{y}^{\text{target}}(n)$, producing a Normalized Root-Mean-Square Error (NRMSE). The NRMSE has an absolute interpretation: it does not depend on the arbitrary scaling of the target $\mathbf{y}^{\text{target}}(n)$ and the value of 1 can be achieved with a simple constant output $\mathbf{y}(n)$ set to the mean value of $\mathbf{y}^{\text{target}}(n)$. This suggests that a reasonable model of a stationary process should achieve the NRMSE accuracy between zero and one.

The normalization and the square root parts are more for human interpretability, as the optimal output $\mathbf{y}^{\text{target}}$ minimizing any MSE is equivalent to the one minimizing (27.1), as long as no additional penalties or weighting are introduced into the equation, such as discussed in Sections 27.4.2 and 27.4.6.

ESNs use an RNN type with leaky-integrated discrete-time continuous-value units. The typical update equations are

$$\tilde{\mathbf{x}}(n) = \tanh(\mathbf{W}^{\text{in}}[1; \mathbf{u}(n)] + \mathbf{W}\mathbf{x}(n-1)), \quad (27.2)$$

$$\mathbf{x}(n) = (1-\alpha)\mathbf{x}(n-1) + \alpha\tilde{\mathbf{x}}(n), \quad (27.3)$$

where $\mathbf{x}(n) \in \mathbb{R}^{N_x}$ is a vector of reservoir neuron activations and $\tilde{\mathbf{x}}(n) \in \mathbb{R}^{N_x}$ is its update, all at time step n , $\tanh(\cdot)$ is applied element-wise, $[; ;]$ stands for a vertical vector (or matrix) concatenation, $\mathbf{W}^{\text{in}} \in \mathbb{R}^{N_x \times (1+N_u)}$ and $\mathbf{W} \in \mathbb{R}^{N_x \times N_x}$ are the input and recurrent weight matrices respectively, and $\alpha \in (0, 1]$ is the leaking rate. Other sigmoid wrappers can be used besides the tanh, which however is the most common choice. The model is also sometimes used without the leaky integration, which is a special case of $\alpha = 1$ and thus $\tilde{\mathbf{x}}(n) \equiv \mathbf{x}(n)$.

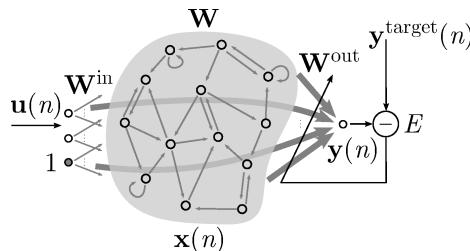


Fig. 27.1. An echo state network

The linear readout layer is defined as

$$\mathbf{y}(n) = \mathbf{W}^{\text{out}}[1; \mathbf{u}(n); \mathbf{x}(n)], \quad (27.4)$$

where $\mathbf{y}(n) \in \mathbb{R}^{N_y}$ is network output, $\mathbf{W}^{\text{out}} \in \mathbb{R}^{N_y \times (1+N_u+N_x)}$ the output weight matrix, and $[.;.;.]$ again stands for a vertical vector (or matrix) concatenation. An additional nonlinearity can be applied to $\mathbf{y}(n)$ in (27.4), as well as feedback connections \mathbf{W}^{fb} from $\mathbf{y}(n-1)$ to $\tilde{\mathbf{x}}(n)$ in (27.2). A graphical representation of an ESN illustrating our notation and the idea for training is depicted in Figure 27.1.

The original method of RC introduced with ESNs [16] was to:

1. generate a large random reservoir RNN ($\mathbf{W}^{\text{in}}, \mathbf{W}, \alpha$);
2. run it using the training input $\mathbf{u}(n)$ and collect the corresponding reservoir activation states $\mathbf{x}(n)$;
3. compute the linear readout weights \mathbf{W}^{out} from the reservoir using linear regression, minimizing the MSE between $\mathbf{y}(n)$ and $\mathbf{y}^{\text{target}}(n)$;
4. use the trained network on new input data $\mathbf{u}(n)$ computing $\mathbf{y}(n)$ by employing the trained output weights \mathbf{W}^{out} .

In subsequent sections we will delve deeper into the hidden intricacies of this procedure which appears so simple on the surface, and spell out practical hints for the concrete design choices that wait on the way. More specifically, Step 1 is elaborated on in Section 27.3; Step 2 is done by Equations (27.2) and (27.3), with initialization discussed in Section 27.4.5; Step 3 is formally defined and options explained in Section 27.4 with additional options for some particular applications in Section 27.5; and Step 3 is again performed by Equations (27.2), (27.3), and (27.4).

27.3 Producing a Reservoir

For producing a good reservoir it is important to understand what function it is serving.

27.3.1 Function of the Reservoir

In practice it is important to keep in mind that the reservoir acts **(i)** as a nonlinear expansion and **(ii)** as a memory of input $\mathbf{u}(n)$ at the same time.

There is a parallel between RC and kernel methods in ML. The reservoir can be seen as (i) a nonlinear high-dimensional expansion $\mathbf{x}(n)$ of the input signal $\mathbf{u}(n)$. For classification tasks, input data $\mathbf{u}(n)$ which are not linearly separable in the original space \mathbb{R}^{N_u} , often become so in the expanded space \mathbb{R}^{N_x} of $\mathbf{x}(n)$, where they are separated by \mathbf{W}^{out} . In fact, employing the “kernel trick” to integrate over all possible reservoirs is also possible in the context of RC, even though not really practical [12].

At the same time, (ii) the reservoir serves as a memory, providing temporal context. This is a crucial reason for using RNNs in the first place. In the tasks where memory is not necessary, non-temporal ML techniques implementing functional mappings from current input to current output should be used.

Both aspects (i) and (ii) combined, the reservoir, being an input-driven dynamical system, should provide a rich and relevant enough signal space in $\mathbf{x}(n)$, such that the desired $\mathbf{y}^{\text{target}}(n)$ could be obtained by linear combination from it. There is however some trade-off between (i) and (ii) when setting the parameters of the reservoir [50], which we will explain in more detail.

27.3.2 Global Parameters of the Reservoir

Given the RNN model (27.2), (27.3), the reservoir is defined by the tuple $(\mathbf{W}^{\text{in}}, \mathbf{W}, \alpha)$. The input and recurrent connection matrices \mathbf{W}^{in} and \mathbf{W} are generated randomly according to some parameters discussed later and the leaking rate α is selected as a free parameter itself.

In analogy to other ML, and especially NN, approaches, what we call “parameters” here could as well be called “meta-parameters” or “hyper-parameters”, as they are not concrete connection weights but parameters governing their distributions. We will call them “global parameters” to better reflect their nature, or simply “parameters” for brevity.

The defining global parameters of the reservoir are: the size N_x , sparsity, distribution of nonzero elements, and spectral radius of \mathbf{W} ; scaling(-s) of \mathbf{W}^{in} ; and the leaking rate α . We will now proceed in this order to give more details on each of these design choices and intuitions on how to make them. Then, in Section 27.3.3, we will summarize by advising how to prioritize these global parameters and tune the really important, or rather task-specific, ones in a principled way.

Size of the Reservoir. One obviously crucial parameter of the model (27.2), (27.3) is N_x , the number of units in the reservoir.

The general wisdom is that the bigger the reservoir, the better the obtainable performance, *provided appropriate regularization measures are taken against overfitting* (see Section 27.4.1). Since training and running an ESN is computationally cheap compared to other RNN approaches, reservoir sizes of order 10^4 are not uncommon [47]. The bigger the space of reservoir signals $\mathbf{x}(n)$, the easier it is to find a linear combination of the signals to approximate $\mathbf{y}^{\text{target}}(n)$. In our experience the reservoir can be too big only when the task is trivial and there is not enough data available $T < 1 + N_u + N_x$.

For challenging tasks use as big a reservoir as you can afford computationally.

That being said, computational trade-offs are important. In academic settings, when comparing different approaches instead of going for the best possible performance, authors often limit their reservoir sizes for convenience and compatibility of results. Even when going for the best performance, starting with the biggest possible reservoir from the beginning is cumbersome.

Select global parameters with smaller reservoirs, then scale to bigger ones.

The tuning of global parameters (described below) often needs multiple trials, thus each should not consume too much time. Good parameters are usually transferable to bigger reservoirs, but some trials with big reservoirs can also be done to confirm this.

A lower bound for the reservoir size N_x can roughly be estimated by considering the number of independent real values that the reservoir must remember from the input to successfully accomplish the task. The maximal number of stored values, called memory capacity, in ESN can not exceed N_x [17].

N_x should be at least equal to the estimate of independent real values the reservoir has to remember from the input to solve its task.

For i.i.d. inputs $\mathbf{u}(n)$, this estimate is N_u times a rough estimate of how many time steps the inputs should be remembered to solve the task. While the result in [17] is precise for i.i.d. inputs, in practice there are often temporal and inter-channel correlations in $\mathbf{u}(n)$, that make it somewhat “compressible”. Also, the shapes of the “forgetting curves” of the reservoirs are typically not rectangular (depend on other parameters), i.e., the forgetting is not instantaneous but gradual. As a result, reservoir can often make do with smaller sizes.

Sparsity of the Reservoir. In the original ESN publications it is recommended to make the reservoir connections sparse, i.e., make most of elements in \mathbf{W}^{in} equal to zero. In our practical experience also sparse connections tend to give a slightly better performance. In general, sparsity of the reservoir does not affect the performance much and this parameter has a low priority to be optimized. However, sparsity enables fast reservoir updates if sparse matrix representations are used.

Connect each reservoir node to a small fixed number of other nodes (e.g., 10) on average, irrespective of the reservoir size. Exploit this reservoir sparsity to speedup computation.

If regardless of reservoir size, a fixed fanout number is chosen, the computational cost of network state updates grows only linearly with the network size instead of quadratically. This greatly reduces the cost of running big reservoirs. The computational savings require virtually no additional effort when the programming environment supports efficient representation and operations with sparse matrices, which many do.

Distribution of Nonzero Elements. The matrix \mathbf{W} is typically generated sparse, with nonzero elements having either a symmetrical uniform, discrete bi-valued, or normal distribution centered around zero. Different authors prefer different distributions. We usually prefer a uniform distribution for its continuity of values and boundedness. Gaussian distributions are also popular. Both distributions give virtually the same performance which depends on the other parameters discussed here. The discrete bi-valued distribution tends to give a slightly less rich signal space (there is a non-zero probability of identical neurons), but might make analysis of what is happening in the reservoir easier. The width of the distributions does not matter, as it is reset in a way explained in the next section.

The input matrix \mathbf{W}^{in} is usually generated according to the same type of distribution as \mathbf{W} , but typically dense.

Spectral Radius. One of the most central global parameters of an ESN is spectral radius of the reservoir connection matrix \mathbf{W} , i.e., the maximal absolute eigenvalue of this matrix. It scales the matrix \mathbf{W} , or viewed alternatively, scales the width of the distribution of its nonzero elements.

Typically a random sparse \mathbf{W} is generated; its spectral radius $\rho(\mathbf{W})$ is computed; then \mathbf{W} is divided by $\rho(\mathbf{W})$ to yield a matrix with a unit spectral radius; this is then conveniently scaled with the ultimate spectral radius to be determined in a tuning procedure.

For the ESN approach to work, the reservoir should satisfy the so-called echo state property: the state of the reservoir $\mathbf{x}(n)$ should be uniquely defined by the fading history of the input $\mathbf{u}(n)$ [16]. In other words, for a long enough input $\mathbf{u}(n)$, the reservoir state $\mathbf{x}(n)$ should not depend on the initial conditions that were before the input.

Large $\rho(\mathbf{W})$ values can lead to reservoirs hosting multiple fixed point, periodic, or even chaotic (when sufficient nonlinearity in the reservoir is reached) spontaneous attractor modes, violating the echo state property.

$\rho(\mathbf{W}) < 1$ ensures echo state property in most situations.

Even though it is possible to violate the echo state property even with $\rho(\mathbf{W}) < 1$, this is unlikely to happen in practice. More importantly, the echo state property often holds for $\rho(\mathbf{W}) \geq 1$ for nonzero inputs $\mathbf{u}(n)$. This can be explained by the strong $\mathbf{u}(n)$ pushing activations of the neurons away from 0 where their $\tanh()$ nonlinearities have a unitary slope to regions where this slope is smaller, thus reducing the gains of the neurons and the effective strength of feedback connections. Intuitively speaking, due to activation-squashing nonlinearities, strong inputs “squeeze out” the autonomous activity from the reservoir activations. This means, that for nonzero $\mathbf{u}(n)$ $\rho(\mathbf{W}) < 1$ is not a necessary condition for the echo state property and optimal $\rho(\mathbf{W})$ values can sometimes be significantly greater than 1.

In practice $\rho(\mathbf{W})$ should be selected to maximize the performance, with the value 1 serving as an initial reference point.

As a guiding principle, $\rho(\mathbf{W})$ should be set greater for tasks where a more extensive history of the input is required to perform it, and smaller for tasks where the current output $\mathbf{y}(n)$ depends more on the recent history of $\mathbf{u}(n)$. The spectral radius determines how fast the influence of an input dies out in a reservoir with time, and how stable the reservoir activations are [50].

The spectral radius should be greater in tasks requiring longer memory of the input.

Input Scaling. The scaling of the input weight matrix \mathbf{W}^{in} is another key parameter to optimize in an ESN. For uniformly distributed \mathbf{W}^{in} we usually define the input scaling a as the range of the interval $[-a; a]$ from which values of \mathbf{W}^{in} are sampled; for normal distributed input weights one may take the standard deviation as a scaling measure.

In order to have a small number of freely adjustable parameters, often all the columns of \mathbf{W}^{in} are scaled together using a single scaling value. However, the scaling of the first column of \mathbf{W}^{in} corresponding to the bias input to the reservoir units in (27.2) can be optimized separately from the rest. If the remaining “active” input channels contribute to the task in very different ways, it is also advised to optimize their scalings separately.

Scale the whole \mathbf{W}^{in} uniformly to have few global parameters in ESN. However, to increase the performance:

- scale the first column of \mathbf{W}^{in} (i.e., the bias inputs) separately;
- scale other columns of \mathbf{W}^{in} separately if channels of $\mathbf{u}(n)$ contribute differently to the task.

This varies the number of free global parameters to set for \mathbf{W}^{in} from 1 up to $N_u + 1$.

It was suggested in the original ESN publications to scale and shift the input data, optimizing the magnitude of both. But the same effect can be achieved by scaling the input weights of the active inputs and the bias, respectively.

Still, input data normalization is advisable for ESNs just as for any other ML approach. This puts each learning task into a more standardized setting. It may be helpful to have the range of the input data values bounded. For example, apply the $\tanh(\cdot)$ squashing to $\mathbf{u}(n)$ if its distribution is unbounded.

Otherwise the outliers can throw the reservoir state $\mathbf{x}(n)$ into some “unfamiliar” regions not well covered by the usual working trajectories of $\mathbf{x}(n)$ for which the global parameters have been optimized or the outputs learned. This can lead to a virtual loss of useful memory (due to saturations in the activation nonlinearities) or unpredictable outputs at these points, respectively.

It is advisable to normalize the data and may help to keep the inputs $\mathbf{u}(n)$ bounded avoiding outliers (e.g., apply $\tanh(\cdot)$ to $\mathbf{u}(n)$ if it is unbounded).

Input scaling determines how nonlinear the reservoir responses are. For very linear tasks \mathbf{W}^{in} should be small, letting units operate around the 0 point where their activation $\tanh(\cdot)$ is virtually linear. For large \mathbf{W}^{in} , the units will get easily saturated close to their 1 and -1 values, acting in a more nonlinear, binary switching manner. While $\rho(\mathbf{W})$ also affects the nonlinearity, the reservoir activations become unstable when increasing $\rho(\mathbf{W})$, as explained in Section 27.3.2, before it can make the reservoir highly nonlinear.

The amount of nonlinearity the task requires is not easy to judge. Finding a proper setting benefits from experience and intuitive insight into nonlinear dynamics. But also the masters of RC (if there are such) use trial and error to tune this characteristic.

Looking at (27.2), it is clear that the scaling of \mathbf{W}^{in} , together with the scaling of \mathbf{W} (i.e., $\rho(\mathbf{W})$) determines the proportion of how much the current state $\mathbf{x}(n)$ depends on the current input $\mathbf{u}(n)$ and how much on the previous state $\mathbf{x}(n-1)$, respectively. The respective sizes N_u and N_x should also be taken into account.

The input scaling regulates:

- the amount of nonlinearity of the reservoir representation $\mathbf{x}(n)$ (also increasing with $\rho(\mathbf{W})$);
- the relative effect of the current input on $\mathbf{x}(n)$ as opposed to the history (in proportion to $\rho(\mathbf{W})$).

It has been empirically observed that the representation of the different principle components of $\mathbf{u}(n)$ in $\mathbf{x}(n)$ is roughly proportional to the square root of their magnitudes in $\mathbf{u}(n)$ [11]. In other words, the reservoir tends to flatten the spectrum of principal components of $\mathbf{u}(n)$ in $\mathbf{x}(n)$ – something to keep in mind when choosing the right representation or preprocessing of the data. For example, if smaller principal components carry no useful information it might be helpful to remove them from the data by Principal Component Analysis (PCA) before feeding them to a reservoir, otherwise they will get relatively amplified there.

Leaking Rate. The leaking rate α of the reservoir nodes in (27.3) can be regarded as the speed of the reservoir update dynamics discretized in time. We

can describe the reservoir update dynamics in continuous time as an Ordinary Differential Equation (ODE)

$$\dot{\mathbf{x}} = -\mathbf{x} + \tanh(\mathbf{W}^{\text{in}}[1; \mathbf{u}] + \mathbf{W}\mathbf{x}). \quad (27.5)$$

If we make an Euler's discretization of this ODE (27.5) in time, taking

$$\frac{\Delta \mathbf{x}}{\Delta t} = \frac{\mathbf{x}(n+1) - \mathbf{x}(n)}{\Delta t} \approx \dot{\mathbf{x}}, \quad (27.6)$$

we arrive at exactly (up to some time indexing conventions) the discrete time equations (27.2)(27.3) with α taking the place of the sampling interval Δt . Thus α can be regarded as the time interval in the continuous world between two consecutive time steps in the discrete realization. Also, empirically the effect of setting α is comparable to that of re-sampling $\mathbf{u}(n)$ and $\mathbf{y}^{\text{target}}(n)$ when the signals are slow [27, 41]. The leaking rate α can even be adapted online to deal with time wrapping of the signals [27, 22]. Equivalently, α can be introduced as a time constant in (27.5), if keeping $\Delta t \equiv 1$.

While there are some slight variations alternative to those in (27.3), of how to do leaky integration (e.g., [22]), the version (27.3) has emerged as preferred, because it guarantees that $\mathbf{x}(n)$ never goes outside the $(-1, 1)$ interval.

Set the leaking rate α in (27.3) to match the speed of the dynamics of $\mathbf{u}(n)$ and/or $\mathbf{y}^{\text{target}}(n)$.

This can, again, be difficult and subjective to determine in some cases. Especially when the timescales of $\mathbf{u}(n)$ and $\mathbf{y}^{\text{target}}(n)$ are quite different. This is one more of the global parameters to be tuned by trial and error.

When the task requires modeling the time series producing dynamical system on multiple time scales, it might be useful to set different leaking rates to different units (making α a vector $\alpha \in \mathbb{R}^{N_x}$) [43], with a possible downside of having more parameters to optimize.

Alternatively, the leaky integration (27.3) can be seen as a simple digital low-pass filter, also known as exponential smoothing, applied to every node. Some contributions even suggest applying more powerful filters for this purpose [53, 14].

In some cases setting a small α , and thus inducing slow dynamics of $\mathbf{x}(n)$, can dramatically increase the duration of the short-term memory in ESN [23].

27.3.3 Practical Approach to Reservoir Production

Prioritizing Parameters. While all the ESN reservoir parameters discussed in Section 27.3.2 have their guiding intuitions in setting them, fixing some of them is more straightforward than others.

The main three parameters to optimize in an ESN reservoir are:

- input scaling(-s);
- spectral radius;
- leaking rate(-s).

These three parameters, discussed in Sections 27.3.2, 27.3.2, and 27.3.2 respectively, are very important for a good performance and are quite task-specific.

The reservoir size N_x almost comes as an external restriction (Section 27.3.2), and the rest of the parameters can be set to reasonable default values: reservoir sparseness (Section 27.3.2), weight distribution (Section 27.3.2), or details of the model (27.2)(27.3). It is still worth investigating several options for them, as a lower priority.

As explained before, the performance can also be additionally improved in many cases by “splitting” a single parameter into several. Setting different scalings to the columns of \mathbf{W}^{in} (corresponding to the bias input and possibly to different dimensions of input if they are of different nature) can go a long way. Also, setting leaking rates α differently for different units (e.g., by splitting them to several sub-populations with constant value) can help a lot in multi-timescale tasks.

Setup for Parameter Selection. One of the main advantages of ESNs is that learning the outputs is fast. This should be exploited in evaluating how good a reservoir generated by a particular set of parameters is.

The most pragmatic way to evaluate a reservoir is to train the output (27.4) and measure its error.

Either validation or training error can be used. Validation is, of course, preferred if there is a danger of overfitting. Training error has the advantage of using less data and in some cases no need to rerun a trained network with it. If a validation data set is necessary for the output training (as explained in Section 27.4), the error on it might be utilized with no additional cost, as a compromise between the training and a yet separate second validation set.

If training of the output and validation is not fast enough, smaller initial reservoirs (as stated in Section 27.3.2), or a reduced representative data set can be used. For the same reason it is often an overkill to use a k -fold cross-validation in global parameter optimization, at least in initial stages, unless the data are really scarce.

It is important to keep in mind that the randomly generated reservoirs even with the same parameters vary slightly in their performance. This variance is ever present but is typically more pronounced with smaller reservoirs than with bigger ones; the random variations inside of a big reservoir tend to “average

out". It is nonetheless important to keep this random fluctuation of performance separate from the one caused by different parameter values.

To eliminate the random fluctuation of performance, keep the random seed fixed and/or average over several reservoir samples.

Fixing a random seed in the programming environment before generating the reservoirs makes the random aspect of the reservoirs identical across trials and thus the experiments deterministically repeatable. Using a single reservoir is faster, but with an obvious danger of below-average performance and/or overfitting the parameters to a particular instance of a randomly generated reservoir: good parameters might not carry over well to a different software implementation, or, e.g., different size of a reservoir.

Manual Parameter Selection. Manual selection of parameters is unavoidable to some extent in virtually all ML approaches. Even when parameters are learned or selected through automated search, it is typically necessary to set meta-parameters (or rather "meta-meta-parameters") for these procedures.

When manually tuning the reservoir parameters, change one parameter at a time.

Changes in several parameters at once often have opposing effects on performance, but it is impossible to tell which contributed what. A reasonable approach is to set a single parameter to a well enough value before starting changing another one, and repeating this until the performance is satisfactory.

It is also advisable to take notes or log the performance automatically for extended optimizations, in order not to "go in circles" when repeating the same parameter values.

An empirical direction of a gradient can be estimated for a parameter, making a small change to it and observing the change in performance. However, the error landscapes are often non-convex and trying distant values of the parameters can sometimes lead to dramatic improvements.

Always plot samples of reservoir activation signals $\mathbf{x}(n)$ to have a feeling of what is happening inside the reservoir.

This may reveal that $\mathbf{x}(n)$ are over-saturated, under-activated, exhibiting autonomous cyclic or chaotic behavior, etc. Overall, plotting information additional to the error rate helps a lot in gaining more insight into how the parameters should be changed.

Typically, good average performance is not found in a very narrow parameter range, thus a very detailed fine-tuning of parameters does not give a significant improvement and is not necessary.

Automated Parameter Selection. Since manual parameter optimization might quickly get tedious, automated approaches are often preferred.

Since ESNs have only a few parameters requiring more careful tuning, *grid search* is probably the most straightforward option. It is easy enough to implement with a few nested loops, and high-level ML programming libraries, such as Oger (mentioned in Section 27.6) in the case of RC, often have ready-made routines for this.

A reasonable approach is to do a coarser grid search over wider parameter intervals to identify promising regions and then do a finer search (smaller steps) in these regions. As mentioned, typically the grid must not be very dense to achieve a good performance.

The best performance being on a boundary of a covered grid is a good indication that the optimal performance might be outside the grid.

In general, meta-parameter or hyper-parameter optimization is a very common topic in many branches of ML and beyond. There are numerous generic optimization methods applicable to this task described in the literature. They are often coping with much larger search spaces than a grid search is effectively capable of, such as random search, or more sophisticated methods trying to model the error landscape (see, e.g., [2]). They are in principle just as well applicable to ESNs with a possibility of also including in the optimization the parameters of second importance.

There is also a way to optimize the global parameters of the reservoir through a gradient descent [22]. It has, however, not been widely applied in the literature.

27.3.4 Pointers to Reservoir Extensions

There are also alternative ways of generating and adapting reservoirs suggested in the literature, including deterministic, e.g., [39], and data-specific, e.g., [36], ones. With a variety of such methods the modern field of RC has evolved from using the initial paradigm of a fixed reservoir and only training a readout from it, to also adapting the reservoir but differently from the readout, using generic, unsupervised, or even supervised methods. In some cases a hardware system is used as a reservoir and thus is predetermined by its specific features. See [29] and updated in Chapter 2 of [30] for a classification and overview. The classical ESN approach described here, however, still holds its ground for its simplicity and performance.

27.4 Training Readouts

27.4.1 Ridge Regression

Since readouts from an ESN are typically linear and feed-forward, the Equation (27.4) can be written in a matrix notation as

$$\mathbf{Y} = \mathbf{W}^{\text{out}} \mathbf{X}, \quad (27.7)$$

where $\mathbf{Y} \in \mathbb{R}^{N_y \times T}$ are all $\mathbf{y}(n)$ and $\mathbf{X} \in \mathbb{R}^{(1+N_u+N_x) \times T}$ are all $[1; \mathbf{u}(n); \mathbf{x}(n)]$ produced by presenting the reservoir with $\mathbf{u}(n)$, both collected into respective matrices by concatenating the column-vectors horizontally over the training period $n = 1, \dots, T$. We use here a single \mathbf{X} instead of $[1; \mathbf{U}; \mathbf{X}]$ for notational brevity.

Finding the optimal weights \mathbf{W}^{out} that minimize the squared error between $\mathbf{y}(n)$ and $\mathbf{y}^{\text{target}}(n)$ amounts to solving a typically overdetermined system of linear equations

$$\mathbf{Y}^{\text{target}} = \mathbf{W}^{\text{out}} \mathbf{X}, \quad (27.8)$$

where $\mathbf{Y}^{\text{target}} \in \mathbb{R}^{N_y \times T}$ are all $\mathbf{y}(n)$, with respect to \mathbf{W}^{out} in a least-square sense – i.e., a case of linear regression. In this context \mathbf{X} can be called the *design matrix*. The system is overdetermined, because typically $T \gg 1 + N_u + N_x$.

There are standard well-known ways to solve (27.8), we will discuss a couple of good choices here.

Probably the most universal and stable solution to (27.8) in this context is ridge regression, also known as regression with Tikhonov regularization:

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}} \mathbf{X}^T \left(\mathbf{X} \mathbf{X}^T + \beta \mathbf{I} \right)^{-1}, \quad (27.9)$$

where β is a regularization coefficient explained in Section 27.4.2, and \mathbf{I} is the identity matrix.

The most generally recommended way to learn linear output weights from an ESN is ridge regression (27.9)

We start with this method because it should be the first choice, even though it is not the most trivial one. We will explain different aspects of this method in the coming sections together with reasons for why it should be preferred and alternatives that in some cases can be advantageous.

27.4.2 Regularization

To assess the quality of the solution produced by training, it is advisable to monitor the actual obtained output weights \mathbf{W}^{out} . Large weights indicate that \mathbf{W}^{out} exploits and amplifies tiny differences among the dimensions of $\mathbf{x}(n)$, and can be very sensitive to deviations from the exact conditions in which the network has been trained. This is a big problem in the setups where the network receives its output as the next input. The slight deviation of the output from the expected value quickly escalates in subsequent time steps. Ways of dealing with such setup are explained in Section 27.5.

Extremely large \mathbf{W}^{out} values may be an indication of a very sensitive and unstable solution.

To counteract this effect is exactly what the regularization part $\beta \mathbf{I}$ in the ridge regression (27.9) is for. Instead of just minimizing RMSE (27.1), ridge regression (27.9) solves

$$\mathbf{W}^{\text{out}} = \arg \min_{\mathbf{W}^{\text{out}}} \frac{1}{N_y} \sum_{i=1}^{N_y} \left(\sum_{n=1}^T (y_i(n) - y_i^{\text{target}}(n))^2 + \beta \|\mathbf{w}_i^{\text{out}}\|^2 \right), \quad (27.10)$$

where $\mathbf{w}_i^{\text{out}}$ is the i th row of \mathbf{W}^{out} and $\|\cdot\|$ stands for the Euclidean norm. The objective function in (27.10) adds a regularization, or weight decay, term $\beta \|\mathbf{w}_i^{\text{out}}\|^2$ penalizing large sizes of \mathbf{W}^{out} to the square error between $\mathbf{y}(n)$ and $\mathbf{y}^{\text{target}}(n)$. This is a sum of two objectives, a compromise between having a small training error and small output weights. The relative “importance” between these two objectives is controlled by the regularization parameter β .

Use regularization (e.g., (27.9)) whenever there is a danger of overfitting or feedback instability.

In (27.9) the optimal regularization coefficient β depends on the concrete instantiation of the ESN. It should be selected individually for a concrete reservoir based on validation data.

Select β for a concrete ESN using validation, without rerunning the reservoir through the training data.

There is no need to rerun the model through the data with every value β , because none of the other variables in (27.9) are affected by its changes. Memory permitting, there is also no need to rerun the model with the (small) validation dataset, if you can store \mathbf{X} for it and compute the validation output by (27.7). This makes testing β values computationally much less expensive than testing the reservoir parameters explained in Section 27.3.

The optimal values of β can vary by many magnitudes of size, depending on the exact instance of the reservoir and length of the training data. If doing a simple exhaustive search, it is advisable to search on a logarithmic grid.

Setting β to zero removes the regularization: the objective function in (27.10) becomes equivalent to RMSE (27.1), making the ridge regression a generalization of a regular linear regression. The solution (27.9) with $\beta = 0$ becomes

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}} \mathbf{X}^T \left(\mathbf{X} \mathbf{X}^T \right)^{-1}, \quad (27.11)$$

known as normal equations method for solving linear regression (27.8). In practice, however, setting $\beta = 0$ often leads to numerical instabilities when inverting $(\mathbf{X} \mathbf{X}^T)$ in (27.11). This too recommends using a logarithmic scale for selecting β where it never goes to zero. The problem can in some cases also be alleviated by using a pseudoinverse instead of the real inverse in (27.11).

A Gaussian process interpretation of the linear readout gives an alternative criterion for setting β directly [4].

A similar regularization effect to Tikhonov (27.9) can be achieved by adding scaled white noise to $\mathbf{x}(n)$ in (27.3) – a method that predates ridge regression in ESNs [16]. Like in ridge regression, i.i.d. noise emphasizes the diagonal of $(\mathbf{X}\mathbf{X}^T)$. The advantage is that it is also propagated through \mathbf{W} in (27.2), modeling better the effects of noisy signals in the reservoir. The output learns to recover from perturbed signals, making the model more stable with feedback loops (Section 27.5). The downside of this noise immunization is that the model needs to be rerun with each value of the noise scaling.

27.4.3 Large Datasets

Ridge regression (27.9) (or the Wiener-Hopf solution (27.11) as a special case) also allows a one-shot training with virtually unlimited amounts of data.

Notice that the dimensions of the matrices $(\mathbf{Y}^{\text{target}}\mathbf{X}^T) \in \mathbb{R}^{N_y \times N_x}$ and $(\mathbf{X}\mathbf{X}^T) \in \mathbb{R}^{N_x \times N_x}$ do not depend on the length T of the training sequence in their sizes. The two matrices can be updated by simply adding the corresponding results from the newly incoming data. This one-shot training approach in principle works with an unlimited amount of data – neither complexity of working memory, nor time of the training procedure (27.9) itself depend on the length of data T .

With large datasets collect the matrices $(\mathbf{Y}^{\text{target}}\mathbf{X}^T)$
and $(\mathbf{X}\mathbf{X}^T)$ incrementally for (27.9).

The eventual limitation of the straightforward summation comes from the finite precision of floating point numbers – adding large (like in the so-far accumulated matrix) and small (like the next update) numbers becomes inaccurate. A better summation scheme, such as a hierarchical multi-stage summation where the two added values are always of similar magnitude (e.g., coming from the same amount of time steps), or Kahan summation [24] that compensates for the accumulating errors, should be used instead.

With very large datasets, a more accurate summation
scheme should be used for accumulating $(\mathbf{Y}^{\text{target}}\mathbf{X}^T)$
and $(\mathbf{X}\mathbf{X}^T)$.

Using extended precision numbers here could also help, as well as in other calculations.

27.4.4 Direct Pseudoinverse Solution

A straightforward solution to (27.8) is

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}} \mathbf{X}^+, \quad (27.12)$$

where \mathbf{X}^+ is the Moore-Penrose pseudoinverse of \mathbf{X} . If $(\mathbf{X}\mathbf{X}^T)$ is invertible the (27.12) in essence becomes equivalent to (27.11), but works even when it is not. The direct pseudoinverse calculation typically exhibits high numerical stability. As a downside, it is expensive memory-wise for large design matrices \mathbf{X} , thereby limiting the size of the reservoir N_x and/or the number of training samples T . Since there is virtually no regularization, the system of linear equations (27.8) should be well overdetermined, i.e., $1 + N_u + N_x \ll T$. In other words, the task should be difficult relatively to the capacity of the reservoir so that overfitting does not happen.

Use direct pseudoinverse (27.12) to train ESNs with high precision and little regularization when memory and run time permit.

Many modern programming libraries dealing with linear algebra have implementations of a matrix pseudoinverse, which can be used “off the shelf”. However, implementations vary in their precision, computational efficiency, and numerical stability.

For high precision tasks, check whether the regression $(\mathbf{Y}^{\text{target}} - \mathbf{W}^{\text{out}} \mathbf{X}) \mathbf{X}^+$ on the error $\mathbf{Y}^{\text{target}} - \mathbf{W}^{\text{out}} \mathbf{X}$ is actually all $= \mathbf{0}$, and add it to \mathbf{W}^{out} if it is not.

This computational trick should not work in theory (the regression on the error should be equal to zero), but sometimes does work in practice in Matlab [28], possibly because of some internal optimizations.

Some high level linear algebra libraries have ready-made subroutines for doing regression, i.e., solving linear least-squares as in (27.8), where the exact methods are not made explicit and can internally be chosen depending on the conditioning of the problem. The use of them has an obvious disadvantage of lacking the control on the issues discussed here.

A powerful extension of the basic ESN approach is training (very) many (very small) ESNs in parallel and averaging their outputs, which in some cases has drastically improved performance [19, 22]. This might be not true for tasks requiring large memory, where one bigger reservoir may still be better than several smaller ones.

Averaging outputs from multiple reservoirs increases the performance.

27.4.5 Initial Transient

Usually $\mathbf{x}(n)$ data from the beginning of the training run are discarded (i.e., not used for learning \mathbf{W}^{out}) since they are contaminated by initial transients. To keep notation simple let us assume they come before $n = 1$.

For long sequences discard the initial time steps of activations $\mathbf{x}(n)$ for training that are affected by initial transient.

The initial transient is in essence a result of an arbitrary setting of $\mathbf{x}(0)$, which is typically $\mathbf{x}(0) = \mathbf{0}$. This introduces an unnatural starting state which is not normally visited once the network has “warmed up” to the task. The amount of time steps to discard depends on the memory of the network (which in turn depends on reservoir parameters), and typically are in the order of tens or hundreds.

However, if the data consists of multiple short separate sequences (like in sequence classification), “the initial transient” might be the usual working mode of the ESN. In this case discarding the precious (possibly all!) data might be disadvantageous. See Section 27.4.7 for more on this. Note, that you would want to reset the state to some initial (the same) value before each sequence to make the classification of sequences independent.

With multiple sequences of data the time steps on which the learning should be performed should be concatenated in \mathbf{X} and $\mathbf{Y}^{\text{target}}$, the same way as there would only be a single long sequence.

A generalization of discarding data is presented in the next Section 27.4.6.

27.4.6 Regression Weighting

In the regression learning of ESNs it is easy to make some time steps count more than others by weighting the minimized square error differently. For this, the time steps of the square error are weighted with a weight vector $s(n) \in \mathbb{R}$:

$$E(\mathbf{y}, \mathbf{y}^{\text{target}}) = \frac{1}{N_y} \sum_{i=1}^{N_y} \sum_{n=1}^T s(n) (y_i(n) - y_i^{\text{target}}(n))^2. \quad (27.13)$$

This error is minimized with the same learning algorithms, but at each time step n the vectors $[1; \mathbf{u}(n); \mathbf{x}(n)]$ and $\mathbf{y}^{\text{target}}(n)$ are element-wise multiplied with $\sqrt{s(n)}$ before collecting them into \mathbf{X} and $\mathbf{Y}^{\text{target}}$. Higher values of $s(n)$ put more emphasis on minimizing the error between $\mathbf{y}(n)$ and $\mathbf{y}^{\text{target}}(n)$. Putting a weight $s(n)$ on a time step n has the same effect as if $[1; \mathbf{u}(n); \mathbf{x}(n)]$ and $\mathbf{y}^{\text{target}}(n)$ have appeared $s(n)$ times in the training with a regular weight $s(n) = 1$. Setting $s(n) = 0$ is equivalent to discarding the time steps from training altogether.

Use weighting to assign different importance to different time steps when training.

This weighted least squares scheme can be useful in different situations like discarding or weighting down the signals affected by the initial transients, corrupted or missing data, emphasizing the ending of the signal [6], etc. It is fully compatible with ridge regression (27.9) where the weighting (27.13) is applied to the square error part of the objective function in (27.10).

For an even more refined version different channels of $\mathbf{y}(n)$ can be trained separately and with different $s(n)$. The weighting can for example be used to counter an imbalance between positive vs. negative samples in a classification or detection task.

27.4.7 Readouts for Classification

When the task is to classify separate short time series, the training is typically set up such that the output $\mathbf{y}(n)$ has a dimension for every class and $\mathbf{y}^{\text{target}}(n)$ is equal to one in the dimension corresponding to the correct class and zero everywhere else. The model is trained to approximate $\mathbf{y}^{\text{target}}(n)$ and the class for single sequence $\mathbf{u}(n)$ is very often decided by

$$\text{class}(\mathbf{u}(n)) = \arg \max_k \left(\frac{1}{|\tau|} \sum_{n \in \tau} y_k(n) \right) = \arg \max_k ((\Sigma \mathbf{y})_k), \quad (27.14)$$

where $y_k(n)$ is the k th dimension of $\mathbf{y}(n)$ produced by ESN from $\mathbf{u}(n)$, τ is some integration interval (can be the length of the whole sequence $\mathbf{u}(n)$), and $\Sigma \mathbf{y}$ stands for a shorthand notation of $\mathbf{y}(n)$ time-averaged over τ .

There is a better way to do this. Notice, that in this case

$$\Sigma \mathbf{y} = \frac{1}{|\tau|} \sum_{n \in \tau} \mathbf{y}(n) = \frac{1}{|\tau|} \sum_{n \in \tau} \mathbf{W}^{\text{out}}[1; \mathbf{u}(n); \mathbf{x}(n)] = \quad (27.15)$$

$$= \mathbf{W}^{\text{out}} \frac{1}{|\tau|} \sum_{n \in \tau} [1; \mathbf{u}(n); \mathbf{x}(n)] = \mathbf{W}^{\text{out}} \Sigma \mathbf{x}, \quad (27.16)$$

where $\Sigma \mathbf{x}$ is a shorthand for $[1; \mathbf{u}(n); \mathbf{x}(n)]$ time-averaged over τ . The form (27.16) is a more efficient way to compute $\Sigma \mathbf{y}$, since there is only one multiplication with \mathbf{W}^{out} .

More importantly, (27.16) can be used to make training more efficient and powerful. For a given short sequence $\mathbf{u}(n)$, instead of finding \mathbf{W}^{out} that minimizes $E(\mathbf{y}^{\text{target}}(n), \mathbf{y}(n))$ for every $n \in \tau$, it is better to find the one that minimizes the error between the time-averaged values $E(\mathbf{y}^{\text{target}}, \Sigma \mathbf{y})$. In this case $\mathbf{y}(n)$ (which is not actually explicitly computed) is allowed to deviate from $\mathbf{y}^{\text{target}}(n)$ as long as the time-averaged $\Sigma \mathbf{y}$ is close to $\mathbf{y}^{\text{target}}$. Here $\mathbf{y}^{\text{target}} \equiv \Sigma \mathbf{y}^{\text{target}} = \mathbf{y}^{\text{target}}(n) = \text{const}$ for a single short sequence.

To classify sequences, train and use readouts from time-averaged activations $\Sigma \mathbf{x}$ (27.16), instead of $\mathbf{x}(n)$.

Note that weighting is still possible both among the short sequences and inside each sequence over the intervals τ , using weighted average instead of a simple one. Actually, weighting over τ is often recommendable, emphasizing the ending of the sequence where the whole information to make the classification decision has been fed into the reservoir.

To retain information from different times in the short sequence, weighted averages $\Sigma_1\mathbf{x}, \dots, \Sigma_k\mathbf{x}$ over several time intervals τ_1, \dots, τ_k during the short sequence can be computed and concatenated into an extended state $\Sigma_*\mathbf{x} = [\Sigma_1\mathbf{x}; \dots; \Sigma_k\mathbf{x}]$. This extended state $\Sigma_*\mathbf{x}$ can be used instead of $\Sigma\mathbf{x}$ for an even more powerful classification. In this case \mathbf{W}^{out} is also extended to $\mathbf{W}_*^{\text{out}} \in \mathbb{R}^{N_y \times k \cdot (1 + N_u + N_x)}$.

Concatenate weighted time-averages over different intervals to read out from for an even more powerful classification.

Since the short sequences are typically of different lengths (the advantage of using temporal classification methods), the intervals τ_1, \dots, τ_k should be scaled to match the length of each sequence.

The techniques so far described in this section effectively reduce the time series classification to a static data classification problem by reducing the variable-length inputs $\mathbf{u}(n)$ to fixed-size feature vectors $\Sigma_*\mathbf{x} \in \mathbb{R}^{k \cdot (1 + N_u + N_x)}$. There are many powerful machine learning methods available to solve the static classification problem that can be employed at this point, such as logistic regression or maximum margin classifiers. See, e.g., [3] for different options. These methods define the error function differently and offer different, mostly iterative, optimization algorithms.

Different powerful classification methods for static data can be employed as the readout from the time-averaged activations $\Sigma_*\mathbf{x}$.

Among others, the same regression methods as for temporal data can be used to train a linear readout $\mathbf{y} = \mathbf{W}_*^{\text{out}} \Sigma_*\mathbf{x}$ and decide the class by maximum as in (27.14). In this case, for every short sequence $\mathbf{u}(n)$ only one pair of vectors $\mathbf{y}^{\text{target}}$ and $\Sigma_*\mathbf{x}$ is collected into $\mathbf{Y}^{\text{target}}$ and \mathbf{X} respectively for training by (27.9) or (27.12). Since this reduces training data points from the total number of time steps to the number of short sequences, precautions against overfitting should be taken. Such regression training for classification has an advantage of the single-shot closed form solution, however, it is not optimal because it does not directly optimize the correct classification rates.

For temporal pattern recognition tasks in a long sequence (i.e., detection plus classification), $\mathbf{y}^{\text{target}}(n)$ should be designed cleverly. The shapes, durations, and delays of the signals in $\mathbf{y}^{\text{target}}(n)$ indicating patterns in $\mathbf{u}(n)$ are also parameters that have to be optimized; as well as algorithms producing the final recognition (in the form of discrete symbol sequences or annotations) from the continuous

signals $\mathbf{y}(n)$. But this goes beyond the scope of this paper. Alternatively, dynamic programming methods (such as Viterbi algorithm) can be used for trainable recognizers at the output layer, see [10].

27.4.8 Online Learning

Some applications require online model adaptation, e.g., [19]. In such cases the process generating the data is often not assumed to be stationary and is tracked by the constantly adapting model. \mathbf{W}^{out} here acts as an adaptive linear combiner.

The simplest way to train \mathbf{W}^{out} is the method known as the *Least Mean Squares* (LMS) algorithm [9], it has many extensions and modifications. It is a stochastic gradient descent algorithm which at every time step n changes \mathbf{W}^{out} in the direction of minimizing the instantaneous squared error $\|\mathbf{y}^{\text{target}}(n) - \mathbf{y}(n)\|^2$. LMS is a first-order gradient descent method, locally approximating the error surface with a hyperplane. This approximation is poor when curvature of the error surface is very different in different directions, which is signified by large eigenvalue spreads of $\mathbf{X}\mathbf{X}^T$. In such a situation the convergence performance of LMS is unfortunately severely impaired.

An alternative linear readout learning to LMS, known in linear signal processing as the *Recursive Least Squares* (RLS) algorithm, is insensitive to the detrimental effects of eigenvalue spread and boasts a much faster convergence. It explicitly at each time step n minimizes a square error that is exponentially discounted going back in time:

$$E(\mathbf{y}, \mathbf{y}^{\text{target}}, n) = \frac{1}{N_y} \sum_{i=1}^{N_y} \sum_{j=1}^n \lambda^{n-j} (y_i(j) - y_i^{\text{target}}(j))^2, \quad (27.17)$$

where $0 < \lambda \leq 1$ is the error “forgetting” parameter. This weighting is not unlike the one discussed in Section 27.4.6 where $s^{(n)}(j) = \lambda^{n-j}$ at time step n . RLS can be seen as a method for minimizing (27.17) at each time step n similar to Wiener-Hopf (27.11), but optimized by keeping and updating the estimate of $(\mathbf{X}\mathbf{X}^T)^{-1}$ from time $n-1$ instead of recomputing it from scratch. The downside of RLS is it being computationally more expensive (quadratic in number of weights instead of linear like LMS) and notorious for numerical stability issues. Demonstrations of RLS for ESNs are presented in [19, 18]. A careful and comprehensive comparison of variants of RLS as ESN readouts is carried out in a Master’s thesis [25], which may be helpful for practitioners.

The BackPropagation-DeCorrelation (BPDC) [44] and FORCE [46] learning algorithms discussed in Section 27.5.3 are two other powerful methods for online training of single-layer readouts with feedback connections from the reservoirs.

27.4.9 Pointers to Readouts Extensions

There are also alternative ways of training outputs from the reservoirs suggested in the literature, e.g., Gaussian process [4], copula [5], or Support Vector Machine [42] style outputs. See [29] and updated in Chapter 2 of [30] for an overview.

27.5 Dealing with Output Feedbacks

27.5.1 Output Feedbacks

Even if the reservoir is kept fixed, for some tasks the trained readouts are fed back to the reservoir and thus the training process changes its dynamics. In other words, a recurrence exists between the reservoir and the trained readout. Pattern generation is a typical example of such task. This can be realized in two ways. Either by feedback connections $\mathbf{W}^{\text{fb}} \in \mathbb{R}^{N_x \times N_y}$ from the output to the reservoir, replacing (27.2) with

$$\tilde{\mathbf{x}}(n) = \tanh(\mathbf{W}^{\text{in}}[1; \mathbf{u}(n)] + \mathbf{W}\mathbf{x}(n-1) + \mathbf{W}^{\text{fb}}\mathbf{y}(n-1)), \quad (27.18)$$

or by looping the output $\mathbf{y}(n-1)$ as an input $\mathbf{u}(n)$ for the next update step n in (27.2), in effect turning a trained one step predictor ESN into a pattern generator. Note that these two options are equivalent and are just a matter of notation: $\mathbf{u}(n)$ and \mathbf{W}^{in} instead of $\mathbf{y}(n-1)$ and \mathbf{W}^{fb} , respectively. The same principles thus apply to producing \mathbf{W}^{fb} as to \mathbf{W}^{in} . In some cases, however, both external input and output feedback can be present.

This extends the power of RC, because it no longer relies on fixed random input-driven dynamics to construct the output, but the dynamics are adapted to the task. This power has its price, because stability issues arise here.

Use output feedbacks to the reservoir only if they are necessary for the task.

This may include tasks that simply cannot be learned well enough without feedbacks. Feedbacks enable reservoirs to achieve universal computational capabilities [33] and can in practice be beneficial even where they are not an integral part of the task [28].

In order to avoid falling prey to the same difficulties as with full RNN training algorithms, two strategies are used in RC when learning outputs with feedbacks:

- Breaking the feedback loop during the training, Section 27.5.2;
- Adapting \mathbf{W}^{out} online with specialized algorithms in the presence of real feedbacks, Section 27.5.3.

27.5.2 Teacher Forcing

The first strategy is to disengage the recurrent relationship between the reservoir and the readout using *teacher forcing* and treat output learning as a feedforward task. This is done by feeding the desired output $\mathbf{y}^{\text{target}}(n-1)$ through the feedback connections \mathbf{W}^{fb} in (27.18) instead of the real output $\mathbf{y}(n-1)$ while learning (Figure 27.2a). The target signal $\mathbf{y}^{\text{target}}(n)$ “bootstraps” the learning process and if the output is learned with high precision (i.e., $\mathbf{y}(n) \approx \mathbf{y}^{\text{target}}(n)$), the recurrent system runs much in the same way with the real $\mathbf{y}(n)$ in feedbacks

after training as it did with $\mathbf{y}^{\text{target}}(n)$ during training (Figure 27.2b). In a pure pattern generator setup with no additional inputs, $\mathbf{u}(n)$ and \mathbf{W}^{in} may not be present at all – after training the ESN is run to autonomously generate a pattern, using teacher forcing initially to start the pattern. As noted before, this is equivalent to training a one time step predictor without feedbacks \mathbf{W}^{fb} and looping its output $\mathbf{y}(n-1)$ as input $\mathbf{u}(n)$ through \mathbf{W}^{in} .

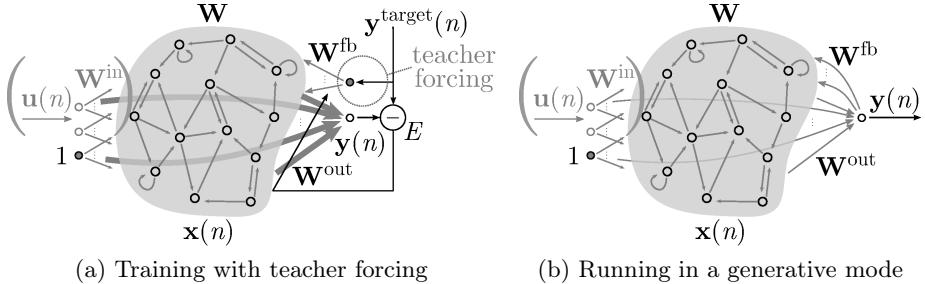


Fig. 27.2. An ESN with output feedbacks trained with teacher forcing

For simple tasks, feed $\mathbf{y}^{\text{target}}(n)$ instead of $\mathbf{y}(n)$ in (27.18) while learning to break the recurrence.

This way \mathbf{W}^{out} can be learned in the same efficient batch mode by linear regression, as explained before.

Teacher forcing applied to speedup error backpropagation RNN training is also discussed in chapter [54] of this book.

There are some caveats here. The approach works very well if the output can be learned precisely [16]. However, if this is not the case, the distorted feedback leads to an even more distorted output and feedback at the next time step, and so on, with the actual generated output $\mathbf{y}(n)$ quickly diverging from the desired $\mathbf{y}^{\text{target}}(n)$. Even with well-learned outputs the dynamical stability of the autonomous running system is often an issue.

Stability with Feedbacks. In both cases regularization of the output by ridge regression or/and noise immunization, as explained in Section 27.4.2, is the key to success.

Regularization by ridge regression or noise is crucial to make teacher-forced feedbacks stable.

Some additional options to the ones in Section 27.4.2 are available here. One is adding scaled noise to the forced teacher signal $\mathbf{y}^{\text{target}}(n)$, emulating an imperfectly learned $\mathbf{y}^{\text{target}}(n)$ by $\mathbf{y}(n)$ and making the network robust to this. In fact, a readout can be trained to ignore some inputs or feedbacks altogether by feeding strong noise into them during training [20].

Another is doing training in several iterations and feeding back the signals that are in between the perfect $\mathbf{y}^{\text{target}}(n)$ and the actual $\mathbf{y}(n)$ obtained from the previous iteration. For example, a one time step prediction of the signal by the ESN (as opposed to running with real feedbacks) can be used as teacher forcer for the next iteration of training [19]. This way the model learns to recover from the directions of deviations from the correct signal that it actually produces, not from just random ones as in the case with noise; while at the same time the teacher signal does not diverge from the target too far.

Another recently proposed option is to also regularize the recurrent connections \mathbf{W} themselves. A one-shot relearning of \mathbf{W} with regularization (similar to ridge regression (27.9) for \mathbf{W}^{out}) to produce the same $\mathbf{x}(n)$, as the one from the initially randomly generated \mathbf{W} , reduces the recurrent connection strengths and helps making the ESN generator more stable [38, 37].

27.5.3 Online Learning with Real Feedbacks

The second strategy to deal with output feedbacks in ESNs is using online (instead of one-shot) learning algorithms to train the outputs \mathbf{W}^{out} while the feedbacks are enabled and feed back the (yet imperfectly) learned output, not the teacher signal. This way the model learns to stabilize itself in the real generative setting.

General purpose online learning algorithms, such as discussed in Section 27.4.8, can be used for this. However, there exist a couple of online RC learning algorithms that are specialized in training outputs with feedbacks, and in fact would not work without them.

BackPropagation-DeCorrelation (BPDC) [44] is such a highly optimized RC online learning algorithm which runs with a linear time complexity in the number of connections. The algorithm is said to be insensitive to reservoir settings and capable of tracking quickly changing signals. As a downside of the latter feature, the trained network forgets the previously seen data and is highly biased by the recent data. Some remedies for reducing this effect are reported in [45].

A recent RC approach named *FORCE* learning uses the RLS (Section 27.4.8) online learning algorithm to vigorously adapt \mathbf{W}^{out} in the presence of the real feedbacks [46]. By the initial fast and strong adaptation of \mathbf{W}^{out} the feedbacks $\mathbf{y}(n)$ are kept close to the desired $\mathbf{y}^{\text{target}}(n)$ already from the beginning of the learning process, similar to teacher forcing. The algorithm benefits from initial spontaneous chaotic activations inside the reservoir which are then subdued by the feedbacks. It appears that *FORCE* learning is well suited to yield very stable and accurate neural pattern generators.

27.6 Summary and Implementations

We have presented many practical aspects for successfully applying ESNs. Some of them are not universal and should be filtered depending on a particular task. They are also not the only possible approaches, and can most likely be improved

upon. They collect, however, the best practices accumulated in the field over the ten years from its start, and should serve well as guiding principles for ESN researchers and practitioners.

Implementing an ESN is relatively straightforward – minimalistic one-page self-contained code examples in several programming languages are available through <http://reservoir-computing.org/software/minimal>. There is also a number of ready-made and expandable software libraries available which incorporate many of the techniques described here. A collection of open source RC toolboxes in different programming languages and varying degrees of sophistication can be found at <http://reservoir-computing.org/software>. The most comprehensive of them is the Oger toolbox in Python <http://reservoir-computing.org/oger>.

The <http://reservoir-computing.org> website is an overall good hub of reservoir computing related resources, where new people can also register and contribute.

Acknowledgments. The author was supported through funds from the European FP7 projects ORGANIC and AMARSi. He would like to specially thank Herbert Jaeger for proof-reading this chapter and valuable suggestions, as well as the anonymous reviewers, and the whole reservoir computing community on whose collective wisdom this chapter is to some extent based.

References

- [1] Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks* 5(2), 157–166 (1994)
- [2] Bergstra, J.S., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: Shawe-Taylor, J., Zemel, R.S., Bartlett, P., Pereira, F.C.N., Weinberger, K.Q. (eds.) *Advances in Neural Information Processing Systems 23 (NIPS 2010)*, pp. 2546–2554 (2011)
- [3] Bishop, C.M.: *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus (2006)
- [4] Chatzis, S.P., Demiris, Y.: Echo state Gaussian process. *IEEE Transactions on Neural Networks* 22(9), 1435–1445 (2011)
- [5] Chatzis, S.P., Demiris, Y.: The copula echo state network. *Pattern Recognition* 45(1), 570–577 (2012)
- [6] Daukantas, S., Lukoševičius, M., Marozas, V., Lukoševičius, A.: Comparison of “black box” and “gray box” methods for lost data reconstruction in multichannel signals. In: Proceedings of the 14th International Conference “Biomedical Engineering”, Kaunas, pp. 135–138 (2010)
- [7] Dominey, P.F., Ramus, F.: Neural network processing of natural language: I. sensitivity to serial, temporal and abstract structure of language in the infant. *Language and Cognitive Processes* 15(1), 87–127 (2000)
- [8] Doya, K.: Bifurcations in the learning of recurrent neural networks. In: Proceedings of IEEE International Symposium on Circuits and Systems, vol. 6, pp. 2777–2780 (1992)

- [9] Farhang-Boroujeny, B.: Adaptive Filters: Theory and Applications. Wiley (1998)
- [10] Graves, A.: Supervised Sequence Labelling with Recurrent Neural Networks. PhD thesis, Technical University Munich, Munich, Germany (2008)
- [11] Hermans, M., Schrauwen, B.: Memory in reservoirs for high dimensional input. In: Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN 2010), pp. 1–7 (2010)
- [12] Hermans, M., Schrauwen, B.: Recurrent kernel machines: Computing with infinite echo state networks. *Neural Computation* 24(1), 104–133 (2012)
- [13] Hochreiter, S., Schmidhuber, J.: Long short-term memory. *Neural Computation* 9(8), 1735–1780 (1997)
- [14] Holzmann, G., Hauser, H.: Echo state networks with filter neurons and a delay and sum readout. *Neural Networks* 23(2), 244–256 (2010)
- [15] Ilies, I., Jaeger, H., Kosuchinas, O., Rincon, M., Sakénas, V., Vaškevičius, N.: Stepping forward through echoes of the past: forecasting with echo state networks. Short report on the winning entry to the NN3 financial forecasting competition (2007),
http://www.neural-forecasting-competition.com/downloads/NN3/methods/27-NN3_Herbert_Jaeger_report.pdf
- [16] Jaeger, H.: The “echo state” approach to analysing and training recurrent neural networks. Technical Report GMD Report 148, German National Research Center for Information Technology (2001)
- [17] Jaeger, H.: Short term memory in echo state networks. Technical Report GMD Report 152, German National Research Center for Information Technology (2002)
- [18] Jaeger, H.: Adaptive nonlinear system identification with echo state networks. In: Advances in Neural Information Processing Systems 15 (NIPS 2002), pp. 593–600. MIT Press, Cambridge (2003)
- [19] Jaeger, H., Haas, H.: Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless communication. *Science* 304(5667), 78–80 (2004)
- [20] Jaeger, H.: Generating exponentially many periodic attractors with linearly growing echo state networks. Technical Report No. 3, Jacobs University Bremen (2006)
- [21] Jaeger, H.: Echo state network. *Scholarpedia* 2(9), 2330 (2007)
- [22] Jaeger, H., Lukoševičius, M., Popovici, D., Siewert, U.: Optimization and applications of echo state networks with leaky-integrator neurons. *Neural Networks* 20(3), 335–352 (2007)
- [23] Jaeger, H.: Long short-term memory in echo state networks: Details of a simulation study. Technical Report No. 27, Jacobs University Bremen (2012)
- [24] Kahan, W.: Pracniques: further remarks on reducing truncation errors. *Communications of the ACM* 8(1), 40 (1965)
- [25] Küçükemre, A.U.: Echo state networks for adaptive filtering. Master’s thesis, University of Applied Sciences Bohn-Rhein-Sieg, Germany (2006),
<http://reservoir-computing.org/publications/2006-echo-state-networks-adaptive-filtering>
- [26] LeCun, Y.A., Bottou, L., Orr, G.B., Müller, K.-R.: Efficient BackProp. In: Orr, G.B., Müller, K.-R. (eds.) NIPS-WS 1996. LNCS, vol. 1524, pp. 9–50. Springer, Heidelberg (1998)
- [27] Lukoševičius, M., Popovici, D., Jaeger, H., Siewert, U.: Time warping invariant echo state networks. Technical Report No. 2, Jacobs University Bremen (May 2006)
- [28] Lukoševičius, M.: Echo state networks with trained feedbacks. Technical Report No. 4, Jacobs University Bremen (February 2007)
- [29] Lukoševičius, M., Jaeger, H.: Reservoir computing approaches to recurrent neural network training. *Computer Science Review* 3(3), 127–149 (2009)

- [30] Lukoševičius, M.: Reservoir Computing and Self-Organized Neural Hierarchies. PhD thesis, Jacobs University Bremen, Bremen, Germany (2011)
- [31] Lukoševičius, M., Jaeger, H., Schrauwen, B.: Reservoir computing trends. KI - Künstliche Intelligenz, pp. 1–7 (2012)
- [32] Maass, W., Natschläger, T., Markram, H.: Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Computation* 14(11), 2531–2560 (2002)
- [33] Maass, W., Joshi, P., Sontag, E.D.: Principles of real-time computing with feedback applied to cortical microcircuit models. In: *Advances in Neural Information Processing Systems 18* (NIPS 2005), pp. 835–842. MIT Press, Cambridge (2006)
- [34] Martens, J., Sutskever, I.: Learning recurrent neural networks with Hessian-free optimization. In: *Proc. 28th Int. Conf. on Machine Learning* (2011)
- [35] Martens, J., Sutskever, I.: Training Deep and Recurrent Networks with Hessian-free Optimization. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) *NN: Tricks of the Trade*, 2nd edn. LNCS, vol. 7700, pp. 479–535. Springer, Heidelberg (2012)
- [36] Ozturk, M.C., Xu, D., Príncipe, J.C.: Analysis and design of echo state networks. *Neural Computation* 19(1), 111–138 (2007)
- [37] Reinhart, F.R., Steil, J.J.: A constrained regularization approach for input-driven recurrent neural networks. *Differential Equations and Dynamical Systems* 19, 27–46 (2011)
- [38] Reinhart, F.R., Steil, J.J.: Reservoir regularization stabilizes learning of echo state networks with output feedback. In: *Proceedings of the 19th European Symposium on Artificial Neural Networks, ESANN 2011* (2011) (in Press)
- [39] Rodan, A., Tiño, P.: Minimum complexity echo state network. *IEEE Transactions on Neural Networks* 22(1), 131–144 (2011)
- [40] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In: *Neurocomputing: Foundations of Research*, pp. 673–695. MIT Press, Cambridge (1988)
- [41] Schrauwen, B., Defour, J., Verstraeten, D., Van Campenhout, J.: The Introduction of Time-Scales in Reservoir Computing, Applied to Isolated Digits Recognition. In: de Sá, J.M., Alexandre, L.A., Duch, W., Mandic, D.P. (eds.) *ICANN 2007*. LNCS, vol. 4668, pp. 471–479. Springer, Heidelberg (2007)
- [42] Shi, Z., Han, M.: Support vector echo-state machine for chaotic time-series prediction. *IEEE Transactions on Neural Networks* 18(2), 359–372 (2007)
- [43] Siewert, U., Wustlich, W.: Echo-state networks with band-pass neurons: towards generic time-scale-independent reservoir structures. Internal status report, PLANET intelligent systems GmbH (2007), <http://reslab.elis.ugent.be/node/112>
- [44] Steil, J.J.: Backpropagation-decorrelation: recurrent learning with $O(N)$ complexity. In: *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN 2004)*, vol. 2, pp. 843–848 (2004)
- [45] Steil, J.J.: Memory in Backpropagation-Decorrelation $O(N)$ Efficient Online Recurrent Learning. In: Duch, W., Kacprzyk, J., Oja, E., Zadrożny, S. (eds.) *ICANN 2005*. LNCS, vol. 3697, pp. 649–654. Springer, Heidelberg (2005)
- [46] Sussillo, D., Abbott, L.F.: Generating coherent patterns of activity from chaotic neural networks. *Neuron* 63(4), 544–557 (2009)
- [47] Trienbach, F., Jalalvand, A., Schrauwen, B., Martens, J.-P.: Phoneme recognition with large hierarchical reservoirs. In: *Advances in Neural Information Processing Systems 23* (NIPS 2010), pp. 2307–2315. MIT Press, Cambridge (2011)
- [48] Verstraeten, D., Schrauwen, B., Stroobandt, D.: Reservoir-based techniques for speech recognition. In: *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN 2006)*, pp. 1050–1053 (2006)

- [49] Verstraeten, D., Schrauwen, B., D'Haene, M., Stroobandt, D.: An experimental unification of reservoir computing methods. *Neural Networks* 20(3), 391–403 (2007)
- [50] Verstraeten, D., Dambre, J., Dutoit, X., Schrauwen, B.: Memory versus non-linearity in reservoirs. In: Proc. Int. Neural Networks (IJCNN) Joint Conf., pp. 1–8 (2010)
- [51] Werbos, P.J.: Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE* 78(10), 1550–1560 (1990)
- [52] Williams, R.J., Zipser, D.: A learning algorithm for continually running fully recurrent neural networks. *Neural Computation* 1, 270–280 (1989)
- [53] Wyffels, F., Schrauwen, B., Verstraeten, D., Stroobandt, D.: Band-pass reservoir computing. In: Hou, Z., Zhang, N. (eds.) *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN 2008)*, Hong Kong, pp. 3204–3209 (2008)
- [54] Zimmermann, H.-G., Tietz, C., Grothmann, R.: Forecasting with Recurrent Neural Networks: 12 Tricks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) *NN: Tricks of the Trade*, 2nd edn. LNCS, vol. 7700, pp. 687–707. Springer, Heidelberg (2012)

Forecasting with Recurrent Neural Networks: 12 Tricks

Hans-Georg Zimmermann, Christoph Tietz, and Ralph Grothmann

Siemens AG, Corporate Technology
 Otto-Hahn-Ring 6, D-81739 Munich, Germany
 Hans_Georg.Zimmermann@siemens.com

Abstract. Recurrent neural networks (RNNs) are typically considered as relatively simple architectures, which come along with complicated learning algorithms. This paper has a different view: We start from the fact that RNNs can model any high dimensional, nonlinear dynamical system. Rather than focusing on learning algorithms, we concentrate on the design of network architectures. Unfolding in time is a well-known example of this modeling philosophy. Here a temporal algorithm is transferred into an architectural framework such that the learning can be performed by an extension of standard error backpropagation.

We introduce *12* tricks that not only provide deeper insights in the functioning of RNNs but also improve the identification of underlying dynamical system from data.

28.1 Introduction

In many business management disciplines, complex planning and decision-making can be supported by quantitative forecast models that take into account a wide range of influencing factors with non-linear cause and effect relationships. Furthermore, the uncertainty in forecasting should be considered. The procurement of raw materials or demand planning are prime examples: The timing of copper purchases can be optimized with accurate market price forecasts, whereas precise forecasts of product sales increase the delivery reliability and reduce costs. Likewise technical applications, e.g. energy generation, also require the modeling of complex dynamical systems.

In this contribution we deal with time-delay recurrent neural networks (RNNs) for time series forecasting and introduce *12* tricks that not only ease the handling of RNNs, but also improve the forecast accuracy. The RNNs and associated tricks are applied in many of our customer projects from economics and industry.

RNNs offer significant benefits for dealing with the typical challenges associated with forecasting. With their universal approximation properties [11], RNNs can model high-dimensional, non-linear relationships. The time-delayed information processing addresses temporal structures. In contrast, conventional econometrics generally uses linear models (e.g. autoregressive models (AR), multivariate linear

regression) which can be efficiently estimated from historical data, but provide only an inadequate framework for non-linear dynamical systems [12].

In Section 28.2 we introduce the so-called correspondence principle for neural networks (*Trick 1*). For us neural networks are a class of functions which can be transformed into architectures. We will work only with algorithms that process information locally within the architectures. As we will outline, for some problems it is easier to start off with the NN architecture and formulate the equations afterwards and for other problems vice versa. The locality of the algorithms enables us to model even large systems. The correspondence principle is the basis for different RNN models and associated tricks.

We will start with a basic RNN in state space formulation for the modeling of partly autonomous and partly externally driven dynamical systems (so-called open systems). The associated parameter optimization task is solved by (finite) unfolding in time, which can be handled by a shared weights extension of standard backpropagation. Dealing with state space models, we are able to utilize memory effects. Therefore, there is no need of a complicated input preprocessing in order to represent temporal relationships. Nevertheless, learning of open dynamical systems tends to focus on the external drivers and, thus, neglects the identification of the autonomous part. On this problem *Trick 2* enforces the autonomous flow of the dynamics and thus, enables long-term forecasting. *Trick 3* finds a proper initialization for the first state vector of recurrent neural network in the finite unfolding.

Typically we do not know all external drivers of the open dynamical system. This may cause the identification of pseudo causalities. *Trick 4* is the extension of the RNN with an error correction term, resulting in a so-called error correction neural network (ECNN), which enables us to handle missing information, hidden external factors or shocks on the dynamics. ECNNs are an appropriate framework for low-dimensional dynamical systems with less than 5 target variables. For the modeling of high-dimensional systems on low dimensional manifolds as in electrical load curves *Trick 5* adds a coordinate transformation (so-called bottleneck) to the ECNN.

Standard RNNs use external drivers in the past and assume constant environmental conditions from present time on. For fast changing environments this is a questionable assumption. Internalizing the environment of the dynamics into the model, leads to *Trick 6*, so-called historically consistent neural networks (HCNN). The special feature of the HCNN is that it not only models the individual dynamics of interest, but also models the external drivers. This leads to a closed dynamical system formulation. Therefore, HCNNs are symmetric in their input and output variables, i.e. the system description does not draw any distinction between input, output and internal state variables.

In practice, HCNNs are difficult to train, because the models have no input signals and are unfolded across the complete data horizon. This implies that we learn from a single data pattern, which is the unique data history. In *Trick 7* we therefore introduce an architectural teacher forcing to make the best possible use of the data from the observables and to accelerate training of the HCNN.

The HCNN models the dynamics for all of the observables and their interaction in parallel. For this purpose a high-dimensional state transition matrix is required. A fully connected state transition matrix can, however, lead to a signal overload during the training of the neural network using error backpropagation through time (EBTT). With Trick 8 we solve this problem by introducing sparse state transition matrices.

The information flow within a HCNN is from the past to present and future time, i.e. we have a causal model to explain the highly-interacting non-linear dynamical systems across multiple time scales. Trick 9 extends this modeling framework with an information flow from the future into past. As we will show this enables us to incorporate the effects of rational decision making and planning into the modeling. The resulting models are called *causal-retro-causal* historically consistent neural networks (CRCNNs). Likewise to HCNNs, CRCNNs are difficult to train. In Trick 10 we extend the basic CRCNN architecture by an architectural teacher forcing mechanism, which allows us to learn the CRCNN using the standard EBTT algorithm. Trick 11 introduces a way to improve the modeling of deterministic chaotic systems.

Finally, Trick 12 is dedicated to the modeling of uncertainty and risk. We calculate ensembles of either HCNNs or CRCNNs to forecast probability distributions. Both modeling frameworks give a perfect description of the dynamic of the observables in the past. However, the partial observability of the world results in a non-unique reconstruction of the hidden variables and thus, different future scenarios. Since the genuine development of the dynamics is unknown and all paths have the same probability, the average of the ensemble is the best forecast, whereas the ensemble bandwidth describes the market risk.

Section 28.3 summarizes the primary findings of this contribution and points to future directions of research.

28.2 Tricks for Recurrent Neural Networks

Trick 1. The Correspondence Principle for Neural Networks

In order to gain a deeper understanding in the functioning and composition of RNNs we introduce our first conceptual trick, which is called correspondence principle between equations, architectures and local algorithms. The correspondence principle for neural networks (NN) implies that any equation for a NN can be portrayed in graphical form by means of an architecture which represents the individual layers of the network in the form of nodes and the matrices between the layers in the form of edges. This correspondence is most beneficial in combination with local optimization algorithms that provide the basis for the training of the NNs. For example, the error back propagation algorithm needs only locally available information from the forward and backward flow of the network in order to calculate the partial derivatives of the NN error function[13]. The use of local algorithms here provides an elegant basis for the expansion of the neural network towards the modeling of large systems. Used in combination with

an appropriate (stochastic) learning rule, it is possible to use the gradients as a basis for the identification of robust minima[8].

Now let us introduce a basic recurrent neural network (RNN) in state space formulation. We start from the assumption that a vector time series y_τ is created by an open dynamical system, which can be described in discrete time τ using a state transition and output equation[3]:

$$\begin{aligned} \text{state transition } s_{\tau+1} &= f(s_\tau, u_\tau) \\ \text{output equation } y_\tau &= g(s_\tau) \end{aligned} \quad (28.1)$$

The hidden time-recurrent state transition equation $s_{\tau+1} = f(s_\tau, u_\tau)$ describes the upcoming state $s_{\tau+1}$ by means of a function of the current system state s_τ and of the external factors u_τ . The system formulated in the state transition equation can therefore be interpreted as a partially autonomous and partially externally driven dynamic. We call this an open dynamical system.

The output, also called observer, equation y_τ uses the non-observable system state s_τ in every time step τ to calculate the output of the dynamic system y_τ . The data-driven system identification is based on the selected parameterized functions $f()$ and $g()$. We chose the parameters in $f()$ and $g()$ such that an appropriate error function is minimized (see Eq. 28.2).

$$\frac{1}{T} \sum_{\tau=1}^T (y_\tau - y_\tau^d)^2 \rightarrow \min_{f,g} . \quad (28.2)$$

The two functions $f()$ and $g()$ are estimated using the quadratic error function (Eq. 28.2) in such a way that the average distance between the observed data y_τ^d and the system outputs y_τ across a number of observations $\tau = 1, \dots, T$ is minimal.

Thus far, we have given a general description of the state transition and the output equation for open dynamical systems. Without loss of generality we can specify the functions $f()$ and $g()$ by means of a recurrent neural network (RNN)[11, 15]:

$$\begin{aligned} \text{state transition } s_{\tau+1} &= \tanh(As_\tau + Bu_\tau) \\ \text{output equation } y_\tau &= Cs_\tau \end{aligned} \quad (28.3)$$

Eq. 28.3 specifies an RNN with weight matrices A , B and C to model the open dynamical system. The RNN is designed as a non-linear state-space model, which is able to approximate any function $f()$ and $g()$. We choose the hyperbolic tangent $\tanh()$ as the activation function for the hidden network layer $s_{\tau+1}$. The output equation is specified as a linear function. The RNN output is generated by a superposition of two components: (i) the autonomous dynamics (coded in A), which accumulates information over time (memory), and (ii) the influence of external factors (coded in B).

Note, that the state transition in Eq. 28.3 does not need an additional matrix leading the hyperbolic tangent $\tanh()$ activation function, since the additional

matrix can be merged into matrix A . Furthermore, without loss of generality we can use a linear output equation in Eq. 28.3. If we would have a non-linearity in the output equation (Eq. 28.3), it could be merged in the state transition equation (Eq. 28.3). For details see Schäfer et al. [11].

We use the technique of finite unfolding in time[10] to solve the temporal system identification, i.e. for the selection of appropriate matrices A , B and C to minimize the error function (Eq. 28.1). The underlying idea here is that any RNN can be reformulated to form an equivalent feedforward neural network, if matrices A , B and C are identical in the individual time steps (shared weights). Fig. 28.1 depicts the resulting RNN.

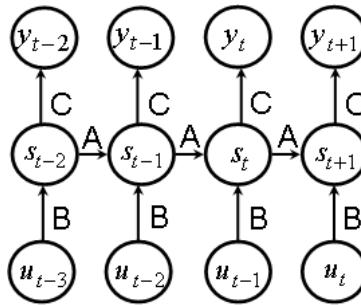


Fig. 28.1. Basic RNN unfolded in time with shared weights A , B and C

One advantage of the shared matrices is the moderate number of free parameters (weights), which reduces the risk of over-fitting[18]. The actual training is conducted using error backpropagation through time (EBTT) together with a stochastic learning rule[13, 10]. For algorithmic solutions, the reader is referred to the overview article by B. Pearlmutter[9].

Trick 2. Overshooting

In applications we often observed that RNNs tend to focus on only the most recent external inputs in order to explain the dynamics. To balance the information flow, we use a trick called overshooting. Overshooting extends the autonomous system dynamics (coded matrix A) into the future (here $t+2$, $t+3$, $t+4$)[18] (see Fig. 28.2). In order to describe the development of the dynamics in one of these future time steps adequately, matrix A must be able to transfer information over time. The different instances of matrix A refer to the same prototype matrix A . Thus the shared weights principle allow us to maintain the locality of the correspondence principle (see Trick 1). By this we can compute consistent multi-step forecasts. A corresponding RNN architecture is depicted in Fig. 28.2. For the RNN we typically use an input preprocessing $u_\tau = x_\tau - x_{\tau-1}$ as the transformation for the raw data x . This avoids trends in the input or

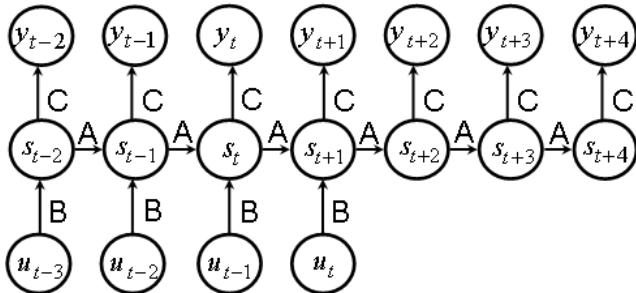


Fig. 28.2. RNN incorporating overshooting

target variables of the RNN. The missing external inputs $u_{\tau>0}$ in the future can be interpreted as a constant environment ($u_{\tau>0} \approx 0$). The effectiveness of overshooting depends on the strength of the autonomous dynamics. The stronger the autonomous flow, the better is the forecast accuracy for the future overshooting time steps. Furthermore, overshooting has an implication for the learning itself. Without overshooting, RNNs have the tendency to focus only on short-term input-output relationships. With overshooting the learning has to work out mid- to long-term input-output relationships.

Summarizing, it should be noted, that overshooting generates additional valuable forecast information about the analyzed dynamical system and acts as a regularization method for the learning.

Trick 3. Handling the Uncertainty of the Initial State

One of the difficulties with finite unfolding in time is to find a proper initialization for the first state vector of the RNN. Our next trick takes care of this problem. An obvious solution is to set the first state s^0 equal to zero. We assume that the unfolding includes enough (past) time steps such that the misspecification of the initialization phase is overwritten along the state transitions. In other words, the network accumulates information over time and thus, may eliminate the impact of the arbitrary initial state on the network outputs.

Beyond this the modeling can be improved if we are able to make the unfolded RNN less sensitive from the unknown initial state s^0 . A first approach to stiff the model against the unknown s^0 is to apply a noise term Θ to the state s^0 . A fixed noise term Θ which is drawn from a certain noise distribution is clearly inadequate to handle the uncertainty of the initial state. Since we do not know what is an appropriate level of noise, we have to find a way to estimate the noise level. We propose to apply an adaptive noise Θ , which fits best to the level of uncertainty of the unknown s^0 . The characteristic of the adaptive noise term Θ is automatically determined as a by-product of the error backpropagation algorithm.

The basic idea is as follows [8]: We use the residual error ϵ of the neural network as computed by error backpropagation for s^0 . The residual error ϵ as

measured at the initial state s^0 can be interpreted as the uncertainty which is due to the missing information about the true initial state vector s^0 . We disturb the initial state s^0 with a noise term Θ which follows the distribution of the residual error ϵ . Given the uncertain initial states, learning tries to fulfill the output-target relationships along the dynamics. As a result of the learning we get a state transition matrix in form of a contraction, which squeezes out the initial uncertainty. A corresponding network architecture is depicted in Fig. 28.3.

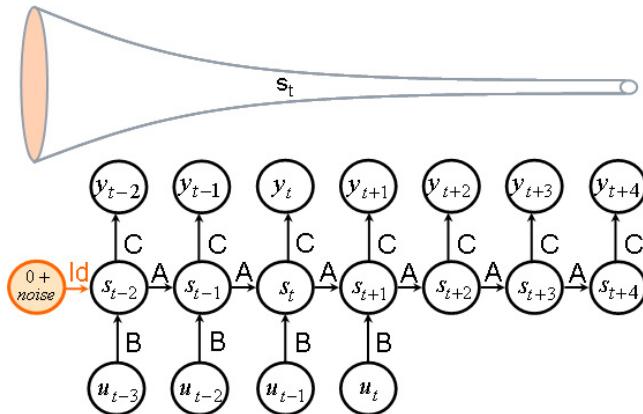


Fig. 28.3. Handling the uncertainty of the initial state s^0 by applying adaptive noise

It is important to notice, that the noise term Θ is drawn from the observed residual errors without any assumption on the underlying noise distribution. The desensitization of the network to the initial state vector s^0 can therefore be seen as a self-scaling stabilizer of the modeling.

In general, a time discrete trajectory can be seen as a sequence of points over time. Such a trajectory is comparable to a fine thread in the internal state space. The trajectory is very sensitive to the initial state vector s^0 . If we apply noise to s^0 , the trajectory becomes a tube in the internal state space. Due to the characteristics of the adaptive noise term, the tube contracts over time. This enforces the identification of a stable dynamical system.

Trick 4. Error Correction Neural Networks (ECNN)

A weakness of the RNN (see Fig. 28.1 or 28.2) is, that modeling might be disturbed by unknown external influences or shocks. As a remedy, the next trick called error correction neural networks (ECNN) introduces an additional term $z_\tau = \tanh(y_\tau - y_\tau^d)$ in the state transition (28.4). The term can be interpreted as a correctional factor: The model error $(y_\tau - y_\tau^d)$ at time τ quantifies the misfit and may help to adjust the model output afterwards.

$$\begin{aligned} \text{state transition } & s_{\tau+1} = \tanh(As_\tau + Bu_\tau + D \tanh(y_\tau - y_\tau^d)) \\ \text{output equation } & y_\tau = Cs_\tau \end{aligned} \quad (28.4)$$

In Eq. 28.4 the model output y_τ is computed by Cs_τ and compared with the observation y_τ^d . The output clusters of the ECNN which generate error signals during the learning phase are $z_\tau(\tau \leq t)$ and $y_\tau(\tau > t)$. Have in mind, that the target values of the sequence of output clusters z_τ are zero, because we want to optimize the compensation mechanism $y_\tau - y_\tau^d$ between the expected value y_τ and its observation y_τ^d . The additional non-linear squashing function in $z_\tau = \tanh(y_\tau - y_\tau^d)$ absorbs large errors respectively shocks. A special case occurs at all future time steps $t + 1$: here we have no compensation y_{t+1}^d of the internal expected value, and thus the system is offering a forecast $y_{t+1} = Cs_{t+1}$.

The system identification task (see Eq. 28.2) is once again solved by finite unfolding in time [3]. Fig. 28.4 depicts the resulting ECNN[15].

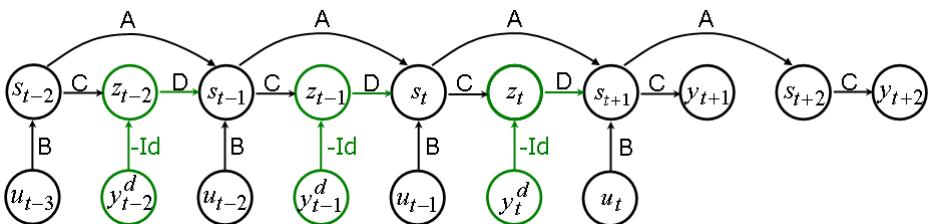


Fig. 28.4. ECNN incorporating overshooting. Note, that $-Id$ is the fixed negative of an identity matrix, while $z_{t-\tau}$ are output clusters to model the error correction mechanism.

The ECNN has two different inputs: (i) the externals u_τ , which directly influence the state transition, and (ii) the targets y_τ^d . Only the difference between y_τ and y_τ^d has an impact on $s_{\tau+1}$. In the future $\tau > t$, we have no compensation for y_τ and thus compute forecasts $y_\tau = Cs_\tau$. We have successfully applied ECNN models to predict the demand of products and product groups within the context of supply chain management.

Trick 5. Variant-Invariant Separation

For the modeling of high-dimensional dynamical systems, the next trick called variant-invariant separation extends the architecture of the ECNN (Fig. 28.5) by a coordinate transformation to reduce the dimensionality of the original forecast problem. The dimension reduction is realized in form of a bottleneck sub-network (Fig. 28.5, left). The compressor E removes time invariant structures from the dynamics. The reconstruction of the complete dynamics (decompression) is done by matrix F . The bottleneck is implicitly connected to the ECNN via the shared matrices E and F [15]. We compress the high-dimensional vector y_t^d to a lower dimensional vector x_t using a bottleneck neural network. The vector x_t contains all relevant information about y_t^d . The ECNN predicts the low dimensional vector x_τ instead of the high dimensional vector y_τ . The error correction compares

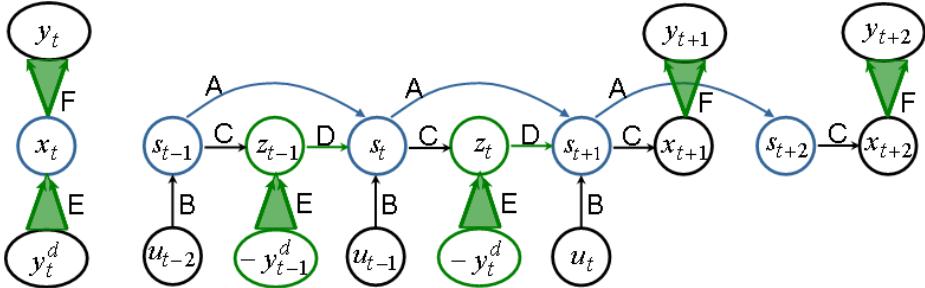


Fig. 28.5. ECNN with Variant-Invariant Separation

the expectations $x_\tau = Cs_\tau$ with the transformed observations $-x_\tau^d = E(-y_\tau^d)$. Note, that the negative inputs $-y_\tau^d$ are required by the ECNN to generate the transformed targets $-x_\tau^d$. In our experience we found, that the training of the extended ECNN (Fig. 28.5) is very robust, i.e. the coordinate transformation and the forecasting can be trained in parallel. The time-invariant information is not lost in the bottleneck network, but simply relegated to lower components of variance of the representation. Furthermore, node pruning can be applied to the middle layer. Due to shared weights the result of a pruning in the bottleneck network is transferred to the ECNN branch as well.

We have successfully applied the ECNN depicted in Fig. 28.5 to forecast electrical load curves and traffic flows. In load forecasting the typical application scenario is that one has to forecast the load curve in 15 minute time buckets, i.e. 96 observations per day. To avoid the error accumulation of a pure iterative model it is more useful to forecast the load curve day-by-day. From our experience a load curve (i.e. 96 observations per day) can be compressed to an $\dim \approx 8$ dimensional indicator vector from which the load curve can in turn be reconstructed. From a mathematical viewpoint this approach corresponds to the modeling of dynamical systems on manifolds. More complicated manifolds can be generated by deep bottleneck neural networks.

Trick 6. Historically Consistent Neural Networks (HCNNs)

Many real-world technical and economic applications can be seen in the context of large systems in which various (non-linear) dynamics interact with one another (in time). Unfortunately only a small sub-set of variables can be observed. From the sub-set of observed variables we have to reconstruct the hidden variables of the large system in order to understand the dynamics. Here the term *observables* includes the input and output variables in conventional modeling approaches (i.e. $y_\tau := (y_\tau, u_\tau)$). This indicates a consistency problem in the RNN (Fig. 28.1) or ECNN (Fig. 28.4) respectively: on the output side, the RNN (ECNN) provides forecasts of the dynamics in the observables y_τ , whereas the input side assumes that the observables y_τ will not change from present time on. This lack of consistency represents a clear contradiction within the model

framework. If, on the other hand, we are able to implement a model framework in which common descriptions and forecasts can be used for the trend in all of the observables, we will be in a position to close the open system – in other words, we will model a closed large dynamic system.

The next trick called historically consistent neural networks (HCNNs) introduces a model class which follows the design principles for modeling of large dynamic systems and overcomes the conceptual weaknesses of conventional models. Equation 28.5 formulates the historically consistent neural network (HCNN).

$$\begin{aligned} \text{state transition } s_{\tau+1} &= A \tanh(s_\tau) \\ \text{output equation } y_\tau &= [Id, 0]s_\tau \end{aligned} \quad (28.5)$$

The joint dynamics for all observables is characterized in the HCNN (28.5) by the sequence of states s_τ . The observables ($i = 1, \dots, N$) are arranged on the first N state neurons s_τ and followed by non-observable (hidden) variables as subsequent neurons. The connector $[Id, 0]$ is a fixed matrix which reads out the observables. The initial state s_0 is described as a bias vector. The bias s_0 and matrix A contain the only free parameters.

Like standard RNNs (Fig. 28.1) HCNNs also have universal approximation capabilities. The proof for the RNN can be found in [11]. Fig. 28.6 outlines the proof for HCNNs. As depicted in Fig. 28.6 the proof of the universal approximation capabilities for HCNN can be divided in six steps:

1. The output equation is shifted one time step into the future and the resulting $s_{\tau+1}$ is substituted by the system transition equation.
2. By combining outputs and state variables into an extended state we get an extended state transition equation. The output of the system is derived from the first components of the extended internal state.
3. For the extended state transition equation we apply the feedforward universal approximation theorem. At least for a finite time horizon this guarantees a small approximation error. Note, that in RNNs at least one large component of the state vector together with the hyperbolic tangent can mimic a bias vector. Thus, we have omitted the explicit notation of a bias vector in the NN equations.
4. In this step we remove one of the two matrices within the state transition equation. We apply a state transformation $r_\tau = As_\tau$. This results in two state transition equations.
5. The two state transition equations can be reorganized in one state transition, which has twice the dimensionality of the original equation.
6. Rewriting the matrix located on front of the tanh activation function results in the claimed formulation for closed systems.

Instead of being applied inside the tanh activation function, matrix A is used outside the tanh activation function. This has the advantage that the states and thus, also the system outputs are not limited to the finite state space $(-1; 1)^n$ created by the $\tanh(\cdot)$ nonlinearity. The output equation has a simple and application independent form. Note, that we can only observe the first elements of the state vector.

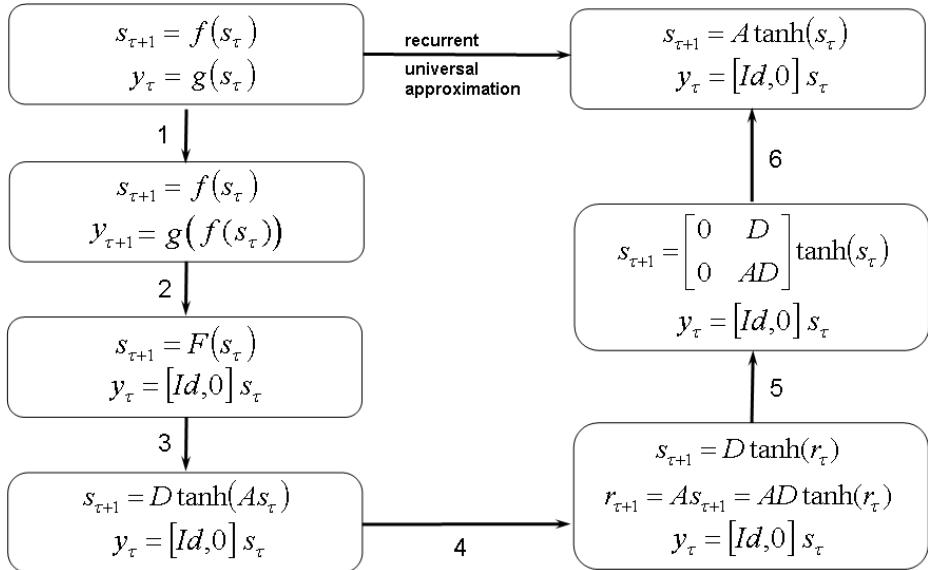


Fig. 28.6. Proof of the universal approximation capabilities for HCNNs

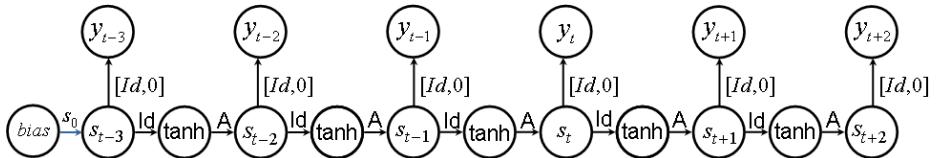


Fig. 28.7. Historically Consistent Neural Network (HCNN)

Fig. 28.7 depicts the HCNN architecture. The HCNN states s_τ are hidden layers with tanh squashing. The forecasts are supplied by the output layers y_τ . There are no target values available for the future time steps. The expected values $y_{\tau>t}$ can be read out at the corresponding future time steps of the network.

Since the HCNN model has no inputs, we have to unfold the neural network along the complete data history. This is different to small recurrent neural networks (see e.g. Fig. 28.2), where we construct training data patterns in form of sliding windows. The HCNN learns the large dynamics from a single history of observations (i.e. a single training data pattern). Forecasting commodity prices with HCNN, we unfold the neural network over 440 trading days in the past to predict the next 20 days.

Trick 7. Architectural Teacher Forcing (ATF) for HCNNs

In practice we observe that HCNNs are difficult to train since the models do not have any input signals and are unfolded across the complete data set. Our next

trick, architectural teacher forcing (ATF) for HCNNs, makes the best possible use of the data from the observables and accelerates the training of the HCNN [20, 14, 9]. The HCNN with integrated teacher forcing is shown in Fig. 28.8 below. In the HCNN with ATF (Fig. 28.8) the expected values for all observables up

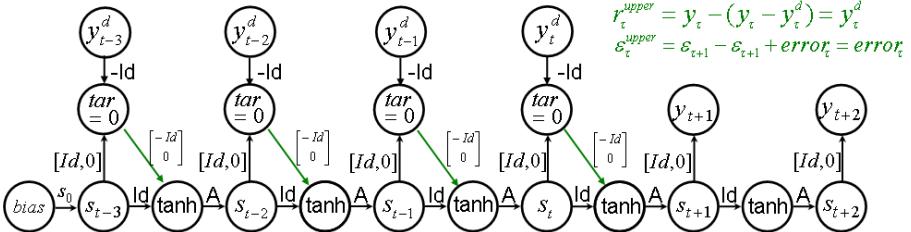


Fig. 28.8. HCNN with Architectural Teacher Forcing (ATF)

to time $\tau = t$ are replaced with the actual observations. The output layers of the HCNN are given fixed target values of zero. The negative observed values $-y_\tau^d$ for the observables are added to the output layer. This forces the HCNN to create the expected values y_τ to compensate for the negative observed values $-y_\tau^d$. The content of the output layer, i.e. $y_\tau - y_\tau^d$, is now transferred to the first N neurons of the hidden layer r_τ on a component-by-component basis with a minus symbol. In addition, we copy the expected values y_τ from the state s_τ to the intermediate hidden layer r_τ . As a result, the expected values y_τ on the first N components of the state vector s_τ are replaced by the observed values $y_\tau^d = y_\tau - (y_\tau - y_\tau^d)$ (Fig. 28.8). All connections of the ATF mechanism are fixed. Following the ATF step, the state transition matrix A is applied, to move the system into the next time step. By definition, we have no observations for the observables in future time steps. Here, the system is iterated exclusively upon the expected values. This turns an open into a closed dynamic system. The HCNN in Fig. 28.8 is equivalent to the architecture in Fig. 28.7, if it converges to zero error in the training. In this case we have solved the original problem.

Trick 8. Sparsity, Dimensionality vs. Connectivity and Memory

HCNNs may have to model ten, twenty or even more observables in parallel over time. It is clear that we have to work with high dimensional dynamical systems (e.g. in our commodity price models we use $dim(s) = 300$). The iteration with a fully connected state transition matrix A of such a dimension is dangerous: Sometimes the matrix vector operations will produce large numbers which will be spread in the recursive computation all over the network and will generate an arithmetic overflow. To avoid this phenomena we can choose a sparse matrix A . Thus, the linear algebra does not accumulate large numbers and the spread of large numbers through the network is damped by the sparsity too.

We have to answer the questions which dimensionality and which sparsity we will choose. In [16] we have worked out that dimensionality and sparsity are related to another pair of meta-parameters: Connectivity (*con*) and memory length (*mem*). Connectivity is defined as the number of nonzero elements in each row of matrix A . The memory length is the number of steps from which we have to collect information in order to reach a Markovian state, i.e. the state vector contains all necessary information from the past. We propose the following parameterization for the state dimension ($\dim(s)$) and sparsity [16]:

$$\text{Dimension of } A \quad \dim(s) = \text{con} \cdot \text{mem} \quad (28.6)$$

$$\text{Sparsity of } A \quad \text{Sparsity} = \text{random} \left(\frac{\text{con}}{\text{mem} \cdot \text{con}} \right) = \text{random} \left(\frac{1}{\text{mem}} \right) \quad (28.7)$$

Eq. 28.7 represents the insight that a sparse system conserves information over a longer time period before it diffuses in the network. For instance a shift register is very sparse and behaves only as a memory, whereas in a fully connected matrix the superposition of information masks the information sources. Let us assume that we have initialized the state transition matrix with a uniform random sparse matrix A . Following Eq. 28.7 the more dense parts of A will model the faster sub-dynamics within the overall dynamics, while the highly sparse parts of A will focus on slow subsystems. As a result a sparse random initialization allows the combined modeling of systems on different time scales.

Unfortunately, Eq. 28.6 favors very large dimensions. Our earlier work on the subject (see [16]) started with the predefinition of the systems memory length *mem*, because for RNNs the memory length is equal to the length of the past unfolding in time. On the other hand, connectivity has to be chosen larger than the number of the observables. Working with HCNNs the memory length is less important, because we unfold the neural network along the whole data horizon. Here the connectivity plays the superior role. From our experience we know that the EBTT algorithm works stably with a connectivity which is equal or smaller than 50 ($\text{con} \leq 50$). For computational performance we usually limit the state dimensionality to $\dim(s) = 300$. This implies a sparsity of $50/300 \approx 17\%$. We leave the fine tuning of the parameters to the EBTT learning.

We propose to initialize the neural network with a randomly chosen sparsity grid. The sparsity grid is therefore chosen arbitrary and not optimized by e.g. pruning algorithms. This raises the question if a random sparse initialization biases the network towards inferior solutions. This is handled by ensemble forecasts. We have performed ensemble experiments with different sparsity grids versus ensembles based on the same sparsity grid. We found, that the average of the ensemble as well as the ensemble width are unaffected by the initialization of the sparsity grid (for more details on ensemble forecasting see Trick 12). These considerations hold only for large systems.

Trick 9. Causal-Retro-Causal Neural Networks (CRCNNs)

The fundamental idea of the HCNN is to explain the joint dynamics of the observables in a causal manner, i.e. with an information flow from the past to

the future. However, rational planning is not only a consequence of a causal information flow but also of anticipating future developments and responding to them on the basis of a certain goal function. This is similar to the adjoint equation in optimal control theory [6]. In other words a retro causal information flow is equivalent to asking for the motivation of a behavior as a goal. In turn, this is the anchor point for the reconstruction of the dynamics.

In order to incorporate the effects of rational decision making and planning into the modeling, the next trick introduces causal-retro-causal neural networks (CRCNNs). The idea behind the CRCNN is to enrich the causal information flow within the HCNN, which is directed from the past to the future, by a retro-causal information flow, directed from the future into the past. The CRCNN model is given by the following set of equations 28.8.

$$\begin{aligned} \text{causal state transition} \quad s_\tau &= A \tanh(s_{\tau-1}) \\ \text{retro-causal state transition} \quad s'_\tau &= A' \tanh(s'_{\tau+1}) \\ \text{output equation} \quad y_\tau &= [Id, 0]s_\tau + [Id, 0]s'_\tau. \end{aligned} \quad (28.8)$$

The output equation y_τ of the CRCNN (Eq. 28.8) is a mixture of causal and retro-causal influences. The dynamics of all observables is hence explained by a sequence of causal (s_τ) and retro-causal states s'_τ using transition matrices A and A' for the causal and retro-causal information flow. Upon the basis of Eq. 28.8, we draw the network architecture for the CRCNN as depicted in Fig. 28.9.

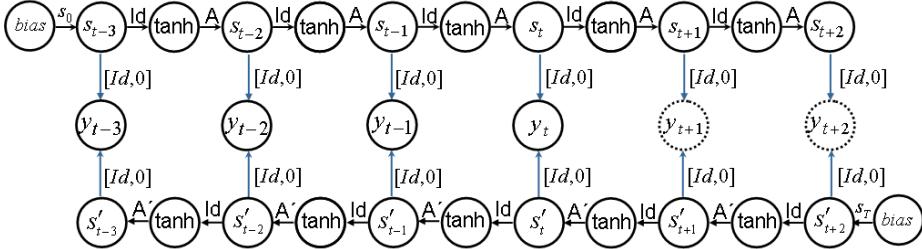


Fig. 28.9. A Causal-Retro-Causal Historically Consistent Neural Network (CRCNN)

Trick 10. Architectural Teacher Forcing (ATF) for CRCNNs

The CRCNN (Fig. 28.9) is also unfolded across the entire time path, i.e. we learn the unique history of the system. Likewise to the training of the HCNN, CRCNNs are difficult to train. Our next trick called architectural teacher forcing (ATF) for CRCNNs formulates TF as a part of the CRCNN architecture, which allows us to learn the CRCNN using the standard EBTT algorithm[3]. ATF enables us to exploit the information contained in the data more efficiently and accelerates the training itself. Fig. 28.10 depicts the CRCNN architecture incorporating ATF.

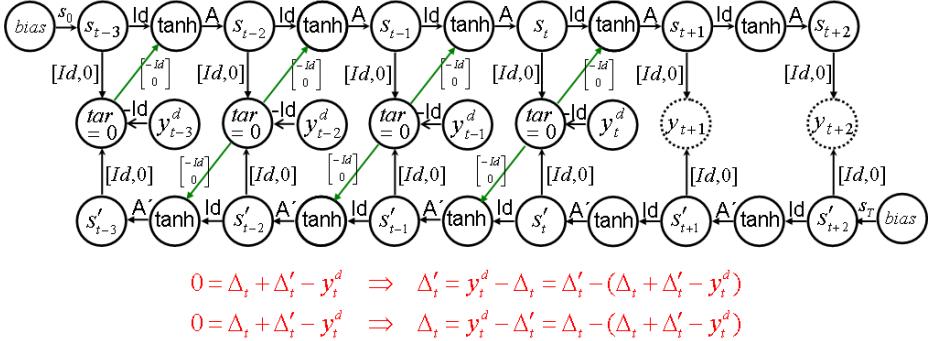


Fig. 28.10. Extended CRCNN with an architectural Teacher Forcing (ATF) mechanism

Let us explain the ATF mechanism in the extended CRCNN model (Fig. 28.10): The extended CRCNN uses a causal-retro-causal network to correct the error of the opposite part of the network in a symmetric manner. In every time step $\tau \leq t$ the expected values y_τ are replaced by the observations y_τ^d using the intermediate $\text{tanh}()$ layers for the causal and for the retro-causal part. Since the causal and the retro-causal part *jointly* explain the observables y_τ , we have to inject the causal into the retro-causal part and vice versa. This is done by compensating the actual observations $-y_\tau^d$ with the output of the causal (Δ_τ) and the retro-causal part (Δ'_τ) within output layers with fixed target values of zero. The resulting content ($\Delta_\tau + \Delta'_\tau - y_\tau^d = 0$) of the output layers is negated and transferred to the causal and retro-causal part of the CRCNN using the fixed $[-Id, 0]'$ connector. Within the intermediate $\text{tanh}()$ layers the expectations of y_τ are replaced with the actual observations y_τ^d , whereby the contribution to y_τ from the opposite part of the CRCNN is considered. Note, that ATF does not lead to a larger number of free network parameters, since all new connections are fixed and are used only to transfer data in the NN. In future direction $\tau > t$ the CRCNN is iterated exclusively on the basis of expected values.

The usage of ATF does not reintroduce an input / output modeling, since we replace the expected value of the observables with their actual observations, while simultaneously considering the causal and retro-causal part of the dynamics. For sufficiently large CRCNNs and convergence of the output error to zero, the architecture in Fig. 28.10 converges towards the fundamental CRCNN architecture shown in Fig. 28.9. The advantage of the CRCNN is, that it allows a fully dynamical superposition of the causal and retro-causal information flows.

The CRCNN depicted in Fig. 28.10 describes a dynamics on a manifold. In every time step the information flows incorporates closed loops, which technically can be seen as equality constraints. These constraints are only implicitly defined through the interaction of the causal and retro-causal parts. The closed loops in the CRCNN architecture (Fig. 28.10) lead to fix-point recurrent substructures

within the model, which are hard to handle with EBTT. As a remedy we propose a solution concept similar to Trick 7: We embed the CRCNN model into a larger network architecture, which is easier to solve and converges to the same solution as the original system. Fig. 28.11 depicts an initial draft for such an embedding.

The extended architecture in Fig. 28.11 is a duplication of the original model depicted in Fig. 28.10. The CRCNN architecture in Fig. 28.11 does not contain closed loops, because we split the ATF mechanism for the causal and retro-causal part into two branches. It is important to notice that these branches are implicitly connected through the shared weights in the causal and retro-causal part. If this architecture converges, the ATF is no longer required and we have two identical copies of the CRCNN model depicted in Fig. 28.9. The solution proposed for the embedding is not the only feasible way to handle the fix-point loops. We will outline alternative solutions in an upcoming paper. The CRCNN is the basis for our projects on forecasting commodity prices.

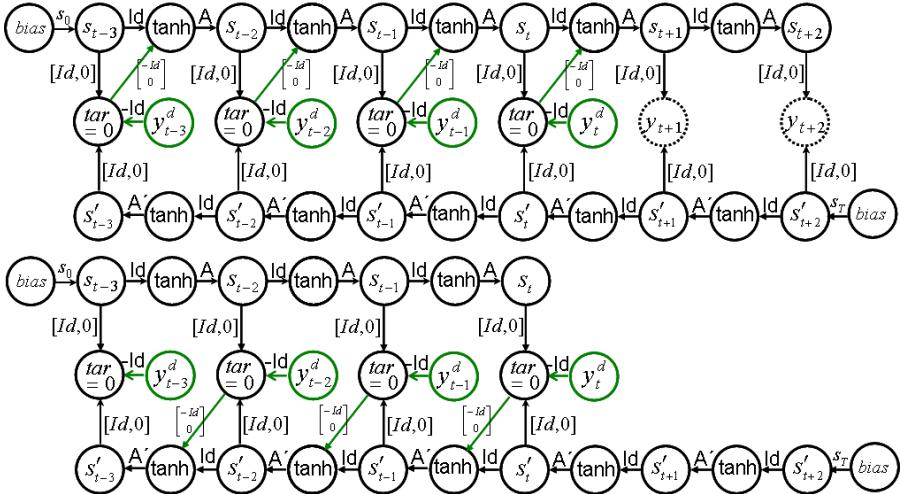


Fig. 28.11. Asymmetric split of ATF in CRC neural networks

Trick 11. Stable & Instable Information Flows in Dynamical Systems

Following Trick 3 it is natural to apply noise in the causal as well as in retro-causal branch (see also Fig. 28.12). This should improve the stability of both time directions resp. information flows. In CRCNNs this feature has a special interpretation: Stability in the causal information flow means that the uncertainty in the beginning is damped along the time path from past to future. Instability in the causal system means that a small disturbance in the past diffuses to very different future scenarios under a chaotic regime (see Fig. 28.12, upper part).

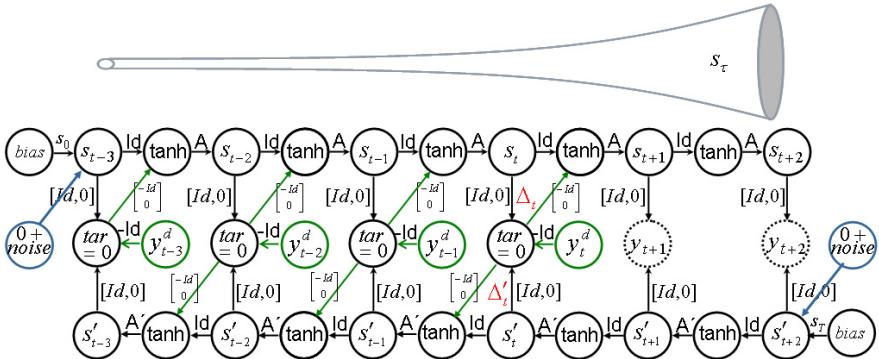


Fig. 28.12. An unstable causal dynamics is converted to a stable retro-causal dynamics

If the causal information flow of a sub-dynamics is unstable, then the retro-causal description of this sub-system is stable. On the other hand, an unstable retro-causal dynamics is stable from a causal perspective. In a combined causal-retro-causal neural network the causal *and* the retro-causal branch can be simultaneously stable, even if the underlying dynamics is partially unstable. In order to enforce the stability of the causal and the retro-causal part we apply noise at the origins of both branches.¹

Trick 12. Uncertainty and Risk

The experience gained during the latest financial crisis has triggered a far-reaching discussion on the limitations of quantitative forecasting models and made investors very conscious of risk[2]. In order to understand risk distributions, traditional risk management uses diffusion models. Risk is understood as a random walk, in which the diffusion process is calibrated by the observed past error of the underlying model[7]. In contrast the next trick called uncertainty and risk focuses on ensemble forecasts in order to provide important insights into complex risk relationships, since internal model (unobserved) variables can be reconstructed from the trend in observed variables (observables).

If the system identification is calculated repeatedly for HCNNs or CRCNNs, an ensemble of solutions will be produced, which all have a forecast error of zero in the past, but which differ from one another in the future. Since every HCNN or CRCNN model gives a perfect description of the observed data, the complete ensemble is the true solution. A way to simplify the forecast is to take the arithmetical average of the individual ensemble members as the expected value, provided the ensemble histogram is unimodal in every time step.

In addition to the expected value, we consider the bandwidth of the ensemble, i.e. its distribution. The form of the ensemble is governed by differences in the reconstruction of the hidden system variables from the observables: for every finite

¹ Thanks to Prof. Jürgen Jost, MPI Leipzig, for a fruitful discussion on this topic.

volume of observations there is an infinite number of explanation models which describe the data perfectly, but differ in their forecasts, since the observations make it possible to reconstruct the hidden variables in different forms during the training. In other words, our risk concept is based on the partial observability of the world, leading to different reconstructions of the hidden variables and thus, different future scenarios. Since all scenarios are perfectly consistent with the history, we do not know which of the scenarios describes the future trend best and risk emerges.

This approach directly addresses the model risk. For HCNN and CRCNN modeling we claim that the model risk is equal to the forecast risk. The reasons can be summarized as follows: First, HCNNs are universal approximators, which are therefore able to describe every market scenario. Second, the form of the ensemble distribution is caused by an underlying dynamics, which interpret the market dynamics as the result of interacting decisions[17]. Third, in experiments we have shown that the ensemble distribution is independent from the details of the model configuration, if we use large models and large ensembles.

Let us exemplify our risk concept. The diagram below (Fig. 28.13, left) shows the approach applied to the Dow Jones Industrial Index (DJI). For the ensemble, a HCNN was used to generate 250 individual forecasts for the DJI. For every forecast date, all of the individual forecasts for the ensemble represent the empirical density function, i.e. a probability distribution over many possible market prices at a single point in time (see Fig. 28.13, right). It is noticeable that the actual development of the DJI is always within the ensemble channel (see gray lines, Fig. 28.13, left). The expected value for the forecast distribution is also an adequate point forecast for the DJI (see Fig. 28.13, right).

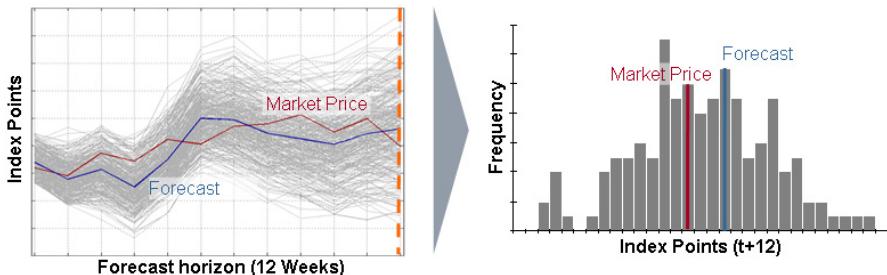


Fig. 28.13. HCNN ensemble forecast for the Dow Jones Index (12 weeks forecast horizon), left, and associated index point distribution for the ensemble in time step $t + 12$, right

28.3 Conclusion and Outlook

Recurrent neural networks model dynamical systems in the form of non-linear state space models. Just like any other NN, the equation of the RNNs, ECNNs, HCNNs or CRCNNs can be expressed as an architecture which represents the

individual layers in the form of nodes and the connections between the layers in the form of links. In the graphical architecture we can apply local learning algorithms like error back propagation and an appropriate (stochastic) learning rule to train the NN[13, 17, 3]. This relationship is called the correspondence principle between equations, architectures and the local algorithms associated with them (**Trick 1**).

Finite unfolding in time of RNN using shared weight matrices enables us to stick to the correspondence principle. Overshooting enforces the autonomous dynamics and enables long-term forecasting (**Trick 2**), whereas an adaptive noise term handles the uncertainty of the finite unfolding in time (**Trick 3**).

ECNN utilizes the previous model error as an additional input. Hence, the learning can interpret the models misfit as an external shock which is used to guide the model dynamics afterwards. This allows us to prevent the autonomous part of the model to adapt misleading inter-temporal causalities. If we know that a dynamical system is influenced by external shocks, the error correction mechanism of the ECNN is an important prestructuring element of the networks architecture to compensate missing inputs (**Trick 4**).

Extending the ECNN by variants-invariants separation, one is able to include additional prior structural knowledge of the underlying dynamics into the model. The separation of variants and invariants with a bottleneck coordinate transformation allows to handle high dimensional problems (**Trick 5**).

HCNNs model not just an individual dynamics, but complex systems made up of a number of interacting sub-dynamics. HCNNs are symmetrical in their input and output variables, i.e. the system description does not draw any distinction between input, output and internal state variables. Thus, an open system becomes a closed system (**Trick 6**). Sparse transition matrices enable us to model different time scales and stabilize the training (**Trick 8**). Causal and retro-causal information flow within an integrated model (CRCNN) can be used to model rational planning and decision making in markets. CRCNNs dynamically combine causal and retro-causal information to describe the prevailing market regime (**Trick 9**). Architectural teacher forcing can be applied to efficiently train the HCNN or CRCNN (**Trick 7 and 10**). An architectural extension (see Fig. 28.12) enables us to balance the causal and retro-causal information flow during the learning of the CRCNN (**Trick 11**).

We usually work with ensembles of HCNN or CRCNN to predict commodity prices. All solutions have a model error of zero in the past, but show a different behavior in the future. The reason for this lies in different ways of reconstructing the hidden variables from the observations and is independent of different random sparse initializations. Since every model gives a perfect description of the observed data, we can use the simple average of the individual forecasts as the expected value, assuming that the distribution of the ensemble is unimodal. The analysis of the ensemble spread opens up new perspectives on market risks. We claim that the model risk of a CRCNN or HCNN is equal to the forecast risk (**Trick 12**).

Work currently in progress aims to improve the embedding of the CRCNN architecture (see Fig. 28.10) in order to simplify and stabilize the training. On

the other hand, we analyze the micro-structure of the ensembles and implement the models in practical risk management and financial market applications.

All NN architectures and algorithms are implemented in the Simulation Environment for Neural Networks (SENN), a product of Siemens Corporate Technology. Work is partially funded by German Federal Research Ministry (BMBF grant Alice, 01 IB10003 A-C).

References

- [1] Calvert, D., Kremer, S.: Networks with Adaptive State Transitions. In: Kolen, J.F., Kremer, S. (eds.) *A Field Guide to Dynamical Recurrent Networks*, pp. 15–25. IEEE (2001)
- [2] Föllmer, H.: Alles richtig und trotzdem falsch?, Anmerkungen zur Finanzkrise und Finanzmathematik. In: MDMV, vol. 17, pp. 148–154 (2009)
- [3] Haykin, S.: *Neural Networks and Learning Machines*, 3rd edn. Prentice Hall (2008)
- [4] Hull, J.: *Options, Futures & Other Derivative Securities*. Prentice Hall (2001)
- [5] Hornik, K., Stinchcombe, M., White, H.: Multilayer Feedforward Networks are Universal Approximators. *Neural Networks* 2, 359–366 (1989)
- [6] Kamien, M., Schwartz, N.: *Dynamic Optimization: The Calculus of Variations and Optimal Control in Economics and Management*, 2nd edn. Elsevier Science (October 1991)
- [7] McNeil, A., Frey, R., Embrechts, P.: *Quantitative Risk Management: Concepts, Techniques and Tools*. Princeton University Press, Princeton (2005)
- [8] Neuneier, R., Zimmermann, H.-G.: How to Train Neural Networks. In: Orr, G.B., Müller, K.-R. (eds.) *NIPS-WS 1996. LNCS*, vol. 1524, pp. 373–423. Springer, Heidelberg (1998)
- [9] Pearlmutter, B.: Gradient Calculations for Dynamic Recurrent Neural Networks. In: Kolen, J.F., Kremer, S. (eds.) *A Field Guide to Dynamical Recurrent Networks*, pp. 179–206. IEEE Press (2001)
- [10] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning Internal Representations by Error Propagation. In: Rumelhart, D.E., McClelland, J.L., et al. (eds.) *Parallel Distributed Processing*, vol. 1: Foundations. MIT Press, Cambridge (1986)
- [11] Schäfer, A.M., Zimmermann, H.-G.: Recurrent Neural Networks Are Universal Approximators. In: Kollias, S.D., Stafylopatis, A., Duch, W., Oja, E. (eds.) *ICANN 2006. LNCS*, vol. 4131, pp. 632–640. Springer, Heidelberg (2006)
- [12] Wei, W.S.: *Time Series Analysis: Univariate and Multivariate Methods*. Addison-Wesley Publishing Company, N.Y. (1990)
- [13] Werbos, P.J.: Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD Thesis, Harvard University (1974)
- [14] Williams, R.J., Zipser, D.: A Learning Algorithm for continually running fully recurrent neural networks. *Neural Computation* 1(2), 270–280 (1989)
- [15] Zimmermann, H.G., Grothmann, R., Neuneier, R.: Modeling of Dynamical Systems by Error Correction Neural Networks. In: Soofi, A., Cao, L. (eds.) *Modeling and Forecasting Financial Data, Techniques of Nonlinear Dynamics*. Kluwer (2002)
- [16] Zimmermann, H.G., Grothmann, R., Schäfer, A.M., Tietz, Ch.: Modeling Large Dynamical Systems with Dynamical Consistent Neural Networks. In: Haykin, S., et al. (eds.) *New Directions in Statistical Signal Processing*. MIT Press, Cambridge (2006)

- [17] Zimmermann, H.G.: Neuronale Netze als Entscheidungskalkül. In: Rehkugler, H., Zimmermann, H.G. (eds.) *Neuronale Netze in der Ökonomie, Grundlagen und wissenschaftliche Anwendungen*. Vahlen, Munich (1994)
- [18] Zimmermann, H.G., Neuneier, R.: Neural Network Architectures for the Modeling of Dynamical Systems. In: Kolen, J.F., Kremer, S. (eds.) *A Field Guide to Dynamical Recurrent Networks*, pp. 311–350. IEEE Press (2001)
- [19] Zimmermann, H.G., Grothmann, R., Tietz, C., von Jouanne-Diedrich, H.: Market Modeling, Forecasting and Risk Analysis with Historical Consistent Neural Networks. In: Hu, B., et al. (eds.) *Operations Research Proceedings 2010, Selected Papers of the Annual Int. Conferences of the German OR Society (GOR)*, Munich. Springer, Heidelberg (September 2011)
- [20] Zimmermann, H.G., Grothmann, R., Tietz, Ch.: Forecasting Market Prices with Causal-Retro-Causal Neural Networks. In: Klatte, D., Lüthi, H.-J., Schmedders, K. (eds.) *Operations Research Proceedings 2011, Selected Papers of the Int. Conference on Operations Research 2011 (OR 2011)*, Zurich, Switzerland. Springer (2012)

Solving Partially Observable Reinforcement Learning Problems with Recurrent Neural Networks

Siegmund Duell^{1,2}, Steffen Udluft¹, and Volkmar Sterzing¹

¹ Siemens AG, Corporate Technology,
Intelligent Systems and Control

² Berlin University of Technology, Machine Learning
 {duell.siegmund.ext,steffen.udluft,volkmar.sterzing}@siemens.com

Abstract. The aim of this chapter is to provide a series of tricks and recipes for neural state estimation, particularly for real world applications of reinforcement learning. We use various topologies of recurrent neural networks as they allow to identify the continuous valued, possibly high dimensional state space of complex dynamical systems. Recurrent neural networks explicitly offer possibilities to account for time and memory, in principle they are able to model any type of dynamical system. Because of these capabilities recurrent neural networks are a suitable tool to approximate a Markovian state space of dynamical systems. In a second step, reinforcement learning methods can be applied to solve a defined control problem. Besides the trick of using a recurrent neural network for state estimation, various issues regarding real world problems such as, large sets of observables and long-term dependencies are addressed.

29.1 Introduction

In this chapter we present a state estimation approach to tackle partially observable reinforcement learning [26] problems in discrete time. Reinforcement learning is the machine learning approach to the optimal control problem. Instead of designing the control strategy, reinforcement learning learns it from actual observations of the system to be controlled. Combined with powerful function approximators like neural networks, impressive results could be achieved [28, 13, 19]. In most real world applications, some form of state estimation is necessary to fulfill the requirements of reinforcement learning.

Consider the task to reduce the emissions of a gas turbine while keeping humming, caused by combustion dynamics, low. A gas turbine consists of a compressor, providing compressed air. Within the combustion chamber, this air is burned with gas to drive the turbine and the coupled generator. The gas is injected through multiple burners. At each burner, the fuel flow is split into different fractions enabling control over the combustion process. The combustion process results in emissions of NOx and CO, which have to be kept below

legal limits. At the same time, the combustion process causes humming. This humming reduces the life time of the machine and can cause fatal damage to the turbine. Reinforcement learning can address the problem of optimizing the combustion process by tuning fuel fractions in an elegant way. Reinforcement learning can also account for seasonal effects and wear because these effects can be found within sensor readings. Approaches that are based on (fixed) physical models are usually unable to find such effects. To successfully develop a reinforcement learning agent, the problem of state estimation (see chapter 30 [23] for other problem solutions to real world reinforcement learning applications) has to be solved, in order to fulfill the requirements of the reinforcement learning framework. Since a gas turbine provides a vast number of sensor readings, such as temperature and pressure readings, mass flows and actor settings such as valve positions and set points of low level controllers, the state is very high dimensional. Even experts cannot anticipate all effects that could be caused by the various subsystems and their interactions. A state estimation approach can overcome this problem.

Various topologies of recurrent neural networks (RNNs) are used for state estimation as they allow to identify the continuous valued, possibly high dimensional state space of real world applications. RNNs explicitly offer possibilities to account for time and memory, in principle they are able to model any type of dynamical system [9, 15, 11, 32]. Because of these capabilities RNNs are a valuable tool for state estimation, especially for real world applications. Based on these estimates, methods of reinforcement learning can be applied to solve a defined control problem.

The chapter is divided into four parts. After a brief introduction to reinforcement learning and its requirements, the main trick of modeling a Markovian state space using an RNN is described in section 29.3. Section 29.4 proceeds with tricks to extract a Markov decision process from a possibly large set of observables and adapts a neural topology further towards the task of state estimation for reinforcement learning. To address the task to capture different time scales of dynamical dependencies a solution to the long-term dependency problem is provided in section 29.5. For all presented tricks, recipes as practical guides are introduced, to avoid potential pitfalls and to improve general applicability. Further, some experiments to demonstrate the applicability of the presented procedures are presented.

29.2 Background

Reinforcement learning usually requires the system of interest to be described as a Markov decision process (MDP) $M := (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where \mathcal{S} and \mathcal{A} denote the state and action spaces, respectively, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ the state transition probabilities, i.e., the probability of entering a successor state s'_{t+1} by executing an action a_t in state s_t , and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ the reward function assigning a transition its immediate cost or gain. The aim of reinforcement learning is to derive a policy $\pi : \mathcal{S} \mapsto \mathcal{A}$ mapping each state to an action that maximizes the

return, i.e., the sum of all (discounted) future rewards. A central characteristic of an MDP is the Markov property which states that the probability of reaching a certain successor state s_{t+1} depends on the current state s_t and action a_t only.

However, in many real-world control problems the current state is not directly accessible. Instead, only a number of observables $z_t \in \mathcal{Z}$ can be used as source of information about the true current state s_t , rendering the MDP into a partially observable Markov decision process (POMDP) $M' := (\mathcal{S}, \mathcal{Z}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{O})$. In addition to the components of an MDP, a POMDP contains an observation space \mathcal{Z} and an (usually unknown) observation function $\mathcal{O} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{Z}$, describing the mapping from state-action pairs to observations.

In the gas turbine setting, presented in the introduction one also has to deal with a POMDP. Various measurements as well as actor settings like valve positions are provided as observables z . The state of the turbine can be influenced by changing the fuel flow valve positions. As a result, an immediate reward signal is calculated, providing information about the quality of performed actions.

A common approach for dealing with POMDPs is to model a distribution of possible current states [12], i.e., a belief state. When using such approaches, the selection of action a_t is based on the most probable current state and additionally its uncertainty [20]. When dealing with technical systems, the partial observability mostly stems from the limited available measurements and the fact that one time step is insufficient to describe the current system state, i.e., most technical systems can also be described as MDPs of higher order. By simply aggregating a sufficient number of prior time slices, such Markov processes can be reduced to MDPs of first order, i.e., the type of MDPs required by reinforcement learning. In many cases a simple aggregation leads to a high dimensional state space and therefore turns out to be impractical in most cases (Bellman's "curse of dimensionality" [2]). Either a deep understanding of the underlying system or, especially in more autonomous settings, a state estimator can overcome this problem. In the following, recurrent neural approaches to model a Markovian state space from partially observable dynamics are derived. The modeled state allows to apply any well understood powerful reinforcement learning algorithms in the offline setting, such as the neural fitted Q iteration [24], the recurrent control neural network [31], or online approaches such as actor critic algorithms [21].

29.3 The Trick of Modeling a Markovian State Space Using a Recurrent Neural Network

A reinforcement learning agent interacts with a system thereby commonly altering its evolution. This development can be described as an open dynamical system. Note that in comparison to the description found in chapter 28 [36], the adjustable external drives of the system, a are explicitly disjoint from other external drivers z . In practice a dynamical system like a gas turbine can be influenced by adjustable external drives like valve positions of fuel or cooling flows but at the same time, other external driver such as ambient conditions might also be relevant. For discrete time grids ($t = 1, \dots, T$ and $T \in \mathbb{N}$) this can be

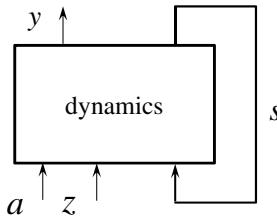


Fig. 29.1. A dynamics can be affected by actions a , performed by a (reinforcement learning) controller but also by external drivers z , e.g., ambient conditions of a gas turbine. Both alter the state s of the dynamics and cause a state transition, resulting in a set of observables y . For a gas turbine, observables are sensor readings such as temperatures or pressure levels as well as actor settings such as valve positions.

represented as a set of equations consisting of a state transition and an output equation [9, 11]:

$$\begin{aligned} s_{t+1} &= f(s_t, a_t, z_t) && \text{state transition} \\ y_t &= g(s_t) && \text{output equation.} \end{aligned} \quad (29.1)$$

Figure 29.1 illustrates equation 29.1. The state transition is a mapping from the present internal state of the system s_t and the influence of external inputs a_t and z_t , to the new state s_{t+1} . In the context of reinforcement learning, the inputs z_t are the agent's (partial) information about the system and a_t represents an action that has been selected by a control strategy. The output equation determines the observable output y_t . In a basic framework, the output is equivalent to the resulting change of the system, in other words the subsequent observables of the system z_{t+1} .

The task of identifying a dynamical system of Eq. 29.1 can be stated as the problem to find (parametrized) functions f and g such that a distance measurement (Eq. 29.2) between the observed data y_t^d and the model output y_t is minimal:

$$\sum_{t=1}^T (y_t - y_t^d)^2 \rightarrow \min_{f,g} \quad (29.2)$$

The identification task of Eq. 29.1 and 29.2 can be formulated by an RNN of the form

$$\begin{aligned} s_{t+1} &= \tanh(As_t + c + Bz_t + Ca_t) && \text{state transition} \\ y_t &= Ds_t && \text{output equation} \end{aligned} \quad (29.3)$$

where A , B , C , and D are weight matrices of appropriate dimensions and c is a bias.

By approximating the functions f and g with an RNN using the weight matrices A , B , C , D , and a bias vector c , of fixed dimensions, the system identification task of Eq. 29.2 is transformed into a parameter optimization problem:

$$\sum_{t=1}^T (y_t - y_t^d)^2 \rightarrow \min_{A,B,C,D,c} \quad (29.4)$$

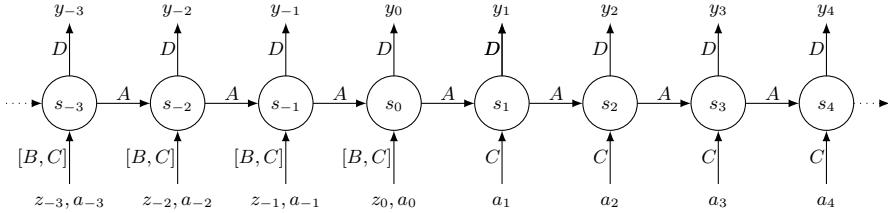


Fig. 29.2. A standard RNN used as state estimator during the training phase. Each circle represents a hidden layer of neurons (called cluster). Each arrow depicts a weight matrix, connecting all neurons between two layers. Note that all hidden clusters as well as all output clusters are connected to a bias (not shown in the figure).

This parameter optimization problem is solved by an RNN with finite unfolding in time using shared weight matrices A, B, C , and D [22, 9]. Fig. 29.2 depicts the resulting spatial neural network architecture. The RNN is trained with backpropagation through time which is a shared weights extension of standard backpropagation [22, 9]. The training of an RNN using shared weights is straightforward and delivers sound results. RNNs are—other than feed forward networks—able to establish memory due to the temporal structure with unfolding towards the past and the future. This allows to model inter-temporal dependencies and latent variables. E.g., the emissions of a gas turbine are delayed multiple steps because of sensor response characteristics and the distance between causing the emissions (within the combustion chamber) and the point of measurement (within the exhaust gas flow).

In order to map the state estimation to the RNN the hidden units s of the RNN (Fig. 29.2) describe the state variables for the reinforcement learning task. In other words, after training, these variables are used as outputs of the state estimation function. The externally available observables and the performed action are used as input vector z_t and action vector a_t , $y_t^d = z_{t+1}$ defines the targets.

The RNN is extended into the future by so-called overshooting [35], i.e., the network is unfolded beyond the present time into the future (Fig. 29.2). This results in a whole sequence of forecasts as outputs. Overshooting regularizes the learning and thus improves the model's performance [35]. Overshooting does not require additional network parameters as shared weight matrices A, B, C , and D are used.

The resulting trained network models the state transition of the underlying dynamical system with respect to the sequence of actions. If the system is able to model the forecast horizon sufficiently well, the Markov property for the hidden cluster s_0 is arbitrarily well approximated [30, 27]. Therefore a sufficiently well trained RNN results in a state estimator for a subsequent use in reinforcement learning algorithms [29].

In this section the basic idea of using an RNN for state estimation was introduced. Subsequent to the trick of using RNNs as state estimators, a series of recipes to overcome various problems when designing neural state estimators

will be presented. First, the important role of actions on the state estimation task and the requirement to generalize over all actions within the action space (Sec. 29.3.1, 29.3.2) is discussed. Next, data and pattern preprocessing methods to provide a suitable training and test set (Sec. 29.3.3, 29.3.4, 29.3.5) are introduced. Thereafter, the learning process is introduced (Sec. 29.3.6, 29.3.7). Finally, the reduced recall-topology of a trained state estimator (Sec. 29.3.8) is presented. The applicability of the approaches is illustrated on the cart-pole problem in Sec. 29.3.9.

29.3.1 Improving the Generalization Capabilities with Respect to Actions

In contrast to the standard definition of an open dynamical system and the description of overshooting [35], the known sequence of actions is used as past and future inputs $a_p, a_{p-1}, \dots, a_0, a_1, \dots, a_f$, where p describes the number of past and f the number of future steps (Fig. 29.2). This is crucial, since the data might have been generated by an arbitrary, possibly unknown control strategy. The resulting network should be capable to model the dynamics of the system based on the influence of these external drivers a_t , rather than predict a sequence of actions from the Markovian state. E.g., the data of a gas turbine might be generated by an arbitrary control strategy. If this effect is neglected, the state estimator is forced to encode the underlying policy within the network. The resulting network is most likely useless since it does not generalize to different action sequences as they occur when following a reinforcement learning policy. In general, it is also recommended to test a trained state estimator for its generalization capabilities towards different actions sequences. This is possible for most systems. E.g., for the gas turbine certain actions are known to reduce emissions, applying them to the turbine should lead to a change of NOx or CO.

29.3.2 A Recipe to Improve the Modeling of State Transitions

The standard RNN shown in Fig. 29.2 uses a single hidden cluster of neurons to encode a state s , the state transition is modeled by the matrix A . In case of state estimation for reinforcement learning we are interested in modeling a state transition, i.e., $s_t \xrightarrow{a_t} s_{t+1}$. This can be explicitly expressed in the neural topology, improving the overall performance of the estimator. Especially generalization capabilities with respect to different actions are improved. Note that a good state estimator not only minimizes the training error of the optimization task described in Eq. 29.4, but also generalizes well to different sequences of actions (Sec. 29.3.1). The neural representation of a state transition is shown in Fig. 29.3.

29.3.3 Scaling of Inputs and Targets

Like in other supervised learning approaches scaling of input and target values is an essential preprocessing step. Usually input scaling to receive a mean of

zero and a standard deviation of one is preferred. Other scaling approaches can be used as well, e.g., to encode prior knowledge about a dynamics. E.g., for concentrations like the emissions of a gas turbine, a logarithm can be used to avoid negative, implausible output values.

29.3.4 Block Validation

As the patterns for an RNN comprise a whole sequence of input and target values, subsequent patterns are highly correlated. Each pattern differs from the preceding one only by a single time slice. Consequently, assigning patterns to the training (used to update weights) or validation set (not used to update weights but to evaluate the performance of the current set of weights) at random would lead to a strong correlation between training and validation set and hence the training and validation error. In order to de-correlate the pattern sets an increased granularity of the partitioning is achieved by using blocks of patterns that do not contain overlapping information. A block is either used for training or testing. Ideally, sets with identical statistical properties are generated.

The usage of blocks is motivated by the occurrence of new *situations*. E.g., a gas turbine might operate at full load and change to a reduced load level. After a transient period, the turbine reaches the new load level. Such processes take minutes, whereas the controller is designed to perform an action every couple of seconds. By grouping the patterns to blocks, the sets are de-correlated and still capture all operational *situations*. Choosing an appropriate block size for the given problem is essential for good results and a reliable test set. If one would simply split a data set of, e.g., two days of operation into two sets, a complete day each, both sets could have significantly different statistical properties and a set might easily miss some operational *situations*.

The random block validation algorithm (Alg. 29.1) addresses these problems. The algorithm uses a data file containing M observations, i.e., M time slices of data vectors. First the size of a block j is determined randomly within pre-defined limits minblocksize and maxblocksize . Next j patterns are generated from subsequential time slices of observations. Each pattern is unfolded over its entire time range. In other words, if the topology defines p past time slices and f future

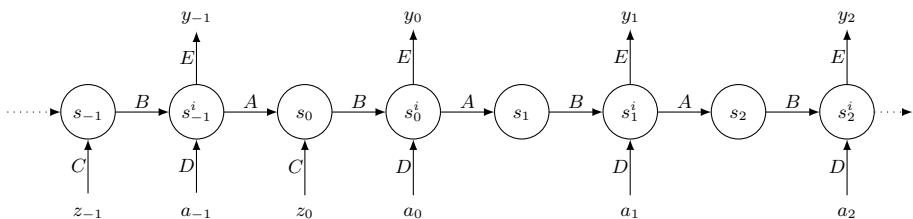


Fig. 29.3. An RNN modeling explicit state transitions depending on an applied action: $s_t \xrightarrow{a_t} s_{t+1}$. Note that all hidden clusters as well as all output clusters are connected to a bias (not shown in the figure).

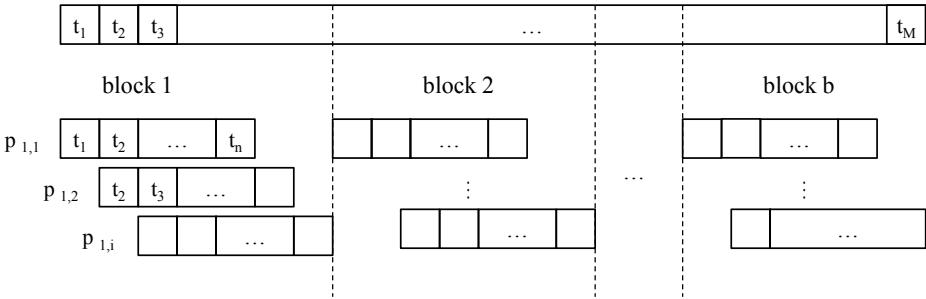


Fig. 29.4. Pattern generated with the block validation procedure. All blocks are de-correlated to provide reliable, uncorrelated test sets (i.e., for validation and generalization tests).

time slices, the entire pattern covers $p + f$ observations. Further each pattern is scaled (see Sec. 29.3.3). The process is repeated until the block is filled or one runs out of observations. Finally the type (training or test set) of a block is determined according to predefined probabilities. A complete block is added to the final pattern set. The resulting generated patterns are organized as shown in Fig. 29.4. The resulting data set fulfills our requirement of a set of patterns, where each pattern set (training set and test set) are de-correlated but have similar properties, since the different operational *situations* are distributed over all sets.

29.3.5 Removal of Invalid Data Patterns

In all applications with training data unfolded in time the problem of invalid patterns can occur. An invalid data pattern contains data that does not fit into a predefined time grid. E.g., every time the gas turbine is restarted, the stream of data is restarted as well and causes a gap. Any pattern that is generated and contains data from the previous shutdown, probably a couple of hours ago, and the new start-up would be invalid. It does not describe the behavior of the turbine and should therefore be removed from the used patterns.

A valid pattern contains, for each step unfolded in time, inputs that match the defined time grid. E.g., if a time grid of $t = \tau$ is selected, a valid pattern unfolds over n equidistant steps $t = \tau, 2\tau, \dots, n\tau$, where n defines the number of unfolded steps of the pattern. Any pattern that has gaps in the time line should be excluded to avoid invalid training patterns. This problem occurs in many real world applications but also for episodic benchmark problems such as a cart-pole or the acrobot [26].

In practice this task can be solved by simply extending the block validation algorithm (Alg. 29.1). Before the subroutine: `generate($tm, \dots, tm + n$)`, each data vector is tested to match the defined time grid. All data vectors violating the grid are ignored.

Algorithm 29.1 . Random block validation

Input: set of observations $\mathcal{O} = \{t_m = (z_m, a_m, r_m) | m = 1, \dots, M\}$
Result: block validated set of patterns for training and testing
 $m := 1$
while $m \leq M$ **do**

- $j := \text{rand}(\min_{\text{blocksize}}, \max_{\text{blocksize}})$ ▷ determine the block size within given limits
- $i := 0$
- while** $(i < j) \wedge (m + n) \leq M$ **do** ▷ generate a block of patterns, containing j patterns, where each pattern covers n time slices, break if insufficient data available
- $\text{pattern} := \text{generate}(t_m, \dots, t_{m+n})$ ▷ generate a valid pattern using the selected scaling
- $\text{block.add}(\text{pattern})$
- $i := i + 1$
- $m := m + 1$

end while

$\text{type} := \text{rand}(p_{\text{training}}, p_{\text{test}})$ ▷ choose the type of the new block according to the specified probabilities p for training and test set

$\text{patternSet.add}(\text{block}, \text{type})$ ▷ Add valid block to final pattern set

$m := m + n$ ▷ Skip data to avoid overlapping pattern, n depicts the network's unfolding in time

end while

return patternSet

29.3.6 Learning Settings

We made good experience using standard backpropagation on the described neural network topologies. This effectively realizes backpropagation through time for a fixed horizon, i.e., over all past and future time slices covered by the neural network. Robust learning is achieved by small randomly chosen batches of patterns with a batch size B of, e.g. $B = 1/1000 N$, where N is the number of all patterns. The weights updates Δw are calculated as

$$\Delta w = -\eta \frac{1}{B} \sum_{i=1}^B \partial E_i / \partial w, \quad (29.5)$$

where η is the learning rate and E_i is the error for pattern i . The learning rate is chosen between 0.01 and 0.001 in most cases.

In many applications the learning process can be sped up significantly by an extension to standard backpropagation called VarioEta [18]. VarioEta scales the weight updates individually by the standard deviation over all patterns

$$\sigma_w = \sqrt{\frac{1}{N} \sum_{i=1}^N (\partial E_i / \partial w - D_w)^2}, \quad (29.6)$$

where

$$D_w = \frac{1}{N} \sum_{i=1}^N \partial E_i / \partial w \quad (29.7)$$

leading to the VarioEta update rule

$$\Delta w' = -\eta \frac{1}{B} \sum_{i=1}^B \partial E_i / \partial w \frac{1}{\sigma_w}. \quad (29.8)$$

29.3.7 Double Rest Learning

The training process of RNNs with shared weights has proven empirically to be reliable and robust. Nonetheless our goal is to get the best training result without wasting time on unnecessary network training. For this purpose we developed the double rest learning procedure (Alg. 29.2).

In principle the same network is trained multiple times on a constant data set but different random initializations of the initial neural weights. If no better network was found after multiple trials of training, the process terminates. A set of pattern that contains a training set and at least one test set used for validation is required. The algorithm is initialized with a maximum number of trials, t_{\max} and

Algorithm 29.2 . Double rest learning

Input: maximal epochs per training e_{\max} , maximal training trials t_{\max} , rest epochs e_{rest} , rest trials t_{rest}

Result: trained neural network

```

 $t := 0$ 
 $t_{\text{total}} := 0$ 
 $v := \infty$ 
while  $t < t_{\text{rest}} \wedge t_{\text{total}} < t_{\max}$  do
    net := restLearning( $e_{\text{rest}}, e_{\max}$ )
     $v_t := \text{net.min}()$ 
    if  $v_t < v$  then
         $v := v_t$ 
        bestNN := net
         $t := 0$ 
    end if
     $t := t + 1;$ 
     $t_{\text{total}} := t_{\text{total}} + 1$ 
end while
return bestNN

```

the number of trials one wants to try to find a better solution, t_{rest} . For the rest learning algorithm (Alg. 29.3) the maximum number of epochs allowed for training a single network, e_{max} and the number of epochs a single network is trained beyond the epoch that achieved the best validation error e_{rest} is required.

Double rest learning starts training single networks using the rest learning algorithm (Alg. 29.3). Whenever a new network is found to have a better validation error than a previous one, the number of rested trials t is reset to 0. The algorithm terminates as soon as no better network was found for t_{rest} trials or the maximum number of trials t_{max} is exceeded.

The rest learning process for a single network follows a similar approach (Alg. 29.3). The algorithm allows us to select the best network according to a validation set. A single network is trained for e epochs. If a better solution is found within e_{rest} , the epoch i with the best validation error is determined. The best set of weights, bestNN , from epoch i is stored, and the network is trained for another $e_{\text{rest}} - i$ epochs. If the validation error did not improve for e_{rest} epochs or the maximum number of training epochs e_{max} is exceeded, the training process terminates.

Algorithm 29.3 . Rest learning

```

Input: maximal epochs per training  $e_{\text{max}}$ , number of rest epochs  $e_{\text{rest}}$ 
Result: trained neural network  $t := 0$   $v_{\text{min}} := \infty$   $t_{\text{total}} := 0$   $e := e_{\text{rest}}$ 
while  $t < e_{\text{rest}} \wedge t_{\text{total}} < e_{\text{max}}$  do
    net.learn( $e$ )                                 $\triangleright$  train the network for  $e$  epochs
     $t_{\text{total}} := t_{\text{total}} + e$ 
     $v := \text{net.min}()$                           $\triangleright$  retrieve the minimal validation error
     $i := \text{net.index}(v)$                        $\triangleright$  retrieve the epoch resulting
                                                in the minimal validation error
    if  $v < v_{\text{min}}$  then
         $v_{\text{min}} := v$ 
         $\text{bestNN} := \text{net.weights}(i)$            $\triangleright$  retrieve the network with the weights,
                                                resulting in the minimal validation error
         $t := i$ 
         $e := e_{\text{rest}}$ 
    else  $t := t + i$   $e := e_{\text{rest}} - i$ 
    end if
end while
return  $\text{bestNN}$ 

```

Double rest learning can be applied to any neural network training where one wants to select the best network from multiple trials using a robust stopping criterion. It is especially useful when the meta-parameters of the training process are subject to optimization and are modified at each trial³. Another advantage

³ In case of identical meta-parameters for all trials, it is adequate (and even more efficient) to use a fixed number of trials, i.e., $t_{\text{max}} = t_{\text{rest}}$.

is the parametrization of the algorithm. Even though there are four parameters, they can be assumed to be constant. For most applications we use: $e_{\text{rest}} = 50$, $e_{\max} = 5000$, $t_{\text{rest}} = 10$, and $t_{\max} = \infty$. This parametrization has shown robust results for a wide range of applications and 500 to 50000 training patterns.

29.3.8 A Recipe to Generate an Efficient State Estimation Function

After training an RNN the neural network is truncated to receive a function of the form:

$$s_t = f(z_t, z_{t-1}, \dots, z_{t-n}, a_t, a_{t-1}, \dots, a_{t-n}), \quad (29.9)$$

where n denotes the length of the considered history of the network. The optimal length n can be found by subsequently reducing the length of the past network until the error of the forecast, i.e., for outputs y_0, y_1, \dots, y_m , increases. An instantaneously increasing error either indicates a perfect choice of n or an insufficient history. Another training with an increased past horizon ensures that all prior required information is provided to the neural network.

The topology of the resulting network is shown in Fig. 29.5. This function can be used to transform observations into tuples of $\{s, a, s'\}$ and apply a reinforcement learning method of choice to solve the control problem.

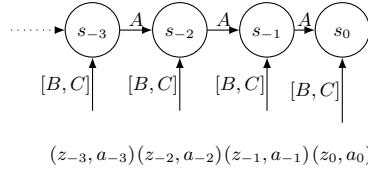


Fig. 29.5. A finalized state estimator after training. The former hidden unit s_0 is now used as output of the function and provides a Markovian state representation for a given sequence of observations and actions. Note that all hidden clusters are connected to a bias (not shown in the figure).

29.3.9 Application of a Neural State Estimator

To demonstrate the capabilities of an RNN as state estimator, we chose the cart-pole problem, which has been extensively studied in control and reinforcement learning theory. Since more than 30 years it serves as a benchmark for new ideas, because it is easy to understand and also quite representative for related questions. The classical problem has been completely solved in the past. E.g., Sutton [26] demonstrated that the pole can be balanced for an arbitrary number of time steps within a remarkable short training sequence. There are two major directions to make the cart-pole problem more challenging. One is to make the task itself more difficult by taking for example two poles [8] or regarding a two dimensional cart [7]. The other one is to make the original problem partially observable [17, 1, 8].

We will focus on the latter, since all other variants provide a Markovian state representation in the first place, and therefore do not require state estimation. A reinforcement learning solution to the original (simulated) benchmark problem, as well as to a real cart-pole system can be found in chapter 30 [23].

Problem Description. The classical cart-pole problem consists of a cart moving on a bounded track and trying to balance a pole on its top. This cart-pole system is illustrated in Fig. 29.6 [17].

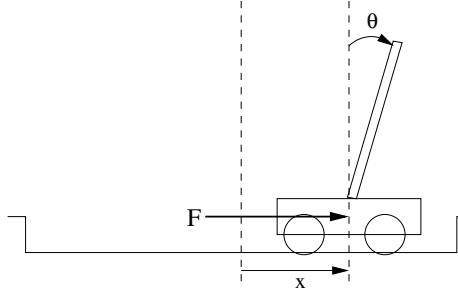


Fig. 29.6. The cart-pole problem system

The system is fully defined through four variables ($t = 1, \dots, T$):

$$\begin{aligned} x_t &:= \text{horizontal cart position} \\ \dot{x}_t &:= \text{horizontal velocity of the cart} \\ \theta_t &:= \text{angle between pole and vertical} \\ \dot{\theta}_t &:= \text{angular velocity of the pole} \end{aligned} \tag{29.10}$$

The goal is to balance the pole for a preferably long sequence of time steps without moving out of the limits. Possible actions are to push the cart left or right with a constant force F . The pole tilts when its angle θ_t is higher than 12 degrees. Either then or when the cart hits one of the boundaries, the system is punished with a negative reinforcement signal. In all other cases the reward is zero.

As already mentioned the system has been extensively studied in its several forms. When the system was studied as partially observable, one usually omitted the two velocities, \dot{x}_t and $\dot{\theta}_t$, i.e., only the cart's position and the angle between the pole and the vertical were given as inputs [17, 1, 8]. Solving this problem is not difficult because the model or algorithm just needs the memory of one past time step to calculate the missing information.

To demonstrate the advantages of RNNs (unfolded in time) only the horizontal position of the cart, x_t is observable [29]. All other information is unknown to the system.

Model Description. To solve the problem described above, an RNN (Sec. 29.3) was used to develop the full dynamics of the cart-pole system. Input z_t and target y_t^d consist of the horizontal cart position x_t as well as the preprocessing transformations $x_t - x_{t-1}$. The input a_t contains the agent's action. No other information is observable by the model. The internal state space s_t is limited to four neurons, allowing the network to reconstruct the complete but only partially observable dynamics (Eq. 29.10) in its internal state space. The network is unfolded ten time steps into the past and future. Preceding experiments have shown that this memory length is sufficient to identify the dynamics. To make the network independent from the last unfolded time slice a technique called cleaning noise is used as a start initialization [10]. The network is trained by backpropagation through time [22, 9].

In a second step the evolved state space is extracted from the RNN, i.e., a state estimation function was exported to calculate estimated states from observations (see Sec. 29.3.8). Then a generalized form of Samuel's adaptive heuristic critic (AHC) algorithm [25] was used to solve the control problem based on the state estimates. Note, that the algorithm has to be started with an already filled lag structure, i.e, the past lags of the state estimator need to be filled with observations in order to provide a first state estimate. Otherwise there is a high probability that the algorithm is faced with a tilted pole in its first learning step, as a minimum of ten uncontrolled time steps would be necessary to fill all the lags.

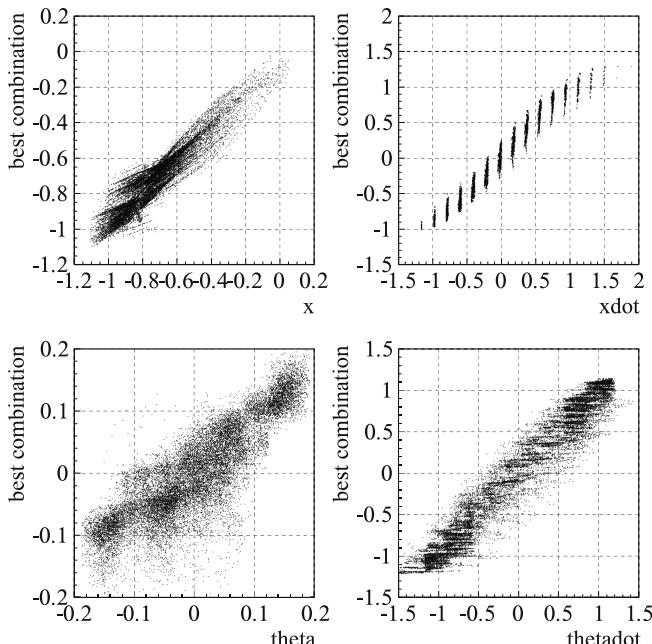


Fig. 29.7. Correlation between the best quadratic combination of the reconstructed state space variables $(s_t)_1, \dots, (s_t)_4$ of the RNN and the original ones (Eq. 29.10)

Results. As a first result the estimation quality of the different state variables of the cart-pole is illustrated in Fig. 29.7. The four plots show the correlation between the original state space variables of the dynamics, $x_t, \dot{x}_t, \theta_t, \dot{\theta}_t$, (Eq. 29.10) and the best linear combination of the reconstructed state space variables $(s_t)_1, \dots, (s_t)_4$ and their squares $(s_t)_1^2, \dots, (s_t)_4^2$ in each case. The high correlation for each state space variable demonstrates the reconstruction quality of the RNN. It also supports the use of RNNs for partially observable reinforcement learning problems.

We compared the results of our approach to a direct application of the AHC algorithm to the problem, i.e., without using an RNN in the first step. Note, that no adaptive binning has been used. In both cases the discretization of the state space was chosen to yield the best results.

Fig. 29.8 plots the achieved number of steps, the pole could be balanced, to the number of trials. The training process was stopped as soon as the first method was able to balance the pole for a minimum of 1000 steps. Fig. 29.8 shows how the RNN approach outperforms a direct application of the AHC algorithm.

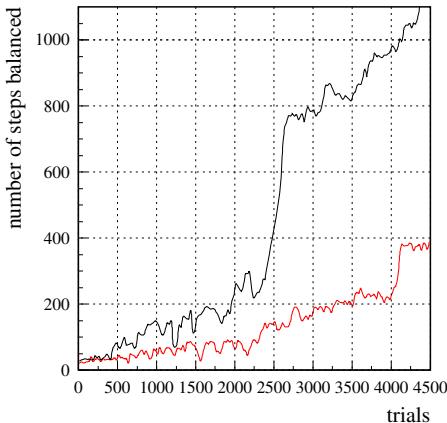


Fig. 29.8. Comparison of the performance in the partially observable cart-pole problem of our RNN approach (upper curve) to a direct application of the AHC algorithm (lower curve)

29.4 The Markov Decision Process Extraction Network

When applying reinforcement learning algorithms to real world problems such as a gas turbine, one not only faces the problem of observations that do not provide the Markov property but in many applications it is not even obvious which variables to consider for the state definition. E.g., the measurements provided by a gas turbine include information about various subsystems. In practice it is often hard to determine if a certain variable has to be considered or a whole set of variables from a certain subsystem can be dismissed.

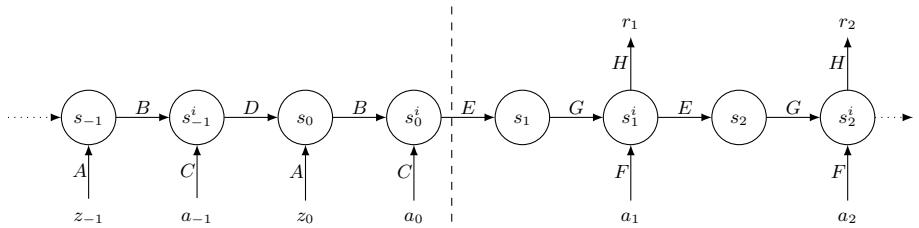


Fig. 29.9. The Markov decision process extraction network (MPEN) consists of a past (left) and a future (right) subnetwork. The input variables are split into two groups: Actions a_t are controllable by the reinforcement learning agent, z_t denote observable variables from the dynamics. The future subnetwork has outputs r_t only. State transitions are modeled by s_t^i as well as s_t . Note that all weight matrices in the past (A, \dots, D) are different from the future matrices (E, \dots, H).

In addition to considering expert knowledge and methods for input selection we developed an approach, designed for optimal control problems, where a reward signal is available. From a reinforcement learning point of view, the performed action a_t , a possibly large set of observables z_t as well as the reward r_t for the given tuple of observation and action is available for all steps in time t . In many real world scenarios, the reward function $r_t = f_r(z_t, a_t)$ is known, because it describes the desired goal of the problem to be addressed and was most likely designed by us. This knowledge can be modeled in an advanced neural state estimator by splitting the network into past and future subnetworks to match following equations:

$$s_{t+1} = f_{\text{past}}(s_t, z_t, a_t), \quad t \leq 0 \quad (29.11)$$

$$s_{t+1} = f_{\text{future}}(s_t, a_t), \quad t > 0. \quad (29.12)$$

The split of the network addresses an inconsistency within the topology of the RNN since observables z are not available for future time slices ($t > 0$). Another topology to address this problem would be the dynamically consistent recurrent neural network [34]. This topology however must predict all observables z which can particularly cause problems when some variables are harder to predict than others. Since the split into past and future allows us to use any target, variables not included in the input set can also be considered. This allows to use the reward signal or, in case of a known reward function, the arguments of the reward function as targets:

$$r_t = g(s_t, a_t), \quad t \geq 0 \quad (29.13)$$

Note that the current state s_t and the applied action a_t are sufficient to describe the expectation value of all relevant reward functions, because the entire

information about the successor state's s_{t+1} probability distribution must be included in these two arguments. This target allows us to change the objective of the neural network to the desired dynamics:

$$\sum_{t=1}^T (r_t - r_t^d)^2 \rightarrow \min_{f,g} \quad (29.14)$$

The resulting topology allows the neural network to accumulate all information required for the Markov property from the provided past observations in the past network, while the future network remodels the state transitions.

It is also possible to encode prior knowledge about the distribution of relevant information over the considered past horizon by adapting the topology of the past network accordingly. A possible modification could be to add additional inputs directly to the hidden cluster s_0 , for instance if they are known to be constant over the considered past horizon. E.g., the ambient temperature, pressure, and humidity for the combustion process of a gas turbine. Since these variables are constant over the considered unfolded past network, they can directly be fed to the cluster s_0 . The only constraint for any input of past information is that it has to be passed through the hidden cluster s_0 which is ultimately used as the output of the Markovian state representation. Therefore, possibilities for topologies of the past network are countless.

The representation of these findings leads to the Markov decision process extraction network (MPEN) shown in Fig. 29.9. The subnetwork on the left (past network) has no explicit target and provides information to the neural branch on the right (future network). The two subnetworks are connected via an arbitrary neural structure, e.g., a weight matrix or a multi-layer perceptron. The future network uses the information provided by s_0 as well as the future actions to learn a system's dynamic capable to predict the sequence of future rewards. Note that future actions are important inputs preventing the network to predict the sequence of actions which induced the state transitions. This is important because action selection can be based on external information which is not included in the set of observables, or might even be unpredictable due to random exploration. See Sec. 29.3.1 for more details regarding this issue.

As proven in [27], an RNN can be used to approximate an MDP by predicting all expected future successor states based on a history of observations. The structure of an RNN forces each hidden cluster s to encode all necessary information to estimate a successor state with respect to the influence of an action. For this reason an RNN must be capable to estimate the expected rewards for each future state because a reward function can only use a state, action, and the resulting successor state as arguments. Therefore, it is sufficient to model a dynamical system predicting the reward for all future time slices. Based on this conclusion, the approach is designed to model the minimal required dynamics of a regarded system [4].

The major advantage of the introduced neural topology over other RNN based state estimators is the capability to model a minimal dynamics from a set of observables without manual selection of variables. A further advantage is the

capability to extract a minimal state space. Networks that need to forecast all observables such as the dynamically consistent RNN [34] encode all information into the state space and are therefore not minimal. This is of special interest if the set of observables contains unimportant variables that are possibly difficult to predict and therefore cause a drop in forecast performance. Additionally, such variables interfere with the training process, since the validation error can be highly influenced by these variables. In other words, resulting largest residuals causing the training result of the entire neural network to be less robust.

29.4.1 Reward Function Design Influences the Performance of a State Estimator

In contrast to the standard RNN, the difficulty to train the neural topology depends on the reinforcement learning problem itself since the definition of the reward function provides the target. We could show that for episodic problems like a cart-pole it is possible to train the network with the standard reward function that gives some positive or negative feedback at the end of trajectories [4]. However, the problem becomes much easier to solve when more information about the quality of the current state is provided. This is reflected in additional gradient information speeding up the learning process. For most real world applications, such a reward function can be easily provided since we usually participate in its design. In case the reward function is known, the arguments of the reward function are usually preferred to be used as targets.

29.4.2 Choosing the Forecast Horizon of a State Estimator

When designing a neural state estimator network, the question about the number of future steps, that should be included into the prediction, arises immediately. Since the networks are unfolded in time towards the past and the future for a limited number of steps, a practical answer to that problem has to be found. Fortunately, the definition of the reinforcement learning problem itself provides an answer. Using a discount factor γ to define the return, limits the significant horizon of future time steps. The return, defining the performance of an agent's strategy, is defined by:

$$\sum_{t=1}^{\infty} r_t \gamma^t. \quad (29.15)$$

In practice one can safely limit the number of future time steps accordingly, since the impact of the rewards decreases exponentially.

29.5 The Trick of Addressing Long Term Dependencies

Most state estimation approaches rely on Takens's theorem [33] which states that a sufficient number of past time slices contain all information necessary to

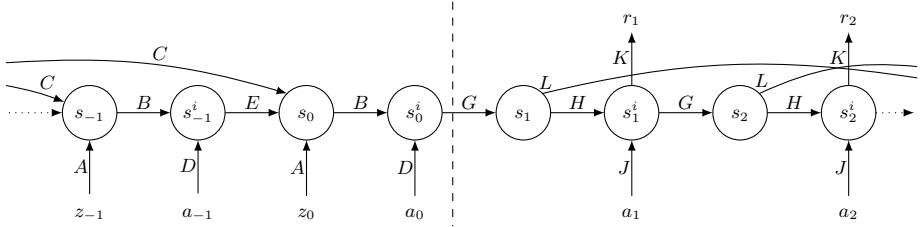


Fig. 29.10. The Markov decision process extraction network with shortcuts (MPEN-S) consists of a past (left) and a future (right) subnetwork. The input variables are split into two groups: Actions a_t are controllable by the reinforcement learning agent, z_t denote observable variables from the dynamics. The future subnetwork has outputs r_t . State transitions are modeled by s_t^i as well as s_t . Note that all weight matrices in the past (A, \dots, E) are different from future matrices (G, \dots, L). All hidden clusters as well as all output clusters are connected to a bias (not shown in the figure).

estimate a Markovian state. Recurrent state estimators show remarkable performance in various applications such as state estimation for gas turbine control [28], but despite the advantages of RNNs there have been concerns regarding their capability to model long-term dependencies [3]. In a system exhibiting long-term dependencies the system's output at time T is dependent on the input at time $t \ll T$ [14]. The problem was discovered by Mozer [16] who found RNNs to be unable to capture global effects in classical music. The main reason for this effect are vanishing gradients in gradient-based learning methods [6, 14]. Long-term dependencies occur in many time series ranging from technical systems to financial data. To overcome this issue the Markov decision process extraction network with shortcuts (MPEN-S) [5] is introduced. It is an extension of the previously introduced MPEN topology (Sec. 29.4), where additional shortcuts connect clusters across a fixed number of time slices (Fig. 29.10). Examples are shortcuts of length n in a network with p steps in the past and f steps into the future part, that connect $s_{-n} \rightarrow s_0$, $s_{-n+1} \rightarrow s_1$, ..., $s_p \rightarrow s_{p+n}$. In the future part, shortcuts of the same length are added from $s_1 \rightarrow s_{n+1}$, $s_2 \rightarrow s_{n+2}$, ..., $s_{f-n} \rightarrow s_f$.

The resulting topology is successfully used for state estimation problems that face the problem of delayed observables or actions that show a delayed effect on a dynamics. For instance, gas turbine emission measurements are delayed by about one to two minutes, whereas the combustion dynamics of the turbine occurs almost instantaneously. Both effects are influenced by the same action applied to a turbine, therefore the underlying state contains information about both, the highly delayed as well as the short term effect.

After presenting a recipe to find a good shortcut length in Sec. 29.5.1, we present experiments to demonstrate the capabilities of the MPEN-S topology. Experiments include a gas turbine simulation with highly delayed effects of action on the dynamics.

29.5.1 A Recipe to Find a Good Shortcut Length

For selecting an adequate shortcut length n , the following heuristic can be considered: The severity of the vanishing-gradient problem is correlated with the number of steps information has to be forwarded within the network. Therefore, the value n is chosen to minimizes the total number of steps information has to travel from any state in the past to the current state s_0 . I.e., $\sum_{i=1}^p \text{steps}(s_0, s_{-i}, n) \rightarrow_n \min$, where $\text{steps}(s_0, s_{-i}, n)$ gives the minimum number of steps to travel from s_{-i} to s_0 , including possible shortcuts. E.g., if $n = 2$, $\text{steps}(s_0, s_{-1}) = 1$, $\text{steps}(s_0, s_{-2}) = 1$, $\text{steps}(s_0, s_{-3}) = 2$, $\text{steps}(s_0, s_{-4}) = 2$. The only information required for this heuristic is the maximum number of past time slices that are assumed to influence the current state. Fig. 29.11 shows results from experiments with different shortcut lengths indicating that the heuristic leads to reasonable results.

29.5.2 Experiments on Long Term Dependencies Problems

To demonstrate the capabilities of the MPEN-S two benchmarks are used, a sequence of random numbers as well as a gas turbine simulation. We compare the MPEN-S with shortcuts of length $n = 4$ to the MPEN. The shortcut length $n = 4$ was chosen according to a heuristic (Sec. 29.5.1). For each architecture and benchmark, 10 networks are trained by online backpropagation with a learning rate of 0.001 on 10,000 observations, of which 30% are used for validation. Evaluation is based on another set of the same size but without noise. To judge the quality of the state estimate, represented by the activation values in s_0 , we test the content of information within s_0 . To do so we feed the estimated states as inputs to a feed forward neural network (two hidden layers with 150 neurons

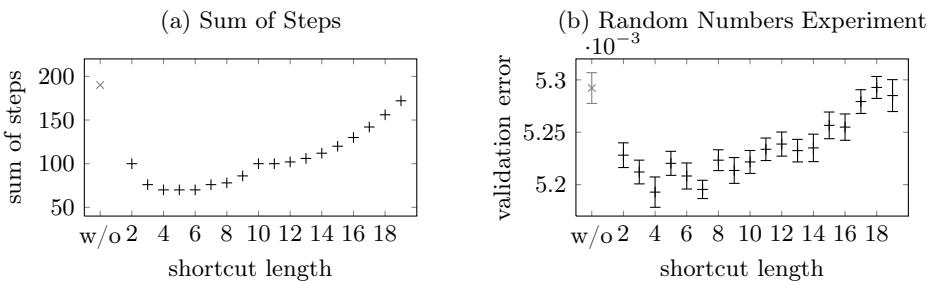


Fig. 29.11. Results from experiments with different shortcut lengths and a past of $p = 20$. (a) shows the sum $\sum_{i=1}^p \text{steps}(s_0, s_{-i}, n)$ of a network without shortcuts and networks with shortcuts of different lengths ($n \in \{4, 5, 6\}$ minimizes the sum). (b) shows the validation errors of these networks for a random numbers experiment (Sec. 29.5.2). The correlation between the sum of steps and the validation error is obvious.

each) whose targets are the true Markovian states of the benchmark applications. In the best case, the estimated states include all relevant information and consequently allow a perfect mapping to the true Markovian states, i.e., the correlation between the target and output is 1.

Random Numbers Experiment. In a first experiment, sequences of equally distributed random numbers $x_i \in [0, 1]$ are used. The network with a past and a future horizon of i steps receives a sequence $x_t, x_{t+1}, \dots, x_{t+i}$ as inputs in the past part of the network. The sequence is also used as targets for the future part of the network, introducing a delay between input and corresponding output. This way, the network has to output an input given at time step t at time step $t+i$ to minimize its error. In addition, equally distributed noise $e \in [-0.05, 0.05]$ is added to the target values for training. The goal of the state estimation for the random numbers problem is to encode information about the past i random numbers.

Gas Turbine Simulation. To demonstrate the capabilities of the presented approach on a problem similar to the real-world application of our interest, a gas turbine simulation is used. The simulation provides a controllable variable *pilot* affecting the emissions and humming of the turbine. The pilot fraction is one of the most important fuel fractions regarding the combustion control. In order to keep the simulation as simple as possible, the combustion tuning is reduced to this variable. The goal of a strategy is to minimize emissions and avoid critical humming, which is reflected in the reward function. While humming reacts instantaneously, the emissions, like in real world, have a long delay and a defined blur over several steps in time. Each step, the simulator provides observations about its current state. The only additional information for the state estimation model is the maximal expected delay d of the simulation. The goal of the state estimator is to encode all relevant information about the past d steps.

Results. Fig. 29.12 illustrates the correlation of the estimated and true Markovian states of the random numbers experiment ((a) and (b)) and the results of the gas turbine simulation experiment ((c) and (d)). Both benchmark results indicate that the MPEN approach is capable of estimating the Markovian states well for small delays. For longer dependencies however, the approximation quality drops significantly, while the MPEN-S can maintain its performance. Table 29.1 shows the average correlation and validation errors of all state variables. The numbers show that the MPEN-S outperforms the MPEN both in reconstruction quality of the Markovian state (resulting in a better correlation) as well as raw forecast performance (lower validation error). The validation error is a good indicator for estimation quality, which is especially relevant for real-world applications.

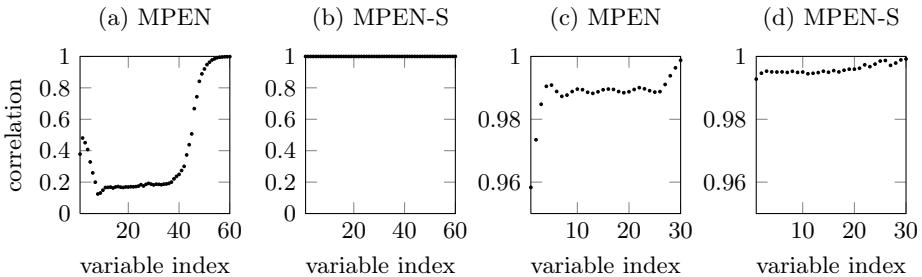


Fig. 29.12. Average correlations of true and estimated Markovian state. A higher variable index indicates a variable closer to the present. (a) and (b) show the estimation quality for the random numbers problem with a delay of 60 steps. (c) and (d) illustrate the estimation quality for the gas turbine simulation with a delay of 30 steps.

Table 29.1. Comparison of the MPEN and the MPEN-S in terms of validation error (MSE) as well as correlation between true and estimated Markovian state

	architecture	experiment	delay	correlation	validation error
MPEN		random numbers	60	0.4	0.9
		gas turbine	30	0.988	0.0160
MPEN-S		random numbers	60	0.99	0.012
		gas turbine	30	0.995	0.0058

29.6 Conclusion

In this chapter we introduced methods for neural state estimation, particularly for reinforcement learning applications. However, the methods can also be applied to other applications that require a Markovian state representation. Further, a set of practical recipes to overcome problems especially relevant to real world applications was presented.

Sec. 29.3 describes the usage of recurrent neural networks (RNNs) as state estimators. Thereafter, a series of recipes to improve the applicability and to overcome possible pitfalls are introduced. A partially observable cart-pole is used to demonstrate the approach in Sec. 29.3.9. In Sec. 29.4, the idea of state estimation based on standard RNNs is further developed towards the Markov decision process extraction network (MPEN). This topology is dynamically consistent and is capable to autonomously model a minimal Markov decision process (MDP) from a large number of observables. Finally, long-term effects are addressed by the Markov decision process extraction network with shortcuts (MPEN-S) in Sec. 29.5. This topology is especially relevant to real world applications where

different effects can occur on various time scales. The capabilities of the MPEN-S are demonstrated using benchmarks, including a gas turbine simulation. Results on the benchmark applications indicate a significant improvement over previous approaches. This is reflected in a smaller validation error of the forecast as well as an improved estimation quality, especially for state variables dependent on highly delayed observables. Another important conclusion to draw from the experiments is the correlation between the validation error and estimation quality. This information is of high value, since in any real-world application one can only rely on the measure of the validation error.

Acknowledgment. Part of this work has been funded by the Federal German Ministry for Education and Research under the grant ALICE, 01 IB10003 A-C.

References

- [1] Bakker, B.: Reinforcement Learning with Long Short-Term Memory. In: Becker, S., Dietterich, T.G., Ghahramani, Y. (eds.) *Advances in Neural Information Processing Systems*, pp. 1475–1482. MIT Press (2002)
- [2] Bellman, R.E.: *Dynamic Programming*. Princeton University Press (1957)
- [3] Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks* 5(2), 157–166 (1994)
- [4] Duell, S., Hans, A., Udluft, S.: The Markov Decision Process Extraction Network. In: Proc. of the 18th European Symposium on Artificial Neural Networks (2010)
- [5] Duell, S., Weichbrodt, L., Hans, A., Udluft, S.: Recurrent Neural State Estimation in Domains with Long-Term Dependencies. In: Proc. of the 20th European Symposium on Artificial Neural Networks (2012)
- [6] Frasconi, P., Gori, M., Soda, G.: Local feedback multilayered networks. *Neural Computation* 4(1), 120–130 (1992)
- [7] Gomez, F., Miikkulainen, R.: 2-D Balancing with Recurrent Evolutionary Networks. In: *Proceedings of the International Conference on Artificial Neural Networks (ICANN 1998)*, pp. 425–430. Springer (1998)
- [8] Gomez, F.: Robust Non-Linear Control through Neuroevolution. PhD thesis, Department of Computer Sciences Technical Report AI-TR-03-3003 (2003)
- [9] Haykin, S.: *Neural networks and learning machines*, vol. 3. Prentice-Hall (2009)
- [10] Haykin, S., Principe, J., Sejnowski, T., McWhirter, J.: *New directions in statistical signal processing: from systems to brain*. MIT Press (2007)
- [11] Kolen, J.F., Kremer, S.C.: *A field guide to dynamical recurrent networks*. IEEE Press (2001)
- [12] Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101, 99–134 (1998)
- [13] Kietzmann, T.C., Riedmiller, M.: The Neuro Slot Car Racer: Reinforcement Learning in a Real World Setting. In: Proc. of the Int. Conf. on Machine Learning and Applications. IEEE (2009)
- [14] Lin, T., Horne, B.G., Tino, P., Giles, C.L.: Learning long-term dependencies in NARX recurrent neural networks. *IEEE Transactions on Neural Networks* 7(6) (1996)
- [15] Medsker, L., Jain, L.: *Recurrent Neural Networks: Design and Application*. International Series on Comp. Intelligence, vol. I. CRC Press (1999)

- [16] Mozer, M.C.: Induction of multiscale temporal structure. In: Advances in Neural Information Processing Systems, vol. 4, pp. 275–282 (1992)
- [17] Meuleau, N., Peshkin, L., Kee-Eung, K., Kaelbling, L.P.: Learning Finite-State Controllers for Partially Observable Environments. In: Proceedings of the Fifteenth International Conference on Uncertainty in Artificial Intelligence (UAI 1999), pp. 427–436 (1999)
- [18] Neuneier, R., Zimmermann, H.-G.: How to Train Neural Networks. In: Orr, G.B., Müller, K.-R. (eds.) NIPS-WS 1996. LNCS, vol. 1524, pp. 373–423. Springer, Heidelberg (1998)
- [19] Peters, J., Schaal, A.: Reinforcement learning of motor skills with policy gradients. *Neural Networks* 21(4) (2008)
- [20] Ramachandran, D.: Knowledge and Ignorance in Reinforcement Learning. PhD thesis, University of Illinois (2011)
- [21] Rosenstein, M.T., Barto, A.G., Si, J., Powell, W., Wunsch, D.: Supervised actor-critic reinforcement learning. In: Handbook of Learning and Approximate Dynamic Programming, pp. 359–380 (2012)
- [22] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. *Nature* 323(9), 533–536 (1986)
- [23] Riedmiller, M.: 10 Steps and Some Tricks to Set Up Neural Reinforcement Controllers. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 735–757. Springer, Heidelberg (2012)
- [24] Riedmiller, M.: Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 317–328. Springer, Heidelberg (2005)
- [25] Samuel, A.L.: Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 210–229 (1959)
- [26] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (1998)
- [27] Schneegass, D.: Steigerung der Informationseffizienz im Reinforcement-Learning. PhD thesis, Luebeck University (2008)
- [28] Schäfer, A.M., Schneegass, D., Sterzing, V., Udluft, S.: A Neural Reinforcement Learning Approach to Gas Turbine Control. In: Proc. of the Int. Joint Conf. on Neural Networks (2007)
- [29] Schäfer, A.M., Udluft, S.: Solving Partially Observable Reinforcement Learning Problems with Recurrent Neural Networks. In: Workshop Proc. of the European Conf. on Machine Learning (2005)
- [30] Schneegass, D., Udluft, S., Martinetz, T.: Neural Rewards Regression for Near-Optimal Policy Identification in Markovian and Partial Observable Environments. In: Proc. of the European Symposium on Artificial Neural Networks, pp. 301–306 (2007)
- [31] Schäfer, A.M., Udluft, S., Zimmermann, H.G.: The Recurrent Control Neural Network. In: Proc. of the European Symposium on Artificial Neural Networks, pp. 319–324 (2007)
- [32] Schäfer, A.M., Zimmermann, H.-G.: Recurrent Neural Networks Are Universal Approximators. In: Kollias, S.D., Stafylopatis, A., Duch, W., Oja, E. (eds.) ICANN 2006. LNCS, vol. 4131, pp. 632–640. Springer, Heidelberg (2006)
- [33] Takens, F.: Detecting strange attractors in turbulence. *Dynamical Systems and Turbulence* 898, 366–381 (1981)

- [34] Zimmermann, H.G., Grothmann, R., Schäfer, A.M., Tietz, C.: Identification and Forecasting of Large Dynamical Systems by Dynamical Consistent Neural Networks. In: New Directions in Statistical Signal Processing: From Systems to Brain, pp. 203–242. MIT Press (2006)
- [35] Zimmermann, H.G., Neuneier, R.: Neural network architectures for the modeling of dynamical systems. In: Kolen, J.F., Kremer, S.C. (eds.) A Field Guide to Dynamical Recurrent Networks, pp. 311–350. IEEE Press (2001)
- [36] Zimmermann, H.G., Tietz, C., Grothmann, R.: Forecasting with Recurrent Neural Networks: 12 Tricks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) NN: Tricks of the Trade, 2nd edn. LNCS, vol. 7700, pp. 687–707. Springer, Heidelberg (2012)

30

10 Steps and Some Tricks to Set up Neural Reinforcement Controllers

Martin Riedmiller

Machine Learning Lab
Computer Science Department
Albert-Ludwigs Universitaet Freiburg
riedmiller@informatik.uni-freiburg.de
<http://ml.informatik.uni-freiburg.de>

Abstract. The paper discusses the steps necessary to set up a neural reinforcement controller for successfully solving typical (real world) control tasks. The major intention is to provide a code of practice of crucial steps that show how to transform control task requirements into the specification of a reinforcement learning task. Thereby, we do not necessarily claim that the way we propose is the only one (this would require a lot of empirical work, which is beyond the scope of the paper), but wherever possible we try to provide insights why we do it the one way or the other. Our procedure of setting up a neural reinforcement learning system worked well for a large range of real, realistic or benchmark-style control applications.

Keywords: Neural reinforcement learning, fitted Q, batch reinforcement learning, learning control.

30.1 Overview

The paper discusses the steps necessary to set up a neural reinforcement controller for successfully solving typical (real world) control tasks. The major intention is to provide a code of practice containing crucial steps necessary to transform control task specifications into the specification and parameterization of a reinforcement learning task. Thereby, we do not necessarily claim that the way we propose is the only one (this would require a lot of empirical work, which is beyond the scope of the paper). But, wherever possible we try to provide insights why we do it the one way or the other. In that spirit, this paper is mainly intended to be a subjective report on how we tackle control problems by reinforcement learning in practice. It is not meant as a general review article and therefore, many related and alternative methods are not mentioned here.

When faced with a real world system, typically a very large number of ways exist to formulate it as a learning problem. This is somewhat different from the situation usually found in reinforcement learning papers, where all the main settings (like state description, actions, control interval) are usually given. In

the following we therefore carefully distinguish between the (real world) control problem (which is given) and the learning problem (which we have to design). Of course, ideally, when the learning task is solved, the resulting policy should fulfill the original controller task. The goal of this paper is to show how we can use the degrees of freedom in the modelling of the learning task to successfully solve the original control task. Our procedure of setting up a neural reinforcement learning system worked well for a large range of real, realistic or benchmark-style control applications, e.g. [9, 24, 10, 26, 7, 19, 13, 6].

30.2 The Reinforcement Learning Framework

30.2.1 Learning in Markovian Decision Processes

The approach for learning controllers followed here tackles control problems as discrete-time Markovian Decision Processes (MDPs). An MDP is described by a set S of states, a set A of actions, a stochastic transition function $p(s, a, s')$ describing the (stochastic) system behavior and an immediate reward or cost function $c : S \times A \rightarrow \mathbf{R}$. The goal is to find an optimal policy $\pi^* : S \rightarrow A$, that minimizes the expected cumulated costs for each state. In particular, we allow S to be continuous, assume A to be finite, and p to be unknown to our learning system (model-free approach). Decisions are taken in regular time steps with a constant cycle time Δ_t .

The underlying learning principle is based on Q-learning [30], a model-free variant of the value iteration idea from dynamic programming. The basic idea is to iteratively learn a value function, Q , that maps state-action pairs to their expected optimal path costs. In Q-learning, the update rule is given by

$$Q_{k+1}(s, a) := (1 - \alpha)Q(s, a) + \alpha(c(s, a) + \gamma \min_b Q_k(s', b)).$$

Here, s denotes the state where the transition starts, a is the action that is applied, and s' is the resulting state. α is a learning rate that has to be decreased in the course of learning in order to fulfill the conditions of stochastic approximation and γ is a discounting factor. It can be shown, that under mild assumptions Q-learning converges for finite state and action spaces, as long as every state action pair is updated infinitely often (see e.g. [3]). Then, in the limit, the optimal Q-function, Q^* , is reached. The optimal policy π^* can then be derived by greedily evaluating the optimal Q-function:

$$\pi^*(s) \in \arg \min_{a \in A} Q^*(s, a)$$

For a detailed introduction to reinforcement learning the reader is referred to the excellent textbooks [3, 27].

30.2.2 Q-Learning with Function Approximation

When dealing with large or even continuous state spaces, a tabular representation of the Q-function comes to its limits or is simply infeasible. A standard way to tackle this, is the use of function approximation to represent the Q-function. We focus on neural networks in the following, but other approximation schemes (like e.g. Gaussian processes [4], CMACs [28, 29], ...) are being used as well.

One big advantage of using neural networks is their capability to generalize to unseen situations - a fact particularly useful in large or continuous state spaces, where one can not expect to experience all situations during training. However, this positive feature has also a negative impact: when the standard Q-learning rule is applied to a certain state transition, it will also influence the value at other inputs in an unforeseeable manner.

To work against this effect, we developed a neural Q-learning framework, that is based on updating batches of transitions instead of single transition updates as in the original Q-learning rule. This approach has turned out to be an instance of the Fitted Q Iteration family of algorithms [5], and was named ‘Neural Fitted Q Iteration (NFQ)’ accordingly [23].

The basic idea underlying NFQ is simple but decisive: the update of the value function is performed considering the complete set of transition experiences instead of single transitions. Transitions are collected in triples of the form (s, a, s') by interacting with the environment. Here, s is the original state, a is the chosen action and s' is the resulting state. The set of experiences is called the sample set \mathcal{D} .

The algorithm is displayed in figure 30.1. It consists of two major steps: The generation of the training set P and the training of these patterns within a multilayer perceptron. The input part of each training pattern consists of the state s^l and action a^l of training experience l . The target value of each pattern is computed as suggested by the Q-learning rule: it is the sum of the transition costs $c(s^l, a^l)$ plus the expected minimal path costs for the successor state s'^l . The latter is computed on the basis of the current estimate of the Q -function, Q_k .

Since at this point, training the Q-function can be done as batch learning of a fixed pattern set, we can use more advanced supervised learning techniques, that converge more quickly and more reliably than ordinary gradient descent techniques. In particular, NFQ uses the Rprop algorithm for fast supervised learning. Rprop adapts its search step size based on the signs of the partial derivatives and has proven to be very fast and yet robust with respect to the choice of its parameters. For a detailed description of Rprop see [17]. The training of the pattern set is executed either for a predefined number of epochs (=complete sweeps through the pattern set), or until the error is below a certain predefined limit. Although simple and straight-forward, training for a fixed number of epochs works well and therefore is our standard choice. For a more detailed discussion about NFQ, please refer to [23].

```

NFQ_main() {
    input: a set of transition samples  $D$ ; output:  $Q$ -value function  $Q_N$ 
    k=0
    init_MLP()  $\rightarrow Q_0$ ;
    Do {
        generate_pattern_set  $P = \{(input^l, target^l), l = 1, \dots, \#D\}$  where:
         $input^l = s^l, a^l,$ 
         $target^l = c(s^l, a^l) + \gamma \min_b Q_k(s^l, b)$ 
        Rprop_training( $P$ )  $\rightarrow Q_{k+1}$ 
        k:= k+1
    } WHILE ( $k < k_{max}$ )
}

```

Fig. 30.1. Main loop of NFQ . k counts the number of iterations, k_{max} denotes the maximum number of iterations. *init_MLP()* returns a multilayer perceptron with randomly initialized weights. *Rprop_training(P)* takes pattern set P and returns a multilayer perceptron that has been trained on P using Rprop as the supervised training method.

30.3 Characteristics of the Control Task

In this work, we consider control scenarios of the following type. The controller has to control a technical system or process such that finally a desired target situation is achieved. The current situation of the system is measured by sensors. Thus, the control goal is usually defined by making one or more sensor values equal to externally given reference values within some tolerance bounds. To do so, the controller has to apply an appropriate sequence of control actions. The control system is realized as a closed-loop system, that acts at discrete time steps. At every time-step, the sensor values are measured, and the controller computes a control action, which is then applied to the process.

Different types of control tasks exist within this framework. An important characterization is if the control task has a defined termination or not. In the first case, control terminates immediately, once a certain goal criterion has been achieved. A typical example would be to drive a mobile robot to a certain target location. The task is terminated, once the target location is reached.

The second case is more challenging: the control task does not end, if a target condition is reached once. Instead, the controller has to actively keep the system in a set of goal states, that all fulfill a certain success criterion. Typically, this criterion is given by the sensor values being equal to their target reference values within a small tolerance band. This is a very common scenario in the control of technical systems. A typical example would be to achieve and hold a certain temperature in a room. From a control perspective, this latter scenario is much more challenging (since it contains the first one as a special case).

30.4 Modeling the Learning Task

This section discusses how to model a given (real-world) control task appropriately within the neural reinforcement learning framework. We discuss alternatives and ‘tricks’ while always trying to stay as close as possible to the framework proposed by the theory of dynamic programming.

30.4.1 State Information

The underlying reinforcement learning framework crucially depends on the assumption of the Markov property of state transitions: the successor state is a (probabilistic) function of the current state and action. As a consequence, state information provided to the learning system must be ‘rich’ enough — ‘rich’ in the sense that the observed state transition does not depend on additional historical information. In a real application, we can not necessarily expect to get the complete state information out of the values provided by the sensors. In classical control theory, the concept of an observer is known to deduce state information out of the sequence of observed sensor information. However, this requires the availability of a system model, which we assume not to have in our learning framework. A standard way to tackle this problem is to provide historical sensor and action information from previous time steps. Since we are learning anyhow, we do not rely on a particular semantic interpretability of the state information. This allows for example to provide more information than necessary or redundant information, to be on the safe side. As a tradeoff, state information should be kept as small and concise as possible to support good generalization, which will generally lead to faster learning and better control performance. In technical dynamical systems, we often use temporal differences of sensor values as an approximation to physically meaningful values like velocity or acceleration.

Like in supervised learning, using meaningful features instead of raw sensor values to enforce generalization is often helpful. However, also like in supervised learning, the design of good features typically requires deep insight into the application (here: knowledge about system behavior which in the strong sense we assume not to have). A current research direction is to autonomously learn meaningful state representations directly out of high-dimensional raw sensor data (like e.g. cameras) [15, 16, 25, 2].

Summary:

- state information must be designed out of sensor information and must be rich enough to support Markov property
- redundant information is not a problem, but it is preferable to keep the input dimensionality as low as possible
- state representation can be transformed into features to enforce generalization
- state information does not necessarily have a human understandable meaning

30.4.2 Actions

The original control task often allows the application of (quasi) continuous control values, typically in a certain range between a minimum and a maximum value. While in principle methods exist to learn continuous control actions (e.g. [11, 22]), we will here focus on providing a discrete set of control actions to our learning system. This is the most common framework in reinforcement learning and corresponds to the framework as presented in section 30.2.

This means, for setting up the learning system, one has to explicitly choose a discrete set of actions within the range of potential control signals. One potential choice is a two action set, consisting out of the minimum and maximum control action ('bang-bang'-control). In classical control theory, such a two-value control policy is the basis of time-optimal controllers. Of course, oscillating back and forth between two extreme control signals, e.g. to keep the system close to a desired sensor output, often is not acceptable when it comes to the control of real hardware. Therefore it is often advisable, to add additional actions to the learning set, e.g. a neutral action that does not put additional energy into the system.

The search space of available policies increases exponentially with the number of actions. Therefore, from the perspective of learning time, one should try to keep the number of available actions limited. However, there is of course a trade-off: a smaller number of actions leads to a coarser control behavior and the learning controller might not be able to fulfill the required accuracy in control.

There is a close interplay with the length of the control interval Δ_t : a larger control interval might require a larger action set to achieve the same level of controllability and vice-versa: the smaller the control interval, the coarser the action set may be, since more frequent interaction (and thus correction) is possible. The dynamic output element framework exploits this close relationship between temporal and structural aspects of the action set to enable more flexible control policies [22].

Trivially, a least requirement to the action set is that a policy must exist, that transfers the system to the goal state and - in case of the non-terminal state framework (see below, section 30.4.4) - keeps it within the goal area.

Summary:

- action set should be kept small to allow fast learning
- tradeoff: more actions can enhance quality or accuracy of control policy
- actions must allow to reach goal states and to keep the system within goal area in the non-terminal goal state setting.

30.4.3 Choice of Control Interval Δ_t

The control interval Δ_t denotes the time between two control interventions of the learning system. In classical control theory, controller design is often assuming that interaction happens in continuous time (like e.g. in classical PID-control).

Therefore one aims to make the control interval Δ_t as small as possible to approximate the assumed continuous time scenario - otherwise, the controller will not work as expected. This is no necessary requirement in the learning framework proposed here. Instead, the controller learns to adapt its behavior to the control interval given - therefore also larger control intervals can be managed. This additional degree of freedom is a big advantage, since for example a slower controller might be realized on less expensive hardware.

As a general tradeoff, learning gets easier, if the control interval is larger, since there are less decision points. On the other hand, the smaller the control interval, the more accurately the system can be controlled. If absolutely no prior knowledge of the system behavior is available, then Δ_t must be chosen empirically. A potential strategy to determine Δ_t is to start with a relatively large time step, which helps to learn faster, and then to refine it until the desired accuracy is achieved.

As already discussed in section 30.4.2 there is a close interplay between the available action set, the control interval Δ_t , and the potential accuracy in control.

Summary:

- the larger the control interval Δ_t , the fewer decisions have to be taken to reach the goal, therefore learning is generally faster
- tradeoff: a smaller control interval Δ_t potentially allows more accurate control and better control quality

30.4.4 The Terminal Goal State and The Non-terminal Goal State Setting

As already discussed in section 30.3, control tasks can either terminate once a goal criterion is met, or (virtually) continue forever. In the latter case, the control goal is not only to reach a state fulfilling certain success criteria, but to actively keep the system in a set of goal states, that all fulfill these success criteria. A typical success criterion for a state could be, for example, that all sensor values correspond to their target values within some tolerance bound. In the following we discuss, how these two control scenarios can be modeled in our learning framework.

For control tasks, mainly two learning scenarios are most appropriate: the terminal goal state and the non-terminal goal state framework. From the learning perspective, the terminal goal state setting is the simpler one. The task is to transfer the controlled system from an initial state to a terminal goal state by an appropriate sequence of actions. Once the goal state is reached, the episode is stopped, and the target Q-value of the last state action pair is set to the transition costs plus final costs of the terminal state.

This can be implemented by computing the target Q values as follows

$$Q(s, u) = \begin{cases} c(s, u) + \gamma \cdot \text{terminal_costs}(s') & , \quad \text{if } s' \in X^+ \\ c(s, u) + \gamma \cdot \min_b Q(s', b) & , \quad \text{else} \end{cases} \quad (30.1)$$

where $c(s, u)$ denotes the immediate costs of a transition (see below). Here, X^+ denotes the set of all terminal goal states, that fulfill the success criteria and by being reached, terminate the control task. For each terminal goal state, *terminal_costs()* assigns the corresponding terminal costs.

On the other side, the non terminal goal state framework is particularly tailored to control tasks, where the controller also has to actively keep the system in a set of goal states. Here, X^+ again denotes the set of all goal states, that fulfill the success criteria, but in contrast to the above framework, the control task is not terminated, when one of those states is reached.

This results in the following rule for the update of the Q-values

$$Q(s, u) = \begin{cases} 0 + \gamma \cdot \min_b Q(s', b) & , \quad \text{if } s' \in X^+ \\ c(s, u) + \gamma \cdot \min_b Q(s', b) & , \quad \text{else} \end{cases} \quad (30.2)$$

where as before $c(s, u)$ denotes the immediate costs of a transition outside the goal region (see below).

This seemingly slight modification has two important consequences: the episode is not stopped, once a state in the goal area is reached, and secondly no ‘grounding’ of the Q values to a terminal value occurs. This has a nasty effect to the value function, when a multilayer perceptron is used to approximate it. Due to interpolation effects and the lack of grounding, the value function tends to steadily increase. We will discuss this problem in further detail in section 30.5.3.

Since learning in the terminal goal state framework is usually easier, it sometimes makes sense to model a per-se non-terminal control problem as a terminal state learning problem. The general idea is to consider goal states with a low change-rate as pseudo terminal states. Then, the terminal goal state framework according to equation 30.1 can be applied. During learning, the task is always stopped, when one of these pseudo terminal states is reached. In the application phase, the policy learned by this procedure is then applied without stopping. The idea behind this is, that whenever the system drifts away from its goal region during application, then the controller immediately brings it back to its goal region.

However, this method is only an approximation to the actually desired behavior. It moreover requires to heuristically define what a ‘low change-rate’ means within the particular setting. So for non terminal control tasks we recommend to use the non terminal goal state learning framework whenever possible. We only wanted to mention this possibility, because sometimes a control task might be too difficult to learn within the non-terminal goal state framework. Then, an approximation by a terminal goal state problem might constitute a practical way to make it work.

Summary:

- in general, terminal goal states make learning easier
- for control applications, often the nonterminal goal state framework is appropriate, since finding a control policy that also stabilizes the system within the target region is required.

30.4.5 Choice of X^+

The set X^+ comprises all states, which fulfill the goal criteria as defined by the control tasks. One typical way to define X^+ is to denote ranges for the values of each state variable. For the state variables that we want to control, we typically define a range around their specified target value, i.e. target value $\pm\delta$, where $\delta > 0$ denotes the allowed tolerance. For other state variables, the allowed ranges might be infinitely large, denoting that we do not care what value they have for judging membership to X^+ .

Again, there is a tradeoff: the smaller we choose X^+ , the more accurate the successfully learned final controller will be. The larger X^+ , the easier it will be to learn, but we also have to accept less accurate controllers as a result.

An important requirement from the perspective of the learning framework is that X^+ is large enough, so that a policy exists, that X^+ can be reached from every starting state. Therefore, the choice of X^+ is highly related to the choice of the action set and the choice of the control interval Δ_t .

In the undiscounted ($\gamma = 1$) nonterminal goal state case, an additional requirement applies for X^+ . It must be chosen such that a policy exists, that keeps the system permanently within X^+ . This policy need not to be known in advance. Again, this requirement implies the interplay between the choice of X^+ , the control interval Δ_t and the available action set.

Summary:

- the larger X^+ , the easier it is to learn
- the smaller X^+ , the more accurate the learned controller will be

30.4.6 Choice of X^-

In many control problems, constraints on the state variables exist, that must not be violated by a successful control policy. The definition of the set of undesired states, X^- constitutes a way to model this requirement within the proposed learning framework. In a typical setting, a state is within X^- whenever a constraint of the original control problem is violated. Whenever a state within X^- is encountered, the control episode is stopped. Below, we show the resulting computation for the target of the Q-value as an extension of the equation for the non-terminal goal state framework (equation 30.2). The application within the terminal goal state framework is straightforward.

$$Q(s, u) = \begin{cases} 0 + \gamma \cdot \min_b Q(s', b) & , \quad \text{if } s' \in X^+ \\ c(s, u) + \gamma \cdot \text{terminal_costs}(s') & , \quad \text{if } s' \in X^- \\ c(s, u) + \gamma \cdot \min_b Q(s', b) & , \quad \text{else} \end{cases} \quad (30.3)$$

The terminal costs for a state within X^- should ideally be larger than the path costs for any successful policy. When using a multilayer perceptron with a sigmoid output function, we typically use a value close to 1 as terminal costs of a state within X^- .

Summary:

- the learning framework allows the modeling of hard constraints on state variables

30.4.7 Choice of Immediate and Final Costs

The choice of the immediate cost function $c(s, u)$ determines the course of the control trajectory until the target region is reached. It is not uncommon in reinforcement learning to make $c(s, u)$ a function of the distance to the target region. This has the advantage, that the immediate costs already contain a local hint to the goal, which may help learning considerably. From the perspective of the control task, however, one has to keep in mind, that the final controller optimizes the path costs to the goal. Optimizing the integrated distances to the goal might not always be the ideal realization of what is actually intended (imagine a situation, where a policy first makes a large error but then reaches the goal in a few time steps, instead of a policy that only makes small errors but for a long period of time). The situation gets even more difficult, if one has to design an immediate cost function that trades off between two or more sensor values, that all have to finally achieve a certain target value.

We therefore prefer a cost formulation, that has the advantage of a very simple and thus broadly applicable cost function:

$$c(s, u) = \begin{cases} 0 & , \text{ if } s' \in X^+ \\ c & , \text{ else} \end{cases} \quad (30.4)$$

where $c > 0$ is a constant value. A reasonable choice of c is that c multiplied by the estimated number of time steps of the optimal policy should be considerably below the maximum path costs that can be represented by the neural network (which, when using a the standard sigmoid output function, is 1).

The immediate cost function proposed above moreover has the advantage, that the learned optimal policy has a clear interpretation: it is the minimum time controller. As a side note: using this immediate cost function, one can also check the ability of a learning system to learn correct value functions: within this framework, learning can only be successful, if the learned value function is actually meaningful, since no hint towards the goal is provided by the immediate cost function.

The terminal cost function is simple as well: terminal costs are 0, if a terminal goal state is reached, and 1, if a constraint is violated. Of course, these values may depend on the potential range of the output values of the neural network.

Summary:

- costs should serve the purpose of meeting the specifications of the original control task as close as possible
- immediate costs may reflect local hints to the goal to help learning but this might not necessarily reflect the intention of the original control task

30.4.8 Discounting

In the above equations, γ with $0 \leq \gamma \leq 1$ denotes a discounting parameter. Using $\gamma < 1$ may make learning the value function easier, since the horizon of the future costs considered is reduced (consider e.g. the extreme case where $\gamma = 0$. Then only the immediate costs are relevant). On the other hand, choosing a discount rate also has an influence on the resulting optimal policies: if $\gamma < 1$, immediate costs that occur later in the sequence are weighted less. One has to make sure, that this is in accordance with the initial control task formulation. We therefore usually prefer a formulation with no discounting, i.e. $\gamma = 1$ and therefore have to make sure that for successful learning, additional assumptions are fulfilled (e.g. the existence of proper policies, which basically means that X^+ can be reached from every state with non-zero probability. For a detailed discussion see e.g. [1]).

Summary:

- discounting requires less assumptions and therefore can make learning simpler and/ or more robust
- it must be checked, whether introducing a discounting rate for the sake of better learning still matches the intention of the original control task.

30.4.9 Choice of X^0

In a typical setup, at the start of each episode, an initial starting state is randomly drawn from the starting state set X^0 . Ideally X^0 is chosen such that it covers the whole range of initial conditions that occur in the original control task.

In tasks, that in average require a large number of steps to reach the goal states, the probability of hitting the goal region by chance can be pretty low. Here, a method that we call the '*growing-competence*'-heuristic [21] might help: First, start with initial states close to the goal area and then incrementally increase the set of starting states until it finally covers the complete original starting state area.

Summary:

- the set of initial starting states for learning should cover the intended working space of the original control problem
- if applicable, then starting with simple states first and then increasing the range might help to improve the learning process dramatically

30.4.10 Choice of the Maximal Episode Length N

Control episodes might take infinitely long — this is inherently the case in the non-terminal goal state framework and can also occur in the terminal goal state setting, if the policy neither finds to the goal region nor crashes. Therefore, while learning, one typically stops the episode after some predefined number

of time steps. This is called the maximal episode length N in the following. Theoretically, within the fitted Q learning framework, N is not a critical choice. It just denotes the number of transitions sampled in a row (this is different from learning methods that rely on complete trajectories). Actually, N might be as low as 2. Then, per episode only one transition sample is collected. From a practical perspective, however, it typically makes more sense to consider longer episodes — in particular, when the policy used to sample the transitions drives the system closer towards the goal region and therefore allows to collect more and more ‘interesting’ transitions.

A rough heuristic that we use is to make N double or three times as large as the expected average time a successful controller will need to reach the target region. If N is chosen too large, then a lot of useless information might be collected - consider for example very long episodes that just contain cycles of ever the same states.

Summary:

- theoretically, the choice of N is not critical
- practically, N can considerably influence learning behavior, since it influences the distribution of the collected transitions.

30.5 Tricks

30.5.1 Scaling the Input Values

Like in normal supervised learning, scaling the input values is an important preprocessing step. Various methods and according explanations are discussed in [14]. As a standard method, in all our learning experiments, we normalize the input values to have mean of 0 and a standard deviation of 1.

Summary:

- like in supervised learning, it is important that all input values have a similar level
- a simple scaling to mean 0 and standard deviation 1 works well in all our learning experiments so far

30.5.2 The X^{++} -Trick

If no explicit terminal state exists (which is the case in the nonterminal goal state framework), the output of the neural network tends to constantly increase from iteration to iteration. This is due to the choice of the transition costs, which are 0 (within the target region) or positive (outside the target region). Therefore, the target value of each state action pair is larger or at least equally large than the evaluation of its successor state. Amplified by the generalization property of the multilayer perceptron, this leads to the tendency to ever increase the output values of all state action pairs.

A simple but effective remedy against this effect is to actually fix the values of some state action pairs to 0. We call the set of such states, for which we assume this to be true, X^{++} . This heuristic is in accordance with a correct working of the value iteration scheme, as long as 0 is the expected optimal path costs for the respective state action pairs in X^{++} . Of course, usually state action pairs for which this is true, cannot be assumed to be known a priori. Therefore, in order to apply this trick, one has to rely on heuristics. One reasonable choice of X^{++} are states, that lie in the center of X^+ , the region of zero transition costs. The reasoning behind this is the following: if one starts at a state x at the center of X^+ , then a good control policy has a very high chance of keeping the system within X^+ forever — which justifies to assign zero path costs to that starting state.

If X^{++} is chosen too large, then for some states within X^{++} the assumption of optimal path costs of 0 may be violated. As a consequence, the resulting policy most likely will not fulfill the expected property of reliably keeping the system within X^+ . On the other hand, if X^{++} is too small, the chance, that a state actually falls into X^{++} is very low and therefore the heuristic becomes ineffective. A remedy against this, is to actually force the learning system to face states in X^{++} . One possibility to do that, is to enforce starting episodes close to X^{++} , another possibility is to introduce artificial state transitions, which is discussed in the context of the *hint-to-goal heuristic* in the next section 30.5.3.

Summary:

- the X^{++} heuristic is a method to prevent the value function to steadily increase
- if applied carefully, it is in perfect accordance with the value iteration scheme

30.5.3 Artificial Training Transitions

In a certain sense, the learning process can be interpreted as spreading its knowledge of the optimal value function from the goal states to the rest of the state space. Therefore, it is crucially required to actually have a reasonable number of state action pairs that lead to a goal state within the overall transition sample set. An obvious recipe would be to try to enforce the occurrence of such goal states, e.g. by starting episodes close to the goal area. However, this is not possible for all systems because they do not allow to set arbitrary initial states.

An unconventional method to cope with the situation is to add artificial state transitions to the sample set. Then, the pattern set used for training consists of actually collected transitions, as well as additionally added artificial transitions. This method was first introduced as part of the *hint-to-goal heuristic* in our first NFQ paper [23] and has meanwhile also been successfully applied by other researchers using other function approximation schemes (e.g. Gaussian processes, [4]). The idea of the *hint-to-goal heuristic* is to introduce artificial state transitions, that start in X^{++} and end in X^{++} . Those states have — by definition of X^{++} — terminal costs of 0. As a consequence, the value function is ‘clamped’

to zero at these input patterns. Supported by the generalization ability of the function approximation, also the neighboring states will tend to have a low and thus attractive value.

If for the artificially introduced state action pairs the optimal path costs are actually zero, the hint-to-goal heuristic will not negatively interfere with the correct working of the value iteration process. An obvious choice therefore is a state action pair, where the state is well embedded in X^+ such that the assumption of optimal path costs of 0 is most likely fulfilled. We usually generate such an artificial state action pair by combining such a state with every action in the action set.

The number of artificial patterns should be chosen such that a ‘reasonable balance’ between experience of success and regular state transitions exists (as a rule of thumb, something between 1:100 and 1:10). This is of course a number, that has to be determined empirically. We are currently working on methods that automatically find such a balance, but this is ongoing work and beyond the scope of this paper.

Summary:

- the *hint-to-goal* heuristic might help to establish a goal region in the value function, if real experiences of success are difficult to achieve during regular learning

30.5.4 Growing Batch

The fitted Q iteration framework originally works with a fixed set of transitions. No particular assumption is made, how these transitions are collected. In the extreme case, these experiences are randomly sampled all over the working space of the controller in advance. In a practical setting, however, this is not always feasible. One reason is, that arbitrary sampling all over the working space is not realizable, since initial states can not be set arbitrarily. Another reason is, that to sample transitions equally over the working space might just be infeasible due to the huge amount of data required to cover the complete space.

Therefore it is desirable to concentrate on regions of the state space that are relevant for the final controller. One method to realize this is the *growing batch* method. The idea is, that one starts with an empty transition set. After the first episode, the value function is updated and the new episode is controlled by exploiting the new value function. Different variants exist, e.g. the value function can only be updated after n episodes, or the number k_{max} of NFQ iterations between two episodes can be varied. In most of our experiments so far we successfully used this growing batch procedure with the choice of $n = k_{max} = 1$.

Summary:

- the *growing batch* method aims at collecting more and more relevant transitions when the performance of the policy increases.

30.5.5 Training the Neural Q-Function

To represent the value function, we use a neural multilayer perceptron. Although it is often believed that setting up such kind of networks is a black art and its parameters are hard to find, we found that this is not particularly critical in the proposed neural fitted Q framework. One crucial point however is to use a powerful training algorithm to train the weights. The *Rprop* learning algorithm combines the advantage of fast learning and uncritical parameter choice [17]. We always use Rprop with its standard parameters. Also we found, that the number of epochs (sweeps through the training set) is not particularly critical. We therefore always train for 300 epochs and get good results. One can also think of ways to monitor the training error and find some stopping criterion to make this more flexible (e.g. to adapt to different network sizes, to different pattern set sizes, etc.), but for the applications we had so far, we found this a minor issue for learning success.

Surprisingly, the same robustness is observed for the choice of the neural network size and structure. In our experience, a multilayer perceptron with 2 hidden layers and 20 neurons per layer works well over a wide range of applications. We use the tanh activation function for the hidden neurons and the standard sigmoid function at the output neuron. The latter restricts the output range of estimated path costs between 0 and 1 and the choice of the immediate costs and terminal costs have to be done accordingly. This means, in a typical setting, terminal goal costs are 0, terminal failure costs are 1 and immediate costs are usually set to a small value, e.g. $c = 0.01$. The latter is done with the consideration, that the expected maximum episode length times the transition costs should be well below 1 to distinguish successful trajectories from failures.

As a general impression, the success of learning depends much more on the proper setting of other parameters of the learning framework. The neural network and its training procedure work very robustly over a wide range of choices.

Summary:

- choice of multilayer perceptron is rather uncritical
- important to have a powerful learning algorithm to adjust the weights
- advantage, if the supervised learning algorithm is not particularly dependent on the choice of its parameters.

30.5.6 Exploration

In reinforcement learning, exploration — the deviation from a greedy exploitation of the current value function — is important to explore the state space. Various suggestions for good exploration strategies have been proposed, e.g. considering a safe control behavior in the learning phase [12]. From our experience with NFQ, a simple ϵ -greedy exploration scheme is often sufficient. This means that in every time step, with a certain probability (e.g. 0.1), the action is chosen randomly instead of greedily exploiting the value function.

In many application cases, we also observe good results even with no explicit exploration at all. This is due to the fact, that the learning process itself — the randomly initialized neural value function, the growing experience, the randomly distributed starting states — already bears a fair amount of randomness. To learn without explicit exploration is also of practical interest. When always acting greedily, the performance achieved in a training episode is already the performance, that the final greedy controller will show. This reduces the effort of additional testing and therefore is particularly interesting for real world tasks.

Summary:

- a simple ϵ greedy exploration scheme is often sufficient
- if the starting states are well distributed in the working space, then in conjunction with the *growing batch* method, even an always greedy exploitation of the current value function works in many cases

30.5.7 Delays

In practical systems delays play a crucial role. Delays may occur both on the sensor side - i.e. a sensor value is available only n time steps later, or on the actor side - a control action has an effect only some time steps later. Simply neglecting these effects typically leads to bad control behavior or even failures. Various methods exist, e.g. to use prediction or filter methods to synchronize the information available to the controller with the current world situation. One simple but effective method is to augment state information with historical information about previous actions applied to the system [31, 20].

Summary:

- in practice, actuator and sensor delays may often be not neglectable
- a simple remedy is to add historical information about previous action values to the current state information

30.6 Experiments

30.6.1 The Control Task

The control task tackled in the following is to control a real cart pole system. While cart-pole is a well known benchmark [27], this real world task is characterized by additional challenging features:

- the initial states can not be set to arbitrary values. We moreover assume, that no human intervention is allowed, in particular, the system can initially only be started with the pole hanging down



Fig. 30.2. The real cart pole system

- the control task is to balance the pole upright with high accuracy and with the cart at a given target position (here: in the middle of the track). The controller therefore not only needs to learn to swing-up the pole from the downright position, but also to do it in such a sophisticated manner, that finally it can be balanced at the requested position.
- one cannot directly control the force applied to the cart, but only the voltage given to the DC motor driving the car. This introduces additional dynamical effects into the system.
- due to communication effects, the sensor information is delayed
- there is considerable noise in both actuation and sensor values.
- there is a discontinuity (jump) in sensor values from $-\pi$ to $+\pi$ when the pole is in the downright position.
- there is a hard constraint: the position of the cart may not be less than -0.25m and more than 0.25m, since the track is bounded.
- the final controller should be able to work from arbitrary initial start states, not only from one position.

The range of control inputs is (quasi) continuous from -12 volt to 12 volt. Sensor information provided by the system is the position of the cart and the pole; no velocity information can be measured. The target values for the sensor values should be reached as fast as possible. The minimum control interval allowed by the hardware is $\Delta_t = 0.01s$.

Since on the real system we can only perform a limited number of experiments, we also report some results on a reliable simulation of the real system (section 30.6.5). The input and output interfaces are exactly the same for both simulated and real plant. The simulation model was derived by parameterizing a physical model of the plant using real data. The accurate match between real and simulated system behavior allowed us to do keep all modelling decisions and learning parameter settings the same for both

the simulated and the real system. Therefore in the following, we only describe the real system setup. An implementation of the simulated plant is available within our open-source learning framework CLSquare available at <http://ml.informatik.uni-freiburg.de/research/clsquare>.

30.6.2 Modeling as a Learning Task

State Description. State information provided to the learning system consists of sensor values of position (p_t), angle (α_t), the normalized temporal difference of these measurements $\frac{p_t - p_{t-1}}{\Delta t}$ and $\frac{\alpha_t - \alpha_{t-1}}{\Delta t}$. The angle is zero, when the pole is upright. The angular value has a discontinuity (a jump from $-\pi$ to $+\pi$) when the pole is hanging down. Nothing particularly is done to resolve this discontinuity. Instead, we expect the learning algorithm to be able to deal with that. To cope with the sensor delay, additionally the value of the previous control action a_{t-1} is added to the state information.

Actions. The action set available for the learning system consists of the ‘standard’ choice of minimal and maximal control signal plus the ‘neutral’ action 0V. Thus $A = \{-12V, 0V, +12V\}$.

Control Interval Δ_t . As defined by the hardware, the minimum length of the control interval is 0.01s. After some initial experiments, we found that a control interval of $\Delta_t = 0.05s$ is still sufficient for an acceptable control quality while at the same time allowing a fast and successful learning process.

Non-terminal Goal State Framework. For the cart-pole task, control must be continued once pole angle and cart position reached their target values to actively keep the system within the goal states. This means, that the correct formulation of the learning problem is the non-terminal goal state setting. As a consequence, every episode is only interrupted, if the system state entered the failure set X^- or if the maximum number of steps per episode, N is reached.

Choice of X^+ . A state is in X^+ , if the following two conditions are fulfilled: the cart position is at most 0.1m away from the target position (here: middle of the track) and the pole angle deviates from 0 rad by maximally ± 0.15 rad. The rest of the state entries is not considered for judging membership to X^+ .

Choice of X^- . A state is in X^- , if the cart position is less than -0.25m or more than 0.25m. This corresponds to the physical boundaries of the track. The rest of the state entries is not considered for judging membership to X^- .

Immediate and Final Cost Functions. As immediate costs we use the standard minimum-time formulation with constant transition costs of 0.01. Thus,

$$c(s, u) = \begin{cases} 0 & , \text{ if } |p_t| \leq 0.1m \text{ and } |\alpha_t| \leq 0.15\text{rad} \\ 0.01 & , \text{ else} \end{cases} \quad (30.5)$$

When a state from X^- is observed, the episode is stopped and final costs of +1 are assigned.

Episode Length. Empirically, a good episode length was found to be $N = 200$.

30.6.3 Applied Tricks

Scaling. We applied our standard input scaling procedure as described in 30.5.1.

Choice of X^{++} and Artificial Transitions. X^{++} contains only one state, namely if all state variables are exactly 0. This corresponds to the center of X^+ . Of course, this state will most likely not occur by chance in the learning process. Therefore, this definition makes only sense in conjunction with adding artificial transitions in the spirit of the *hint-to-goal* heuristic. Here, we used 3 different artificial patterns, namely state (0,0,0,0,0) combined with all 3 actions. These transitions were repeated 100 times in the training pattern set, in order to establish some balance between the (huge) number of normal transitions and those special transitions. The target values for those artificial patterns is 0.

Growing Batch. Learning was implemented as a ‘growing batch’ process. This means, that after every episode, one NFQ iteration (new calculation of Q-target values, supervised learning of the neural Q function) was performed. Then the next episode was controlled by ϵ -greedy exploitation of this new Q function.

Training the Neural Q Function. The neural Q function is represented by a multilayer perceptron with 6 input neurons, two hidden layers with 20 neurons each and one output neuron. Hidden neurons use the tanh activation function, the output neuron uses the standard sigmoid function. Rprop with standard parameters was used for weight updates. In every NFQ iteration step, the network weights of the learning network were randomly initialized between -0.5 and 0.5 before training. The network was trained for 300 epochs per NFQ iteration.

Exploration. No explicit exploration scheme was used for the experiments done here, i.e. the current Q function was always exploited greedily to determine the action. This has the advantage, that the application performance can already be determined during training.

30.6.4 Measuring Quality

The quality of a learning control approach has two important aspects: the quality of the learning process and the quality of the resulting controller. The quality of the learning process is measured by the learning effort needed, usually measured in the number of transitions (or the number of episodes) needed, the quality of the achieved solution with respect to the used cost function, and the reliability of the results over a number of learning trials.

The quality of the resulting controller is measured with respect to the specification of the original control task. Relevant criteria are for example accuracy, robustness, working area, and performance measures like e.g. the time outside the tolerated error zone. For a detailed discussion of different criteria also see [11].

Here, we first report results achieved in a realistic simulation. This allows us to conduct a series of 10 experiments with different seeds of the random generator in reasonable time. For learning on the real system, we used exactly the same setup and parameters. The only difference we made was, that the controller was allowed to learn for a maximum of 500 episodes on the simulated cart-pole and - due to time restrictions - for a maximum of 300 episodes on the real cart-pole system.

30.6.5 Results on the Simulated Cart Pole

For the simulated system, all 10 runs delivered a successful controller. ‘Successful’ means, that for a test set of 100 random initial starting situations, the controller was able to swing up the pole and then steadily keep the system within the desired tolerance. A test run lasted 20s. In average over 10 runs, the best controller was found after an average training of 392 episodes with a standard deviation of 80. The average time needed by the best controllers was 3.23s with a standard deviation of 0.16s.

Table 30.1. Results on the simulated cart-pole for the standard setup, averaged over 10 trials. Shown are the average number of episode to train the best controller and its control performance, measured in time outside the target region. The number in brackets shows the respective standard deviation.

setup	successful trials	best controller at episode	time outside of X^+
Default	10/10	392 (80.7)	3.23s (0.16s)

30.6.6 Results on the Real Cart Pole

The evaluation on the real cart-pole system was slightly different, due to the effort it takes to do experiments with the real device. However, the overall picture of the learning behavior on the simulated and real system was consistent.

We performed three learning trials with different initializations of the random generator. Each learning trial lasted 300 episodes. Besides the reduced number

of maximum episodes, the setup of the learning system was exactly the same as for the simulated system. In all 3 trials performed, successful controllers were learned within less than 300 episodes of training. In particular, the controllers are very robust with respect to varying initial states or to disturbance from outside.

A video documenting learning and final controller performance is available at <http://www.youtube.com/watch?v=Lt-KLtkDlh8>

30.7 Conclusion

This paper discusses many of the basic modeling and methodological tricks to set up a reinforcement learning task. These insights should help to successfully handle a wide range of interesting control problems. The proposed method builds on neural fitted Q iteration (NFQ), a method that considers the complete batch of collected transitions to update the Q function. While the paper is written from the perspective of using a neural network, it should also give useful insights when using other kinds of function approximation schemes.

Current and future work is aiming to further improve the method in several directions. One big direction is to improve NFQ with respect to resulting controller quality (e.g. accuracy, continuous actions, interpretation of control policies, increasing complexity of control tasks, etc). Some steps in this direction have already been made and are discussed in [11]. Another area of ongoing and future research is to further improve NFQ with respect to robustness and autonomy of the learning process. A third area is to improve efficiency with respect to the data required for learning. Beyond that, distributed reinforcement learning algorithms that cooperatively control a complex system in a multi-agent setting is a vital research area. Distributed learning systems that are based on the neural learning framework presented here have been successfully applied in typical multi-agent scenarios like distributed job-shop scheduling [18, 8]).

Acknowledgment. The author wants to especially thank Roland Hafner from Cognit GmbH for the important initial ignition for writing this article.

References

- [1] Bertsekas, D.P.: *Dynamic Programming and Optimal Control*, vol. I, II. Athena Scientific, Belmont (1995)
- [2] Blum, M., Springenberg, J.T., Wülfing, J., Riedmiller, M.: A Learned Feature Descriptor for Object Recognition in RGB-D Data. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), St. Paul, Minnesota, USA (2012)
- [3] Bertsekas, D.P., Tsitsiklis, J.N.: *Neuro Dynamic Programming*. Athena Scientific, Belmont (1996)
- [4] Deisenroth, M.P., Rasmussen, C.E., Peters, J.: Gaussian Process Dynamic Programming. *Neurocomputing* 72(7–9), 1508–1524 (2009)

- [5] Ernst, D., Wehenkel, L., Geurts, P.: Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research* 6, 503–556 (2005)
- [6] Gabel, T., Lutz, C., Riedmiller, M.: Improved Neural Fitted Q Iteration Applied to a Novel Computer Gaming and Learning Benchmark. In: *Proceedings of the IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2011)*, Paris, France. IEEE Press (April 2011)
- [7] Gabel, T., Riedmiller, M.: On Experiences in a Complex and Competitive Gaming Domain: Reinforcement Learning Meets RoboCup. In: *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, Honolulu, USA (2007)
- [8] Gabel, T., Riedmiller, M.: Adaptive Reactive Job-Shop Scheduling with Reinforcement Learning Agents. *International Journal of Information Technology and Intelligent Computing* 24(4) (2008)
- [9] Hafner, R., Riedmiller, M.: Reinforcement learning on an omnidirectional mobile robot. In: *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, Las Vegas (2003)
- [10] Hafner, R., Riedmiller, M.: Neural Reinforcement Learning Controllers for a Real Robot Application. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2007)*, Rome, Italy (2007)
- [11] Hafner, R., Riedmiller, M.: Reinforcement learning in feedback control. *Machine Learning* 27(1), 55–74 (2011), 10.1007/s10994-011-5235-x
- [12] Hans, A., Schneegass, D., Schäfer, A.M., Udluft, S.: Safe exploration for reinforcement learning. In: *ESANN*, pp. 143–148 (2008)
- [13] Kietzmann, T., Riedmiller, M.: The Neuro Slot Car Racer: Reinforcement Learning in a Real World Setting. In: *Proceedings of the Int. Conference on Machine Learning Applications (ICMLA 2009)*, Miami, Florida. Springer (December 2009)
- [14] LeCun, Y., Bottou, L., Orr, G.B., Müller, K.-R.: Efficient backProp. In: Orr, G.B., Müller, K.-R. (eds.) *NIPS-WS 1996. LNCS*, vol. 1524, pp. 9–50. Springer, Heidelberg (1998)
- [15] Lange, S., Riedmiller, M.: Deep auto-encoder neural networks in reinforcement learning. In: *International Joint Conference on Neural Networks (IJCNN 2010)*, Barcelona, Spain (2010)
- [16] Lange, S., Riedmiller, M.: Deep learning of visual control policies. In: *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2010)*, Brugge, Belgium (2010)
- [17] Riedmiller, M., Braun, H.: A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In: Ruspini, H. (ed.) *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, San Francisco, pp. 586–591 (1993)
- [18] Riedmiller, M., Gabel, T.: Distributed Policy Search Reinforcement Learning for Job-Shop Scheduling Tasks. *TPRS International Journal of Production Research* 50(1) (2012); Available online from (May 2011)
- [19] Riedmiller, M., Gabel, T., Hafner, R., Lange, S.: Reinforcement Learning for Robot Soccer. *Autonomous Robots* 27(1), 55–74 (2009)
- [20] Riedmiller, M., Hafner, R., Lange, S., Lauer, M.: Learning to Dribble on a Real Robot by Success and Failure. In: *Proceedings of the 2008 International Conference on Robotics and Automation (ICRA 2008)*, Pasadena CA. Springer (2008) (video presentation)
- [21] Riedmiller, M.: Learning to control dynamic systems. In: Trappl, R. (ed.) *Proceedings of the 13th European Meeting on Cybernetics and Systems Research, EMCSR 1996* (1996)

- [22] Riedmiller, M.: Generating continuous control signals for reinforcement controllers using dynamic output elements. In: European Symposium on Artificial Neural Networks, ESANN 1997, Bruges (1997)
- [23] Riedmiller, M.: Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 317–328. Springer, Heidelberg (2005)
- [24] Riedmiller, M.: Neural reinforcement learning to swing-up and balance a real pole. In: Proc. of the Int. Conference on Systems, Man and Cybernetics, 2005, Big Island, USA (October 2005)
- [25] Riedmiller, M., Lange, S., Voigtländer, A.: Autonomous reinforcement learning on raw visual input data in a real world application. In: Proceedings of the International Joint Conference on Neural Networks, Brisbane, Australia (2012)
- [26] Riedmiller, M., Montemerlo, M., Dahlkamp, H.: Learning to Drive in 20 Minutes. In: Proceedings of the FBIT 2007 Conference, Jeju, Korea. Springer (2007) (Best Paper Award)
- [27] Sutton, R.S., Barto, A.G.: Reinforcement Learning. MIT Press, Cambridge (1998)
- [28] Sutton, R.S.: Generalization in reinforcement learning: Successful examples using sparse coarse coding. In: Touretzky, D.S., Mozer, M.C., Hasselmo, M.E. (eds.) Advances in Neural Information Processing Systems, vol. 8, pp. 1038–1044. MIT Press, Cambridge (1996)
- [29] Timmer, S., Riedmiller, M.: Fitted Q Iteration with CMACs. In: Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2007), Honolulu, USA (2007)
- [30] Watkins, C.J.: Learning from Delayed Rewards. Phd thesis, Cambridge University (1989)
- [31] Walsh, T.J., Nouri, A., Li, L., Littman, M.L.: Planning and Learning in Environments with Delayed Feedback. In: Kok, J.N., Koronacki, J., Lopez de Mantaras, R., Matwin, S., Mladenić, D., Skowron, A. (eds.) ECML 2007. LNCS (LNAI), vol. 4701, pp. 442–453. Springer, Heidelberg (2007)

Author Index

- Andersen, Lars Nonboe 111
Back, Andrew 295
Bengio, Yoshua 437
Bottou, Léon 9, 419
Burns, Ian 295

Caruana, Rich 163
Cireşan, Dan Claudiu 581
Coates, Adam 561
Collobert, Ronan 537, 639

Denker, John S. 235
Duell, Siegmund 709

Farabet, Clément 537
Finke, Michael 311
Flake, Gary William 143
Fritsch, Jürgen 311

Gambardella, Luca Maria 581
Giles, C. Lee 295
Grothmann, Ralph 687

Hansen, Lars Kai 111
Hinton, Geoffrey E. 599
Hirzinger, Gerd 191
Horn, David 131

Intrator, Nathan 131

Kavukcuoglu, Koray 537

Larsen, Jan 111
Lawrence, Steve 295
LeCun, Yann A. 9, 235
Lukoševičius, Mantas 657
Lyon, Richard F. 271

Martens, James 479
Meier, Ueli 581

Mobahi, Hossein 639
Montavon, Grégoire 421, 559, 621, 659
Moody, John 339
Müller, Klaus-Robert 1, 7, 9, 49, 139, 231, 343, 421, 559, 621, 659

Naftaly, Ury 131
Neuneier, Ralph 369
Ng, Andrew Y. 561

Orr, Genevieve B. 1, 7, 9, 49, 139, 231, 343

Plate, Tony 91, 225
Prechelt, Lutz 53

Ratle, Frédéric 639
Riedmiller, Martin 735
Rögnvaldsson, Thorsteinn S. 69

Schmidhuber, Jürgen 581
Schraudolph, Nicol N. 205
Simard, Patrice Y. 235
Sterzing, Volkmar 709
Sutskever, Ilya 479
Svarer, Claus 111

Tietz, Christoph 687
Tsoi, Ah Chung 295

Udluft, Steffen 709

van der Smagt, Patrick 191
Victorri, Bernard 235

Webb, Brandyn J. 271
Weston, Jason 639

Yaeger, Larry S. 271

Zimmermann, Hans-Georg 369, 687

Subject Index

- ACID, 324
- ACID/HNN, 324
- acoustic modeling, 318, 324
 - connectionist, 322
- Adaline, 423
- adaptive heuristic critic (AHC), 722, 723
- annealed importance sampling (AIS), 632
- Apple Computer
 - eMate, 271
 - MessagePad, 271
 - Newton, 271
- architectural optimization, 401–406
- asset allocation, 415
- auto-encoder, 440
 - contractive auto-encoder, 440
 - denoising auto-encoder, 440
 - sparse auto-encoder, 453
- automatic differentiation, 444
- Automatic Relevance Detection, 92
- automatic relevance detection (ARD), 92
- autoregressive model, 301
- averaging, 377
 - averaging rate, 432, 449
 - weight averaging, 432, 449, 633
- backpropagation, 11, 301, 583
 - flatness, 196
- backpropagation-decorrelation, 682
- bag of features, 561, 569
- Bayes rule, 312, 322
- Bayesian
 - *a posteriori* probabilities, 295–297
 - framework, 91
 - hyperparameters, 91
 - model prior, 72
- bias/variance, 71, 356
 - decomposition, 134
 - ensembles, 134
- block validation, 715
- Boltzmann machine, 599, 622
 - centering, 625
 - Hessian, 626
 - parameterization, 625
 - reparameterization, 626
- CART, 321
- centering
 - Boltzmann machine, 621, 625
 - neural network
 - activity, 207
 - error, 207
 - slope, 208
- class
 - equalizing class membership, 299
 - frequencies, 296
 - probabilities, 295
- classification, 120, 311
 - ECG, 299
 - error, 316
 - figure-of-merit, 297
 - handwritten characters, 274
 - k-nearest neighbor, 236
 - LVQ, 236
 - medical, 306
 - memory based, 236
 - modular, 312, 327
 - networks, 92
 - time series, 677
 - trees, 315
- cleaning, 395
- clearning, 395
- clustering, 317
 - agglomerative, 324
- committee
 - committee of neural networks, 585, 589
- computational graph, 487
- conditional random field, 433
- conditioning, 191, 626
 - condition number, 27, 450, 471, 628
- confusion matrix, 299, 317
- conjugate gradient, 32, 304, 450, 484, 544, 553
 - conjugate directions, 33
 - conjugate gradient truncation, 510
 - convergence, 513
 - damping, 496
 - Fletcher-Reeves, 33
 - line search, 32, 512
 - Polak-Ribiere, 33
 - preconditioned, 484
 - trust-region methods, 507
- context

- context driven search, 283
- geometric, 286
- lexical, 284
- contrastive divergence, 601, 614
- persistent contrastive divergence, 625
- controller
 - action, 710, 740
 - closed-loop system, 738
 - control interval, 735, 740
 - cycle time, 736
 - neural reinforcement controller, 735
 - state, 710, 739
- convergence, 304
 - almost sure convergence, 423
 - convergence of conjugate gradient, 513
 - convergence speed, 423
- convolutional neural network, 590
- correspondence principle, 689
- covariance, 26
- covariance estimation, 390
- credit assignment, 222
- cross-validation, 113, 334, 357
 - generalized (GCV), 356
 - nonlinear, 357
- damping, 496
 - conjugate gradient truncation, 510
 - K-means damped update, 566
 - line search, 512
 - scale-sensitive damping, 501
 - structural damping, 504
 - Tikhonov damping, 498
 - trust-region methods, 507
- data
 - artificially generated, 304
 - autonomous vehicle navigation, 177
 - bilinear problem, 77
 - CIFAR-10, 575, 577
 - COIL100, 652
 - combustion engine, 78
 - data noise, 397
 - dataset extension, 583
 - Deterding's vowel, 150, 215
 - distribution, 179
 - ECG, 299
 - elastic distortions, 584
 - German bond, 415
 - hill-plateau, 146
 - Index of Industrial Production, 344, 361
 - Mackey-Glass, 123
 - MIT-BIH Arrhythmia database, 300
 - MNIST, 581, 583, 633, 650
 - NETtalk, 181
 - NIST1, 247
 - NIST2, 247
 - overlapping distributions, 305
 - partitioning, 408
 - Pascal Large Scale Challenge, 432
 - Peterson and Barney, 120
 - pneumonia, 171
 - power load, 78
 - Proben1, 59
 - RCV1, 432
 - riverflow, 78
 - STL-10, 575, 577
 - sunspots, 78, 132
 - Switchboard, 329, 334
 - two-spirals, 148
 - US Postal Service, 247
 - Wikipedia, 651
- debugging, 463
 - activation statistics, 466
 - gradient verification, 428, 463
 - layer-wise analysis, 466, 629, 636
 - restricted Boltzmann machines, 615
 - verifying Gauss-Newton products, 495
- decimation
 - of d.c. errors, 207
 - of linear errors, 208
- decision boundary, 304
- decision tree, 313, 321
- phonetic, 326
- deep network
 - deep Boltzmann machine (DBM), 623
 - locally connected (LC-DBM), 623
 - deep K-means, 574
 - dependency testing, 574
 - energy correlation, 574
 - embedding, 639
 - deformable prototype, 237
 - dendrogram, 326
 - density estimation, 384, 622
 - conditional, 384
 - dictionary, 562
- dilemma
 - Observer-Observation Dilemma, 391, 399
 - Structure-Speed-Dilemma, 395
- distance
 - Euclidean, 237
 - invariant, 239
 - tangent, 238, 242

- divide & conquer, 312
- Doonesbury effect, 271, 289
- dynamic programming, 736
 - value iteration, 736
- dynamical system, 712, 713
 - instable information flow, 702
 - open dynamical system, 714
 - risk, 703
 - stable information flow, 702
 - uncertainty, 703
- early stopping, 73, 406, 448
 - in MTL nets, 181
- echo state network, 659
- effectice number of parameters, 357
- effective degrees of freedom, 94
- efficiency
 - of stopping criteria, 58
- embedding, 468, 639
 - deep embedding, 644
 - ISOMAP, 642
 - Laplacian eigenmaps, 642
 - multidimensional scaling, 642
 - neighbors graph, 646
 - siamese networks, 642
 - symbol embedding, 469
 - temporal embedding, 646
- empirical, 421
- empirical fisher diagonal, 523
- ensemble
 - averaging, 131
 - forecast, 349
 - error, 383, 388
 - approximation error, 192, 425
 - cross-entropy, 92, 228, 296
 - cross-validation, 113
 - error-in-variables, 396
 - estimate, 55
 - estimation error, 425
 - evolution, 55
 - idealized, 54
 - realistic, 55
 - generalization, 53
 - generalized least square estimation, 387
 - increase, 57
 - Kullback-Leibler, 325
 - Likelihood
 - log, 384
 - scaled, 322
 - LnCosh, 383
 - local minima, 55
 - mean squared sensitivity error, 300
 - metrics, 175
 - normalized root mean square error, 661
 - optimization error, 425
 - prediction error, 355
 - final (FPE), 356
 - generalized (GPE), 357
 - squared (PSE), 356
 - residual pattern, 192
 - root mean square error, 660
 - roundoff error, 225
 - training, 53
 - validation, 53
 - word error rate, 334
 - error bar estimation, 388–390
 - estimation
 - conditional density, 384
 - covariance, 390
 - density, 384
 - error bar, 388–390
 - generalization error, 55
 - robust, 383–388
 - with CDEN, 384
 - with LnCosh, 383
 - evidence, 91
 - expected risk, 421
 - factoring posteriors, 323
 - false positive rate, 299
 - feature selection, 180
 - feed-forward network, 480
 - Hessian, 194
 - Jacobian, 194
 - linearly augmented, 191–202
 - feedback, 680
 - output feedback connections, 680
 - finite differences, 428
 - flat spots, 228
 - focus of attention, 177
 - forecasting, 687, 713, 720, 729, 730
 - function approximation, 709
 - GARCH, 387
 - Gauss-Newton matrix, 489
 - generalized, 489
 - multiplication, 492
 - generalization, 10, 56, 92, 113, 329, 334, 639
 - reuse, 468
 - generalized Gauss-Newton
 - estimating the diagonal, 524

- Gibbs sampling, 601, 623
 - alternating Gibbs sampling, 601, 623
- gradient descent, 422, 442
 - convergence, 23
 - divergence, 24
 - regularization parameters, 116
- graphics processing unit (GPU), 466, 542, 556, 582, 587
 - convolution kernel, 591
 - kernel, 591
 - thread, 591
- handwriting recognition, 246, 312
 - architecture, 275
 - online, 271
- Hessian, 25, 194, 446
 - backpropagating diagonal Hessian, 36
 - Boltzmann machine, 626
 - conditioning, 221
 - eigenvalue spread, 38
 - eigenvalues, 38
 - exact multiplication by the Hessian, 487
 - maximum eigenvalue, 27, 42
 - online computation, 43
 - power method, 42
 - Taylor expansion, 42
 - minimum Eigenvalue, 22
 - product of Hessian and vector, 37
 - shape, 40
 - singular, 195–196
 - square Jacobian approximation, 34, 35
- Hessian-free optimization, 479, 483
 - damping, 496
 - debugging, 495
 - parallelism, 494
- heteroscedasticity, 384, 387
- hidden Markov models, 319
- hidden units, 450, 610
 - number of, 181, 302, 450
- hierarchical, 313
 - classifier, 333, 336
 - features, 574
 - hierarchy of neural networks, 323, 324, 327, 329
- hint-to-goal heuristic, 747
- hyperparameters, 91, 347, 446
 - automated grid search, 458
 - coordinate descent, 458
 - grid search, 456
 - layer-wise optimization, 459
 - manual search, 456
- multi-resolution search, 458
- semi-supervised learning, 645
- ill-conditioned, 191
- ill-posed problem, 70
- image recognition, 569
- information divergence, 324
- initialization
 - conjugate gradient, 517
 - K-means, 565
- input
 - contrast normalization, 564
 - force, 370
 - momentum, 370
 - normalization, 16, 30
 - decorrelation, 17
 - equalization, 17
 - zero mean, 16
 - preprocessing, 374, 563, 714, 716
 - representation, 274
 - scaling, 714
 - spherling, 32
 - squared inputs, 374
 - symbolic, 468
 - whitening, 32, 564
- invariance transformations, 263
 - diagonal hyperbolic transformation, 265
 - elastic distortions, 584
 - parallel hyperbolic transformation, 264
 - rotation, 264
 - scaling, 264
 - thickening, 265
 - X-translation, 263
 - Y-translation, 264
- Jacobian, 34, 35, 194, 441
- K-means, 423, 561, 562, 566
 - encoding, 571
 - hard assignment, 571
 - hyperparameters, 570
 - patch size, 570
 - pooling, 571
 - receptive fields, 570
 - initialization, 565
 - receptive field, 572
 - sparsity, 566
 - spherical K-means, 562
 - training procedure, 566
- Kahan summation, 674
- kernel principal component analysis, 630

- Krylov subspace, 484
- Kullback-Leibler, 325
- labeled data, 572
- large-scale learning, 424
 - trade-offs, 426
- Lasso, 423
- LBFGS, 544, 553
- leaky integration, 661, 667
- learning, 391
 - batch, 13, 544
 - advantages, 14
 - BFGS, 34
 - bold driver, 213
 - condition, 191–202
 - conjugate gradient, 32, 148, 193, 225, 230
 - curriculum learning, 470
 - FORCE learning, 682
 - Gauss-Newton, 34
 - gradient descent, 296
 - hierarchical, 180
 - hill-climbing, 299
 - hints, 179
 - K-means, 561, 562
 - Levenberg-Marquardt, 35, 193
 - life-long, 163
 - multi-relational, 468
 - multitask, 164, 468
 - Newton, 31, 148
 - online learning, 13, 441, 679
 - Perceptron, 28
 - Quasi-Newton, 34, 193, 392
 - rankprop, 172
 - rate, 20, 185
 - adaptation, 21
 - annealing, 14
 - momentum, 21
 - Rprop, 59, 737
 - second order, 31, 193–194, 225, 230
 - semi-supervised learning, 642
 - single task, 164
 - stochastic, 13
 - advantages, 13
 - stochastic diagonal Levenberg Marquardt, 40
 - stopping criterion, 59
 - tangent propagation, 256
 - variable metric, 193
 - vario- η , 212, 392, 717
 - learning rate, 332, 429, 430, 432, 447, 606, 717
 - adaptive, 333, 448, 472
 - annealing, 281
 - individual, 40
 - maximal, 25
 - optimal, 25
 - Levenberg-Marquardt heuristic, 506
 - Lie group, 259
 - linear models, 429
 - linearly augmented feed-forward network, 191–202
 - equivalence with feed-forward network, 198
 - universal approximation, 198
 - liquid state machine, 659
 - local minimum, 191–202
 - local versus global features, 143, 155, 159
 - long-range dependencies, 713
 - long-term dependencies, 482, 691, 727, 730
 - loss function
 - convex, 489
 - non-convex, 493
 - second derivative, 493
 - Lua, 539
 - associative arrays, 539
 - closures, 540
 - performance, 541
 - LVCSR, 318, 319, 322
 - M-estimators, 386
 - macroeconomic forecasting, 343
 - Markov decision process, 710, 736
 - extraction network (MPEN), 725, 727, 729
 - with shortcuts, 727, 729
 - partially observable (POMDP), 711, 723
 - policy, 736
 - Markov property, 711, 714, 723, 725, 729
 - Markov-chain Monte Carlo methods, 109
 - Markovian state, 727
 - mean-variance approach, 389
 - medical
 - diagnosis, 171
 - risk prediction, 171
 - memory allocation, 545
 - mini-batch, 443, 544, 603, 717
 - Hessian-free optimization, 526
 - size of the mini-batch, 448
 - missing values, 617

- mixture
 - densities, 320, 324
 - of Gaussians, 323, 385
- model
 - complexity, 329
 - interpretation, 349
 - selection, 316, 348, 353, 446
 - architecture, 354
 - input, 354
- momentum, 449, 607
- monitoring
 - errors, 428
 - overfitting, 605
 - progress of learning, 604, 615, 630, 632
 - reconstruction error, 605
- Moore-Penrose pseudoinverse, 675
- MTL, 164
- multi-core machines, 466
- network information criterion (NIC), 357
- neural network
 - architecture, 371–382
 - bottleneck network, 372
 - capacity, 181
 - correspondence principle, 689
 - efficient implementation, 538
 - expert, 324
 - feed-forward, 121, 123
 - interaction layer, 375
 - internal preprocessing, 371
 - linear, 28
 - multi-layer perceptron, 28, 313, 329, 581
 - neural fitted Q-iteration, 737
 - neural reinforcement controller, 735
 - radial basis function, 374
 - size, 181
 - vanilla, 346
- noise
 - Gaussian, 383, 391
 - Laplacian, 383
 - parameter noise, 393
- noise reduction, 221
- noise/nonstationarity tradeoff, 345
- non-linearity, 454
 - hard-limiting, 471
 - softsigm, 471
- nonlinear calibration, 307
- outliers, 372, 383, 386
 - information shocks, 383
- output representation, 176
- overfitting, 53, 55, 91
 - mini-batch overfitting, 529
- overshooting, 714
- partition function, 600, 622
 - ratio of partition functions, 631
- pattern generation, 680
- penalty
 - factors, 91
 - implicit, 393
 - implied curvature, 400
- Perceptron, 423
- performance
 - criteria, 299
 - measures, 299
- polyphones, 321
- portfolio management, 389
- positive predictivity, 299
- power method, 42
- preconditioning, 427
 - conjugate gradient, 519
 - preconditioner design, 521
- predicting
 - posteriors, 313
- preprocessing, 370–371, 456, 563
 - by bottleneck network, 372
 - by diagonal connector, 371
- price-performance ratio, 56
- prior, 72
 - bias reduction, 289
 - knowledge, 242, 317
 - probabilities, 295
 - scaling, 296
- probabilistic sampling, 298
- pronunciation graph, 319
- pruning, 359, 401–406
 - Early-Brain-Damage, 402
 - input variables, 348, 359
 - Instability-Pruning, 405
 - Inverse-Kurtosis, 403
 - of nodes, 401
 - of weights, 401
 - sensitivity-based (SBP), 359
 - Stochastic-Pruning, 401
- Python, 541
- Q-learning, 736
 - artificial training transitions, 747
 - cost function, 744
 - discounting, 736, 745
 - growing batch, 748

- maximal episode length, 745
- neural fitted Q-iteration, 737
- neural Q-function, 749
- policy, 736
- X++ trick, 746
- quantized weights, 282

- R operator, 445, 487
- radial basis function network, 146, 155
- random seeds, 455
- recurrent neural network, 313, 482, 659, 687, 710, 712, 720, 721, 724
 - causal-retro-causal (CRCNN), 699
 - connectivity, 698
 - dimensionality, 698
 - dynamically consistent, 724
 - echo state network, 659
 - error correction (ECNN), 693
 - historically consistent (HCNN), 695
 - initial state problem, 692
 - liquid state machine, 659
 - memory, 698
 - overshooting, 691, 713
 - shortcut connections, 208
 - sparsity, 698
 - unfolding in time, 713
 - variant-invariant separation, 694
- regression, 676
 - least mean squares, 679
 - linear regression, 672
 - recursive least squares, 679
 - ridge regression, 672
 - weighted least squares, 677
- regularization, 54, 71, 91, 111, 114, 181, 241, 349, 639, 672
 - adaptive, 115
 - choice of, 115
 - for recurrent networks, 351
 - smoothing, 350
 - Tikhonov regularization, 672
- reinforcement learning, 709, 712, 735
 - cart-pole problem, 750
 - exploration, 749
 - neural reinforcement controller, 735
- representational capacity, 304
- reservoir, 660
 - parameters, 663
 - size, 663
 - sparsity, 664
 - spectral radius, 665
- rest learning, 719

- double rest learning, 718
- restricted Boltzmann machine, 599
- discrimination, 615
- risk, 390
- robust estimation, 383–388
 - with CDEN, 384
 - with LnCosh, 383
- robust learning, 717

- scaling, 296
- scripting language, 538
- search
 - for hyperparameters, 92
- second order gradient descent, 422, 472, 479, 544
- segmentation
 - character, 273
 - word, 287
- semi-supervised learning, 642
 - deep embedding, 643
 - label propagation, 642
 - LapSVM, 643
- sensitivity, 299
 - average absolute gradient, 360
 - average gradient, 360
 - delta output, 362
 - evolution, 362
 - output gradient, 362
 - pruning, 359
 - RMS gradient, 360
- separation of structure and noise, 399
- shuffle test, 167
- shuffling, 427
- SIMD operations, 547
 - NEON, 547
 - SSE, 542, 547, 582
- softmax, 228, 331
- software
 - BLAS, 466, 541, 542, 546, 550, 551
 - CUDA, 542, 545, 549, 556
 - EBLearn, 552
 - JanusRTk, 329
 - Just In Time compiler, 539
 - LAPACK, 541, 546
 - Lush, 542
 - Numpy, 543, 550
 - OpenMP, 545, 548, 556
 - SENN, 416
 - SN, 542
 - SWIG, 541
 - Theano, 445, 550, 552

- Torch5, 552
- Torch7, 537, 542
- source separation, 568
- sparsity, 430
 - feature sparsity, 453
 - input sparsity, 467
 - K-means, 566
 - sparse coding, 562
 - sparsity of activation, 453, 609
 - weight sparsity, 303
- speech recognition, 296, 312, 318
 - connectionist, 312
 - statistical, 314, 318
 - system, 330
- squared inputs, 143
- stability
 - structural instability, 382
- state estimation, 709, 714, 720, 724, 727, 729
- state tying, 322
- STL, 164
- stochastic gradient descent, 332, 422, 442, 544, 553
 - averaged SGD, 431
 - stopping criterion, 56, 719
 - early stopping, 406
 - effectiveness, 62
 - efficiency, 56, 58, 62
 - final stopping, 413
 - late stopping, 407
 - predicate, 56
 - robustness, 58, 62
 - rules for selecting, 57
 - threshold
 - for stopping training, 56
 - time versus error tradeoff, 63
 - tradeoff
 - time versus error, 55, 58
 - variance, 60
 - SVM, 423, 432
 - symbolic differentiation, 445
 - Takens’s theorem, 726
 - tangent propagation, 253
 - targets
 - embeddings, 380
 - forces, 380
 - random, 378
 - teacher forcing, 680
 - architectural teacher forcing, 697, 700
 - temporal pattern recognition, 678
 - time series
 - linear models, 345
 - noise, 345
 - nonlinearity, 345
 - nonstationarity, 345
 - prediction, 123, 133, 176
 - Torch7, 537, 550, 552
 - CUDA, 550
 - neural networks, 543
 - packages, 544
 - Tensor, 542, 544
 - TH, 551
 - training
 - procedure, 406–414
 - transfer, 163
 - inductive, 163
 - sequential vs. parallel, 179
 - tricks
 - backpropagating diagonal Hessian, 36
 - backpropagating second derivatives, 36
 - choice of targets, 19
 - computing Hessian information, 35
 - data warping, 279
 - derivative roundoff error, 225
 - ensemble with different initial weights, 131
 - error emphasis, 281
 - frequency balancing, 298
 - Hessian
 - finite difference, 35
 - maximum eigenvalue, 42
 - minimum Eigenvalue, 22
 - square Jacobian approximation, 35
 - hyperparameters, 91
 - initializing the weights, 20
 - multitask learning, 164
 - negative training, 278
 - nonlinear cross-validation, 357
 - normalizing output errors, 276
 - post scaling, 298
 - prior frequency balancing, 280
 - prior scaling, 296
 - roundoff error, 225
 - sensitivity-based pruning, 359
 - shuffling examples, 15
 - sigmoid, 17
 - squared input units, 143
 - tangent distance, 248
 - elastic tangent distance, 249
 - hierarchy of distances, 251
 - smoothing, 249

- speeding, 251
- tangent propagation, 259
- training big nets, 312
- variance reduction with ensembles, 131
- weight decay parameter estimate, 75, 77
- triphone models, 321
- unequal misclassification costs, 295
- unit
 - binomial, 613
 - Gaussian, 612
 - logistic, 226
 - radial basis function, 22
 - rectified linear, 613
 - sigmoid, 17
 - softmax, 611
 - squared input, 143
 - tanh, 229
- user adaptation, 291
- vanishing gradient, 728
- variable metric, 193
- VC dimension, 241
- vector quantization, 562
- ventricular contraction
 - premature ventricular contraction, 301
 - supraventricular contraction, 301
- visualization, 349, 361, 465
 - filters, 465, 636
 - Hinton diagram, 465
 - learning trajectory, 466
 - local tangent vectors, 465
 - sampling, 465
 - t-SNE, 466
- weight
 - updates
 - magnitude of, 298
 - weight decay, 73, 118, 121, 123, 303, 451, 608, 673
 - L1 regularization, 452
 - L2 regularization, 452
 - parameter estimate, 74, 75
 - simulation, 79
- weight sharing, 468, 713, 718
- weights initialization, 454, 606
 - fan-in, 454
- well determined parameters, 94