An Investigation of the Gradient Descent Process in Neural Networks

Barak A. Pearlmutter

January 16, 1996
CMU-CS-96-114

School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:

Professor David S. Touretzky, Chair
Professor Scott E. Fahlman
Professor James L. McClelland
Professor Geoffrey E. Hinton, University of Toronto

© 1996 Barak A. Pearlmutter

This research was sponsored in part by the Office of Naval Research under contract numbers N00014-86-K-0678 to Dave Touretzky at Carnegie Mellon University and N00014-92-J-4062 to John Moody at Yale University and the Oregon Graduate Institute, by the National Science Foundation under contract numbers EET-8716324 to Scott Fahlman and Dave Touretzky at Carnegie Mellon University and ECS-9114333 to John Moody at Yale University and the Oregon Graduate Institute, and by the Hertz Foundation under a Hertz Fellowship to the author. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the ONR, the NSF, or the U.S. government.

Abstract

Usually gradient descent is merely a way to find a minimum, abandoned if a more efficient technique is available. Here we investigate the detailed properties of the gradient descent process, and the related topics of how gradients can be computed, what the limitations on gradient descent are, and how the second-order information that governs the dynamics of gradient descent can be probed. To develop our intuitions, gradient descent is applied to a simple robot arm dynamics compensation problem, using backpropagation on a temporal windows architecture. The results suggest that smooth filters can be easily learned, but that the deterministic gradient descent process can be slow and can exhibit oscillations. Algorithms to compute the gradient of recurrent networks are then surveyed in a general framework, leading to some unifications, a deeper understanding of recurrent networks, and some algorithmic extensions. By regarding deterministic gradient descent as a dynamic system we obtain results concerning its convergence, and a quantitative theory of its behavior when a saturating or "robust" error measure is used. Since second-order structure enters into the dynamics, an efficient exact technique for multiplying a vector by the Hessian of a gradient system is derived and applied.

Keywords: neural networks, control, optimization, gradient descent, Newton method, convergence rate, Hessian

Dedicated to the memory of
Rabbi Fishel Alter Pearlmutter, Alav Hashalom.

Contents

1 Overview **1**

1.1 Summary of Contributions 3

2 Dynamics Compensation **4**

2.1 Overview of Control Theory 4

2.1.1 Linear Control Theory 5

2.1.2 Linearization and Dynamics Compensation 6

2.2 Simulated Arm 7

2.3 Experiments on an Actual Robot Arm 9

2.4 Analysis of the Weights as Convolutions 13

2.5 Conclusion 17

3 Recurrent Networks **20**

3.1 Introduction 20

3.1.1 Why Recurrent Networks 20

3.1.2 Why Hidden Units 22

3.1.3 Continuous vs. Discrete Time 24

3.2 Learning in Networks with Fixedpoints 25

3.2.1 Will a Fixedpoint Exist? 25

3.2.2 Problems with Fixedpoints 26

3.2.3 Recurrent Backpropagation 27

3.2.4 Deterministic Boltzmann Machines 31

3.3 Non-fixedpoint Gradient Calculation 32

3.3.1 Backpropagation Through Time 33

3.3.2 Real Time Recurrent Learning . . . 36
3.3.3 More Efficient Online Techniques . . . 37
3.3.4 Time Constants . . . 38
3.3.5 Time Delays . . . 38
3.3.6 Extending RTRL to Time Constants and Time Delays . . . 40
3.4 Some Simulations . . . 41
3.4.1 Exclusive Or . . . 41
3.4.2 A Circular Trajectory . . . 43
3.4.3 A Figure Eight . . . 44
3.4.4 A Rotated Figure Eight . . . 44
3.4.5 Computational Neuroscience: A Simulated Leech . . . 46
3.5 Stability and Perturbation Experiments . . . 46
3.6 Other Non-fixedpoint Techniques . . . 47
3.6.1 "Elman Nets" . . . 47
3.6.2 The Moving Targets Method . . . 48
3.6.3 Feedforward Networks with State . . . 49
3.6.4 Teacher Forcing In Continuous Time . . . 49
3.6.5 Jordan Nets . . . 51
3.6.6 Teacher Forcing, RTRL, and the Kalman Filter . . . 52
3.7 Learning with Scale Parameters . . . 52
3.8 Summary and Conclusion . . . 53
3.8.1 Complexity Comparison . . . 53
3.8.2 Speeding the Optimization . . . 54
3.8.3 Prospects and Future Work . . . 55
3.8.4 Conclusions . . . 55

4 The Convergence of Gradient Descent 56
4.1 Introduction . . . 56
4.1.1 Asymptotic convergence and choice of learning rate . . . 57
4.1.2 Asymptotic convergence with momentum . . . 58
4.1.3 Choice of both α and η . . . 59

4.2 Second-order momentum 61
4.3 Simulations 62
4.4 Nonquadratic error surfaces 63
4.4.1 Saturating error surfaces 64
4.4.2 Superquadratic error surfaces 68
4.5 Alternate formulations of momentum 72
4.6 Transient behavior 75
4.7 Automatic adjustment of η 76
4.8 Why assume optimal α 77
4.9 Weight precision 78
4.10 Changes of variables 78
4.11 Conclusions and speculations 80

5 Multiplication by the Hessian **92**
5.1 Introduction 92
5.2 The Relation Between the Gradient and the Hessian 93
5.3 The $\mathcal{R}\{\cdot\}$ Technique 95
5.4 Application of the $\mathcal{R}\{\cdot\}$ Technique to Various Networks 96
5.4.1 Simple Backpropagation Networks 96
5.4.2 Recurrent Backpropagation Networks 98
5.4.3 Deterministic Boltzmann Machines 99
5.4.4 Stochastic Boltzmann Machines 99
5.4.5 Weight Perturbation 101
5.5 Practical Applications 102
5.5.1 Finding Eigenvalues and Eigenvectors 103
5.5.2 Multiplication by the Inverse Hessian 103
5.5.3 Step Size and Line Search 103
5.5.4 Optimization of Stochastic Gradient Descent 105
5.5.5 Second-Order Optimization Techniques 107
5.6 Summary and Conclusion 109

6 Final Words **112**

6.1 Applications of Recurrent Networks 112
6.1.1 Generative Continuous Backpropagation 112
6.1.2 Signal Processing and Sensor Integration 114
6.2 Applications of Multiplication by the Hessian 116

Bibliography **118**

Acknowledgments

My graduate study was supported by a Fannie and John Hertz Fellowship for most of the time spent working on this thesis. Additional support came from Carnegie Mellon University, and from grants to Scott Fahlman, David Touretzky, and John Moody.

In the individual chapters, I have thanked those who were involved in the work described. Here, I would like to take the opportunity to thank those who more globally influenced my life and career. My parents were both teachers, and whatever love for knowledge or capacity for learning that I possess is due to them, and to the wonderful upbringing they gave me.

The Computer Science Department at Cranberry Lemon University was an exciting place that combined cooperation, high level work, and a reverence for function over form. My friends from my days at CMU are numerous, and if in my efforts to thank everyone who deserves it I fail to mention someone, please be assured that it is only because my memory is episodic rather than serial.

Olin Shivers proselytized Scheme to me when I was still young. If I deviated from the one true λ-form, it was surely a failing of my own soul. Kevin Lang, my fellow λ-heretic, was a constant enigma, alternating between the most elementary questions and the most profound answers and comments. Sometimes I was able to distinguish them. Mark Derthick made sure I'd never feel sorry for myself. Lin Chase taught me the best of office politics, and showed me what it means to be a free spirit. Bruce Horn's enthusiasm and deep caring remain embedded in my soul. I'll never cease to admire Rick Szeliski's understated way of developing and presenting profound and important results. Troubles and provocations flow smoothly over him, like water over a stone. Richard Wallace and Ken Goldberg showed me how science can be fun, and more importantly, how fun can be science.

Akaysha Tang provided her love, and I can only hope that her continuing attempts to cure me of my many bad habits meet in disappointment. One habit she has successfully cured me of, however, is procrastinating about finishing my thesis, a cure for which both I and my mother are deeply grateful.

This thesis consists mainly of elaborations and footnotes to offhand comments of Geoff Hinton's. His influence would itself take a chapter to describe—my dream is to do work one tenth as significant as his.

David Touretzky is a true mensch. He gave me opportunities that I frittered away, but now that I have finally grabbed the brass ring that he is responsible for placing and holding in front of me, I would like to thank him for all that he has done—and for all that he has put up with.

Chapter 1

Overview

This thesis takes the gradient descent process seriously. Typically, gradient descent is treated as just a way to find the minimum of an error function, to be abandoned if a more efficient optimization technique is found. Here, we instead investigate the detailed properties of the gradient descent process, and the related topics of how gradients can be computed, and how second-order information, which governs the local dynamics of a gradient system, can be efficiently obtained in high-dimensional systems.

To give a global overview: we try gradient descent on a typical problem, in order to develop our intuitions; we review and unify gradient calculation techniques for recurrent networks; we apply dynamical systems methods to the analysis of gradient descent; and we find an efficient way to extract the second-order information relevant to the gradient descent process.

In chapter 2 we review applications of neural network function approximators to the dynamics compensation problem, and we use neural network function approximators to learn a particular dynamics compensation task. In the process, we find that pre-processing the signals from the plant into velocities and accelerations is unnecessary, as the network spontaneously develops filters to compute these quantities. This motivates us to attempt to reduce the number of free parameters of our networks by investigating temporally continuous recurrent neural networks.

In chapter 3 we discuss and review recurrent neural networks, and derive some algorithms for computing the derivatives of error measures of the performance of these networks with respect to their internal parameters, or *weights*. These algorithms calculate the *gradient* of the system, the sensitivity of the error with respect to the free parameters. This gradient is in turn used in various procedures that modify the weights to minimize the error. This is normally done using simple gradient descent, a process discussed in chapter 4.

In some simulated control experiments with indirect neural network control architectures, learning was very slow, and some unexpected gentle rhythmic oscillations

were noted. In chapter 4 we treat the deterministic gradient descent process as a dynamical system, and show how its parameters can be controlled to optimize its rate of convergence, and what the limits on this rate are. After reviewing known results on simple gradient descent and gradient descent with momentum, we present a novel analysis of gradient descent with second-order momentum. We then note some anomalies that arise when this linear analysis is applied in the nonlinear error surfaces that are used in practice, and perform a novel nonlinear dynamics system analysis of gradient descent on a saturating error surface, and on a narrowing valley. This analysis results in a satisfying quantitative theory which explains the previously mysterious oscillatory phenomena, which has been observed but only discussed informally, becoming part of the backpropagation folklore.

One way of thinking about the limits on the convergence of deterministic gradient descent is that, although the system can be well approximated by a locally linear model, this linear system is ill conditioned, and the control parameters, *i.e.* the learning rate and momentum, cannot decouple the system along the eigenvectors of the Hessian, and so cannot decompose the single ill conditioned system into multiple better conditioned systems. This decoupling could be done easily if one knew the eigenvectors of the Hessian of the system, but even computing the Hessian itself is prohibitive (in both space and time) for the high dimensional systems in which we are interested.

In chapter 5, by viewing the Taylor expansion of the error made in chapter 4 as a general formula, rather than only in the specific context of gradient descent, we are led to a highly efficient technique for computing not the Hessian itself, but rather its product with a vector. This technique applies to a wide class of gradient systems, including feedforward systems like backpropagation networks, recurrent fixedpoint networks, stochastic Boltzmann machines, and weight perturbation networks. This leads to a new class of iterative algorithms for computing properties of the Hessian without computing the full matrix itself. For instance, the power method can use this technique to calculate the principal eigenvector of the Hessian, or to estimate it well in a stochastic setting. A trivial conjugate-gradient method can use it to multiply an arbitrary vector by the inverse Hessian, sometimes even more efficiently than could be done were the entire inverse Hessian already known. We use the power method to find the principal eigenspace of the Hessian, and then decouple minimization in the principal eigenspace from that in the remaining orthogonal space, speeding stochastic gradient descent. (Unfortunately, initial benchmarks using this technique, although promising, were not superior to other known methods.) A fast approximate line search can be based on this procedure, and has been used in the Scaled Conjugate Gradient algorithm. The shape of the Eigenvalues spectrum can be estimated using just a score of multiplications by the Hessian. There are also some potential applications to generalization.

Chapter 6 concludes the thesis with a discussion of potential applications and directions for future research.

1.1 Summary of Contributions

Because of the organization of this thesis, consisting of alternation of tutorial and background information with new results, and because its breadth may limit some readers' interest to a portion of the thesis, it may be helpful to have a roadmap of the original contributions contained herein. For that reason, I have compiled below, in stylized bullet form, a summary of the important contributions.

- Time constants of hidden units can be learned using backpropagation through continuous time.
- Modifications of the backpropagation through time (BPTT) and and real-time recurrent learning (RTTL) procedures that can learn time delays.
- Continuous-time BPTT can be used to train continuous-time deterministic recurrent networks to be effective oscillatory pattern generators which are responsive to their environmental input.
- Teacher-forcing, which has hitherto been on somewhat shaky theoretical ground, can be understood as learning the *derivative* of the target function.
- Second-order momentum is useless, even in the deterministic gradient case—previously an open theoretical question.
- The previously-rumored discrepancy between the linear theory of gradient descent and gradient descent in backpropagation networks in practice can be characterized and precisely accounted for by a dynamic system analysis of gradient descent on a nonlinear error surface.
- A novel $O(n)$-time and $O(n)$-space algorithm finds the exact product of the Hessian matrix with a vector—a quantity previously only approximated using $O(n^2)$-space and $O(n^2)$-time (and worse) methods.

Chapter 2

Dynamics Compensation

Capsule: Some of the work described in this chapter was included in Goldberg and Pearlmutter (1988, 1989).

2.1 Introduction: An Idiosyncratic Overview of Control Theory

For a less revisionist picture of control, the reader is directed to one of the popular textbooks, such as Craig (1986) or VanLandingham (1985)

The field of control might be abstractly defined as the study of methods for manipulating the inputs to a system (called a *plant* or, here, sometimes an *arm*) in order to cause it to behave in a desired fashion. This is conventionally divided into open-loop control, in which the inputs are not modulated in response to the outputs of the plant, and closed-loop control, in which the inputs respond to the outputs of the plant being controlled.

The field of control is further characterized by the nature of the systems to be dealt with. They are in general continuous, rather than combinatorial (although the increasingly important area of dynamic programming control (Bellman, 1957) violates this dictum) and the control inputs are likewise continuous. Even with these restrictions, this would leave problems such as path planning within the domain of control. Indeed, problems like path planning are considered to be on the borders of control theory: essentially, when path planning does not involve combinatorial considerations, it is clearly control theory; when it does, it depends on the techniques being used, *etc.* To grossly oversimplify, mainstream control theory is mostly about low-level control, in which the desired response of the plant is known, and the job of the controller is to ensure that the plant continuously tracks a given reference signal.

Under these conditions, almost all plants are well approximated as linear systems in a local neighborhood containing both the current state of the plant and its desired state, plus some constant term.

2.1.1 Linear Control Theory

The optimal control of linear systems is the field of linear control theory. In linear control theory, an n^{th} order plant can be characterized by the linear equations

$$\begin{aligned} x(t+1) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) \end{aligned}$$

where the vector x is the n-dimensional state of the system, the vector u is the control input, t is time, and the vector y is the output available to the feedback controller.

It can be shown that such a system is *observable,* meaning that its internal state x can be determined by observation of y, if and only if the observability matrix

$$M_o = [C|CA|\cdots|CA^{n-1}]$$

(where the vertical bars denote matrix concatenation) is of rank n.

It can be also be shown that such a linear system has the dual property of *controllability*, meaning that the system can be driven to a desired state $x = x_d$ by suitable choice of u, if and only if the controllability matrix

$$M_c = [B|AB|\cdots|A^{n-1}B]$$

is of rank n. If the system is controllable then any point can be reached in n timesteps, and if not then only the subspace spanned by the columns of M_c can be reached.

Similar notions hold for continuous time linear systems, which can in fact be treated by the above formalism by assuming the system to be sampled at some rate.

It is straightforward to design a suitable controller given knowledge of A, B, and C. In a *proportional derivative* or *PD* controller the control signals are generated by a linear function of the difference between the plant output (and its first derivative) and the target values. These simple PD controllers suffice across a broad range of control situations, but frequently exhibit long-term error in the presence of persistent disturbances. Their extension, the *proportional integral derivative* or *PID* controller, adds a simplistic ability to compensate for persistent disturbances by adding to the PD controller's outputs a linear function of the integrated distance from their targets of the plant outputs, to counteract long-term disturbances. Such PID controllers are adequate for many real-world control applications.

There are two broad families of techniques for generating suitable controllers for a plant: direct methods and indirect methods. In indirect methods, one uses feedback

from the plant to determine the parameters of a plant model, and then uses this plant model to generate a suitable controller. In direct methods, one uses the feedback from the plant to directly construct a controller, without attempting to model the plant.

2.1.2 Linearization and Dynamics Compensation

Given the above linear control theory, and the further theory that allows one to easily create an optimal controller for a given linear system, one approach to control a plant is to assume that the constant terms on local linear approximations to a plant are small, and use a linear controller suitable to the region of state space which the plant occupies at each instant. Thus, one is faced with the problem of either finding a locally linear model of the plant, if one is performing indirect control; or of finding a controller suitable for the current local linear neighborhood, assuming one is doing direct control.

A problem with this approach is that the assumption that the constant offsets are small is typically violated. The obvious response is to cancel this constant term precisely, at each point in phase space, and use the above approach. This problem, of canceling the offsets, is called *dynamics compensation* and involves the construction of an inverse dynamical model of the plant. This inverse dynamical model solves for the required plant inputs needed to exert forces that maintain the plant's dynamic state.

To be concrete: for a robot arm, the torques necessary to obtain given accelerations at a joint depend on the configuration of the arm. For instance, the shoulder motor must exert greater torque to accelerate an arm when the elbow is extended than when it is bent. Also, if zero torque is exerted at all the joints, the arm will slump downward because of gravity. Compensating for this force of gravity is also part of dynamics compensation—but typically, for a robot arm, the problem is decoupled into forces that do not change with motion, the kinematic dynamics, and those that are a function of velocity, like centrifugal forces.

Similarly, gravity and centrifugal (coriolis) forces exert toques on the joints of a robot arm, and these depend on the configuration and velocity of the arm. There are two ways to handle such effects: to ignore them, and allow the controller to treat these forces as disturbances to be corrected, or to predict them and exert compensatory forces. The latter dynamics compensation approach demands that the compensatory control offsets be computed from the outputs of the plant. This is conventionally done by using physics and intimate knowledge of the structure of the arm (Hollerbach, 1980; Brady, Hollerbach, Johnson, Lozano-Perez, and Mason, 1982). In this chapter we show the feasibility of using neural networks to learn these compensatory forces.

There are many hundreds of papers on learning dynamics compensation, from table-based approaches (Raibert and Horn, 1978; Atkeson and Reinkensmeyer, 1988, 1990) to algebraic calculation of the inverse dynamics based on a model with few

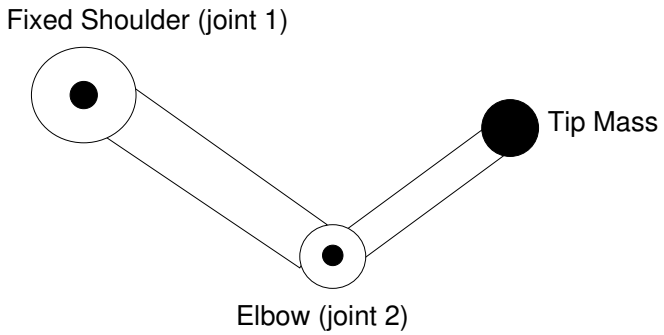


Figure 2.1: The two degree of freedom simulated direct drive arm taken from Brady *et al.* (1982).

free parameters (An, Atkeson, and Hollerbach, 1988a, 1988b) to neural networks (Kuperstein and Wang, 1990; Tzirkel-Hancock and Fallside, 1991; Kawato, Setoyama, and Suzuki, 1988; Goldberg and Pearlmutter, 1989). In this chapter we present an expanded version of this last paper.

2.2 Simulated Arm

The arm sketched in figure 2.1 was simulated. Its equations of motion, and values for the free parameters, were taken from Brady *et al.* (1982), and the gravitational term was set to zero. A network with 10 hidden units which was trained on 298 points, each of which consisted of a point chosen randomly and uniformly from the phase space of a simulated arm[1] and a corresponding random desired acceleration. The input to the network consisted of the positions, velocities and desired accelerations of the two joints, and the required output was the appropriate torques to exert. The network was trained in batch mode all the way to the bottom of its local minimum, *i.e.* to the point where the derivative of the error was nearly zero.[2] This training took 12,000 epochs (in an epoch the network is exposed to the entire corpus of training cases and the weights are updated) while the learning parameters were constantly adjusted for maximum learning speed.

Figure 2.2 shows a Hinton diagram of the weights of the trained network. It is

[1]The phase space of the arm consists of the positions and velocities of the two joints.

[2]In general, when training on a training corpus and testing using a different testing corpus, performance on the training corpus rises monotonically to an asymptote while performance on the test corpus first rises to a maximum and then falls to an asymptote. Many researchers, quite reasonably, use the best test corpus performance achieved as the generalization rate, which can be seen as a form of regularization. In our work, we use the more pessimistic asymptotic test corpus performance.

Figure 2.2: This Hinton diagram shows the weights in a recursive fashion: each of the twelve large structures represents a unit. The bottom row of each unit displays the weights from the inputs, the center two bars display the weights from the hidden units, and the two positions on the right in the top row represent the two output units. The remaining weight is the bias connection. The 10 units in the lower two rows are hidden; the two output units at the top are the torques for joints 1 and 2, from left to right. The input units (not displayed) are, from left to right, the position, velocity and desired acceleration of the shoulder joint and the position, velocity and desired acceleration of the elbow joint. The values displayed in the upper left hand corners of the units are biases. The magnitude of the largest weight is 9.63.

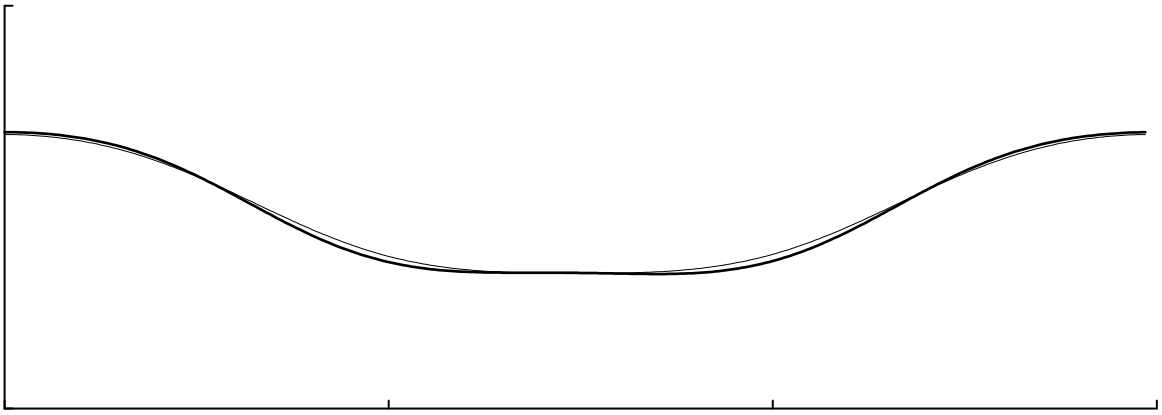
RMSS Error

Random Training Set	Random Test Set	Trajectory E
0.0682	0.0829	0.0676

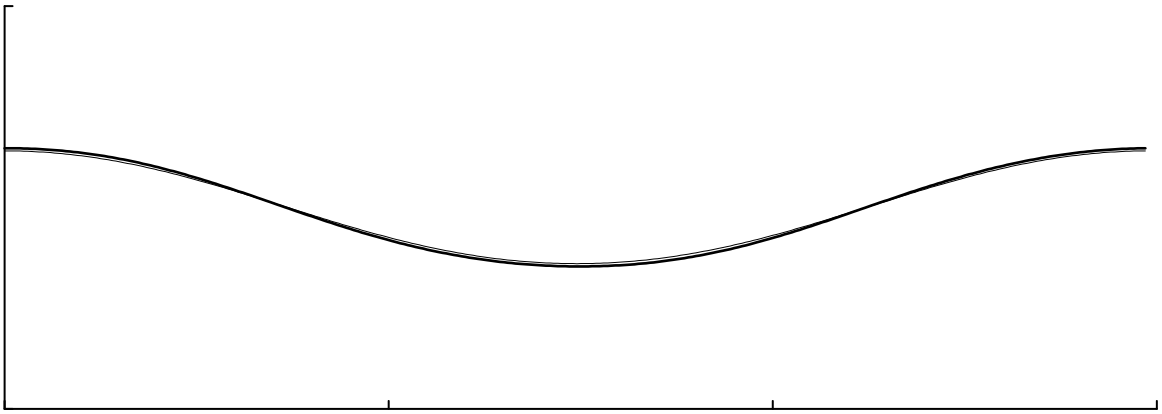
Table 2.1: Root mean square/standard deviation (RMSS) errors on various synthetic data sets. The training set and random test set each consist of 298 points chosen randomly with a uniform distribution from phase space and desired accelerations. Trajectory E involves moving both the shoulder and elbow joints through a sinusoid.

interesting to try to figure out how the network is performing this task; the hidden units each seem to be sensitive to only a particular range of velocities of the shoulder joint (joint 1). The position of the shoulder is almost completely ignored (the blob in the low left hand corner of each unit is nearly absent), as it should be because of the rotational symmetry of the plant and the lack of gravitational forces.

Quantitative measures of the performance of this network, both its training set and on some testing sets are shown in table 2.1. The excellent performance on the random test set indicates that the network has generalized well from its sample of 298 points. The performance on trajectory E, as well as the graphs shown in figure 2.3, show that the network performs well in that portion of phase space in which the



Torque for trajectory A joint 1



Torque for trajectory E joint 1

Figure 2.3: The network of figure 2.2 was used to generate torques to be applied in some simulated trajectories. The fine line is the required torque for the simulated arm, and the bold line is the torque output by the network.

trajectories lie.

2.3 Experiments on an Actual Robot Arm

The actual robot arm used for these experiments is the blue CMU Direct Drive Arm II, used with the kind and invaluable assistance of Pradeep Khosla. Other experiments were run on Lee Weiss' 2D direct drive arm, and we express our gratitude to him. Regretably, logistic difficulties prevented us from including data gathered from his arm in this document.

In our experiments with real robot arms, we primarily used variations of the architecture depicted in figure 2.4. The input to the networks consisted solely of a temporal window of positions for joint i, namely $x_i(t-n\Delta t), \ldots, x_i(t), \ldots, x_i(t+m\Delta t)$ for $i = 1, 2$; explicit velocity and acceleration data were not given to the networks. The required output was $\tau_i(t)$ for $i = 1, 2$, the torque vector that was applied to joint i at time t. This technique assumes that the torques are not varying too rapidly, an assumption which was guaranteed by our use in all the experiments described here of a Δt of 10 milliseconds.

A difficulty with this approach is gathering adequate sample data from the arm. Rather than attempt to sample all of phase space well, which is very difficult, we decided to use a family of trajectories and to sample a distribution on this family to obtain both training and testing trajectories. We trained the networks on 5 randomly chosen trajectories, and tested them on another randomly chosen 2. Plots of the actual torques overlayed with plots of the torques predicted by the network are shown in figure 2.5. We show here networks with three different window sizes: $n = m = 5$, $n = m = 10$, and $n = m = 20$, where n is the number of timesteps included in the window prior to the current time, and m is the number after. Thus the total window size is $n + m + 1$. After some experimentation we settled on simply using ten hidden units in all of our networks.

Table 2.2 shows performance by networks of various window sizes on both the data they were trained on and some independent data drawn from the same distribution (*i.e.* different trajectories but drawn from the same ensemble of trajectories) as the training data. This is the usual technique for testing generalization.

All of these networks had ten hidden units, and each was trained exhaustively, as in the earlier experiments with the simulated arm. Observe that performance on the training set gets better as the window gets larger, while performance on the test set first improves as window size grows, and then worsens. This evidence of a tradeoff between window size and generalization is a special case of the ubiquitous phenomenon of a bias/variance tradeoff. We conjecture that the improved generalization between a window of size $n = m = 5$ and a window of size $m = n = 10$ is caused by the fact that a window of size five simply does not see enough data to make sufficiently accurate estimates of the acceleration. In contrast, the network with a window of size of $m = n = 20$ seems to have used a portion of its extra capacity to memorize some of the training set, thus improving performance on the training set in a way that impairs generalization.

Evidence of this memorization is visible in figure 2.8, where some of the hidden units have receptive fields which have isolated black and white dots, an indication that they respond to some particular pattern of noise that occured in the training set.

Figures 2.6, 2.7, and 2.8 show Hinton diagrams of the weights developed by the networks after being trained on the training corpus well beyond the point of dimin-

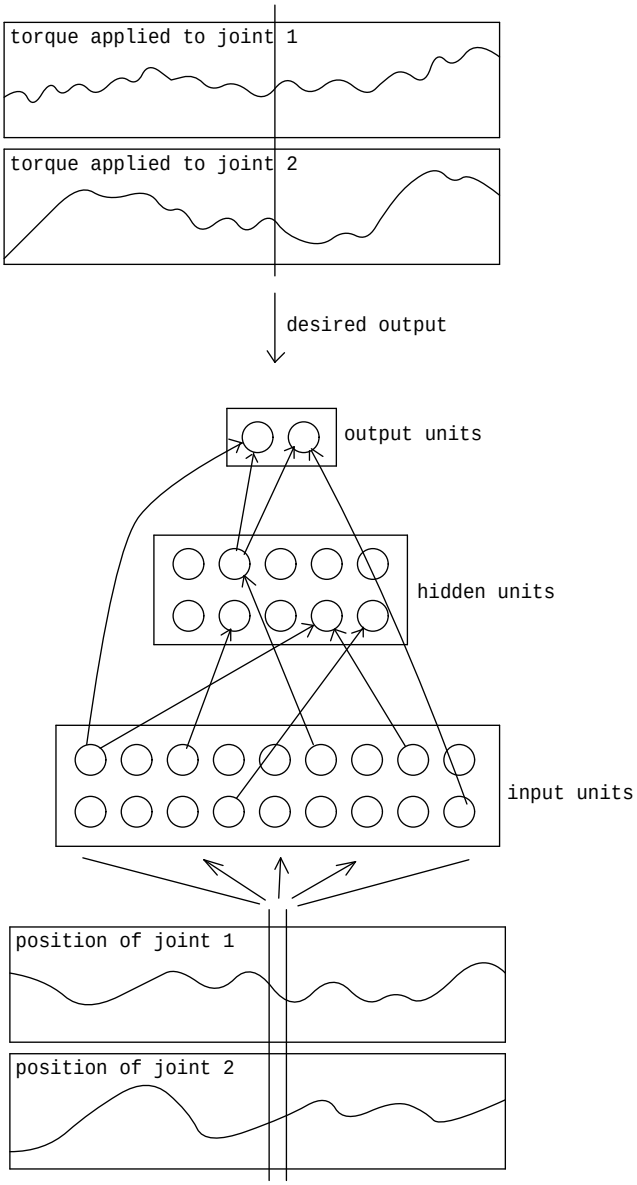


Figure 2.4: This architecture is used extensively below. The input is a window of joint positions; the desired output is the torques applied to the two joints at the center of the temporal window. The output units receive input from both the hidden units and the input units.

ishing returns, *i.e.* well beyond the point that the test set error began to rise. The top two are the output units, with the shoulder torque on the left and the elbow torque on the right. The rest are hidden units. Within each unit, the two stripes on the bottom show the weights of the incoming connections from the input units, with time starting on the left and moving to the right, and the position of the shoulder

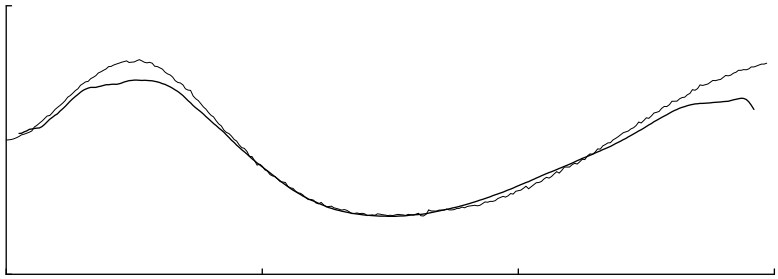
Window Size	Training Data	Test Trajectory E	Test Trajectory I
11	0.01296	0.04675 (3.6)	0.03573 (2.8)
21	0.00489	0.02224 (4.5)	0.02445 (5.0)
41	0.00435	0.02862 (6.6)	0.04074 (9.4)

Table 2.2: RMS errors of networks with different window sizes on both testing and training data. The ratio between the error shown and the error on the training set is in parenthesis.

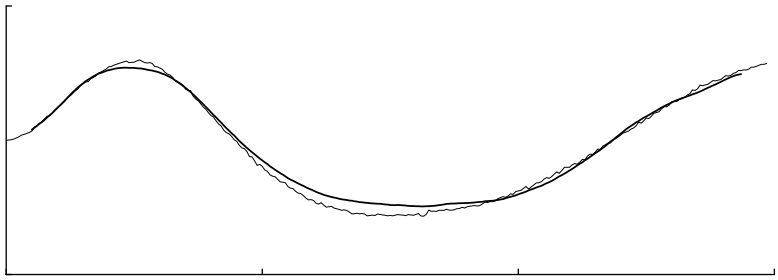
on the bottom row and the elbow just above it. The top two dots on each of the hidden units are the weights of its outgoing connections to the output units, which are also displayed in the middle portion of the output units. The single remaining unexplained blob is each unit's bias, the strength of a connection from a unit which is always on. White blobs are positive and black blobs are negative.

An attempt to figure out how the networks work yields some insights, although a complete understanding is virtually impossible. The easiest things to interpret are the weights of connections from the input units, which form temporally smooth filters shaped to detect a linear combination of position, velocity, and acceleration. At first glance most of the units appear to have one-dimensional mexican hat or difference of gaussian receptive fields, which leads one to suspect that they respond almost solely to acceleration. But on closer examination one sees that the zero crossings are frequently quite asymmetric, an indication that velocity is also being responded to. A yet closer examination, possible only with access to the actual weights, shows that the sum of all the elements of the filters is usually not zero, so the absolute position also has some influence. The linear combination of acceleration and velocity stands out in the network of figure 2.8, in which some of the filters are strikingly asymmetric. It should be remembered that the networks had no built in notion of temporal adjacency, but developed these filters purely in order to map each input to the appropriate output.

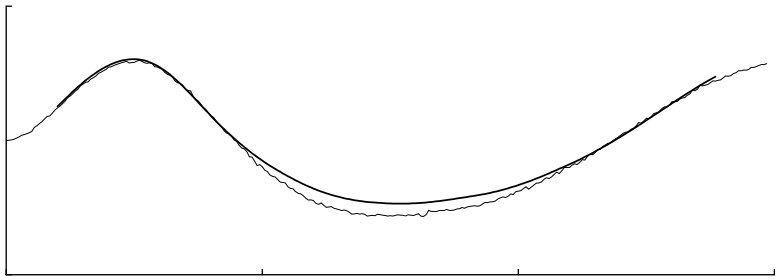
The roles of the individual hidden units are much more difficult to fathom. Since the weights are so large, most units are saturated most of the time. Each unit has a transition window in which it is not saturated which is reached only under the correct circumstances. For example, a unit might respond to $3.4x_1 + 1.2dx_1/dt - 2.1x_2$, being saturated at 0 if this value is less than 2.3 and at 1 if the value is greater than 2.6, and having a non-binary value only within that narrow range. Thus, the input space is chopped up into soft hyperplanes along these dimensions. The use made of the hidden units by the output units provides little information about their roles in the network in intuitive terms, as the values are used in concert, canceling each other out delicately under various circumstances. In a word, the networks are not modular: it is difficult to understand the roles of the various pieces in isolation.



Window Size 11 (see figure 2.6)



Window Size 21 (see figure 2.7)



Window Size 41 (see figure 2.8)

Figure 2.5: These graphs show torque profiles that are to drive joint 1 (the shoulder) through trajectory E, which was the more difficult of the two test trajectories. The "correct" torque profile is drawn with a fine line, while the torque profiles generated by networks are drawn with bold lines. Note the high frequency artifacts in the target profiles, which result from physical chatter and vibration. These are due to the gain of the PD controller being near the limits of the system.

2.4 Analysis of the Weights as Convolutions

We can interpret the weights between the first and third layers (shown in the upper right-hand parts of figures 2.6 2.7, and 2.8) as temporally smooth filters shaped to detect a linear combination of position, velocity, and acceleration. As we saw before, on close examination one can see Mexican-hat shaped receptive fields with asymmetric

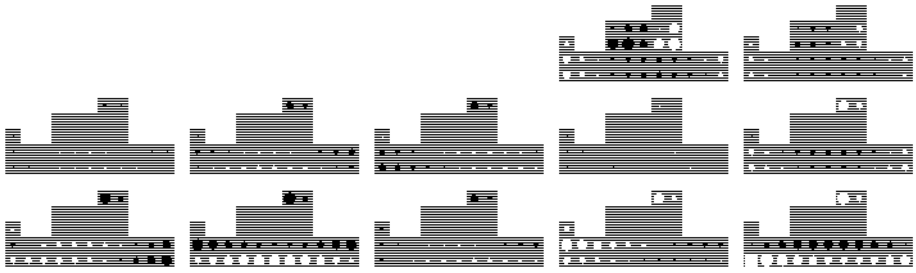


Figure 2.6: This network has a window size of $m = n = 5$. The largest weight has a magnitude of 18.9.

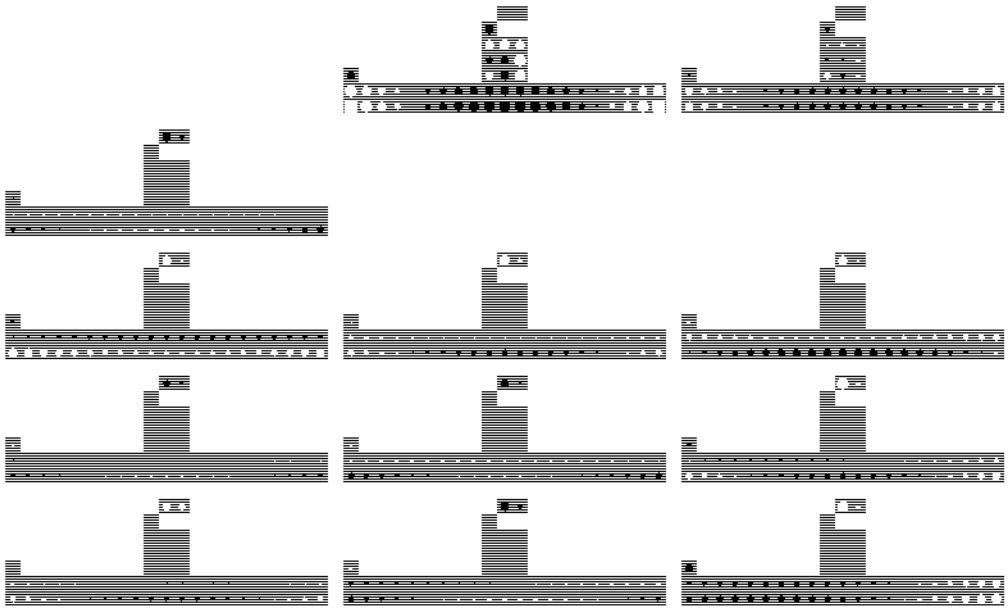


Figure 2.7: This network has a window size of $m = n = 10$. The largest weight has a magnitude of 11.7.

zero crossings, an indication that both position and velocity are being responded to. Because these filters are applied to the input at each possible offset, we can view them as convolution functions and analyze them formally in those terms.

If we imagine the weights to be samples from a continuous function that is convolved with the position of the joint, we can decompose this function $f(x)$, $-w \leq x \leq w$, into a constant part f_c, an odd part $f_o(x)$, and an even part $f_e(x)$, using the equations

$$f_c = \frac{1}{2w} \int_{-w}^{w} f(x)\, dx$$

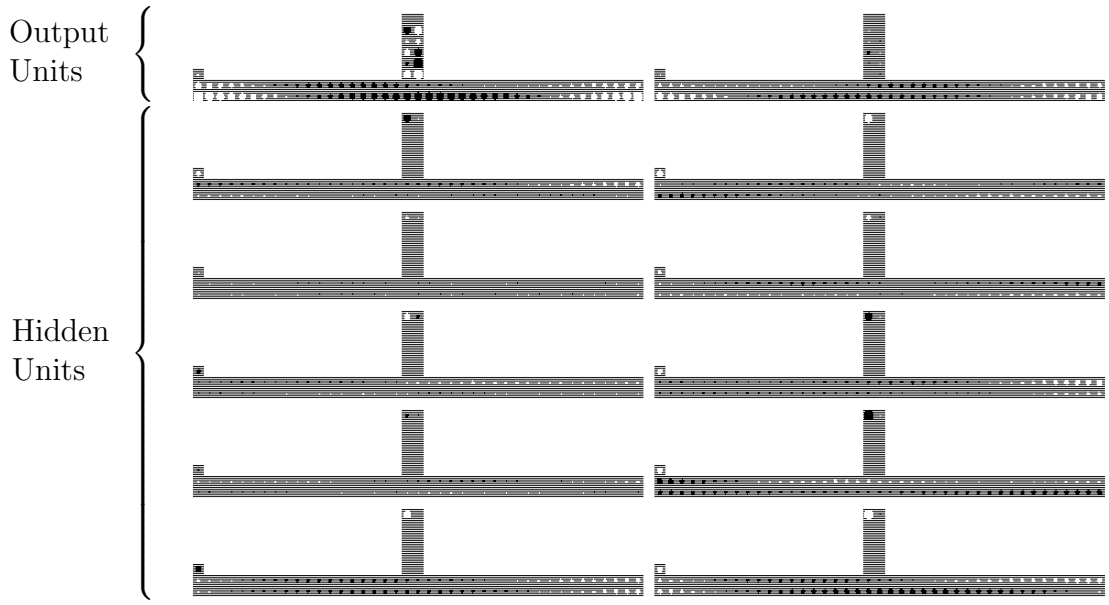


Figure 2.8: This network has a window size of $m = n = 20$. The largest weight has a magnitude of 4.2.

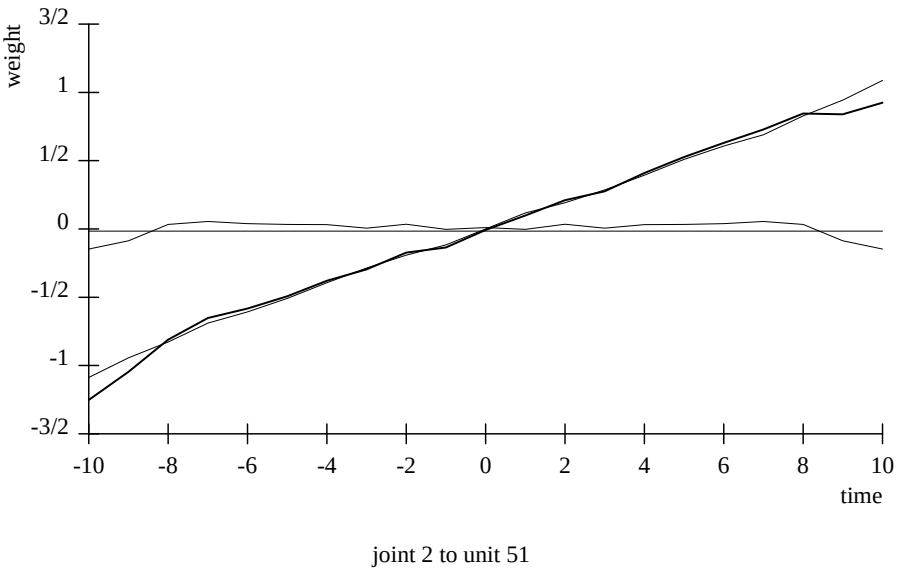


Figure 2.9: A graph of the inputs to a unit from the network with a window size of 21 regarded as a convolution function (bold line) and decomposed into constant, even, and odd components. These weights appear to form a velocity filter.

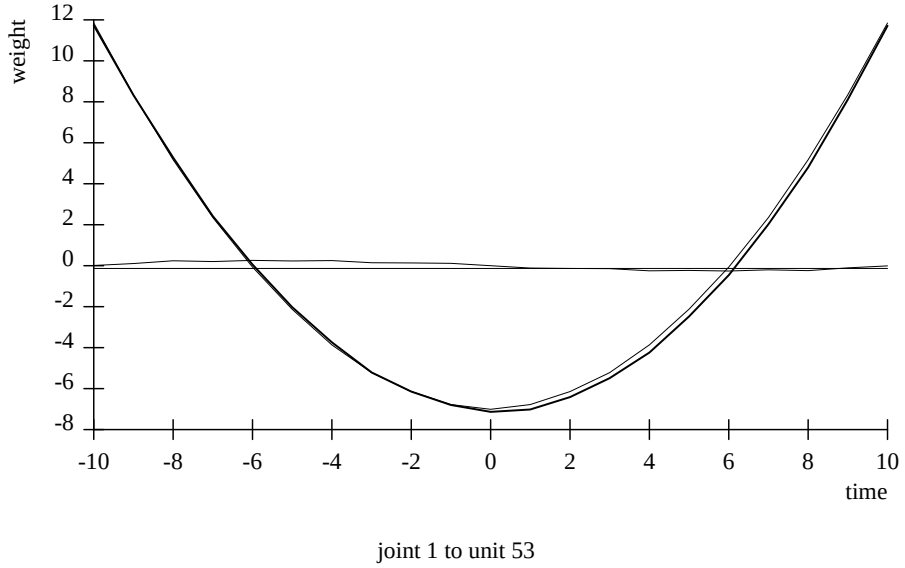


Figure 2.10: A graph of the inputs to a unit from the network with a window size of 21 regarded as a convolution function (bold line) and decomposed into constant, even, and odd components. These weights appear to form an acceleration filter.

$$\begin{aligned} f_e(x) &= \frac{f(x) + f(-x)}{2} - f_c \\ f_o(x) &= \frac{f(x) - f(-x)}{2} \end{aligned}$$

which have the property that $f_o(x) = f_o(-x)$, $f_e(x) = -f_e(-x)$, and $f(x) = f_e(x) + f_o(x) + f_c$. The f_c term is not strictly necessary, but using it to take out the pure dc component enables the visual display of the position sensitivity, and also permits the visual interpretation of the odd component as being acceleration (and higher order derivative) sensitive, without contaminating it with position sensitivity, which would then be visually indiscernible.

Figure 2.9 shows that the weights on the connections from the elbow joint to a particular unit from the network with a window size of 21 can be understood as a simple velocity filter. Figure 2.10 shows another unit from that network whose weights from the shoulder joint form a simple acceleration filter, and figure 2.11 shows a unit which is activated by a linear combination of velocity and acceleration.

The filter functions we have examined so far have been extremely smooth. This is generally the case in the network with window size 21, but in the network with window size 41 a new phenomenon appears. In figure 2.12 we see some curves with rough edges. Below, when we look at generalization, we shall partly account for this behavior. Notice that in figure 2.12, the odd component crosses zero near the edges of the window, suggesting that this unit responds in part to jerk (the derivative of

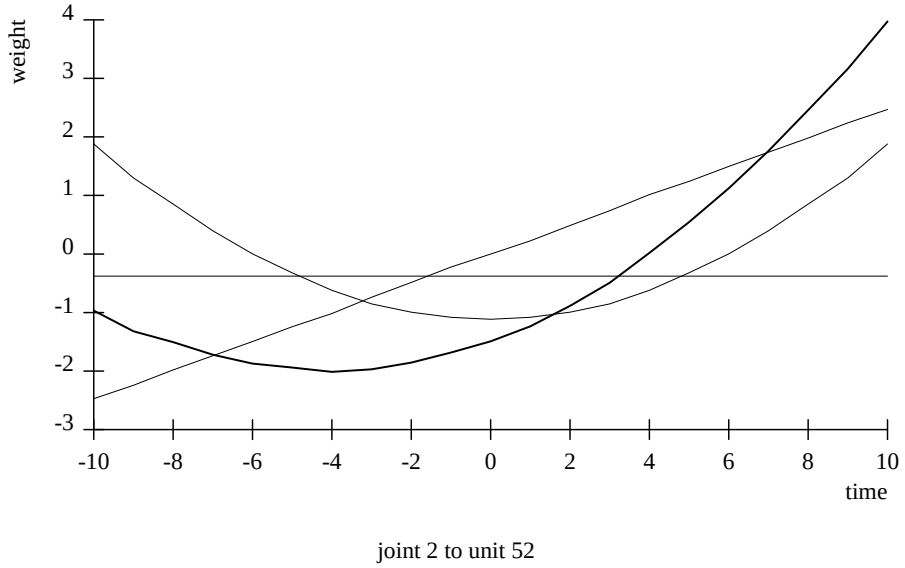


Figure 2.11: A graph of the inputs to a unit from the network with a window size of 21 regarded as a convolution function (bold line) and decomposed into constant, even, and odd components. These weights appear to form a filter that responds to a combination of velocity and acceleration.

acceleration.)

The concern that any function can be decomposed into even and odd componants, and that this sort of analysis of even random weights would result in finding filters for velocity, acceleration, *etc*, can be easily dismissed. In figure 2.13 we see a pattern of weights which is not smooth, and thus the decomposition in these terms proves entirely unilluminating.

2.5 Conclusion

In this chapter we have seen that a simple simulated inverse kinematics problem and an inverse dynamics problem are both easily solved using standard backpropagation neural network technology. In the latter case, temporal windows were a crucial additional technique used to allow the extraction of relevent higher-order properties of the input signal (velocity, acceleration, *etc.*) without the necessity for providing these inputs manually. Performance on both training and testing sets was good, with the errors generally being smaller than perturbations caused by other effects upon the system. Examination of the weights of the trained networks gave insight into their function, and trying networks of different sizes explored various points along the bias/variance spectrum. As a toy application of neural networks, this effort was

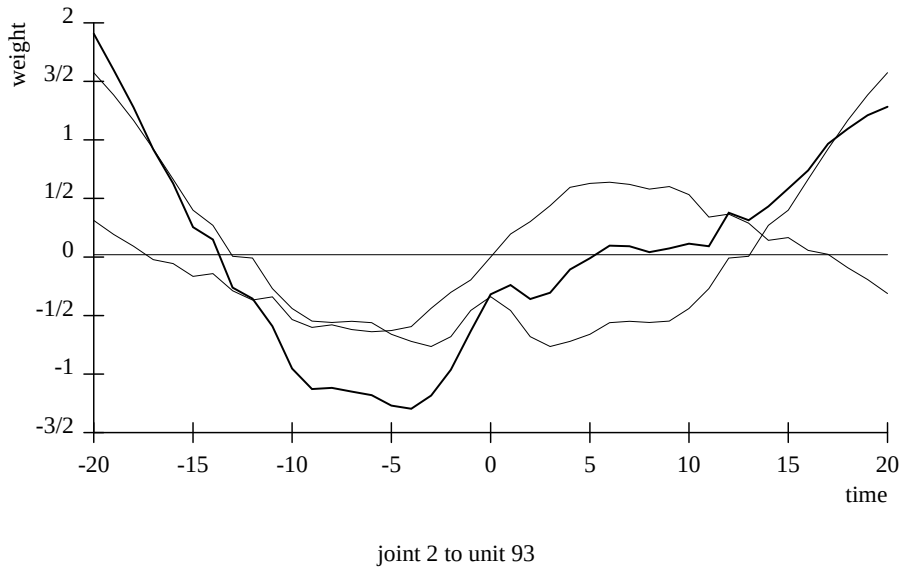


Figure 2.12: A graph of the inputs to a unit from the network with a window size of 41 regarded as a convolution function (bold line) and decomposed into constant, even, and odd components. These weights appear to form a filter that responds to a combination of acceleration and jerk, and also to a particular pattern of noise.

successful. However, the training was slow, and we therefore now turn our attention to why training can be slow, how the factors leading to slow training can be measured, and how the training can be accelerated.

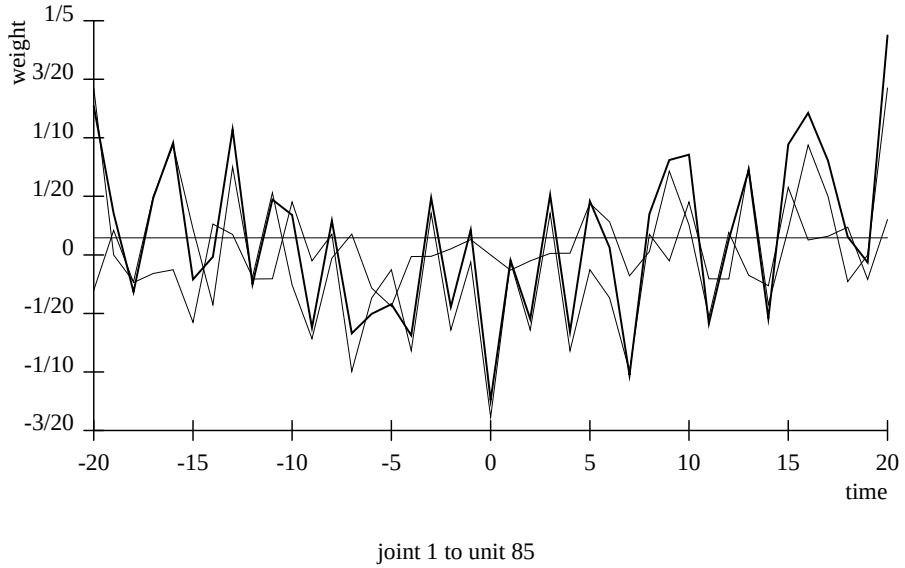


Figure 2.13: A graph of the inputs to a unit from the network with a window size of 41 regarded as a convolution function (bold line) and decomposed into constant, even, and odd components. These weights do not appear to be amenable to analysis as a temporal filter, as they are not even piecewise smooth.

Chapter 3

Gradient Calculation for Dynamic Recurrent Neural Networks

Some of the work described in this chapter was included in Pearlmutter (1989a, 1989b, 1990a, 1990b, 1992b, 1995)

Capsule: We survey learning algorithms for recurrent neural networks with hidden units, and put the various techniques into a common framework. We discuss fixedpoint learning algorithms, namely recurrent backpropagation and deterministic Boltzmann Machines, and non-fixedpoint algorithms, namely backpropagation through time, Elman's history cutoff, and Jordan's output feedback architecture. Forward propagation, an online technique that uses adjoint equations, and variations thereof, are also discussed. In many cases, the unified presentation leads to generalizations of various sorts. We discuss advantages and disadvantages of temporally continuous neural networks in contrast to clocked ones, and continue with some "tricks of the trade" for training, using, and simulating continuous time and recurrent neural networks. We present some simulations, and at the end, address issues of computational complexity and learning speed.

3.1 Introduction

3.1.1 Why Recurrent Networks

The motivation for exploring recurrent architectures is their potential for dealing with two sorts of temporal behavior. First, recurrent networks are capable of settling to a

solution that satisfies many constraints (McClelland, Rumelhart, and Hinton, 1986), as in a vision system which relaxes to an interpretation of an image which maximally satisfies a complex set of conflicting constraints (Marr and Poggio, 1976, 1978; Hinton, 1977; Davis and Rosenfeld, 1981; Szeliski, 1986), a system which relaxes to find a posture for a robot satisfying many criteria (Hinton, 1976), and models of language parsing (Waltz and Pollack, 1985). Although algorithms suitable for building systems of this type are reviewed to some extent below, such as the algorithm used in Qian and Sejnowski (1989), the bulk of this chapter is concerned with the problem of causing networks to exhibit particular desired detailed temporal behavior, which has found application in signal processing (Nerrand, Roussel-Ragot, L. Personnaz, and Marcos, 1993; Karjala, Himmelblau, and Miikkulainen, 1992), speech and language processing (Watrous, Laedendorf, and Kuhn, 1990; Poddar and Unnikrishnan, 1991; Albesano, Gemello, and Mana, 1992), and neuroscience (Lockery, Fang, and Sejnowski, 1990; Doya, Boyle, and Selverston, 1993; Doya, Selverston, and Rowat, 1994).

It should be noted by engineers that many real-world problems which one might think would require recurrent architectures for their solution turn out to be solvable with feedforward architectures, sometimes augmented with preprocessed inputs such as tapped delay lines, and various other architectural embellishments (Chauvin and Rumelhart, 1995; Gorman and Sejnowski, 1988; Lang and Hinton, 1988; Waibel, Hanazawa, Hinton, Shikano, and Lang, 1989; Lang, Hinton, and Waibel, 1990; Lang and Hinton, 1990; Tank and Hopfield, 1987a; Day and Davenport, 1993; Jordan, 1989; on Underspecified Target Trajectories, 1990; Weigend, Rumelhart, and Huberman, 1991; de Vries and Principe, 1991; Back and Tsoi, 1991; Juang, Kung, and Camm, 1991; Nerrand *et al.*, 1993; Hush and Horne, 1993; de Vries and Principe, 1992; Maxwell, Giles, Lee, and Chen, 1986; LeCun, Boser, Denker, Henderson, Howard, Hubbard, and Jackel, 1989; Tsung and Cottrell, 1995). For this reason, if one is interested in solving a particular problem, it would be only prudent to try a variety of non-recurrent architectures before resorting to the more powerful and general recurrent networks.

This chapter is concerned with learning algorithms for recurrent networks themselves, and not with recurrent networks as elements of larger systems, such as specialized architectures for control (Kawato *et al.*, 1988; Jordan and Jacobs, 1990; Narendra and Parthasarathy, 1990; Miller, Sutton, and Werbos, 1990). Also, since we are concerned with learning, we will not discuss the computational power of recurrent networks considered as abstract machines (Siegelmann and Sontag, 1991; Kilian and Siegelmann, 1993; Siegelmann and Sontag, 1993). Although we consider techniques for trajectory learning, we will not review practical applications thereof. In particular, grammar learning, although intriguing and progressing rapidly (Cottrell and Small, 1983; Cleeremans, Servan-Schreiber, and McClelland, 1989; Watrous and Kuhn, 1992; Giles, Miller, Chen, Sun, Chen, and Lee, 1992; Mozer and Das, 1993; Das, Giles, and Sun, 1993; Kolen, 1994; Das and Mozer, 1994), typically involves recurrent neural networks as components of more complex systems, and also at present is inferior in

practice to discrete algorithmic techniques (Angluin, 1987; Lang, 1992). Grammar learning is therefore beyond our scope here. Similarly, learning of multiscale phenomena, which again typically consists of larger systems containing recurrent networks as components (Hochreiter, 1991; Mozer, 1992; Schmidhuber, 1992c, 1992d), will not be discussed.

3.1.2 Why Hidden Units

We will restrict our attention to training procedures for networks which may include *hidden units,* units which have no particular desired behavior and are not directly involved in the input or output of the network. They can be thought of as non-observable intermediate stages in the process leading from input to output.

With the practical successes of backpropagation, it seems gratuitous to expound the virtues of hidden units and internal representations. Hidden units make it possible for networks to discover and exploit regularities of the task at hand, such as symmetries or replicated structure (Hinton, 1986; Sejnowski, Kienker, and Hinton, 1986), and training procedures capable of exploiting hidden units, such as the Boltzmann machine learning procedure (Ackley, Hinton, and Sejnowski, 1985) and backpropagation (Rumelhart, Hinton, and Williams, 1986b; Werbos, 1974; Parker, 1985; Howard, 1960), are behind much of the current excitement in the neural network field (Touretzky and Pomerleau, 1989). Also, training algorithms that do not operate with hidden units, such as the Widrow-Hoff LMS procedure (Widrow and Hoff, 1960), can be used to train recurrent networks without hidden units, so recurrent networks without hidden units reduce to *non*-recurrent networks without hidden units, and therefore do not need special learning algorithms.

Consider a neural network governed by the equations

$$\frac{dy}{dt} = f(y(t), w, I(t)) \tag{3.1}$$

where y is the time-varying state vector, w the parameters to be modified by the learning, and I a time-varying vector of external input. Given some error metric $E'(y,t)$, our task is to modify w to reduce $E = \int E'(y,t)dt$. Our strategy will be gradient descent, so the main portion of our work will be finding algorithms to calculate the gradient $\nabla_w E$, the vector whose elements are $\partial E/\partial w_i$.

The above formulation is for a continuous time system. The alternative to this is a clocked system, which obeys an equation of the form $y(t + \Delta t) = f(y(t), w, I(t))$. Without loss of generality, for clocked systems we will use $\Delta t = 1$, giving

$$y(t + 1) = f(y(t), w, I(t)), \tag{3.2}$$

with t an integer.

Certainly, barring high-frequency components in I, the behavior of (3.1) can be precisely duplicated by (3.2) with suitable choice of f in the latter. For this reason, in order to determine the practical tradeoffs of one against the other, we must consider particular functional forms for f. We will consider the most common neural network formulation,

$$\frac{dy_i}{dt} = -y_i + \sigma(x_i) + I_i \tag{3.3}$$

where y_i is the state or activation level of unit i,

$$x_i = \sum_j w_{ji} y_j \tag{3.4}$$

is the total input to unit i, w_{ij} is the strength of the connection from unit i to unit j, and σ is a differentiable function.[1] The initial conditions $y_i(t_0)$ and driving functions $I_i(t)$ are the inputs to the system.

This defines a rather general dynamic system. Even assuming that the external input terms $I_i(t)$ are held constant, it is possible for the system to exhibit a wide range of asymptotic behaviors. The simplest is that the system reaches a stable fixedpoint; in the next section, we will discuss two different techniques for modifying the fixedpoints of networks that exhibit them.

More complicated possible asymptotic behaviors include limit cycles and even chaos. Later, we will describe a number of gradient based training procedures that can be applied to training networks to exhibit desired limit cycles, or particular detailed temporal behavior. We will not discuss specialized non-gradient methods for learning limit cycle attractors, such as Baird (1990), Baird and Eeckman (1991). Although it has been theorized that chaotic dynamics play a significant computational role in the brain (Skarda and Freeman, 1987; Freeman, 1987), there are no specialized training procedures for chaotic attractors in networks with hidden units. However, Crutchfield and McNamara (1987) and Lapedes and Farber (1987) have had success with the identification of chaotic systems using models without hidden state, and there is no reason to believe that learning the dynamics of chaotic systems is more difficult than learning the dynamics of non-chaotic ones.

Special learning algorithms are available for various restricted cases. There are fixedpoint learning algorithms (for details see Pineda (1987), Almeida (1987), Hinton (1989), Baldi and Pineda (1991), or for a survey see Pearlmutter (1990a)) that take advantage of the special relationships holding at a fixedpoint to reduce the storage requirements to $O(m)$, the number of weights, and the time requirements to the time required for the network to settle down. There are continuous-time feed-forward

[1] Typically $\sigma(\xi) = (1+e^{-\xi})^{-1}$, in which case $\sigma'(\xi) = \sigma(\xi)(1-\sigma(\xi))$, or the scaled $\sigma(\xi) = \tanh(\xi)$, in which case $\sigma'(\xi) = (1+\sigma(\xi))(1-\sigma(\xi)) = 1-\sigma^2(\xi)$. The latter symmetric squashing function is usually preferable, as it leads to a better conditioned Hessian, which speeds gradient descent (LeCun, Kanter, and Solla, 1991). However, the former was used in all the simulations presented in this chapter.

learning algorithms that are as efficient in both time and space as algorithms for pure feedforward networks, but are applicable only when w is upper-triangular but not necessarily zero-diagonal, in other words, when the network is feedforward except for recurrent self-connections (surveyed in Pearlmutter (1990b), or see Gori, Bengio, and de Mori (1989), Kuhn (1987), Mozer (1989), Uchiyama, Shimohara, and Tokunaga (1989), Day and Davenport (1993) for more detail.)

Later, we will describe a number of training procedures that, for a price in space or time, do not rely on such restrictions and can be applied to training networks to exhibit desired limit cycles, or particular detailed temporal behavior.

3.1.3 Continuous vs. Discrete Time

We will be concerned predominantly with continuous time networks, as in (3.3). However, all of the learning procedures we will discuss can be equally well applied to discrete time systems, which obey equations like (3.2). Continuous time has advantages for expository purposes, in that the derivative of the state of a unit with respect to time is well defined, allowing calculus to be used instead of tedious explicit temporal indexing, making for simpler derivations and exposition.

When a continuous time system is simulated on a digital computer, it is usually converted into a set of simple first order difference equations, which is formally identical to a discrete time network. However, regarding the discrete time network running on the computer as a simulation of a continuous time network has a number of advantages. First, more sophisticated and faster simulation techniques than simple first order difference equations can be used (Press, Flannery, Teukolsky, and Verrerling, 1988). Second, even if simple first order equations are used, the size of the time step can be varied to suit changing circumstances; for instance, if the network is being used for a signal processing application and faster sensors and computers become available, the size of the time step could be decreased without retraining the network. Third, because continuous time units are stiff in time, they tend to retain information better through time. Another way of putting this is that their bias in the learning theory sense is towards temporally continuous tasks, which is certainly advantageous if the task being performed is in fact temporally continuous.

Another advantage of continuous time networks is somewhat more subtle. Even for tasks which themselves have no temporal content, such as constraint satisfaction, the natural way for a recurrent network to perform the required computation is for each unit to represent nearly the same thing at nearby points in time. Using continuous time units makes this the default behavior; in the absence of other forces, units will tend to retain their state through time. In contrast, in discrete time networks, there is no *a priori* reason for a unit's state at one point in time to have any special relationship to its state at the next point in time.

A pleasant added benefit of units tending to maintain their states through time

is that it helps make information about the past decay more slowly, speeding up learning about the relationship between temporally distant events (Bengio, Simard, and Frasconi, 1994; Hihi and Bengio, 1996; Hochreiter, 1991).

3.2 Learning in Networks with Fixedpoints

The fixedpoint learning algorithms we will discuss assume that the networks involved converge to stable fixedpoints.[2] Networks that converge to fixedpoints are interesting because of the class of things they can compute, in particular constraint satisfaction and associative memory tasks. In such tasks, the problem is usually given to the network either by the initial conditions or by a constant external input, and the answer is given by the state of the network once it has reached its fixedpoint. This is precisely analogous to the relaxation algorithms used to solve such things as steady state heat equations, except that the constraints need not have spatial structure or uniformity.

3.2.1 Will a Fixedpoint Exist?

One problem with fixedpoints is that recurrent networks do not always converge to them. However, there are a number of special cases that guarantee convergence to a fixedpoint.

- Some simple linear conditions on the weights, such as zero-diagonal symmetry ($w_{ij} = w_{ji}$, $w_{ii} = 0$) guarantee that the Lyapunov function

$$L = -\sum_{i,j} w_{ij} y_i y_j + \sum_i (y_i \log y_i + (1 - y_i) \log(1 - y_i)) \tag{3.5}$$

  decreases until a fixedpoint is reached (Cohen and Grossberg, 1983). This weight symmetry condition arises naturally if weights are considered to be Bayesian constraints, as in Boltzmann Machines (Hinton and Sejnowski, 1983).

- A unique fixedpoint is reached regardless of initial conditions if $\sum_{ij} w_{ij}^2 < \max(\sigma')$ where $\max(\sigma')$ is the maximal value of $\sigma'(x)$ for any x (Atiya, 1988), but in practice much weaker bounds on the weights seem to suffice, as indicated by empirical studies of the dynamics of networks with random weights (Renals and Rohwer, 1990).

- Other empirical studies indicate that applying fixedpoint learning algorithms stabilizes networks, causing them to exhibit asymptotic fixedpoint behavior

[2] Technically, these algorithms only require that a fixedpoint be reached, not that it be stable. However, it is unlikely (with probability zero) that a network will converge to an *unstable* fixedpoint, and in practice the posibility of convergence to unstable fixedpoints can be safely ignored.

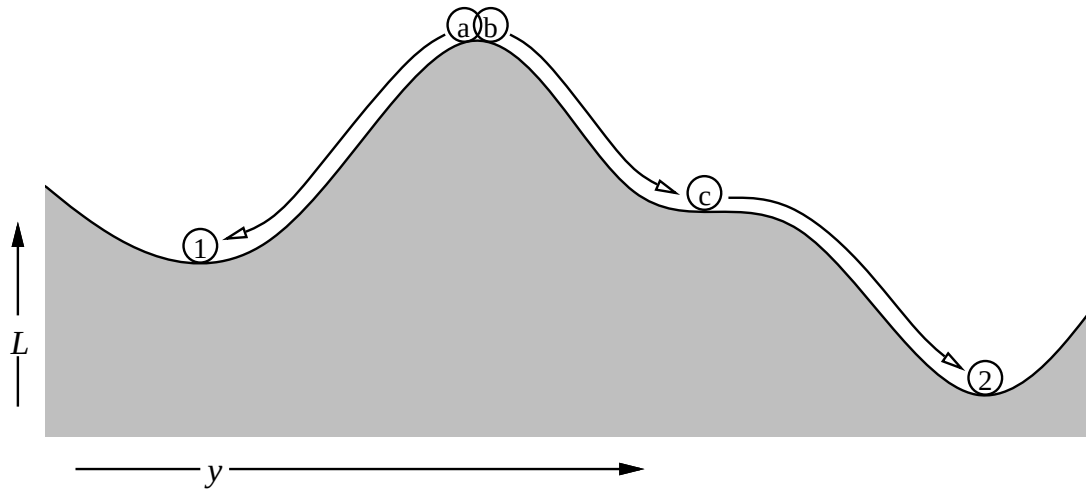


Figure 3.1: This energy landscape, represented by the curved surface, and the balls, representing states of the network, illustrate some potential problems with fixedpoints. The initial conditions **a** and **b** can differ infinitesimally but map to different fixedpoints, so the mapping of initial conditions to fixedpoints is not continuous. Likewise, an infinitesimal change to the weights can change which fixedpoint the system evolves to from a given starting point by moving the boundary between the basins of attraction of two attractors. Similarly, point **c** can be changed from a fixedpoint to a non-fixedpoint by an infinitesimal change to the weights.

> (Allen and Alspector, 1989; Galland and Hinton, 1989). There is as yet no theoretical explanation for this phenomenon, and it has not been replicated with larger networks.

One algorithm that is capable of learning fixedpoints, but does not require the network being trained to settle to a fixedpoint in order to operate, is backpropagation through time (Rumelhart *et al.*, 1986b). This has been used by Nowlan to train a constraint satisfaction network for the eight queens problem, where shaping was used to gradually train a discrete time network without hidden units to exhibit the desired attractors (Nowlan, 1988). However, the other fixedpoint algorithms we will consider take advantage of the special properties of a fixedpoint to simplify the learning algorithm.

3.2.2 Problems with Fixedpoints

Even when it can be guaranteed that a network settles to a fixedpoint, fixedpoint learning algorithms can still run into trouble. The learning procedures discussed

here all compute the derivative of some error measure with respect to the internal parameters of the network. This gradient is then used by an optimization procedure, typically some variant of gradient descent, to minimize the error. Such optimization procedures assume that the mapping from the network's internal parameters to the consequent error is continuous, and can fail spectacularly when this assumption is violated.

Consider mapping the initial conditions $\tilde{y}(t_0)$ to the resultant fixedpoints, $\tilde{y}(t_\infty) = \mathcal{F}(\tilde{y}(t_0))$. Although the dynamics of the network are all continuous, $\mathcal{F}$ need not be. For purposes of visualization, consider a symmetric network, whose dynamics thus cause the state of the network to descend the energy function of equation (3.5). As shown schematically in figure 3.1, even an infinitesimal change to the initial conditions, or to the location of a ridge, or to the slope of an intermediate point along the trajectory, can change which fixedpoint the system ends up in. In other words, $\mathcal{F}$ is not continuous. This means that as a learning algorithm changes the locations of the fixedpoints by changing the weights, it is possible for it to cross such a discontinuity, making the error jump suddenly; and this remains true no matter how gradually the weights are changed.

3.2.3 Recurrent Backpropagation

It was shown independently by Pineda (1987) and Almeida (1987) that the error backpropagation algorithm (Parker, 1985; Rumelhart *et al.*, 1986b; Werbos, 1974) is a special case of a more general error gradient computation procedure. The backpropagation equations are

$$\begin{aligned} x_i &= \sum_j w_{ji} y_j \\ y_i &= \sigma(x_i) + I_i \qquad (3.6) \\ z_i &= \sigma'(x_i) \sum_j w_{ij} z_j + e_i \qquad (3.7) \\ \frac{\partial E}{\partial w_{ij}} &= y_i z_j \qquad (3.8) \end{aligned}$$

where z_i is the ordered partial derivative of E with respect to y_i as defined in Werbos (1974), E is an error measure over $y(t_\infty)$, and $e_i = \partial E / \partial y_i(t_\infty)$ is the simple derivative of E with respect to the final state of a unit. In the original derivations of backpropagation, the weight matrix is assumed to be triangular with zero diagonal elements, which is another way of saying that the connections are acyclic. This ensures that a fixedpoint is reached, and allows it to be computed very efficiently in a single pass through the units. But the backpropagation equations remain valid even with recurrent connections, assuming a fixedpoint is found.

If we assume that equation (3.3) reaches a fixedpoint, which we will denote $y(t_\infty)$, then equation (3.6) must be satisfied. And if (3.6) is satisfied, and assuming we can

find z_i that satisfy (3.7), then (3.8) will give us the derivatives we seek, even in the presence of recurrent connections. (For a simple task, Ottaway, Simard, and Ballard (1989) reports that reaching the precise fixedpoint is not crucial to learning.)

One way to compute a fixedpoint for (3.6) is to relax to a solution. By subtracting y_i from each side, we get

$$0 = -y_i + \sigma(x_i) + I_i.$$

At a fixedpoint, $dy_i/dt = 0$, so the equation

$$k\frac{dy_i}{dt} = -y_i + \sigma(x_i) + I_i$$

has the appropriate fixedpoints. Now we note that when $-y_i + \sigma(x_i) + I_i$ is greater than zero, we can reduce its value by increasing y_i, so under these circumstances dy_i/dt should be positive, so k should be greater than zero. We can choose $k = 1$, giving (3.3) as a technique for relaxing to a fixedpoint of (3.6).

Equation (3.7) is linear once y is determined (y appears in the equation through the intermediate variable x, and also through the error terms e_i), so (3.7) has a unique solution. Any technique for solving a set of linear equations could be used. Since we are computing a fixedpoint of (3.6) using the associated differential equation (3.3), it is tempting to do the same for (3.7) using

$$\frac{dz_i}{dt} = -z_i + \sigma'(x_i)\sum_j w_{ij}z_j + e_i. \tag{3.9}$$

These equations admit to direct analog implementation. In a real analog implementation, different time constants would probably be used for (3.3) and (3.9), and under the assumption that the time y and z spend settling is negligible compared to the time they spend at their fixedpoints and that the rate of weight change η is slow compared to the speed of presentation of new training samples, the weights would likely be updated continuously by an equation like

$$\frac{dw_{ij}}{dt} = -\eta\frac{dE}{dw_{ij}} = -\eta y_i z_j \tag{3.10}$$

or, if a momentum term $0 < \alpha < 1$ is desired,

$$\frac{d^2w_{ij}}{dt^2} + (1-\alpha)\frac{dw_{ij}}{dt} + \eta y_i z_j = 0. \tag{3.11}$$

Simulation of an Associative Network

In this section we will simulate a recurrent backpropagation network learning a higher order associative task, that of associating three pieces of information: two four bit shift registers, A and B, and a direction bit, D. If D is off, then B is equal to A. If D

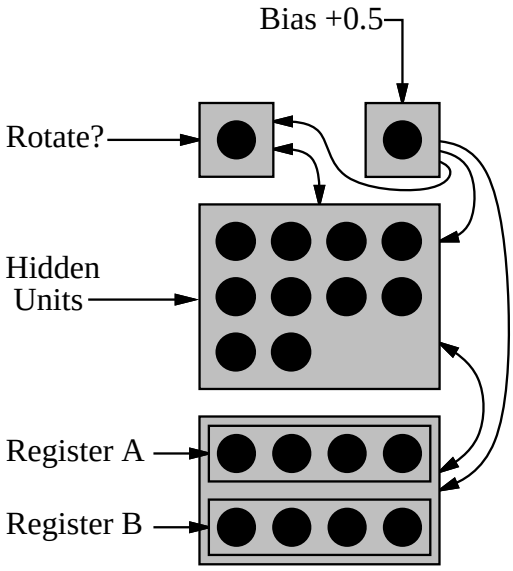


Figure 3.2: The architecture of a network to solve an associative version of the four bit rotation problem.

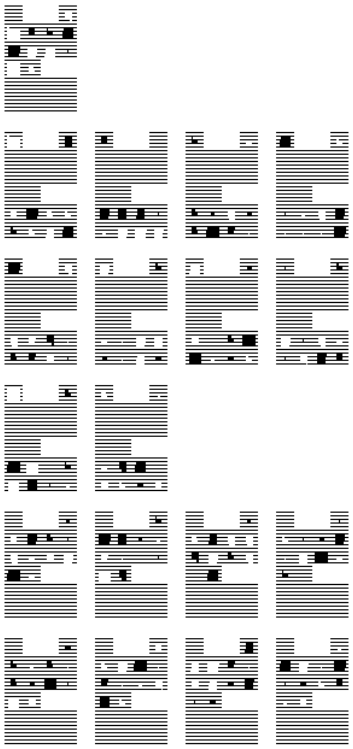


Figure 3.3: A Hinton diagram of weights learned by the network of figure 3.2.

is on, then B is equal to A rotated one bit to the right, with wraparound. The task is to reconstruct one of these three pieces of information, given the other two.

The architecture of the network is shown in figure 3.2. Three groups of visible units hold A, B, and D. An undifferentiated group of ten hidden units is fully and bidirectionally connected to all the visible units. There are no connections between visible units. An extra unit, called a bias unit, is used to implement thresholds. This unit has no incoming connections, and is forced to always have a value of 1 by a constant external input of 0.5. Connections go from it to each other unit, allowing units to have biases, which are equivalent to the negative of the threshold, without complicating the mathematics. Inputs are represented by an external input of $+0.5$ for an on bit, -0.5 for an off bit, and 0 for a bit to be completed by the network.

The network was trained by giving it external inputs that put randomly chosen consistent patterns on two of the three visible groups, and training the third group to attain the correct value. The error metric was the squared deviation of each I/O unit from its desired state, except that units were not penalized for being "too correct."[3] All 96 patterns were successfully learned, except for the ones which were

[3]A unit with external input could be pushed outside the [0,1] bounds of the range of the $\sigma(\cdot)$

Figure 3.4: Network state for all the cases in the four bit rotation problem. This display shows the states of the units, arranged as in figure 3.2. Each row of six shows one value for register A. There are $2^4 = 16$ such rows. Within each row, the three diagrams on the left show the network's state when completing the direction bit, register B, and register A, unshifted. The right three are the same, except with a shift. Note that all completions are correct except in the two cases where the rotation bit can not be determined from the two shift registers, namely a pattern of 0000 or 1111.

ambiguous, as shown in the state diagrams of figure 3.4. The weights after this training, which took about 300 epochs, are shown in figure 3.3. By inspection, many weights are large and decidedly asymmetric; but during training, no instabilities were observed. The network consistently settled to a fixedpoint within twenty simulated time units. When the network was tested on untrained completion problems, such

used.

as reconstructing D as well as half of A and B from partially, but unambiguously, specified A and B, performance was poor. Redoing the training with weight symmetry enforced, however, caused the network to learn not only the training data but also to do well on these untrained completions.

Qian and Sejnowski (1989) successfully applied the Pineda (1987), Almeida (1987) recurrent backpropagation learning procedure to learning weights for a relaxation procedure for dense stereo disparity problems with transparent surfaces. By training on examples, they were able to learn appropriate weights instead of deriving them from a simplified and unrealistic analytical model of the distribution of surfaces to be encountered, as is usual.

3.2.4 Deterministic Boltzmann Machines

The Mean Field form of the stochastic Boltzmann Machine learning rule, or MFT Boltzmann Machines, (Peterson and Anderson, 1987a) have been shown to descend an error functional (Hinton, 1989). Stochastic Boltzmann Machines themselves (Ackley *et al.*, 1985) are beyond our scope here; instead, we give only the probabilistic interpretation of MFT Boltzmann Machines, without derivation.

In a a deterministic Boltzmann Machine, the transfer function of (3.3) is $\sigma(\xi) = (1+e^{-\xi/T})^{-1}$, where T is the *temperature,* which starts at a high value and is gradually lowered to a *target temperature* each time the network is presented with a new input; without loss of generality, we assume this target temperature to be $T = 1$. The weights are assumed to be symmetric and zero-diagonal. Input is handled in a different way than in the other procedures we discuss: the external inputs I_i are set to zero, and a subset of the units, rather than obeying (3.3), have their values set externally. Such units are said to be *clamped.*[4]

In learning, a set of *input units* (states over which we will index with α) are clamped to some values, the output units are similarly clamped to their correct corresponding values, the network is allowed to settle, and the quantities

$$p_{ij}^{+} = \langle y_i y_j \rangle = \sum_{\alpha,\beta} P(\alpha) y_i^{(\alpha,\beta)} y_j^{(\alpha,\beta)}. \tag{3.12}$$

are accumulated, where $\langle \cdot \rangle$ denotes an average over the environmental distribution, the + superscript denote clamping of both input and output, and α is used to index the input units and β indexes the output units. The same procedure is then repeated, but with the output units (states of which we will index by β) not clamped, yielding

$$p_{ij}^{-} = \langle y_i y_j \rangle = \sum_{\alpha} P(\alpha) y_i^{(\alpha)} y_j^{(\alpha)} \tag{3.13}$$

[4]The clamping convention is for notational convenience only, as the two input schemes are formally equivalent, in that clamping can be used to implement external input (by allocating an extra unit whose outgoing weights are frozen at one), and external inputs can be used to implement clamping, by freezing to zero all the incoming weights of the unit to be clamped.

where the $-$ superscript denotes clamping of only the inputs and not the outputs. At this point, it is the case that

$$\frac{\partial G}{\partial w_{ij}} = p_{ij}^{+} - p_{ij}^{-} \tag{3.14}$$

where

$$G = \sum_{\alpha,\beta} P(\alpha) \log \frac{P(\beta|\alpha)}{P^{-}(\beta|\alpha)} \tag{3.15}$$

is a measure of the information theoretic difference between the clamped and unclamped distribution of the output units given the clamped input units. $P^{-}(\beta|\alpha)$ measures how probable the network says β is given α, and its definition is beyond the scope of this chapter, while $P(\beta|\alpha)$ is the probability of β being the correct output when α is the input, as given by the target distribution to be learned.

This learning rule (3.14) is a version of Hebb's rule in which the sign of synaptic modification is alternated, positive during the "waking" phase and negative during the "hallucinating" phase.

Even before the learning rule was rigorously justified, deterministic Boltzmann Machines were applied to a number of tasks (Peterson and Anderson, 1987b, 1987a). Although weight symmetry is assumed in the definition of energy which is used in the definition of probability, and is thus fundamental to these mathematics, it seems that in practice weight asymmetry can be tolerated in large networks (Galland and Hinton, 1989). This makes MFT Boltzmann Machines the most biologically plausible of the various learning procedures we discuss, but it is difficult to see how it would be possible to extend them to learning more complex phenomena, like limit cycles or paths through state space. We now turn out attention to networks capable of these more complex feats.

3.3 Computing the Gradient Without Assuming a Fixedpoint

Now we get to the heart of the matter—the computation of $\nabla_w E$, the gradient of the error E with respect to the vector of free parameters w, where the error is not defined at a fixedpoint but rather is a function of the network's detailed temporal behavior. The techniques we will discuss here, like those of section 3.2, are quite general purpose: they can accommodate hidden units as well as various architectural embellishments, such as second-order connections (Hinton and Lang, 1985; Maxwell *et al.*, 1986; Sejnowski, 1986; Watrous and Kuhn, 1992), weight sharing (Lang and Hinton, 1990; LeCun *et al.*, 1989), and in general any of the architectureal modifications made to neural networks to customize them for their problem domain. We will consider two major gradient calculation techniques, and then a few more derived from

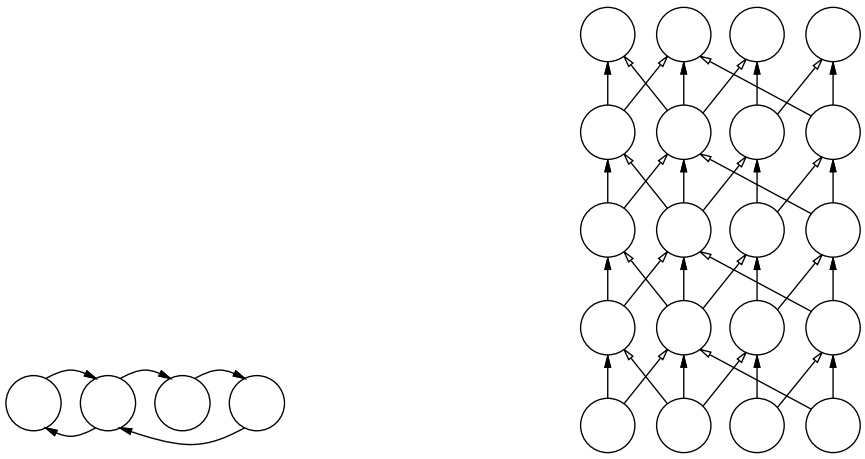


Figure 3.5: A recurrent network is shown on the left, and a representation of that network unfolded in time through four time steps is shown on the right.

them. The first is the obvious extension of backpropagation through time (BPTT) to continuous time (Werbos, 1990; Pearlmutter, 1989a; Howard, 1960).

3.3.1 Backpropagation Through Time

The fixedpoint learning procedures discussed above are unable to learn non-fixedpoint attractors, or to produce desired temporal behavior over a bounded interval, or even to learn to reach their fixedpoints quickly. Here, we turn to a learning procedure suitable for such non-fixedpoint situations. This learning procedure essentially converts a network evolving through time into a network whose activation is flowing through a number of layers, translating time into space, as shown in figure 3.5. Backpropagation then becomes applicable. The technique is therefore called Backpropagation Through Time, or BPTT.

Consider minimizing $E(\mathbf{y})$, some functional of the trajectory taken by $\mathbf{y}$ between t_0 and t_1. For instance, $E = \int_{t_0}^{t_1}(y_0(t) - d(t))^2 dt$ measures the deviation of $y_0(t)$ from the function $d(t)$, and minimizing this E would teach the network to have $y_0(t)$ imitate $d(t)$. Below, we derive a technique for computing $\partial E(\mathbf{y})/\partial w_{ij}$ efficiently, thus allowing us to do gradient descent in the weights so as to minimize E. Backpropagation through time has been used to train discrete time networks to perform a variety of tasks (Rumelhart *et al.*, 1986b; Nowlan, 1988). Here, we will derive the continuous time version of backpropagation through time, as in Pearlmutter (1989a), and use it in two toy domains.

In this derivation, we take the conceptually simple approach of unfolding the continuous time network into a discrete time network with a step of Δt, applying backpropagation to this discrete time network, and taking the limit as Δt approaches zero to get a continuous time learning rule. The derivative in (3.3) can be approximated

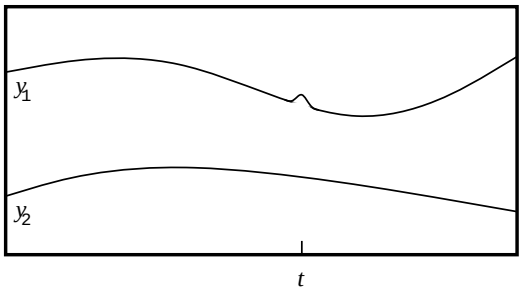


Figure 3.6: The infinitesimal changes to $\mathbf{y}$ considered in $e_1(t)$.

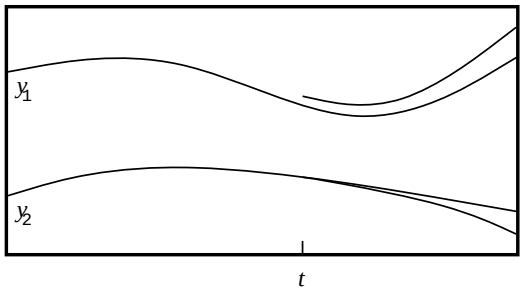


Figure 3.7: The infinitesimal changes to $\mathbf{y}$ considered in $z_1(t)$.

with

$$\frac{dy_i}{dt}(t) \approx \frac{y_i(t+\Delta t) - y_i(t)}{\Delta t}, \tag{3.16}$$

which yields a first order difference approximation to (3.3),

$$\tilde{y}_i(t+\Delta t) = (1-\Delta t)\tilde{y}_i(t) + \Delta t \sigma(\tilde{x}_i(t)) + \Delta t I_i(t). \tag{3.17}$$

Tildes are used throughout for temporally discretized versions of continuous functions.

Let us define e_i to be the first variation of E with respect to the function $y_i(t)$, as in figure 3.6,

$$e_i(t) = \frac{\delta E}{\delta y_i(t)}. \tag{3.18}$$

In the usual case E is of the form

$$E = \int_{t_0}^{t_1} f(\mathbf{y}(t), t)\, dt \tag{3.19}$$

so $e_i(t) = \partial f(\mathbf{y}(t), t)/\partial y_i(t)$. Intuitively, $e_i(t)$ measures how much a small change to y_i at time t affects E if everything else is left unchanged.

As usual in backpropagation, let us define

$$\tilde{z}_i(t) = \frac{\partial^+ E}{\partial \tilde{y}_i(t)} \tag{3.20}$$

where the ∂^+ denotes the ordered derivative of Werbos (1988b), with variables ordered here by time and not unit index. Intuitively, $\tilde{z}_i(t)$ measures how much a small change to $\tilde{y}_i$ at time t affects E when this change is propagated forward through time and influences the remainder of the trajectory, as in figure 3.7. Of course, z_i is the limit of $\tilde{z}_i$ as $\Delta t \to 0$. This z is the δ of the standard backpropagation "generalized δ rule."

We can use the chain rule for ordered derivatives to calculate $\tilde{z}_i(t)$ in terms of the $\tilde{z}_j(t+\Delta t)$. According to the chain rule, we add all the separate influences that varying $\tilde{y}_i(t)$ has on E. It has a direct contribution of $\Delta t e_i(t)$, which comprises the

first term of our equation for $\tilde{z}_i(t)$. Varying $\tilde{y}_i(t)$ by $d\tilde{y}_i(t)$ has an effect on $\tilde{y}_i(t+\Delta t)$ of $d\tilde{y}_i(t)\ (1-\Delta t)$, giving us a second term, namely $(1-\Delta t)\tilde{z}(t+\Delta t)$.

Each weight w_{ij} makes $\tilde{y}_i(t)$ influence $\tilde{y}_j(t+\Delta t)$, $i \neq j$. Let us compute this influence in stages. Varying $\tilde{y}_i(t)$ by $d\tilde{y}_i(t)$ varies $\tilde{x}_j(t)$ by $d\tilde{y}_i(t)\ w_{ij}$, which varies $\sigma(\tilde{x}_j(t))$ by $d\tilde{y}_i(t)\ w_{ij}\ \sigma'(\tilde{x}_j(t))$, which varies $\tilde{y}_j(t+\Delta t)$ by $d\tilde{y}_i(t)\ w_{ij}\ \sigma'(\tilde{x}_j(t))\ \Delta t$. This gives us our third and final term, $\sum_j w_{ij}\ \sigma'(\tilde{x}_j(t))\ \Delta t\ \tilde{z}_j(t+\Delta t)$. Combining these,

$$\tilde{z}_i(t) = \Delta t\, e_i(t) + (1-\Delta t)\tilde{z}_i(t+\Delta t) + \sum_j w_{ij}\sigma'(\tilde{x}_j(t))\Delta t\tilde{z}_j(t+\Delta t). \tag{3.21}$$

If we put this in the form of (3.16) and take the limit as $\Delta t \to 0$ we obtain the differential equation

$$\frac{dz}{dt} = \frac{df(y,w,I)}{dy}z + \frac{\delta E}{\delta y} \tag{3.22}$$

$$\frac{dE}{dw} = \int_{t_0}^{t_1} y\frac{df(y,w,I)}{dw}z dt. \tag{3.23}$$

with boundary condition $z(t_1) = 0$. Thus we have derived appropriate adjoint equations to (3.1). They are similar to the analogous discrete-time backwards error equations,

$$z(t-1) = \frac{df(y,w,I)}{dy}z + \frac{\partial E}{\partial y(t)} \tag{3.24}$$

$$\frac{dE}{dw} = \sum_t y\frac{df(y,w,I)}{dw}z. \tag{3.25}$$

where the error to be minimized is E. If this error is of the usual form of an integral $E = \int E'(y(t),t)dt$ then we get the simple form $\delta E/\delta y = dE'/dy$.

For the particular form of (3.3), this comes to

$$\frac{dz_i}{dt} = z_i - e_i - \sum_j w_{ij}\sigma'(x_j)z_j. \tag{3.26}$$

For boundary conditions note that by (3.18) and (3.20) $\tilde{z}_i(t_1) = \Delta t e_i(t_1)$, so in the limit as $\Delta t \to 0$ we have $z_i(t_1) = 0$.

Consider making an infinitesimal change dw_{ij} to w_{ij} for a period Δt starting at t. This will cause a corresponding infinitesimal change in E of $y_i(t)\sigma'(x_j(t))\Delta t z_j(t) dw_{ij}$. Since we wish to know the effect of making this infinitesimal change to w_{ij} throughout time, we integrate over the entire interval, yielding

$$\frac{\partial E}{\partial w_{ij}} = \int_{t_0}^{t_1} y_i\sigma'(x_j)z_j dt. \tag{3.27}$$

One can also derive (3.26), (3.27) and (3.37) using the calculus of variations and Lagrange multipliers, as in optimal control theory (Bryson, 1962; Dreyfus, 1965). In

fact, the idea of using gradient descent to optimize complex systems was explored by control theorists in the late 1950s. Although their mathematical techniques and algorithms are identical to those reviewed here, and thus handled hidden units, they refrained from exploring systems with so many degrees of freedom, perhaps in fear of local minima.

It is also interesting to note that the recurrent backpropagation learning rule (section 3.2.3) can be derived from these. Let I_i be held constant, assume that the network settles to a fixedpoint, and let E be integrated for one time unit before t_1. As $t_1 \to \infty$, (3.26) and (3.27) reduce to the recurrent backpropagation equations (3.9) and (3.8), so in this sense backpropagation through time is a generalization of recurrent backpropagation.

There are two ways to go about finding such derivations. One is direct, using the calculus of variations (Bryson, 1962). The other is to take the continuous time equations, approximate them by difference equations, precisely calculate the adjoint equations for this discrete time system, and then approximate back to get the continuous time adjoint equations, as in Pearlmutter (1990a). An advantage of the latter approach is that, when simulating on a digital computer, one actually simulates the difference equations. The derivation ensures that the simulated adjoint difference equations are the precise adjoints to the simulated forward difference equations, so the computed derivatives contain no approximation errors.

3.3.2 Real Time Recurrent Learning

An online, exact, and stable, but computationally expensive, procedure for determining the derivatives of functions of the states of a dynamic system with respect to that system's internal parameters has been discovered and applied to recurrent neural networks a number of times (Werbos, 1982; Robinson and Fallside, 1988; Gherrity, 1989; Williams and Zipser, 1989); for reviews see also Pearlmutter (1990b, 1990a), Narendra and Parthasarathy (1991). It is called by various researchers *forward propagation*, *forward perturbation*, or *real time recurrent learning,* RTRL. Like BPTT, the technique was known and applied to other sorts of systems since the 1950s; for a hook into this literature see Jacobson (1968), Bellman (1973) or the closely related Extended Kalman Filter (Gelb *et al.*, 1974). In the general case of (3.1), RTRL is

$$\frac{dE}{dw} = \int_{t_0}^{t_1} \gamma \frac{\delta E}{\delta y} dt \tag{3.28}$$

where $\gamma(t_0) = 0$ and

$$\frac{d\gamma}{dt} = \frac{df(y, w, I)}{dw} + \frac{df(y, w, I)}{dy}\gamma. \tag{3.29}$$

The γ matrix is the sensitivity of the states $y(t)$ to a change of the weights w.

Under the assumption that the weights are changing slowly, RTRL can be made an online algorithm by updating the weights continuously instead of actually integrating

(3.28),

$$\frac{dw}{dt} = -\eta\gamma\frac{\delta E}{\delta y}, \tag{3.30}$$

where η is the learning rate, or, if a momentum term $0 < \alpha < 1$ is also desired,

$$\alpha\frac{d^2w}{dt^2} + (1-\alpha)\frac{dw}{dt} + \eta\gamma\frac{\delta E}{\delta y} = 0. \tag{3.31}$$

For the special case of a fully connected recurrent neural network, as described by (3.3), applying the general RTRL formulas above yields

$$\frac{d\gamma_{ijk}}{dt} = \frac{\partial f_k}{\partial y_k}\gamma_{ijk} + ([j=k]y_j + \sum_l w_{lk}\gamma_{ijl})\frac{\partial f_k}{\partial net_k} \tag{3.32}$$

$$\frac{dw_{ij}}{dt}(t) = -\eta\sum_k \frac{\partial g}{\partial y_k}(t)\gamma_{ijk}(t). \tag{3.33}$$

Regretably, the computation of γ is very expensive, and also non-local. The γ array has nm elements, where n is the number of states and m the number of weights, which is typically on the order of n^2. Updating γ requires $O(n^3m)$ operations in the general case, but the particular structure of a neural network causes some of the matrices to be sparse, which reduces the burden to $O(n^2m)$. This remain too high to make the technique practical for large networks. Nevertheless, because of its ease of implementation, RTRL is used by many researchers working with small networks.

3.3.3 More Efficient Online Techniques

One way to reduce the complexity of the RTRL algorithm is to simply leave out elements of γ that one has reason to believe will remain approximately zero. This approach, in particular ignoring the coupling terms which relate the states of units in one module to weights in another, has been explored by Zipser (1990).

Another is to use BPTT with a history cutoff of k units of time, termed BPTT(k) by Williams and Peng (1990), and make a small weight change each timestep. This obviates the need for epochs, resulting in a purely online technique, and is probably the best technique for most practical problems.

A third is to take blocks of s timesteps using BPTT, but use RTRL to encapsulate the history before the start of each block. This requires $O(s^{-1}n^2m + nm)$ time per step, on average, and $O(nm + sm)$ space. Choosing $s = n$ makes this $O(nm)$ time and $O(nm)$ space, which dominates RTRL. This technique has been discovered independently a number of times (Williams and Zipser, 1995; Schmidhuber, 1992b).

Finally, one can note that, although the forward equations for y are nonlinear, and therefore require numeric integration, the backwards equations for z in BPTT

are linear. Since the dE/dw terms are linear integrations of the z, this means that they are linear functions of the external inputs, namely the e_i terms. As shown by (Sun, Chen, and Lee, 1992), this allows one, during the forward pass, to compute a matrix relating the external error signal to the elements of ∇_w, allowing a fully online algorithm with $O(nm)$ time and space complexity.

3.3.4 Time Constants

A major advantage of temporally continuous networks is that one can add additional parameters that control the temporal bahavior in ways known to relate to natural tasks. An example of this is time constants, which were learned in the context of neural networks in Mozer (1989), Hochreiter (1991), Mozer (1992), Nguyen and Cottrell (1993). If we add a time constant T_i to each unit i, modifying (3.3) to

$$T_i \frac{dy_i}{dt} = -y_i + \sigma(x_i) + I_i, \tag{3.34}$$

and carry these terms through the derivation of section 3.3.1, equations (3.26) and (3.27) become

$$\frac{dz_i}{dt} = \frac{1}{T_i} z_i - e_i - \sum_j \frac{1}{T_j} w_{ij} \sigma'(x_j) z_j. \tag{3.35}$$

and

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{T_j} \int_{t_0}^{t_1} y_i \sigma'(x_j) z_j dt. \tag{3.36}$$

In order to learn these time constants rather than just set them by hand, we need to compute $\partial E(\mathbf{y})/\partial T_i$. If we substitute $\rho_i = T_i^{-1}$ into (3.34), find $\partial E/\partial \rho_i$ with a derivation similar to that of (3.27), and substitute T_i back in we get

$$\frac{\partial E}{\partial T_i} = -\frac{1}{T_i} \int_{t_0}^{t_1} z_i \frac{dy_i}{dt} dt. \tag{3.37}$$

3.3.5 Time Delays

Consider a network in which signals take finite time to travel over each link, so that (3.4) is modified to

$$x_i(t) = \sum_j w_{ji} y_j(t - \tau_{ji}), \tag{3.38}$$

τ_{ji} being the time delay along the connection from unit j to unit i. Let us include the variable time constants of section 3.3.4 as well. Such time delays merely add analogous time delays to (3.35) and (3.36),

$$\frac{dz_i}{dt}(t) = \frac{1}{T_i} z_i(t) - e_i(t) - \sum_j w_{ij} \sigma'(x_j(t + \tau_{ij})) \frac{1}{T_j} z_j(t + \tau_{ij}), \tag{3.39}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{T_j} \int_{t_0}^{t_1} y_i(t) \sigma'(x_j(t + \tau_{ij})) z_j(t + \tau_{ij}) dt, \tag{3.40}$$

while (3.37) remains unchanged. If we set $\tau_{ij} = \Delta t$, these modified equations alleviate concern over time skew when simulating networks of this sort, obviating any need for accurate numerical simulations of the differential equations and allowing simple difference equations to be used without fear of inaccurate error derivatives.

Instead of regarding the time delays as a fixed part of the architecture, we can imagine modifiable time delays. Given modifiable time delays, we would like to be able to learn appropriate values for them, which can be accomplished using gradient descent by

$$\frac{\partial E}{\partial \tau_{ij}} = \int_{t_0}^{t_1} z_j(t) \sigma'(x_j(t)) w_{ij} \frac{dy_i}{dt}(t - \tau_{ij}) dt. \tag{3.41}$$

Watrous *et al.* (1990) applied recurrent networks with immutable time delays in the domain of speech. Feedforward networks with immutable time delays (TDNNs) have been applied with great success in the same domain by Lang *et al.* (1990). A variant of TDNNs which learn the time delays was explored by Bodenhausen (1990). The synapses in their networks, rather than having point taps, have gaussian envelopes whose widths and centers were both learned. Similar synaptic architectures using alpha function envelopes (which obviate the need for a history buffer) whose parameters were learned were proposed and used in systems without hidden units (Tank and Hopfield, 1987b; de Vries and Principe, 1991). A continuous time feedforward network with learned time delays was successfully applied to a difficult time-series prediction task by Day and Davenport (1993).

In the sections on time constants and delays, we have carried out the derivative derivations for BPTT. All the other techniques also remain applicable to this case, with straightforward derivations. The analogous derivations for RTRL are carried out in Pearlmutter (1990a). However, we will not here simulate networks with modifiable time delays.

An interesting class of architectures would have the state of one unit modulate the time delay along some arbitrary link in the network or the time constant of some other unit. Such a "higher order time delay" architecture seems appropriate for tasks in which time warping is an issue, such as speech recognition. The gradients with respect to higher order time delay can be readily calculated by appropriate augmentation of either BPTT or RTRL.

In the presence of time delays, it is reasonable to have more than one connection between a single pair of units, with different time delays along the different connections. Such "time delay neural networks" have proven useful in the domain of speech recognition (Lang and Hinton, 1988; Lang *et al.*, 1990; Waibel *et al.*, 1989; Watrous, 1988). Having more than one connection from one unit to another requires us to modify our notation somewhat; weights and time delays are modified to take a single index, and we introduce some external apparatus to specify the source and destina-

tion of each connection. Thus w_i is the weight on a connection between unit $\mathcal{L}(i)$ and unit $\mathcal{R}(i)$, and τ_i is the time delay along that connection. Using this notation we write (3.38) as

$$x_i(t) = \sum_{j\,|\mathcal{L}(j)=i} w_j y_{\mathcal{R}(j)}(t - \tau_j). \tag{3.42}$$

Our equations would be more general if written in this notation, but readability would suffer, and the translation is quite mechanical.

3.3.6 Extending RTRL to Time Constants and Time Delays

We have seen that BPTT can be easily applied to these new sorts of free parameters we have been adding to our networks, namely time constants and time delays. Other gradient calculation procedures also can be naturally applied to these new sorts of free parameters. In this section, we apply RTRL, first to incorporate time constants and then time delays.

If we begin with (3.34), first we must generalize (3.32) and (3.33) to correctly modify the weights in the presence of time constants. If we substitute k for i in (3.34), take the partial with respect to w_{ij}, and substitute in γ where possible, we have a the differential equation for γ

$$T_k \frac{\gamma_{kij}}{dt} = -\gamma_{kij} + \sigma'(x_k) \sum_l w_{lk} \gamma_{lij}, \tag{3.43}$$

nearly the same as (3.32) except for a time constant.

We can derive analogous equations for the time constants themselves; define

$$q_j^i(t) = \frac{\partial y_i(t)}{\partial T_j}, \tag{3.44}$$

take the partial of (3.3) with respect to T_j, and substitute in q. This yields

$$T_i \frac{dq_j^i}{dt} = -q_j^i - \frac{dy_i}{dt} + \sigma'(x_i) \sum_k w_{ki} q_j^k \tag{3.45}$$

which can be used to update the time constants using the continuous update rule

$$\frac{d\,T_i}{dt} = -\eta \sum_j e_j q_i^j. \tag{3.46}$$

Similarly, let us derive equations for modifying the time delays of section 3.3.5. Define

$$r_{ij}^k(t) = \frac{\partial y_k(t)}{\partial \tau_{ij}} \tag{3.47}$$

and take the partial of (3.3) with respect to τ_{ij}, arriving at a differential equations for r,

$$T_k \frac{dr_{ij}^k}{dt} = -r_{ij}^k + \sigma'(x_k)(\underbrace{w_{ij}\frac{dy_i}{dt}(t-\tau_{ij})}_{\text{included if } j=k} - \sum_l w_{lk} r_{ij}^l (t-\tau_{lk})). \tag{3.48}$$

The time delays can be updated online using the continuous update equation

$$\frac{d\tau_{ij}}{dt} = -\eta \sum_k e_k r_{ij}^k. \tag{3.49}$$

3.4 Some Simulations

In the following simulations, we used networks without time delays, but with mutable time constants. As in the associative network of section 3.2.3, an extra input unit whose value was always held at 1 by a constant external input of 0.5, and which had outgoing connections to all other units, was used to implement biases.

Using first order finite difference approximations, we integrated the system $\mathbf{y}$ forward from t_0 to t_1, set the boundary conditions $z_i(t_1) = 0$, and integrated the system $\mathbf{z}$ backwards from t_1 to t_0 while numerically integrating $z_j\,\sigma'(x_j)\,y_i$ and $z_i\,dy_i/dt$, thus computing $\partial E/\partial w_{ij}$ and $\partial E/\partial T_i$. Since computing dz_i/dt requires $\sigma'(x_i)$, we stored it and replayed it backwards as well. We also stored and replayed y_i as it is used in expressions being numerically integrated.

We used the error functional

$$E = \frac{1}{2}\sum_i \int_{t_0}^{t_1} s_i(\,y_i - d_i)^2 dt \tag{3.50}$$

where $d_i(t)$ is the desired state of unit i at time t and $s_i(t)$ is the importance of unit i achieving that state at that time, in this case 0 except when i was an output unit and after some time (5 units) had elapsed for the network to settle down. Throughout, we used $\sigma(\xi) = (1+e^{-\xi})^{-1}$. Time constants were initialized to 1, weights were initialized to uniformly distributed random values between 1 and -1, and the initial values $y_i(t_0)$ were set to $I_i(t_0) + \sigma(0)$. The simulator used first order difference equations (3.17) and (3.21) with $\Delta t = 0.1$.

3.4.1 Exclusive Or

The network of figure 3.8 was trained to solve the xor problem. Aside from the addition of time constants, the network topology was that used in Pineda (1987). We defined $E = \sum_k \frac{1}{2}\int_2^3 (y_o^{(k)} - d^{(k)})^2 dt$ where k ranges over the four cases, d is the correct output, and y_o is the state of the output unit. The inputs to the net $I_1^{(k)}$ and $I_2^{(k)}$

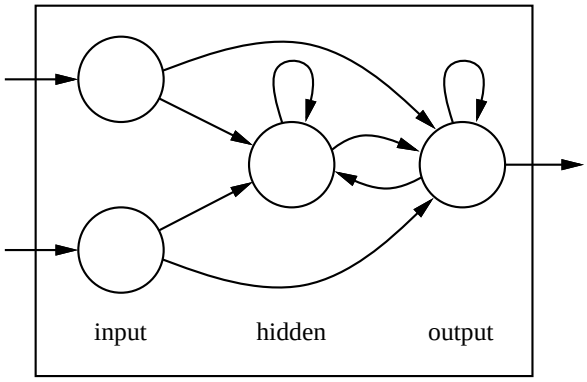


Figure 3.8: The XOR network.

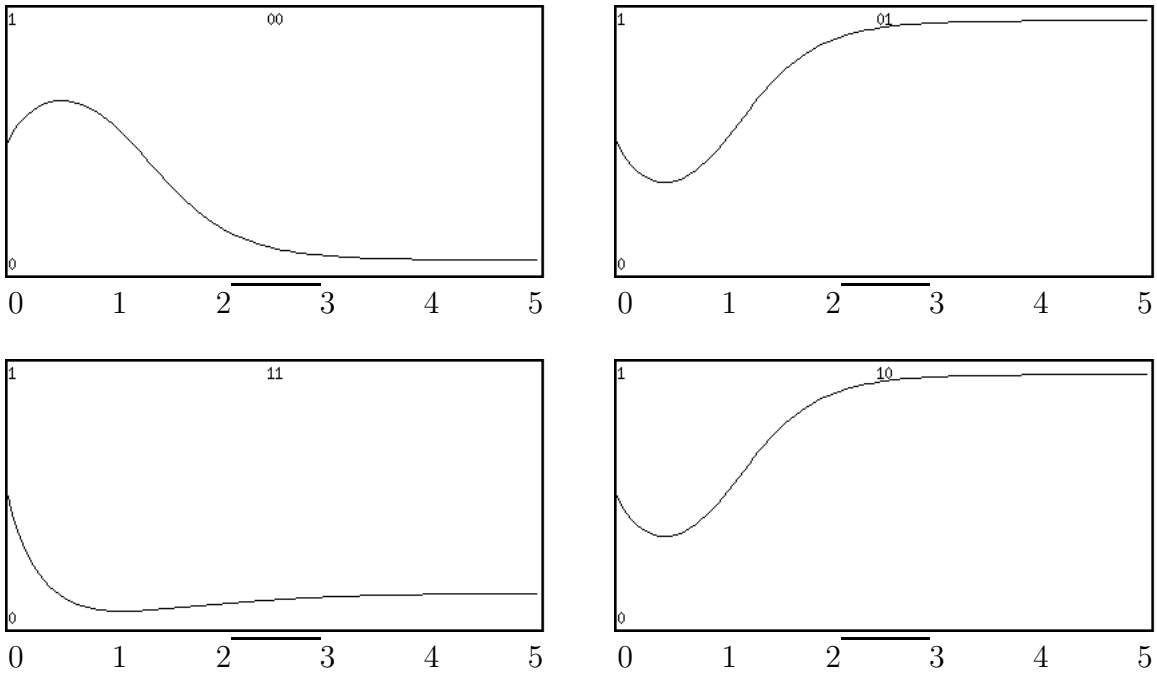


Figure 3.9: The states of the output unit in the four input cases plotted from $t = 0$ to $t = 5$ after 200 epochs of learning. The error was computed only between $t = 2$ and $t = 3$.

range over the four possible boolean combinations in the four different cases. With suitable choice of step size and momentum training time was comparable to standard backpropagation, averaging about one hundred epochs.

For this task it is to the network's benefit for units to attain their final values as quickly as possible, so there was a tendency to lower the time constants towards 0. To avoid small time constants, which degrade the numerical accuracy of the simulation,

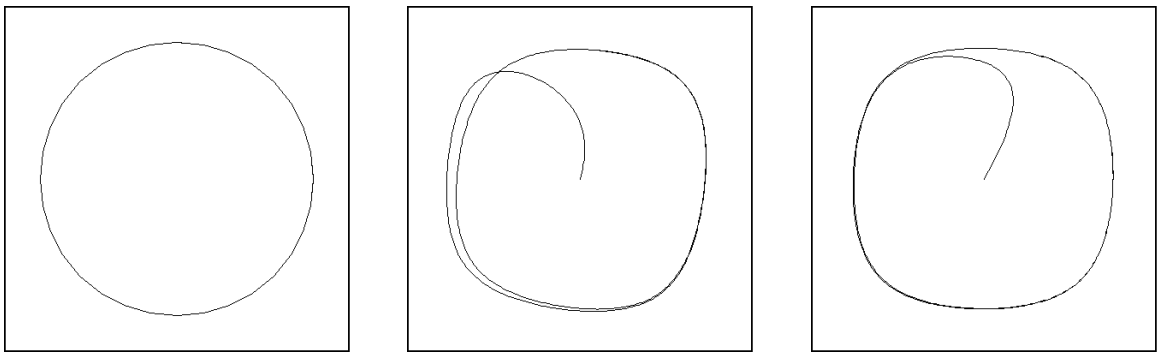


Figure 3.10: Desired states d_1 and d_2 plotted against each other (left); actual states y_1 and y_2 plotted against each other at epoch 1500 (center) and 12000 (right).

we introduced a term to decay the time constants towards 1. This decay factor was not used in the other simulations described below, and was not really necessary in this task if a suitably small Δt was used in the simulation. An easier, and perhaps more justifiable, approach is to simply introduce a minimum time constant; this was done in later simulations.

What is interesting is that that even for this binary task, the network made use of dynamic behavior. After extensive training the network behaved as expected, saturating the output unit to the correct value. Earlier in training, however, we occasionally (about one out of every ten training sessions) observed the output unit at nearly the correct value between $t = 2$ and $t = 3$, but then saw it move in the wrong direction at $t = 3$ and end up stabilizing at a wildly incorrect value. Another dynamic effect, which was present in almost every run, is shown in figure 3.9. Here, the output unit heads in the wrong direction initially and then corrects itself before the error window. A very minor case of diving towards the correct value and then moving away is seen in the lower left hand corner of figure 3.9.

3.4.2 A Circular Trajectory

We trained a network with no input units, four hidden units, and two output units, all fully connected, to follow the circular trajectory of figure 3.10. This more complex trajectory was intended to exhibit a limit cycle somewhat reminiscent of an invertebrate central pattern generator. It was required to be at the leftmost point on the circle at $t = 5$ and to go around the circle twice, with each circuit taking 16 units of time. The environment does not include desired outputs between $t = 0$ and $t = 5$, and during this period the network moves from its initial position at $(0.5, 0.5)$ to the correct location at the leftmost point on the circular trajectory. Although the network was run for ten circuits of its cycle, these overlap so closely that the separate circuits are not visible.

Upon examining the network's internals, we found that it devoted three of its

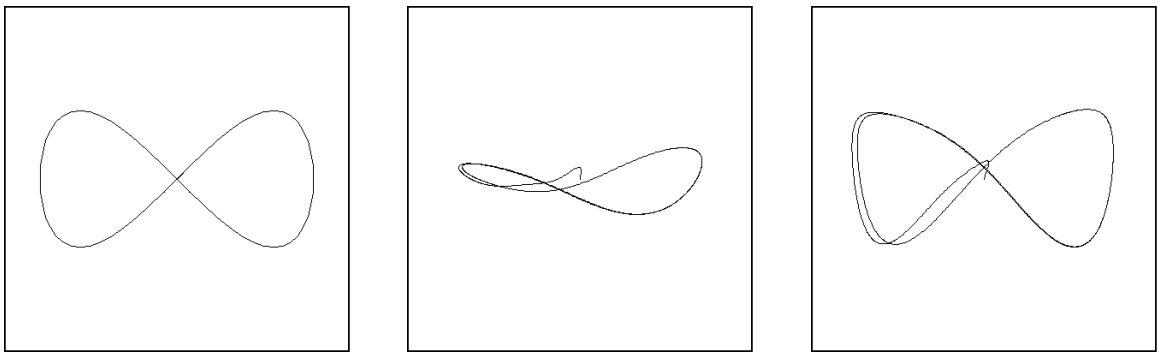


Figure 3.11: Desired states d_1 and d_2 plotted against each other (left); actual states y_1 and y_2 plotted against each other at epoch 3182 (center) and 20000 (right).

hidden units to maintaining and shaping a limit cycle, while the fourth hidden unit decayed away quickly. Before it decayed, it pulled the other units to the appropriate starting point of the limit cycle, and after it decayed it ceased to affect the rest of the network. The network used different units for the limit behavior and the initial behavior, an appropriate modularization.

3.4.3 A Figure Eight

A more interesting problem is one that cannot even in theory be performed without hidden units, such as a figure eight shape. We were unable to train a network with four hidden units to follow the figure eight shape shown in figure 3.11, so we used a network with ten hidden units. Since the trajectory of the output units crosses itself, and the units are governed by first order differential equations, hidden units are necessary for this task regardless of the σ function. Training was more difficult than for the circular trajectory, and shaping the network's behavior by gradually extending the length of time of the simulation proved useful.

From $t = 0$ to $t = 5$ the network moves in a short loop from its initial position at $(0.5, 0.5)$ to where it ought to be at $t = 5$, namely $(0.5, 0.5)$. Following this, it goes through the figure eight shaped cycle every 16 units of time. Although the network was run for ten circuits of its cycle to produce this graph, these overlap so closely that the separate circuits are not visible.

3.4.4 A Rotated Figure Eight

In this simulation a network was trained to generate a figure eight shaped trajectory in two of its units, designated output units. The figure eight was to be rotated about its center by an angle θ which was input to the network through two input units which held the coordinates of a unit vector in the appropriate direction. This was

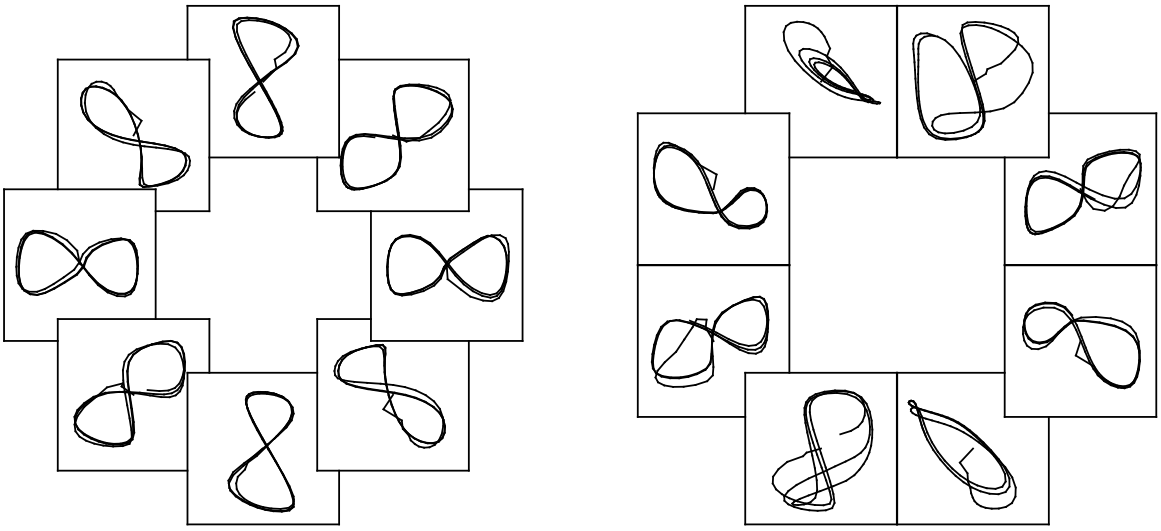


Figure 3.12: The output of the rotated figure eight network at all the trained angles (left) and some untrained angles (right).

intended to model a controlled modulation of a central pattern generator from tonic modulatory input, as in the lobster stomatogastric ganglion (Norris, Coleman, and Nusbaum, 1994). The target vector for the two output units was generated by

$$\text{target} = 0.4 \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} \sin \pi t/16 \\ \cos \pi t/16 \end{pmatrix} + \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \tag{3.51}$$

while the input to the network was simply the angle θ, represented to avoid blemishes as the direction vector

$$\begin{pmatrix} \sin\theta \\ \cos\theta \end{pmatrix}$$

Eight different values of θ, equally spaced about the circle, were used to generate the training data. In experiments with 20 hidden units, the network was unable to learn the task. Increasing the number of hidden units to 30 allowed the network to learn the task, as shown on the left in figure 3.12. But as shown on the right in figure 3.12, generalization is poor when the network is run with the eight input angles furthest from the training angles, *i.e.* 22.5 degrees off.

The task would be simple to solve using second order connections, as they would allow the problem to be decoupled. A few units could be devoted to each of the orthogonal oscillations, and the connections could form a rotation matrix. The poor generalization of the network shows that it is not solving the problem in such a straightforward fashion, and suggests that for tasks of this sort it might be better to use slightly higher order units.

3.4.5 Computational Neuroscience: A Simulated Leech

Lockery, Fang, and Sejnowski (1990), Lockery and Kristan (1990a, 1990b), Lockery, Wittenberg, Kristan, Qian, and Sejnowski (1990), Lockery and Sejnowski (1993) used the continuous time BPTT method discussed above to fit a low level neurophysiological model of the leech local bending reflex to data on sensory and motor neuron activity. They modified the dynamic equations substantially in order to model their system at a low level, using activity levels to represent currents rather than voltages. Their trained model disagreed with human intuition concerning what the synaptic strengths, and in fact signs, would be, but qualitatively matched empirical measurements of interneuron synaptic strengths in the leech *Hirudo medicinalis.*

3.5 Stability and Perturbation Experiments

We can analytically determine the stability of the network by measuring the eigenvalues of Df where f is the function that maps the state of the network at one point in time to its state at a later time. For instance, for a network exhibiting a limit cycle one would typically use the function that maps the network's state at some time in the cycle to its state at the corresponding time in the next cycle. Unfortunately, this gives only a local stability measure, and also does not factor out the effect of hidden units.

In our attempt to judge the stability of the limit cycles exhibited above, rather than calculating Df, where $f(y(t)) = y(t + 16)$, we simply modified the simulator to introduce random perturbations and observed the effects of these perturbations upon the evolution of the system.[5] The two output units in the unrotated figure eight task appear to be phase locked, as their phase relationship remains invariant even in the face of major perturbations. This phase locking is unlike the solution that a human would create by analytically determining weights through decoupling the two output units and using linearized subnets to generate the desired oscillatory behavior, as suggested by Merrick Furst.

The networks to which we introduced these perturbations had been trained to produce simple limit cycles, one in a circular shape, and the other in a figure eight shape. Neither of the networks had any input units; they produced only a single limit cycle.

The limit cycle on the right in figure 3.11 is symmetric, but when perturbations are introduced, as in the right of figure 3.13, symmetry is broken. The portion of the limit cycle moving from the upper left hand corner towards the lower right hand corner

[5]Actually, we wouldn't care about the eigenvalues of Df per se, because we wouldn't care about perturbations in the direction of travel, as these effect only the phase, or perturbations that effect only the hidden units. For this reason, we would want to project these out of the matrix Df before computing the eigenvalues. This effect is achieved automatically in our display in figure 3.13.

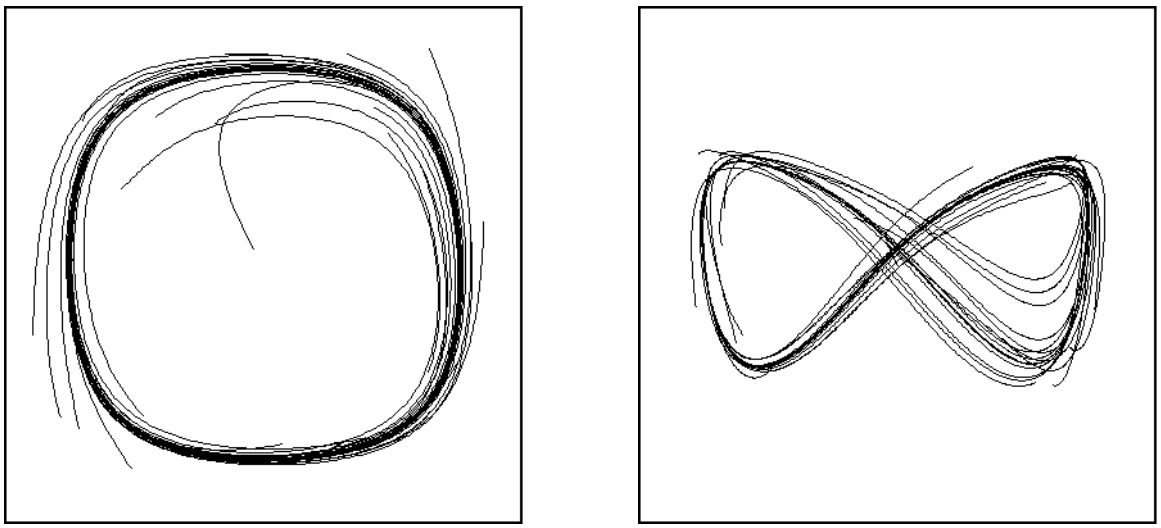


Figure 3.13: The output states y_1 and y_2 plotted against each other for a 1000 time unit run, with all the units in the network perturbed by a random amount about every 40 units of time. The perturbations in the circle network (left) were uniform in ± 0.1, and in the figure eight network (right) in ± 0.05.

has diverging lines, but we do not believe that they indicate high eigenvalues and instability. The lines converge rapidly in the upward stroke on the right hand side of the figure, and analogous unstable behavior is not present in the symmetric downward stroke from the upper right hand corner towards the lower left. Analysis shows that the instability is caused by the initialization circuitry being inappropriately activated. Since the initialization circuitry is adapted for controlling just the initial behavior of the network, when the net must delay at $(0.5, 0.5)$ for a time before beginning the cycle by moving towards the lower left corner, this circuitry is explicitly not symmetric. The diverging lines seem to be caused by this circuitry being activated and exerting a strong influence on the output units while the circuitry itself deactivates.

In fact, Simard, Rayzs, and Victorri (1991) developed a technique for learning the local maximum eigenvalue of the transfer function, optionally projecting out directions whose eigenvalues are not of interest. This technique, which explicitly modulates the behavior we only measured above, has not yet been applied in a control domain.

3.6 Other Non-fixedpoint Techniques

3.6.1 "Elman Nets"

Elman (1990) considers a version of backpropagation through time in discrete time in which the temporal history is cut off. Typically, only one or two timesteps are preserved, at the discretion of the architect. This cutoff makes backpropagation

through time an online algorithm, as the backpropagation to be done to account for the error at each point in time is done immediately. However, it makes the computational expense per time step scale linearly with the number of timesteps of history being maintained. Thus, accuracy of the computed derivative is smoothly traded off against storage and computation.

The real question with Elman networks is whether the contribution to the error from the history that has been cut off is significant. This question can only be answered relative to a particular task. For instance, Elman (1988) finds some problems amenable to the history cutoff, but resorts to full fledged backpropagation through time for other tasks. Cleeremans *et al.* (1989) describe a regular language token prediction task which is difficult for Elman nets when the transition probabilities are equal, but find that breaking this symmetry allows these nets to learn the task.

3.6.2 The Moving Targets Method

LeCun (1985), Grossman, Meir, and Domany (1989), Rohwer (1990) propose a moving targets learning algorithm. Such an algorithm maintains a *target* value for each hidden unit at each point in time. These target values are typically initialized either randomly, or to the units' initial untrained behavior. In learning, two phases alternate. In one phase, the hidden units' targets are improved, such that if the targets are attained better performance would be achieved. In the other phase, the weights are modified such that each unit comes closer to attaining its target values. The error can be regarded as having two terms, one term which penalizes the units being too far from their targets, and another which penalizes the targets for being too far from the values actually attained. This technique has the appeal of decoupling temporally distant actions during the learning of weights, and the disadvantage of requiring the targets to be stored and updated. In the limit, as the internal targets approach equilibrium, the moving targets method becomes equivalent to backpropagation though time.

In continuous time, the moving targets method would entail decoupling the units during learning, and storing a target trajectory for each unit, including the hidden units. The weights would then be modified to make the trajectories consistent with each other, while the trajectories of the hidden units would be similarly modified. Unfortunately, as with teacher forcing (see section 3.6.4), even if the error is driven to very low levels by such a procedure, there would be no guarantee that the resulting network, if allowed to run free, would have dynamics close to that of the forced dynamics.

The primary disadvantage of the technique is that each pattern to be learned must have associated with it the targets for the hidden units, and these targets must be learned just as the weights are. This makes the technique inapplicable for online learning, in which each pattern is seen only once.

3.6.3 Feedforward Networks with State

It is noteworthy that that the same basic mathematical technique of forward propagation can be applied to networks with a restricted architecture, feedforward networks whose units have state (Gori *et al.*, 1989; Kuhn, 1987; Uchiyama *et al.*, 1989). This is the same as requiring the w_{ij} matrix to be triangular, but allowing non-zero diagonal terms. If we let the γ quantities be ordered derivatives, as in standard backpropagation, then this simplified architecture reduces the computational burden substantially. The elimination of almost all temporal interaction makes $\gamma_{ijk} = 0$ unless $i = k$, leaving only $O(n^2)$ auxiliary equations, each of which can be updated with $O(1)$ computation, for a total update burden of $O(n^2)$, which is the same as conventional backpropagation. This favorable computational complexity makes it of practical significance even for large feedforward recurrent networks. But these feedforward networks are outside the scope of this chapter.

3.6.4 Teacher Forcing In Continuous Time

Williams and Zipser (1988) coin the term *teacher forcing,*, which consists of jamming the desired output values into output units as the network runs. The teacher forces the output units to have the correct states, even as the network runs—hence the name. This technique is applied to discrete time clocked networks, as only then does the concept of changing the state of an output unit each time step make sense.

The error is as usual, with the caveat that errors are to be measured before output units are forced, not after. Williams and Zipser (1988) report that their teacher forcing technique radically reduced training time for their recurrent networks, although Pearlmutter (1990a) reports difficulties when teacher forcing was used in networks with a larger number of hidden units.

Williams and Zipser's application of teacher forcing to their networks is dependent on discrete time steps, so applying teacher forcing to temporally continuous networks requires a different approach. The approach we shall take is to add some controls that one imagines being used to control the states of the output units, and use them to keep the output units locked at their desired states. The error function to be minimized will measure the amount of control that it was necessary to exert, with zero error coming only when the no external forces at all need to be exerted.

Let

$$F_i = \frac{1}{T_i}(-y_i + \sigma(x_i) + I_i) \tag{3.52}$$

so that (3.3) is just $dy_i/dt = F_i$, and let us add a new forcing term $f_i(t)$ to (3.3),

$$\frac{dy_i}{dt} = F_i + f_i. \tag{3.53}$$

Using Φ to denote the set of units to be forced, we will let d_i be the trajectory that we will force y_i to follow, for each $i \in \Phi$. So we set

$$f_i = \frac{dd_i}{dt} - F_i \tag{3.54}$$

and $y_i(t_0) = d_i(t_0)$ for $i \in \Phi$ and $f_i = 0$ for $i \notin \Phi$, with the consequence that $y_i = d_i$ for $i \in \Phi$. Now let the error functional be of the form

$$E = \int_{t_0}^{t_1} L(f(t), t)dt, \tag{3.55}$$

where typically $L = \sum_{i \in \Phi} f_i^2$.

We can modify the derivation in section 3.3.1 for this teacher forced system. For $i \in \Phi$ a change to $\tilde{y}_i$ will be canceled immediately, so taking the limit as $\Delta t \to 0$ yields $z_i = 0$. Because of this, it doesn't matter what e_i is for $i \in \Phi$.

We can apply (3.18) to calculate e_i for $i \notin \Phi$. The chain rule is used to calculate how a change in y_i effects E through the f_i, yielding

$$e_i = \sum_{j \in \Phi} \frac{\delta E}{\delta f_j} \frac{\partial f_j}{\partial y_i}$$

or

$$e_i = \sum_{j \in \Phi} \frac{\partial L}{\partial f_j} - \frac{1}{T_j} \sigma'(x_j) w_{ij} \tag{3.56}$$

For $i \notin \Phi$ (3.26) and (3.37) are unchanged, and for $j \notin \Phi$ and any i (3.27) also remains unchanged. The only equations still required are $\partial E / \partial w_{ij}$ for $j \in \Phi$ and $\partial E / \partial T_i$ for $i \in \Phi$. To derive the first, consider the instantaneous effect of a small change to w_{ij}, giving

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{T_j} \int_{t_0}^{t_1} y_i \sigma'(x_j) \frac{\partial L}{\partial f_i} dt. \tag{3.57}$$

Analogously, for $i \in \Phi$

$$\frac{\partial E}{\partial T_i} = -\frac{1}{T_i} \int_{t_0}^{t_1} \frac{\partial L}{\partial f_i} \frac{dy_i}{dt} dt. \tag{3.58}$$

We are left with a system with a number of special cases depending on whether units are in Φ or not. Interestingly, an equivalent system results if we leave (3.26), (3.27), and (3.37) unchanged except for setting $z_i = \partial L / \partial f_i$ for $i \in \Phi$ and setting all the $e_i = 0$. It is an open question whether there is some other way of defining z_i and e_i that results in this simplification.

However, by taking the limit as the step size goes to zero, it is possible to show that the continuous time analogue of teacher forcing is to force the output states to follow desired trajectories, with the error being the difference between the derivative that the network attempts to apply to these units and the derivative of the desired

trajectory. This casts light on teacher forcing in the discrete time case, which can be seen as nearly the same thing.

Regretably it also shows that teacher forcing can result in a network with a systematic bias, or a network which, although when being forced has little error, when running free rapidly drifts far from the desired trajectory, in a qualitative sense, as reported by Williams and Zipser (1988) for some cases where oscillations trained with teacher forcing exhibited radically and systematically lower frequency and amplitude when running free.

3.6.5 Jordan Nets

Jordan (1986) used a backpropagation network with the outputs clocked back to the inputs to generate temporal sequences. Although these networks were used long before teacher forcing, from our perspective Jordan nets are simply a restricted class of teacher forced recurrent networks, in particular, discrete time networks in which the only recurrent connections emanate from output units. By teacher forcing these output units, no real recurrent paths remain, so simple backpropagation through a single time step suffices for training.

The main disadvantage of such an architecture is that state to be retained by the network across time must be manifest in the desired outputs of the network, so new persistent internal representations of temporal structures cannot be created. For instance, it would be impossible to train such networks to perform the figure eight task of section 3.4.3. In the usual control theory way, this difficulty can be partially alleviated by cycling back to the inputs not just the previous timestep's outputs, but also those from a small number of previous timesteps. The tradeoffs between using hidden units to encapsulate temporally hidden structure and using a temporal window of values which must contain the desired information is problem dependent, and depends in essence on how long a hidden variable can remain hidden without being manifested in the observable state variables.

It is easy to construct a continuous time Jordan network, in which the units' values are continuous in time, the output units constantly have corrected values jammed into them from external sources, and the only recurrent connections are from the outputs back to the inputs. Above we explored teacher forcing in the general setting of fully recurrent networks, but when applied to a Jordan network, the result is a system that is no longer truly recurrent, at least as far as learning is concerned. This is because the network maps the current visible state to the next visible state, with no other information retained in the network. For this reason, a continuous time Jordan network is precisely equivalent to training a layered network whose input is the current measured value of the signal we wish the Jordan network to learn, and whose target output is the *first derivative* of this signal to be learned.

3.6.6 Teacher Forcing, RTRL, and the Kalman Filter

Matthews (1990), Williams (1992) have pointed out that RTRL is related to a version of the (Kalman, 1960) filter, in the extension that allows it to apply to nonlinear systems, namely the *extended Kalman filter* (EKF) (Mahra, 1970; Gelb *et al.*, 1974; Anderson and Moore, 1979). The EKF has time and space complexity of the same order as those of RTRL. One advantage of using the EKF (instead of RTRL) for learning the weights of a recurrent neural network, is that the EKF rationalizes teacher forcing: it modifies both the weights and the states on an equal basis. This solves the dilema of teacher forcing: that if the "true output" units are extra added units whose values are directly copied from those of the old output units, teacher forcing fails to maintain synchronization between the network and its teacher. The EKF does not have this problem, in that it would adjust the new extra and the old output units on an equal basis.

Another way of attempting to rationalize teacher forcing is to note that gradient descent itself generates dE/dy in addition to dE/dw terms. One might think this would make it natural to use $\Delta y = -\eta dE/dy$, thus treating the states on an equal basis with the weights. The problem with this, as pointed out by Ron Williams (personal communication) is that it is difficult to determine exactly what this means. Should the derivative be taken just with respect to the current states, or to their histories too? One way alleviate this dilema is to note that, when we change the weights, we wish we had changed them earlier. To this end, it would be natural to change the states to what they would have been had we changed the weights earlier. This gives

$$\Delta y = \frac{dy}{dw}\Delta w. \tag{3.59}$$

The involved matrix, dy/dw, is already available as γ in RTRL.

3.7 Learning with Scale Parameters

The parameters usually modified by neural network learning algorithms are the weights. There are no *a priori* restrictions on these values; they can be positive, negative, or zero, and the behavior of a network is continuous with respect to changes in its weights. These factors, along with the tractable shape of the error surface, make simple gradient descent algorithms, $\Delta w = -\eta dE/dw$, surprisingly effective.

The error term E being used generally contains one term which has to do with how well the network's outputs meet some criterion. Frequently another term is added as an expression of some *a priori* known probability distribtion of the weights. For instance, adding $\sum_i w_i^2$ is equivalent to assuming that the weights are Gaussian distributed. Not adding such a term is equivalent to assuming that the *a priori* distribution on what the weights will turn out to be is flat—not a totally unreasonable

prior (Weigend *et al.*, 1991; Nowlan and Hinton, 1992).

However, we have added some new sorts of parameters, namely time constants and time delays, here represented generically by the vector T. These are *scale parameters,* which differ from positional parameters in a number of ways. The most telling property of a scale parameter is that the dynamics of the system are affected about as much by multiplying a scale parameter by some constant, irrespective of the scale parameter's value. For instance, changing a time constant from 2 seconds to 2.2 seconds can be expected to have about the same qualitative effect as changing it from 200 to 220. Other properties of scale parameters is that they must not become negative, and that as they approach zero, the dynamics of the associated system becomes more and more sensitive to small changes. This means that in practive one must add machinery to enforce the constraint of positiveness, and that gradient descent will become increasingly unstable as a scale parameter approaches zero, due to the system's growing sensitivity to its value. Also, the flat prior is no longer the appropriate zero-knowledge prior.

All these problems can be solved in a single stroke by noting that the correct zero-knowledge hypothesis for scale parameters is not flat in their values, but rather flat in their log values (Skilling, 1989b). In practice, This corresponds to doing gradient descent in $\mathcal{L}_T = \log T$ rather than in T itself; in other words, to not manipulating T directly but rather using $\Delta\mathcal{L}_T = -\eta dE/d\mathcal{L}_T$. Such a policy also solves the practical problems with scale parameters noted above, as it makes the gradient descent process stiffer as T approaches zero, compensating for the system's increased sensitivity in that region, and it naturally enforces $T > 0$ since $T = \exp \mathcal{L}_T > 0$, which enforces this constraint without any additional mechanism. This last property led to the independent invention and use of this technique by Rowat and Selverston (1991).

In addition, weight decay of scale parameters becomes simpler, as decaying $\mathcal{L}_T$ towards zero corresponds to decaying T towards one, which is a reasonable target. Of course, a constant factor can be inserted to make the decay towards some other *a priori* most likely value. Note, however, that the force exerted by the decay term will scale with the log parameter, which is more appropriate, since the additional force exerted should correspond to the change's effect on the dynamics of the system, in order to pass dimensional analysis.

3.8 Summary and Conclusion

3.8.1 Complexity Comparison

Consider a network with n units and m weights which is run for s time steps (variable grid methods (Blom, Sanz-Serna, and Verwer, 1986) would reduce s by dynamically varying Δt) where $s = (t_1 - t_0)/\Delta t$. Additionally, assume that the computation of each $e_i(t)$ is $O(1)$ and that the network is not partitioned.

technique	*time*/Δt	*space*	*online*	*stable*	*local*	*exact*
BPTT, storing y	$O(m)$	$O(sn+m)$	no	yes	yes	yes
RTRL	$O(n^2m)$	$O(nm)$	yes	yes	no	yes
BPTT, only h steps	$O(hm)$	$O(hn+m)$	yes	yes	yes	no
Williams-Peng, h steps	$O(m)$	$O(hn+m)$	yes	yes	yes	no
hybrid BPTT/RTRL	$O(nm)$	$O(nm)$	yes	yes	no	yes
Sun-Chen-Lee	$O(nm)$	$O(n^2+m)$	yes	yes	no	yes
BPTT, recalc. y	$O(m)$	$O(m)$	no	no	yes	yes

Table 3.1: A summary of the complexity of some learning procedures for recurrent networks. In the "storing y" technique we store y as time is run forwards and replay it as we run time backwards computing z. In "y backwards" we do not store y, instead recomputing it as time is run backwards. "Forward propagation" 1 and 2 are the online techniques described in section 3.3.2. The times given are for computing the gradient with respect to one pattern.

Under these conditions, simulating the $\mathbf{y}$ system takes $O(m+n) = O(m)$ time for each time step, as does simulating the $\mathbf{z}$ system. This means that using the technique described in section 3.4, the entire simulation takes $O(m)$ time per time step, the best that could be hoped for. Storing the activations and weights takes $O(n+m) = O(m)$ space, and storing $\mathbf{y}$ during the forward simulation to replay while simulating $\mathbf{z}$ backwards takes $O(sn)$ space, so if we use this technique the entire computation takes $O(sn+m)$ space. If we simulate $\mathbf{y}$ backwards during the backwards simulation of $\mathbf{z}$, the simulation requires $O(n+m)$ space, again the best that could be hoped for. This later technique, however, is susceptible to numeric stability problems.

The online technique of RTRL described in section 3.3.2 requires $O(n^2m)$ time each time step, and $O(nm)$ space. The other techniques discussed in that section require less time and space, and retain all of the advantages of being online (with the possible exception of simplicity of implementation), so it would appear that these new online methods dominate RTRL. These time complexity results are for sequential machines, and are summarized in table 3.1.

Note that in this section we are concerning ourselves with how much computation it takes to obtain the gradient information. This is generally just the inner loop of a more complex algorithm to adjust the weights, which uses the gradient information, such as a stochatic gradient descent algorithm.

3.8.2 Speeding the Optimization

Experience has shown that learning in these networks tends to be "stiff" in the sense that the Hessian of the error with respect to the weights (the matrix of second deriva-

tives) tends to have a wide eigenvalue spread. One technique that has proven useful in this particular situation is that of Jacobs (1988) which was applied by Fang and Sejnowski (1990) to the single figure eight problem described in section 3.4.3 with great success. It was also used in the leech simulations of Lockery *et al.* described in section 3.4.5, again leading to much faster training. For a modern variant of this technique which is suitable to online pattern presentation, see Sutton (1992b, 1992a), Gluck, Glauthier, and Sutton (1992).

For details on how to analyze the speed of gradient descent, and techniques to accelerate its convergence, see chapter 4.

3.8.3 Prospects and Future Work

Control domains are the most natural application for continous time recurrent networks, but signal processing and speech generation (and recognition using generative techniques) are also domains to which this type of network might be naturally applied. Such domains may lead us to complex architectures like those discussed in section 3.3.5. For control domains, it seems important to have ways to force the learning towards solutions that are stable in the control sense of the term.

On the other hand, we can turn the logic of section 3.5 around. Consider a difficult constraint satisfaction task of the sort that neural networks are sometimes applied to, such as the traveling salesman problem (Hopfield and Tank, 1985). Two competing techniques for such problems are simulated annealing (Kirkpatrick, Gelatt, and Vecchi, 1983; Ackley *et al.*, 1985) and mean field theory (Peterson and Anderson, 1987b). By providing a network with a noise source which can be modulated (by second order connections, say) we could see if the learning algorithm constructs a network that makes use of the noise to generate networks that do simulated annealing, or if pure gradient descent techniques are evolved. If a hybrid network evolves, its structure may give us insight into the relative advantages of these two different optimization techniques, and into the best ways to structure annealing schedules.

3.8.4 Conclusions

Recurrent networks are often avoided because of a fear of inordinate learning times and incomprehensible algorithms and mathematics. It should be clear from the above that such fears are unjustified. Certainly there is no reason to use a recurrent network when a feedforward layered architecture suffices; but on the other hand, if recurrence is needed, there are a plethora of learning algorithms available across the spectrum of quiescence vs. dynamics and across the spectrum of accuracy vs. complexity and across the spectrum of space vs. time. These new learning algorithms, and experience with recurrent and temporally continuous networks, has made them much more tractable and practical than they seemed only a few years ago.

Chapter 4

Gradient Descent: Second-Order Momentum and Saturating Error

Capsule: Batch gradient descent, $\Delta w(t) = -\eta dE/dw(t)$, converges to a minimum of quadratic form with a time constant no better than $\frac{1}{4}\lambda_{\max}/\lambda_{\min}$ where $\lambda_{\min}$ and $\lambda_{\max}$ are the minimum and maximum eigenvalues of the Hessian matrix of E with respect to w. It was recently shown that adding a momentum term $\Delta w(t) = -\eta dE/dw(t) + \alpha\Delta w(t-1)$ improves this to $\frac{1}{4}\sqrt{\lambda_{\max}/\lambda_{\min}}$, although only in the batch case. Here we show that second-order momentum, $\Delta w(t) = -\eta dE/dw(t) + \alpha\Delta w(t-1) + \beta\Delta w(t-2)$, can lower this time constant no further.

We then regard gradient descent with momentum as a dynamic system and explore the effect of a nonquadratic error surface, showing that a simple model of nonlinearity, namely saturation of the error, accounts for a variety of effects observed in our simulations, and justifies heuristics that have been popular in the backpropagation community.

Portions of this chapter were published in abbreviated form in Pearlmutter (1992a).

4.1 Introduction

Tesauro, He, and Ahmad (1989) calculate the asymptotic convergence rate of gradient descent optimization of a backpropagation network under the assumption that the output units are asymptotically approaching saturation. Since it is necessary for the weights to grow without bound if units are to approach saturation, their results apply only when the network is not converging to a proper local optimum, but instead the error can always be lowered by enlarging the weights.

Here we address the more conventional case, where the total error does not necessarily approach zero and the gradient descent converges to a proper local optimum at which all weights are bounded. Bounded weights seem desirable in principle, and can be guaranteed by any of a number of well known techniques, such as non-saturated targets, or any of the variants of weight decay.

We also assume that (a) the error has quadratic form at the minimum, an assumption borne out by our simulations, and (b) the problem is hard, in other words, the constraints on the weights are stiff. This latter assumption is used when analyzing convergence rates and deriving optimal values for the learning parameters, we can neglect the easy terms of the error and assume that all effort should be concentrated on the hardest term.

The tight bounds derived for the convergence of gradient descent assume that the learning parameters are set optimally. Finding these optimal values in practice is beyond our scope here, but some techniques for achieving nearly optimal learning rates, although not momentum terms, are surveyed below. The analysis here may itself lead to algorithms that automatically adjust the learning rate and momentum, by providing signatures of optimality and suboptimality. This approach is in contrast to approaches that attempt to accelerate the convergence by going outside of the simple gradient descent method itself (Parker, 1987; Watrous, 1987; Fahlman, 1988).

4.1.1 Asymptotic convergence and choice of learning rate

First, let us derive the well known bound on the convergence rate of simple gradient descent without momentum (Widrow, McCool, Larimore, and Johnson, 1976; Alexander, 1986; Widrow and Stearns, 1985). The same analysis technique will then be applied to more sophisticated variants of gradient descent, giving our main results.

We can analyze the asymptotic convergence rate of the discrete time weight change equation

$$\Delta \mathbf{w} = -\eta \frac{dE}{d\mathbf{w}} \tag{4.1}$$

where $\Delta f(t) \equiv f(t+1) - f(t)$.

Let us assume that the network is near the local optimum $\mathbf{w}^*$ and expand $\mathbf{w}$ about $\mathbf{w}^*$, defining $\mathbf{x} = \mathbf{w} - \mathbf{w}^*$. Let $H = d^2E/d\mathbf{w}^2(\mathbf{w}^*)$ and λ_i, $\mathbf{v}_i$ be the eigenvalues and eigenvectors of H. These eigenvectors are orthogonal because H is symmetric, and each $\lambda_i > 0$ because $\mathbf{w}^*$ is a local minimum.[1] Expressing $\mathbf{x}$ in terms of this eigenvector basis as $\mathbf{c}$ we get

$$c_i(t+1) \approx c_i(t) - \eta c_i(t)\lambda_i = (1 - \eta\lambda_i)c_i(t).$$

[1]Even if some eigenvalues are degenerate, we can still find an orthogonal set of unit basis vectors which are eigenvectors. If the assumption of a local minimum is relaxed slightly and we allow some zero-valued eigenvalues of H, the analysis still holds if we project out the kernel of H.

This is stable and convergent when $|1 - \eta\lambda_i| < 1$, which is strictest for the largest eigenvalue, so the well known

$$0 < \eta < \frac{2}{\lambda_{\max}} \tag{4.2}$$

of Widrow *et al.* (1976) is necessary and sufficient for asymptotic stability and convergence. We can substitute $\eta = 2/\lambda_{\max}$ into the weight change equation to obtain $c_{\min}(t+1) = (1-2s)c_{\min}(t)$, which gives convergence that tightly bounds any achievable in practice, getting a time constant of convergence of $-1/\log(1-2s) = (2s)^{-1} + O(1)$, or

$$E - E^* \succ \exp(-4st) \tag{4.3}$$

where we use $s = \lambda_{\min}/\lambda_{\max}$ for the inverse eigenvalues spread of H and $\succ$ is read "asymptotically converges to zero more slowly than." This is the result of Widrow *et al.* (1976) for LMS, except that here the eigenvalues cannot be determined from the input autocorrelation matrix.

4.1.2 Asymptotic convergence with momentum

Sometimes a momentum term is used, the weight update (4.1) being modified to incorporate a momentum term $\alpha < 1$ (Rumelhart *et al.*, 1986b, equation 16).

$$\Delta\mathbf{w}(t) = -\eta\frac{dE}{d\mathbf{w}}(t) + \alpha\Delta\mathbf{w}(t-1). \tag{4.4}$$

This is sometimes called the "method of acceleration" in the neural network literature, a bit of a misnomer, as the method of acceleration uses line searches (Wilde and Beightler, 1967, page 304). The confusion is understandable, as both rely on the intuition that it might be a good idea to continue to move in the direction the system has been moving, despite local information to the contrary.

The Momentum LMS algorithm, MLMS, has been analyzed by Shynk and Roy (1988), who have shown that the momentum term cannot speed convergence in the online, or stochastic gradient, case. In the batch case, which we consider here, Tuğay and Tanik (1989) have shown that momentum can speed convergence by a square root factor, a result we rederive now.

Consider the matrix

$$M_i = \begin{pmatrix} 0 & 1 \\ -\alpha & 1 + \alpha - \eta\lambda_i \end{pmatrix}$$

since (4.4) can be approximated by $(c_i(t)\ c_i(t+1))^T \approx M_i(c_i(t-1)\ c_i(t))^T$. Iterated application of M_i converges to zero if and only if the absolute values of both eigenvalues of M_i are less than 1, which holds iff $\alpha < 1$ and $0 < \eta < 2(\alpha+1)/\lambda_i$. This is strictest for the largest eigenvalue, so the condition for convergence of gradient

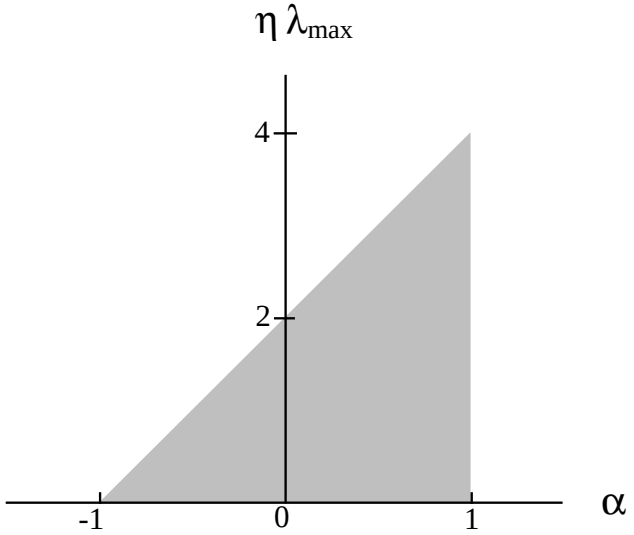


Figure 4.1: Gradient descent with momentum converges if the learning parameters are within the shaded region.

descent with momentum is

$$\alpha < 1 \quad \text{and} \quad 0 < \eta < \frac{2(\alpha + 1)}{\lambda_{\max}} \tag{4.5}$$

which is the region shown in figure 4.1. We see that $\eta < 0$ and $|\alpha| > 1$ are both prohibited, which agrees with physical intuition about the situation, and (4.2) falls out as a special case when $\alpha = 0$.

It was conjectured in Watrous (1987) that momentum serves both to stabilize learning against oscillations and to accelerate learning along "ravines". We have shown above that $\alpha > 0$ can have a stabilizing effect, allowing the system to converge when it would otherwise diverge, and have precisely calculated the magnitude of this effect: momentum can give no more than a factor of 2 increase in the allowable η. We show below that this minor constant factor is overshadowed by a more pronounced reduction of the time constant of convergence given by the direct effect upon convergence of α, rather than by this indirect stabilizing effect, which appears unimportant in practice.

4.1.3 Choice of both α and η

Proceeding as before, we plug in $\eta = 2(1 + \alpha)/\lambda_{\max}$ to get convergence bounding that actually possible for any particular α. Slowest convergence is along the direction of the eigenvector corresponding to the smallest eigenvalue, and convergence

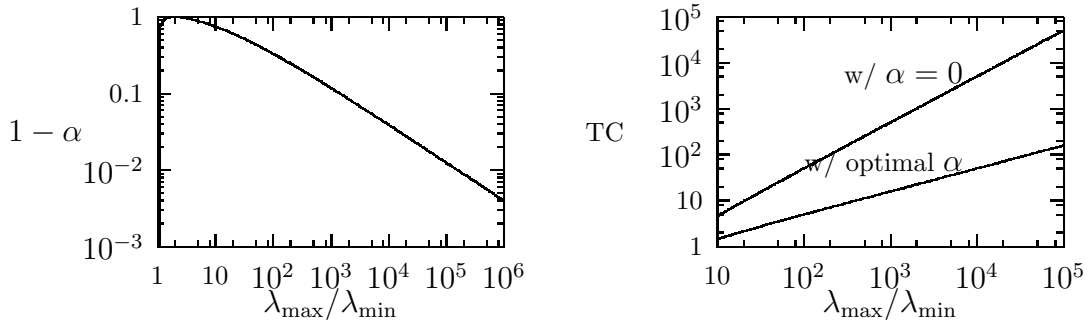


Figure 4.2: Optimal value of α (left) and time constant of convergence with optimal η and $\alpha = 0$ vs. optimal (right).

in this direction is maximized by critical damping, given when the roots of $M_{\min}$'s characteristic polynomial converge. This has only one solution with $\alpha < 1$,

$$\alpha^* = \frac{2 - 4\sqrt{s(1-s)}}{(1-2s)^2} - 1 = 1 - 4\sqrt{s} + O(s) \tag{4.6}$$

which is expressed to take notation advange of the fact that $s < 1$. Substituting in, we get eigenvalues of $M_{\min}$ of

$$\frac{1 - 2\sqrt{s(1-s)}}{1-2s}$$

giving a time constant of convergence of $\mathbf{w}$ of

$$\frac{1}{2\sqrt{s}} + O(1)$$

or

$$E - E^* \succ \exp(-(4\sqrt{s} + O(s))\, t) \tag{4.7}$$

We have seen that momentum can give a substantial improvement in the asymptotic convergence rate, as shown in figure 4.2, and that the advantages of momentum become more pronounced as the eigenvalue spread of $d^2E/d\mathbf{w}^2$ becomes wider, corresponding to a "stiffer" set of constraints on the weights. Such eigenvalue spreads have proven typical of neural networks with many weights or substantial feedback.

On the other hand, the speedup we have derived is less than the backpropagation folk theorem of a speedup of $1/(1-\alpha)$ (Watrous, 1987, section 3.2). The reason this folk theorem fails is that it assumes the gradient to be constant as the system evolves, giving an acceleration of $\sum_i \alpha^i = 1/(1-\alpha)$. But the gradient is not constant in the neighborhood of a minimum. In the stochastic gradient terminology, the folk theorem applies in the initial "fast convergence" phase and the analysis here applies to the later "slow convergence" regime of Tuğay and Tanik (1989).

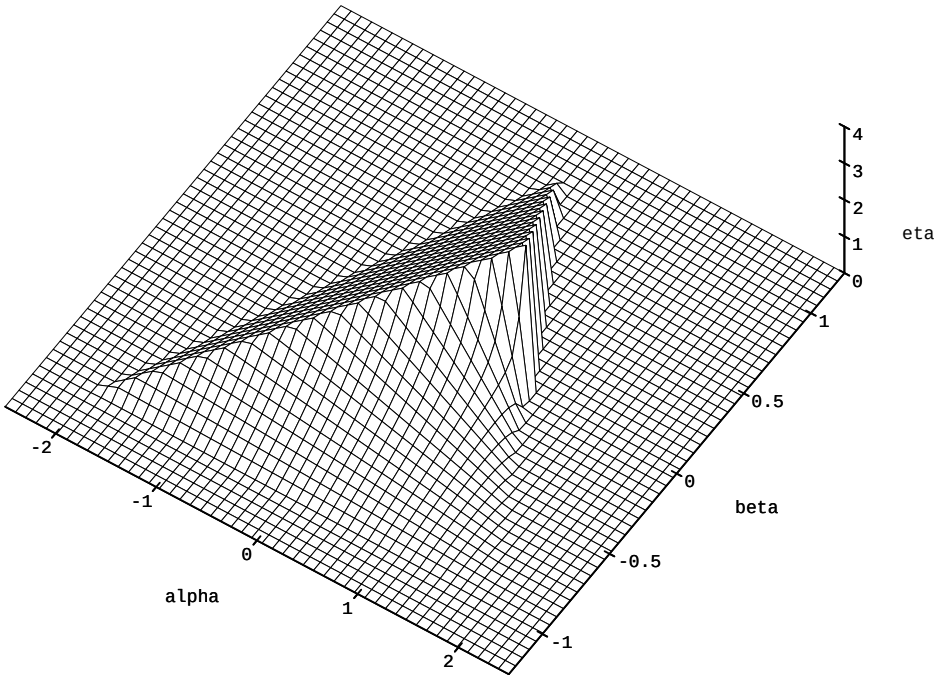


Figure 4.3: Second-order momentum converges if $\eta\lambda_{\max}$ is less than the value plotted as "eta," as a function of α and β. The region of convergence is bounded by four smooth surfaces: three planes and one hyperbola. One of the planes is parallel to the η axis, even though the sampling of the plotting program makes it appear slightly sloped. Another is at $\eta = 0$ and thus hidden. The peak is at 4.

4.2 Second-order momentum

We have shown that the time constant of asymptotic convergence can be changed from $O(\lambda_{\max}/\lambda_{\min})$ to $O(\sqrt{\lambda_{\max}/\lambda_{\min}})$ by going from a first-order system, (4.1), to a second-order system, (4.4). Making a physical analogy, the first-order system corresponds to a circuit with a resistor, and the second-order system adds a capacitor to make an RC oscillator. One might ask whether further gains can be had by going to a third-order system, like adding an inductor, as claimed in Watanabe, Nagata, and Asakawa (1988),

$$\Delta\mathbf{w}(t) = -\eta\frac{dE}{d\mathbf{w}} + \alpha\Delta\mathbf{w}(t-1) + \beta\Delta\mathbf{w}(t-2). \tag{4.8}$$

For convergence, all the eigenvalues of the matrix

$$M_i = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -\beta & -\alpha+\beta & 1-\eta\lambda_i+\alpha \end{pmatrix}$$

in $(c_i(t-1)\ c_i(t)\ c_i(t+1))^T \approx M_i(c_i(t-2)\ c_i(t-1)\ c_i(t))^T$ must have absolute value less than or equal to 1, which occurs precisely when

$$\begin{aligned} -1 &\leq \beta \leq 1 \\ 0 &\leq \eta \leq 4(\beta+1)/\lambda_i \\ \eta\lambda_i/2 - (1-\beta) &\leq \alpha \leq \beta\eta\lambda_i/2 + (1-\beta). \end{aligned}$$

For $\beta \leq 0$ this is most restrictive for $\lambda_{\max}$, but when $\beta > 0$ then $\lambda_{\min}$ also comes into play. Taking the limit as $\lambda_{\min} \to 0$, this gives convergence conditions for gradient descent with second-order momentum of

$$\begin{aligned} -1 \leq\ & \beta \\ \beta - 1 \leq\ & \alpha \leq 1 - \beta \\ \text{when } \alpha \leq 3\beta + 1 :& \\ 0 \leq\ & \eta \leq \frac{2}{\lambda_{\max}}(1 + \alpha - \beta) \\ \text{when } \alpha \geq 3\beta + 1 :& \\ 0 \leq\ & \eta \leq \frac{\beta+1}{\lambda_{\max}\beta}(\alpha + \beta - 1) \end{aligned} \tag{4.9}$$

a region shown in figure 4.3.

Fastest convergence for $\lambda_{\min}$ within this region lies along the ridge, $\alpha = 3\beta + 1$ with $\eta = 2(1 + \alpha - \beta)/\lambda_{\max}$. Unfortunately, although convergence is slightly faster than with first-order momentum, the relative advantage tends to zero as $s \to 0$, giving no asymptotic speedup when $\lambda_{\max} \gg \lambda_{\min}$. For small s, the optimal settings of the parameters are

$$\begin{aligned} \alpha^{**} &= 1 - \frac{9}{4}\sqrt{s} + O(s) \\ \beta^{**} &= -\frac{3}{4}\sqrt{s} + O(s) \\ \eta^{**} &= 4(1 - \sqrt{s}) + O(s). \end{aligned} \tag{4.10}$$

4.3 Simulations

We constructed a standard three layer backpropagation network with 10 input units, 3 sigmoidal hidden units, and 10 sigmoidal output units. 15 associations between random 10 bit binary input and output vectors were constructed, and the weights were initialized to uniformly chosen random values between -1 and $+1$. Training was performed with a square error measure, batch weight updates, targets of 0 and 1, and a weight decay coefficient of 0.01.

To get past the initial transients, the network was run at $\eta = 0.45, \alpha = 0$ for 150 epochs, and at $\eta = 0.3, \alpha = 0.9$ for another 200 epochs. The progress of the error

in this initial regime is shown in figure 4.4. Figure 4.5 shows that momentum can accelerate convergence quite substantially in practice thereafter.

At this point experiments to test the theory in this chapter were carried out: the network was run for 200 epochs for η ranging from 0 to 0.5 and α ranging from 0 to 1.

Figure 4.8 shows that the parameter setting that give the most rapid convergence in practice are the settings predicted by the theory. It is interesting to note that, within the region that does not converge to the minimum, there appear to be two regimes: one that is characterized by apparently chaotic fluctuations of the error, and one which slopes up gradually from the global minimum. In figure 4.6 it can be seen that this sloping region is actually the first period doubling in a transit to chaos, as the single attractor (the minimal error) bifurcates into a binary limit cycle, which gradually rises up until more complex limit cycles, and finally chaotic attractors, are reached.

Figure 4.9 shows that the region of convergence has the qualitative shape predicted in figure 4.1. Calculation of the eigenvalues of $d^2E/d\mathbf{w}^2$ confirms that the location of the dividing line conforms to the theory quantitatively as well.

Since this phenomenon is so atypical of a quadratic minimum in a linear system, which either converges or diverges, and this phenomenon seems important in practice, we decided to investigate a simple system to see if this behavior could be replicated and understood. This is the subject of the next section.

4.4 Nonquadratic error surfaces

The analysis of the sections above may be objected to on the grounds that it assumes the minimum to have quadratic form and then performs an analysis in the neighborhood of that minimum, which is equivalent to analyzing a linear unit. Surely our nonlinear backpropagation networks are richer than that.

A clue that this might be the case was shown in figure 4.8. The region where the system converges to the minimum is of the expected shape, but rather than simply diverging outside of this region, as would a linear system, more complex phenomena are observed, in particular a sloping region.

In this section we will find that this phenomenon, which is at odds with the behavior in the quadratic case, occurs in two cases: when the error surface is saturating, *i.e.* subquadratic; and when the error surface is superquadratic in a particular way which amounts to a ravine whose sides grow progressively steeper.

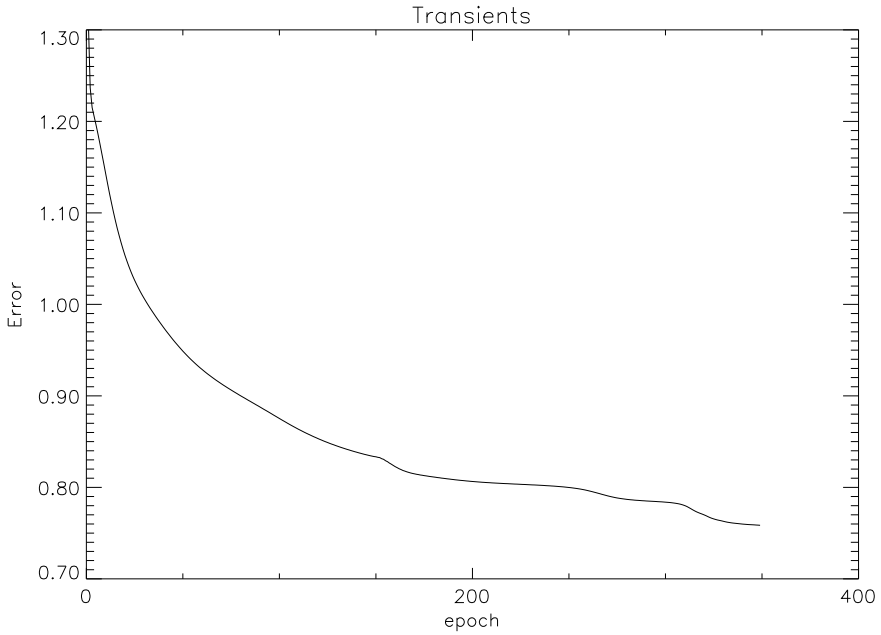


Figure 4.4: The learning curve of the network from epoch 0 to epoch 350, at which point the network was close enough to the minimum to test the theoretical predictions concerning the asymptotic convergence of gradient descent with momentum. To get past the initial transients shown, the network was run at $\eta = 0.45, \alpha = 0$ for 150 epochs, and at $\eta = 0.3, \alpha = 0.9$ for another 200 epochs. Note that the transients appear to be characterized by an asymptotic falling off of the error, followed by a sudden drop which asymptotes in turn. Each of these asymptotic regions is susceptible to the analysis here, if the next dropping off is properly recognized and the learning parameters shifted to the next appropriate regime.

4.4.1 Saturating error surfaces

Acting on the hypothesis that this region is caused by $\lambda_{\max}$ being maximal at the minimum, and gradually decreasing away from it (it must decrease to zero in the limit, since the hidden units saturate and the squared error is thus bounded) we decided to perform a dynamic systems analysis of the convergence of gradient descent on a one dimensional nonquadratic error surface. We chose

$$E = 1 - \frac{1}{1 + w^2} \tag{4.11}$$

which is shown in figure 4.10, as this results in a bounded E.

Letting

$$f(w) = w - \eta E'(w) = \frac{w(1 - 2\eta + 2w^2 + w^4)}{(1 + w^2)^2} \tag{4.12}$$

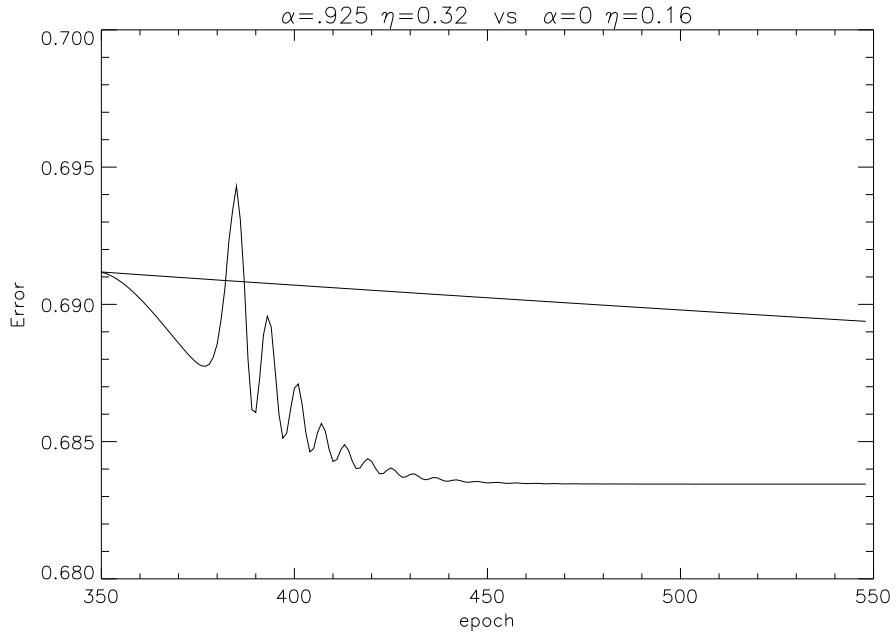


Figure 4.5: Error plotted as a function of time, under two sets of learning parameter settings. In one, $\alpha = 0$ and η is chosen to yield minimal error at the end of the run. In the other, both α and η are chosen to minimize the final error. Both of these were determined empirically rather than theoretically. There exists slightly less aggressive setting of the parameters that suppresses the oscillations but yields a terminal error almost as low.

be our transfer function, a local analysis at the minimum gives $\lambda_{\max} = E''(0) = 2$ which limits convergence to $\eta < 1$. Since the gradient towards the minimum is always less than predicted by a second-order series at the minimum, such η are in fact globally convergent. As η passes 1 the fixpoint bifurcates into the limit cycle

$$w = \pm\sqrt{\sqrt{\eta} - 1}, \tag{4.13}$$

which remains stable until $\eta \to 16/9 = 1.77777\ldots$ from below, at which point the single symmetric binary limit cycle splits into two asymmetric limit cycles, each still of period two. These in turn remain stable until $\eta \to 2.0732261475$ from below, at which point repeated period doubling to chaos occurs. This progression is shown in figure 4.13.

As usual in a bifurcation, w rises sharply as η passes 1. But recall that figure 4.8, with the smooth sloping region, plotted the error E rather than the weights. The analogous graph here is shown in figure 4.12 where we see the same qualitative feature of a smooth gradual rise, which first begins to jitter as the limit cycle becomes asymmetric, and then becomes more and more jagged as the period doubles its way

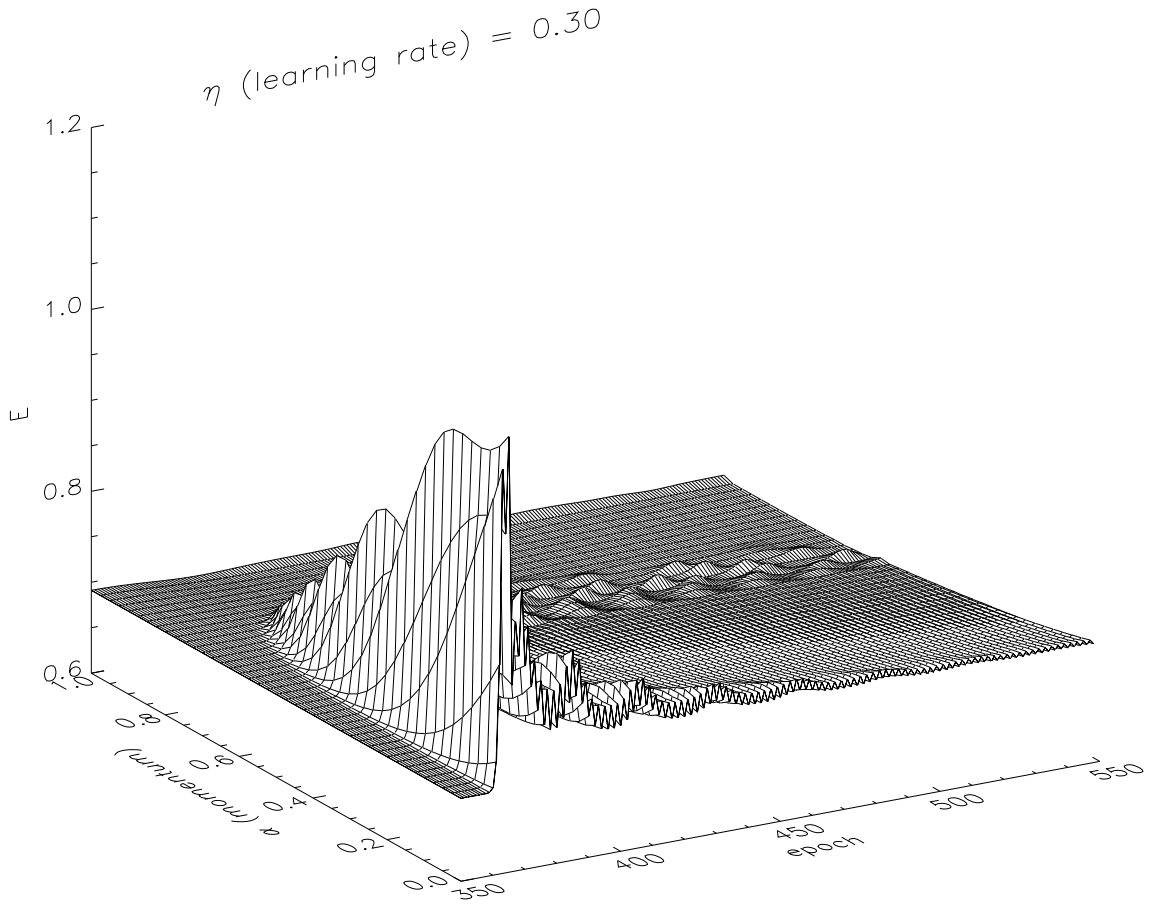


Figure 4.6: Error plotted as a function of time for various values of α as η is held constant at $\eta = 0.30$. Note that for all α below a certain value, about 0.8, the system settles to a limit cycle instead of the minimum.

to chaos. From figure 4.13 it is clear that for higher η the peak error of the attractor will continue to rise gently until it saturates.

Next, we add momentum to the system. This simple one dimensional system duplicates the phenomena we found earlier, as can be seen by comparing figure 4.8 with figure 4.11. We see that momentum delays the bifurcation of the fixed point attractor at the minimum by the amount predicted by (4.5), namely until η approaches $1 + \alpha$. At this point the fixpoint bifurcates into a symmetric limit cycle of period 2 at

$$w = \pm\sqrt{\sqrt{\frac{\eta}{1+\alpha}} - 1}, \tag{4.14}$$

a formula of which (4.13) is a special case. This limit cycle is stable for

$$\eta < \frac{16}{9}(1+\alpha), \tag{4.15}$$

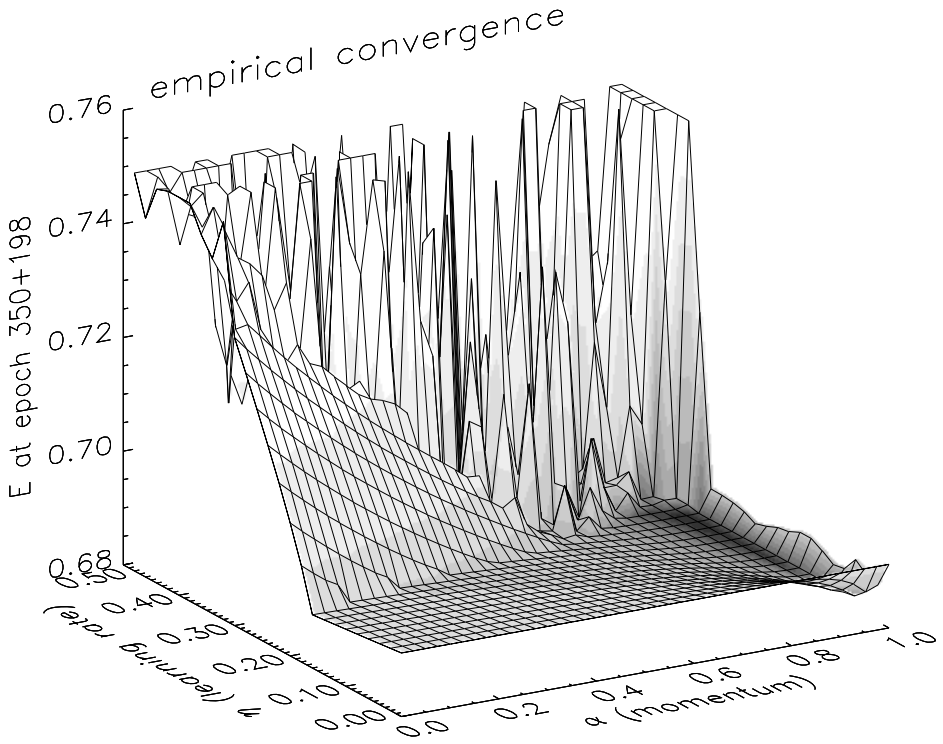


Figure 4.7: The error at epoch 550 as a function of the learning regime, for the network described in section 4.3. Shading is based on the height, but most of the vertical scale is devoted to nonconvergent networks in order to show the mysterious nonconvergent sloping region. The minimum, corresponding to the most darkly shaded point, is on the plateau of convergence at the location predicted by the theory.

but as η reaches this limit, which happens at the same time that w reaches $\pm 1/\sqrt{3}$ (the inflection point of E where $E = 1/4$) the limit cycle becomes unstable. However, for α near 1 the cycle breaks down more quickly in practice, as it becomes haloed by more complex attractors which make it progressively less likely that a sequence of iterations will actually converge to the limit cycle in question. Both boundaries of this strip, $\eta = 1 + \alpha$ and $\eta = \frac{16}{9}(1 + \alpha)$, are visible in figure 4.11, particularly since in the region between them E obeys

$$E = 1 - \sqrt{\frac{1+\alpha}{\eta}} \tag{4.16}$$

The bifurcation and subsequent transition to chaos with momentum is shown for $\alpha = 0.8$ in figure 4.13. This α is high enough that the limit cycle fails to be reached by the iteration procedure long before it actually becomes unstable. Note that this diagram was made with w started near the minimum. If it had been started far from it, the system would usually not reach the attractor at $w = 0$ but instead enter a halo

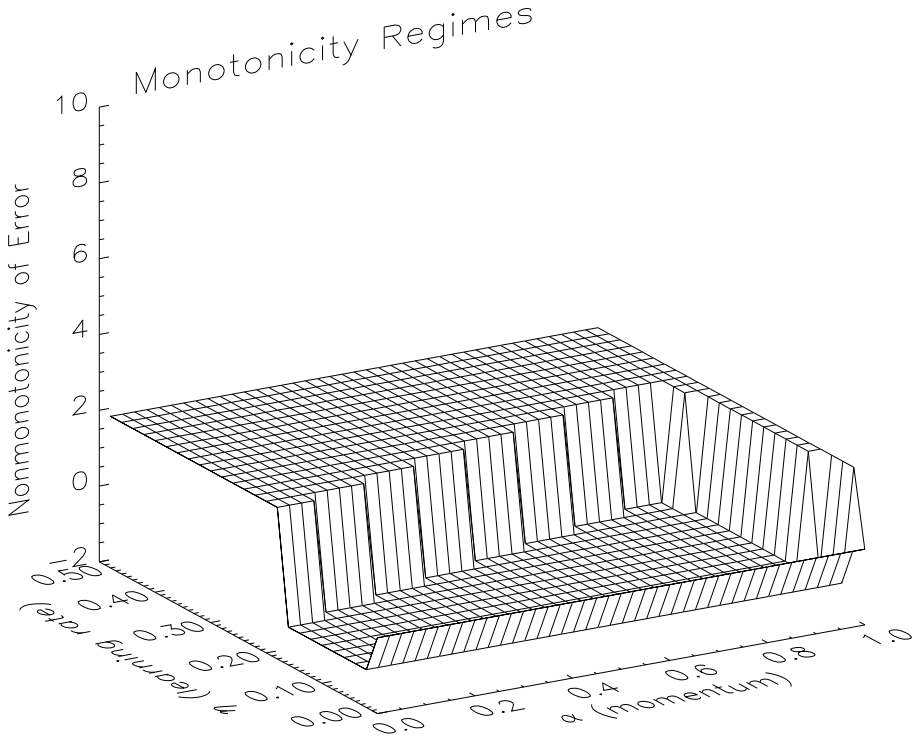


Figure 4.8: The region of parameter space in which the network of section 4.3 is convergent, as measured by a strictly monotonically decreasing error. Learning parameter settings for which the error was strictly decreasing have a low value while those for which it was not have a high one. The lip at $\eta = 0$ has a value of 0, given where the error did not change. The rim at $\alpha = 1$, absent in figure 4.1, corresponds to damped oscillation caused by $\eta > 4\alpha\lambda/(1-\alpha)^2$.

attractor. This accounts for the policy of backpropagation experts, who gradually raise momentum as the optimization proceeds.

4.4.2 Superquadratic error surfaces

The above phenomenon, in which a saturating error surface leads to stable oscillations with a learning rate which is above the threshold required for convergence at the bottom of the minimum, but low enough for convergence farther away from the minimum, matches our intuition, in that the system settles into a state of bouncing around about as close to the minimum as it could be without violating the convergence limitations. We might say that it settles into a state where, on average, the learning rate is about critical, in that it sometime rises above the neighborhood of criticality, and sometimes falls below it, but in general lurks on the threshold.

Here we consider a more counterintuitive situation: a two dimensional ill condi-

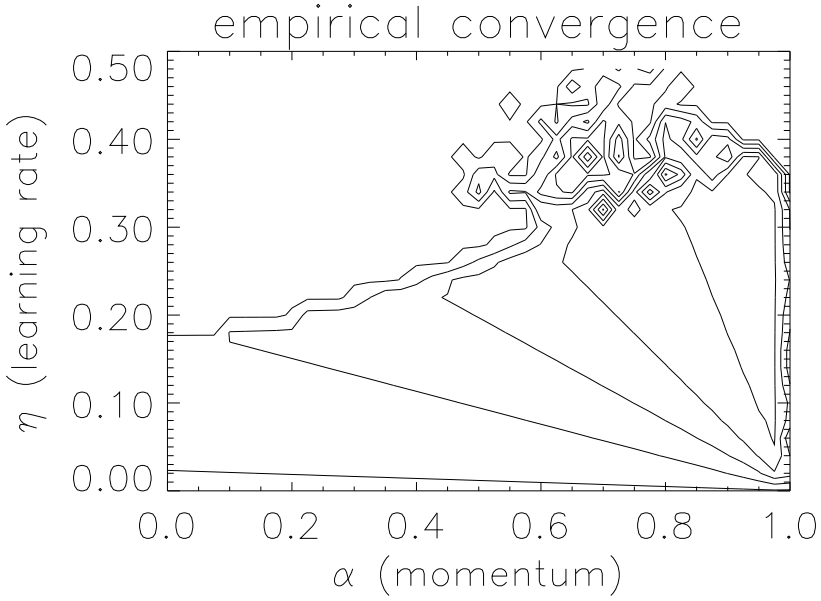


Figure 4.9: Contour plot of the convergent plateau of figure 4.8 shows that the regions of equal error have linear boundaries in the nonoscillatory region in the center, as predicted by the linear theory.

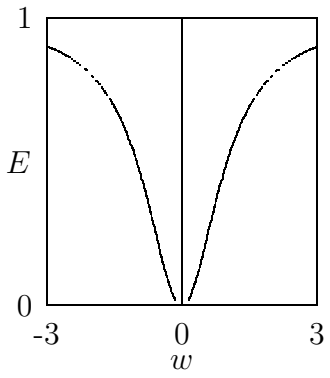


Figure 4.10: A one dimensional tulip-shaped nonlinear error surface $E = 1 - (1 + w^2)^{-1}$.

tioned minimum which is steeper than quadratic, but which, rather than leading to what might be expected, runaway divergence, leads instead to a chaotic orbit near the minimum, qualitatively similar to the subquadratic situation!

In particular, we consider a situation typical of backpropagation networks, in

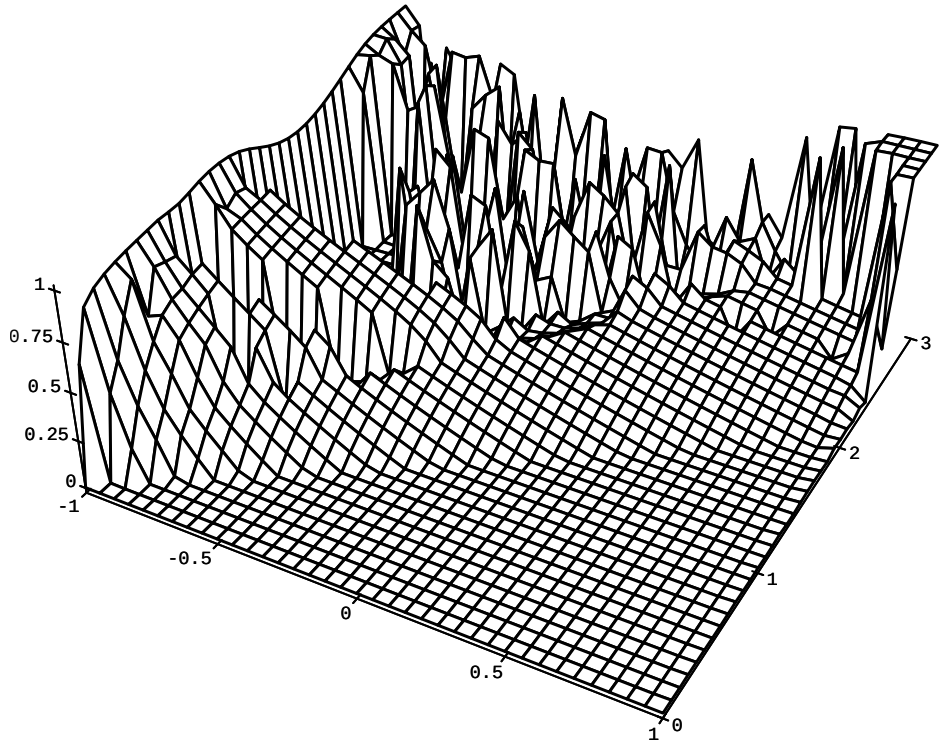


Figure 4.11: E after 50 iterations from a starting point of 0.05, as a function of η and α.

which we are descending a long narrow ravine whose walls grow steeper as we proceed down it. Our simulations are conducted in a toy two dimensional system, namely Rosenbrock's function $E(w_1, w_2) = 100(w_2 - w_1^2)^2 + (1 - w_1)^2$ which has a banana-shaped trough $w_2 = w_1^2$ with a minimum at $w_1 = w_2 = 1$. It qualitatively matches the common situation in backpropagation networks in which the error function is characterized by a ravine whose sides narrow as the optimization proceeds.[2]

At the minimum, the Hessian of Rosenbrock's function has eigenvalues $\lambda = 501 \pm \sqrt{250601} \approx \{0.4, 1000\}$, for an inverse eigenvalues spread of $s \approx 1/2500$. The region of asymptotic stability near the minimum is $\eta < 0.0019968(1 + \alpha)$. For α near 1 this comes to $\eta < 0.004$.

Since in the quadratic situation standard gradient descent with momentum either converges or diverges, in the superquadratic situation here one might expect that when convergence is not attained, the divergence would simply be more violent. However, the simulations of figure 4.17–4.27 show that this is not the case. In fact, as the phase diagram of figure 4.16 shows, there is a considerable region in which the

[2] An intuitive explanation for why this must be so is that, as pointed out by Yann le Cun, in a standard backpropagation network the error is flat to third order at $w = 0$. In other words, both the gradient and the Hessian are zero at $w = 0$.

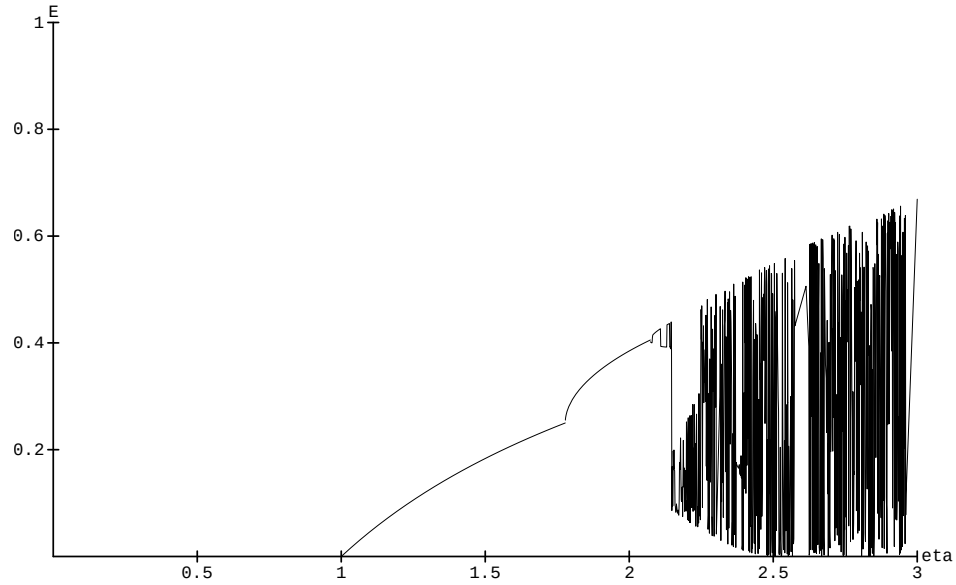


Figure 4.12: E as a function of η with $\alpha = 0$. When convergent, the final value is shown; otherwise E after 100 iterations from a starting point of $w = 1.0$. This a more detailed graph of a slice of figure 4.11 at $\alpha = 0$.

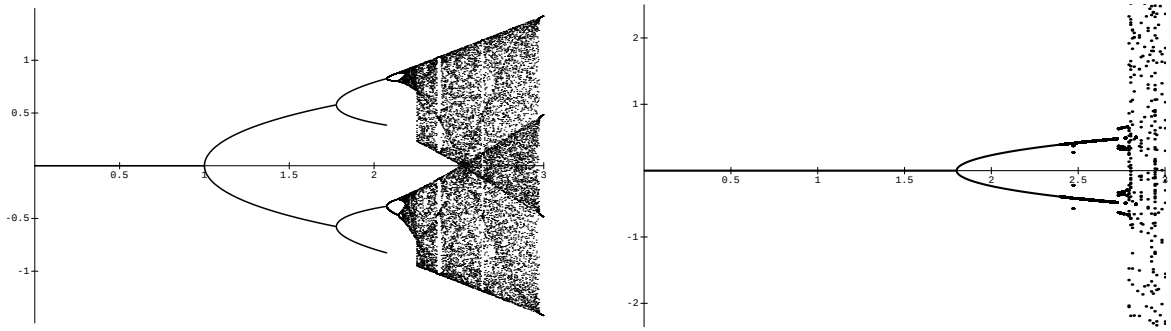


Figure 4.13: The attractor in w as a function of η is shown, with the progression from a single attractor at the minimum of E to a limit cycle of period two, which bifurcates and then doubles to chaos. The x axis is η and the y axis is the corresponding terminal w. On the left $\alpha = 0$ and on the right $\alpha = 0.8$. For the numerical simulations portions of the graphs, iterations 100 through 150 from a starting point of $w = 1$ or $w = 0.05$ are shown.

system repeatedly travels down the ravine until the ravine has gotten steep enough (the increasing steepness is plotted in figure 4.14) to cause divergence, but as the system bounces up the walls of the ravine, the gradient pushes it back towards the mouth, where it can once again converge to the bottom of the ravine, and agains begins its journey along the ravine towards the minimum. This process is illustrated diagrammatically in figure 4.15.

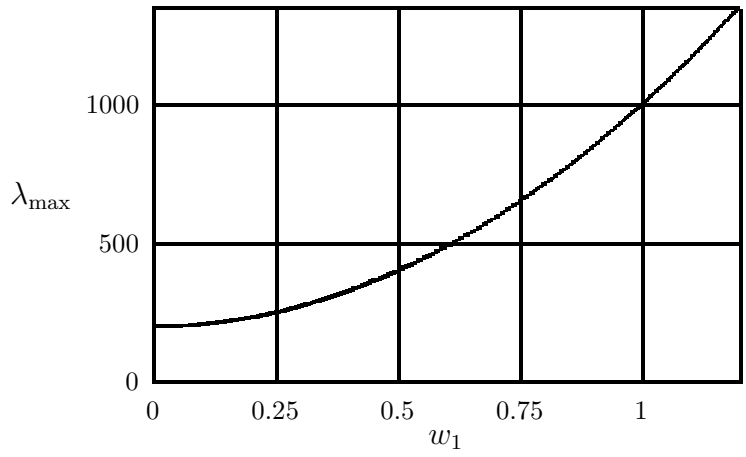


Figure 4.14: The principal eigenvalue $\lambda_{\max}$ of the Hessian of Rosenbrock's function as a function of w_1, measured along the ravine $w_2 = w_1^2$.

Within the ravine $w_2 = w_1^2$, the largest eigenvalue

$$\lambda_{\max} = 101 + 400w_1^2 + \sqrt{9801 + 80800w_1^2 + 160000w_1^4}$$

as plotted in figure 4.14

In figures 4.17–4.27, we show carefully selected attractors representative of the evolution of the system as η is increased, for a range of momenta: $\alpha = 0$, $\alpha = 0.25$, $\alpha = 0.3333333$, $\alpha = 0.5$, $\alpha = 0.8$, $\alpha = 0.9$. Note the evolution from convergence to periodicity, the period doubling to chaos, and finally divergence. The behavior is relatively more complex for higher values of momentum, and the phase space appears to be richer with periodic islands in chaotic regions. Recall that the phase space of the system corresponds to the weights along with their velocities, so when $\alpha \neq 0$ the scatterplots actually show $\mathbf{R}^4 \to \mathbf{R}^2$ projections. For $\alpha = 0.95$ the momentum is high enough that the initial point of $w_1 = w_2 = 0$ is too far from the minimum, and is generally ejected from the neighborhood of the minimum. By starting closer to the minimum at $w_1 = w_2 = 1$, convergence to compact orbits could be attained.

4.5 Alternate formulations of momentum

Assuming that η is held constant, the paired first-order difference equations

$$\begin{aligned} \mathbf{v}(t) &= -\frac{dE}{d\mathbf{w}}(t) + \alpha\mathbf{v}(t-1) \\ \Delta\mathbf{w}(t) &= \eta\mathbf{v}(t) \end{aligned} \tag{4.17}$$

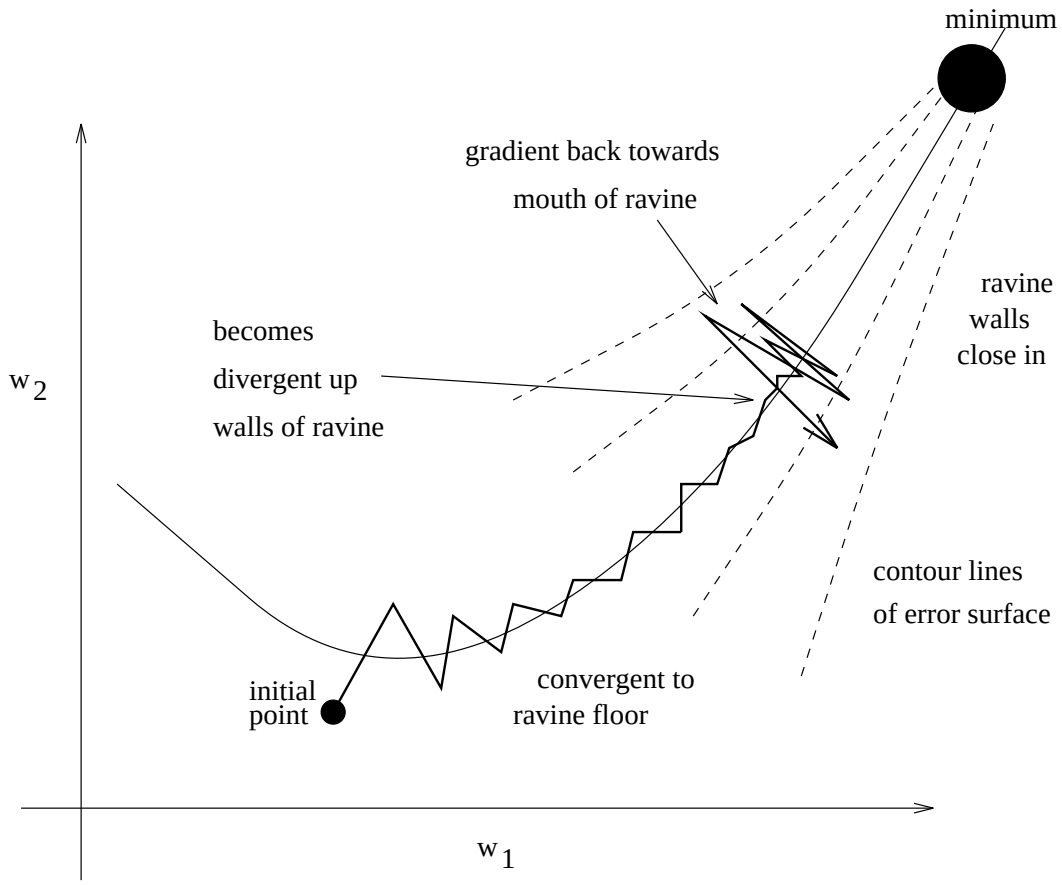


Figure 4.15: An illustration of the process by which a supercritical learning rate and a superquadratic error surface can arrest progress towards the minimum, but lead to oscillations or chaos rather than divergence.

are equivalent to the second-order difference equation formulation above (4.4), as can be shown by some simple algebra. In practice the paired first-order equations here are preferred, as fewer transients are introduced by changes to η.

Analogously, second-order momentum is simulated in practice not by a third-order difference equation (4.8) but by three coupled first-order equations,

$$\begin{aligned} \mathbf{a}(t) &= \alpha_1 \mathbf{a}(t-1) - \frac{dE}{d\mathbf{w}}(t) \\ \mathbf{v}(t) &= \mathbf{a}(t) + \alpha_0 \mathbf{v}(t-1) \\ \Delta\mathbf{w}(t) &= \eta \mathbf{v}(t) \end{aligned}$$

implementing

$$\Delta\mathbf{w}(t) = -\eta \frac{dE}{d\mathbf{w}} + \alpha_0(\Delta\mathbf{w}(t-1) + \alpha_1 \Delta^2 \mathbf{w}(t-2))$$

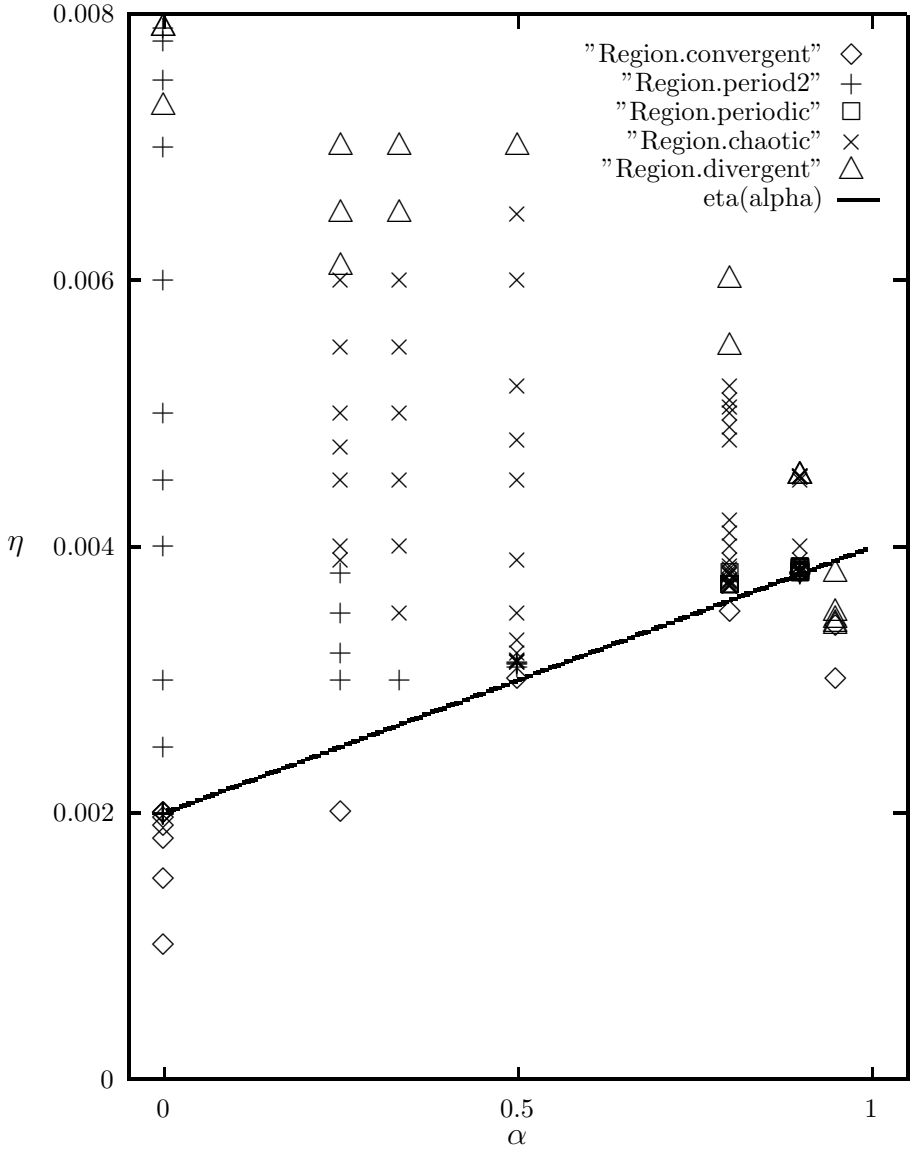


Figure 4.16: Phase diagram. Behavior of the system (optimization of Rosenbrock's function using gradient descent with momentum started at $w_1 = w_2 = 0$) as a function of learning parameters. The system may converge, have period 2, show regular period behavior, exhibit chaos, or diverge, as shown. The convergent region predicted by the principal eigenvalue of the Hessian at the minimum is show as the area below the straight line, which gives the highest theoretically convergent (at the minimum) η as a function of α.

which is related to (4.8) by a simple transformation.

A somewhat common formulation of momentum (Yann le Cun, personal commu-

nication, or Jacobs (Jacobs, 1988, equation 2)) is

$$\Delta \mathbf{w}(t) = -(1-\alpha)\eta' \frac{dE}{d\mathbf{w}} + \alpha \Delta \mathbf{w}(t-1)$$

which differs from (4.4) by the factor of $(1-\alpha)$. In effect, the learning rate is reduced for α near one, $\eta = (1-\alpha)\eta'$. The motivation for this formulation is apparently that without this scaling the "effective learning rate" is $(1-\alpha)^{-1}\eta$ so canceling out this effect is necessary in order to maintain stability over a range of values of α without changing other parameters.

But as we have seen above, multiplying the learning rate by $1-\alpha$ is unnecessary; in fact, $\alpha > 0$ allows the use of a slightly higher η, rather than necessitating a lower one, so if anything $\eta = (1+\alpha)\eta'$ should be used to control for this stabilizing effect. One possible consequence of $\eta = (1-\alpha)\eta'$ would be to poison comparative studies of momentum in which η' is controlled and α varied, as under these circumstances $\alpha > 0$ could not give any asymptotic speedup.

4.6 Transient behavior

In the analyses above, we used

$$\sum_i c_i(0) \exp(-f(\lambda_i)t) \sim \exp(-f(\lambda_{\min})t).$$

where f was positive and nondecreasing in λ_i. Certainly this is true asymptotically. However, sometimes we must turn our attention to the transients. In practice, $c_i(0)$ tends to be larger when λ_i is larger and smaller when λ_i is smaller. This implies that, as learning proceeds, $E - E^*$ will be dominated by the decaying exponentials associated with successively smaller eigenvalues of $d^2E/d\mathbf{w}^2$.

From this perspective, given that the minimal error E^* is unknown, that time is limited, and that the components of the error associated with some of the smallest eigenvalues might start out with such small magnitudes that one can ignore them, a good online strategy is to set the learning parameters as if the currently dominant component is $\lambda_{\min}$. Once this component has decayed enough that it is no longer dominant, the learning parameters can be tuned to the next smaller eigenvalue.

Fortunately, our analyses above apply without change in this case, by simply letting $\lambda_{\min}$ denote the eigenvalue of the currently dominant component of the error. Using the above result, this procedure can be easily shown to be "competitive," in that it will always turn its attention to the real smallest eigenvalue after some amount of time, and that in the interim it will take less than a factor of two extra iterations to reach any given error level.

4.7 Automatic adjustment of η

The maximum convergence rates we calculate assume that the learning rates are set optimally. Were the problem of finding optimal values for the learning rate an intractable problem, this work would be of purely theoretical interest. Fortunately, a number of techniques for automatically adjusting the learning rate are now available. These techniques can do no better than adjusting the parameters optimally, and it appears that in fact they do keep the learning rate near its optimal value.

Vogl, Mangis, Zigler, Zink, and Alkon (1988) report their experiences with a method they developed for accelerating gradient descent. Their technique detects divergence of the error, $E(t+1) > 1.01E(t)$, and reduces the learning rate in response, multiplying η by a number less than one. In the absence of divergence, it raises η slightly, multiplying it by a number slightly greater than 1. They claim that their technique maintains η within a small constant factor of optimal, and verify their claim with some simulations. They do this in the context of momentum, but their use of momentum is very simple: they clear the accumulated velocity when divergence is detected, and otherwise use a constant $\alpha = 0.9$. It is interesting to note their figure plotting learning time T as a function of η, without automatic adjustment of η. The curve exactly follows the power law $T\eta = k$ predicted by (4.3), except that the curve suddenly diverges for η greater than a critical value, presumably $2(1+\alpha)/\lambda_{\max}$.

Tollenaere (1990) reports similar results at tracking the optimal η during gradient descent with his SuperSAB algorithm, which appears identical to the technique of Vogl *et al.* (1988) except that divergence is detected by $dE/d\mathbf{w}(t+1) \cdot dE/d\mathbf{w}(t) < 0$ and no momentum term is included. He also reports the same phenomenon of better performance for larger η, below a critical value at which point divergence appears. A nearly identical algorithm was also developed, again independently, by Silva and Almeida (1990), Du, Hou, and Li (1992), and doubtless others.

An earlier but more sophisticated variant of this idea is the Jacobs (1988) delta-bar-delta rule. It is essentially the same as those above, except that each weight has its own learning rate η_i. The delta-bar-delta rule is a simple heuristic: raise η_i by a small amount κ if $\partial E/\partial w_i(t)\ \partial E/\partial w_i(t-1) > 0$, and multiply it by a constant slightly less than one (typically .8) otherwise. It has proven quite useful, giving massive speedups for very stiff problems with little effort (Fang and Sejnowski, 1990).

The delta-bar-delta approach has been generalized to the online situation and put on more formal grounds by Sutton (1992b).

We are attempting to extend the analysis in this chapter to the case where each weight has its own learning rate, each of which is set optimally, but we do not yet have any concrete results, except that the convergence is limited by the eigenvalues

spread of $\Lambda^{1/2} H \Lambda^{1/2}$, where

$$\Lambda = \begin{pmatrix} \eta_0 & & 0 \\ & \ddots & \\ 0 & & \eta_{n-1} \end{pmatrix}$$

is a diagonal matrix of the individual learning rates.

In the the Pseudo-Newton technique of Becker and LeCun (1989) it is assumed that $d^2E/d\mathbf{w}^2$ is diagonal, and an algorithm is presented to efficiently compute the diagonal terms and (to use terminology which puts their technique in the framework here) they use these to set individual learning rates $\eta_i = \epsilon/(|\partial^2 E/\partial w_i^2| + \mu)$ where ϵ and μ respectively compensate for bounded relative and absolute errors of the estimate of the best value for η_i. It would be interesting to see if, by viewing the algorithm in these terms and filtering the η_i, convergence or stability could be improved.

4.8 Why assume optimal α

The previous section argues that assuming η to be set optimally is reasonable. One might object that this does not justify assuming the momentum term α to be set optimally as well. But although there are no published algorithms for automatically adjusting α, there is good reason to believe that it should be easier to control than η.

One reason is that this work was motivated by the author's observation that, under continuous hand-tuning of the learning parameters, based on only a small amount of feedback $(E, \cos(\angle \mathbf{v}(t), \mathbf{v}(t-1)), |\mathbf{v}|(1-\alpha)/|dE/d\mathbf{w}|)$ substantial speedups were achieved. The author's η was typically kept quite close to the point of divergence, and α was adjusted to a value where slow oscillations of the angle between $dE/d\mathbf{w}$ and $\mathbf{v}$ were barely detectable. From the analysis above, this is nearly the optimal value for α, as it is near critical damping. It was much easier to keep α properly adjusted than to keep η properly adjusted, perhaps because $\lambda_{\max}$ tends to change more rapidly than $\lambda_{\min}$.

Another is that the current techniques, which appear successful at keeping the learning parameters nearly optimal during the gradient descent, are quite crude, and there seems to be room for further developments in this area. Since η can be controlled properly with so little feedback (just the most recent two values of the error, for instance, in Vogl et al.'s method) using more feedback terms, and looking at a window of values stretching back a bit in time, should provide more than enough information to adjust a couple of more slowly changing, and less potentially divergent, learning parameters.

Dabis and Moir (1991) take such an approach to online LMS learning, a related problem. They view LMS as a control system and attempt to design a good controller to minimize the error by using a sliding window and considering frequency response,

span ratios, and phase advance compensation. The result is a robust system with good convergence properties.

To summarize, the problem of controlling η to keep it at a nearly optimal value appears to have been repeatedly solved. Since α is easier to control than η, and has been controlled in related systems, there is good reason to believe that it too can be automatically maintained at a nearly optimal value. There is anecdotal evidence that this is the case, as a number of unpublished "learning rate autopilots" adjust both η and α, and seem to converge to near optimal values of these learning parameters.

4.9 Weight precision

A question of some practical significance, particularly to the design of special purpose hardware, is determining the number of bits of precision required to adequately represent the weights. A simple application of this theory relates the maximum convergent learning rate to the required weight precision.

Assuming that we are using learning parameters that are nearly optimal, we can plug their values into (4.5) to estimate $\lambda_{\max} \approx 2(\alpha + 1)/\eta$.

If we consider each weight w_i to be perturbed by a zero mean Gaussian with variance σ^2, then the perturbation along the direction of the eigenvector corresponding to $\lambda_{\max}$ is also zero mean and also has variance σ^2. A zero-mean perturbation has a zero expected effect on the first-order term of the Taylor expansion of E, so let us consider its effect on the second-order term. This effect is to increase E by $\sum_i \lambda_i r^2/2$ where r is a zero mean σ^2 variance Gaussian. The expected value of this sum is $\sigma^2 \sum_i \lambda_i/2$, which must be between $\sigma^2 n\lambda_{\max}/2$ and $\sigma^2\lambda_{\max}/2$, depending on the distribution of eigenvalues. From empirical studies of the eigenvalue distribution of $d^2E/d\mathbf{w}^2$, both in this work and as reported by Becker and LeCun (1989) (assuming that the unit values are zero mean as given by a tanh activation function instead of a sigmoid to avoid spurious large eigenvalues (LeCun *et al.*, 1991)), a good estimate of the eigenvalues distribution appears to be a uniform distribution. This gives us an expected rise in the error resulting from noise on the weights of standard deviation σ of about

$$\sigma^2 n\lambda_{\max}/4 \approx \sigma^2 n(\alpha + 1)/\eta/2$$

which can be used to give a bound on σ required for a given error tolerance.

4.10 Changes of variables

In the sections above, we computed tight bounds on the possible convergence rates of some variants of gradient descent. The convergence rate of an optimization technique

is one criterion by which we might judge it, but another is its robustness to changes of variables, which we will consider in this section.

If we change the adjustable parameters from $\mathbf{w}$ to $\mathbf{y}$, $\mathbf{y} = f(\mathbf{w})$, f one-to-one, and optimize $E(\mathbf{w})$ by modifying $\mathbf{y}$ rather than $\mathbf{w}$, a very robust optimization algorithm would display the same behavior as if it had been used to adjust $\mathbf{w}$ directly. Of course this is impossible in general, but if we restrict our attention to linear transformations $f(\mathbf{w}) = T\mathbf{w}$ with $|T| \neq 0$, we can gain some insights into the robustness of various gradient based techniques. We could admit translations as well, but since all gradient based techniques are insensitive to translations we omit this detail.

The only current gradient based algorithms that are totally insensitive to all linear transformations are full second-order methods. In most such techniques (eg. Newton's method, or the optimal method of Parker (1987)) the entire matrix $d^2E/d\mathbf{w}^2$ is computed. In others it is gradually approximated, as in the BFGS algorithm of Dennis and Schnabel (1983) found, *e.g.*, in the popular IMSL package's `duminf` routine. Although BFGS is rumored to be the best currently available technique for problems with "small to medium" dimensionality (Watrous, 1987), no fully second-order technique can be practical in a space of high dimension, as all such techniques require storage and computation at least on the order of the number of adjustable parameters squared.

In contrast, consider simple gradient descent, without momentum. This is invariant to the basis used for $\mathbf{w}$, as long as the basis is orthonormal, so gradient descent is therefore insensitive to rigid transformations T. Assuming that the optimal learning rate η is used, then the optimization will also be insensitive to scalings $T = kI$. However, simple gradient descent is not invariant to any T which is not a simple scaling combined with a rigid transformation. An instance of this is a diagonal T where the elements along the diagonal are not all equal.

Gradient descent remains insensitive to rigid transformations and to scaling when momentum is added.

In fact, if we make the odd assumptions that the learning parameters are held constant, not varied to suit current circumstances, and that all the eigenvectors of $d^2E/d\mathbf{w}^2$ are less than $(1-\alpha)^2/(4\eta)$, and the same for the eigenvalues of $d^2E/d\mathbf{y}^2$, then the time constant of asymptotic convergence of gradient descent with momentum will be $2/(1-\alpha)$ regardless of T (except that T must have the property listed above). However, these assumptions are strong and unnatural, and preclude anything near an optimal rate of convergence.

First-order techniques that use a separate learning rate for each adjustable parameter, such as Jacob's delta-bar-delta rule described above, give up insensitivity to arbitrary rigid transformations, but have the potential to gain insensitivity to arbitrary non-rigid diagonal transformations. Jacob's delta-bar-delta rule does not quite achieve this, because the "small constant" κ determines a preferred scale. But were this changed to, say, a slow exponential increase, as in Vogl et al., the delta-bar-delta

rule would be insensitive to diagonal transformations.

Similarly, the Pseudo-Newton technique of Becker and le Cun, also described above, is nearly insensitive to diagonal transformations, except that μ determines a preferred scale.

4.11 Conclusions and speculations

In analyzing the asymptotic convergence of gradient descent with momentum under the assumption that the learning rate and momentum are set optimally, we have found that momentum can provide significant acceleration. However, the convergence rate is still limited by the eigenvalues spread of $d^2E/d\mathbf{w}^2$, getting arbitrarily worse as the spread widens. From a neural networks perspective, the challenge here is to produce a local gradient based optimization algorithm that overcomes this barrier, or to show that changes based purely on local information are intrinsically incapable of doing so.

Finally, note that our analysis uses no special properties of backpropagation, and applies to the general case of asymptotic convergence of gradient descent optimization to a local minimum of quadratic form, except insofar as the particular nonlinearities analyzed are typical of feedforward neural networks.

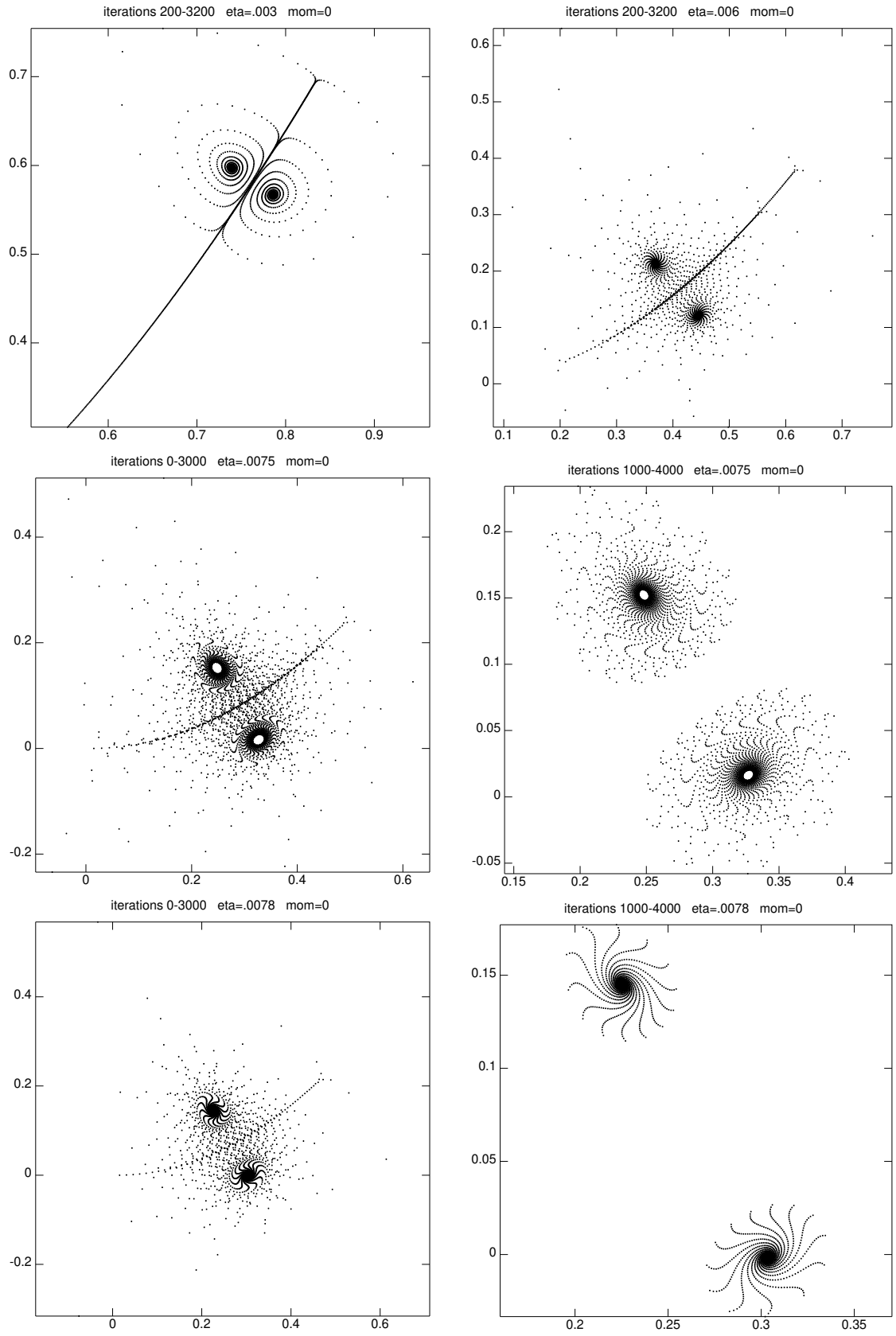


Figure 4.17: Attractors for gradient descent with momentum on Rosenbrock's function started at $w_1 = w_2 = 0$. Learning parameters: $\alpha = 0$, $\eta = .003, \ldots, .0078x$.

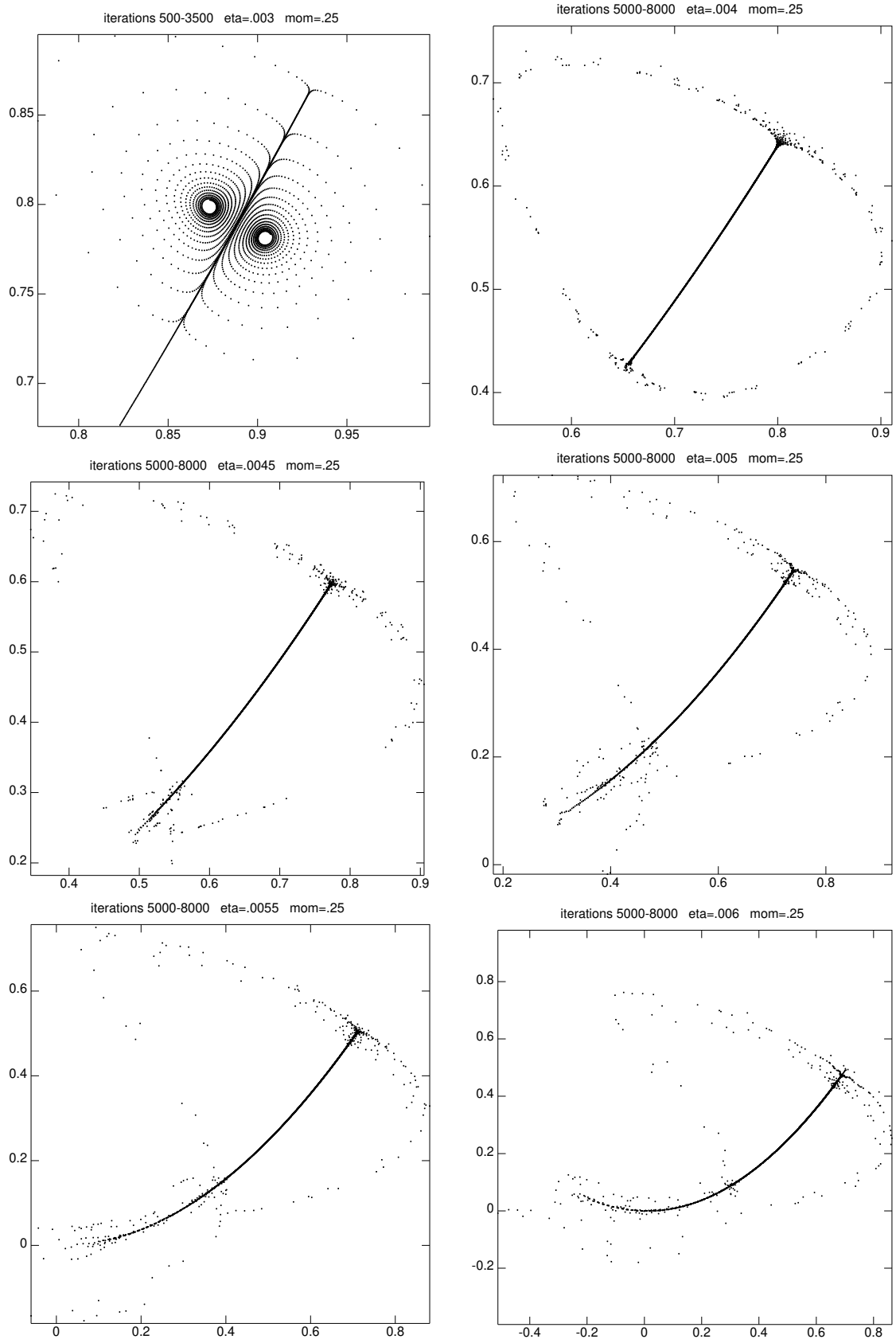


Figure 4.18: Attractors for gradient descent with momentum on Rosenbrock's function started at $w_1 = w_2 = 0$. Learning parameters: $\alpha = .25$, $\eta = .003, \ldots, .006$.

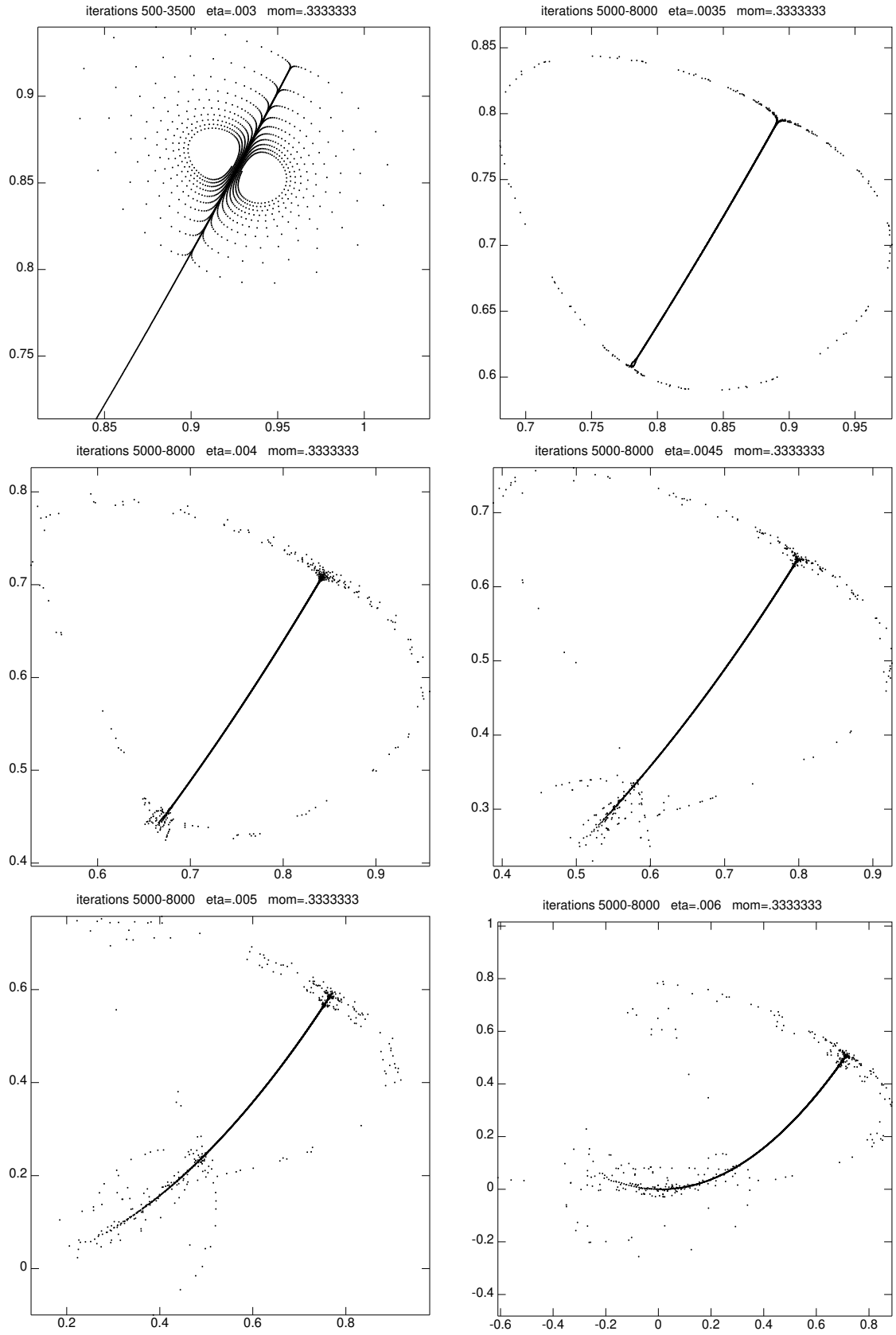


Figure 4.19: Attractors for gradient descent with momentum on Rosenbrock's function started at $w_1 = w_2 = 0$. Learning parameters: $\alpha = .3333333$, $\eta = .003, \ldots, .006$.

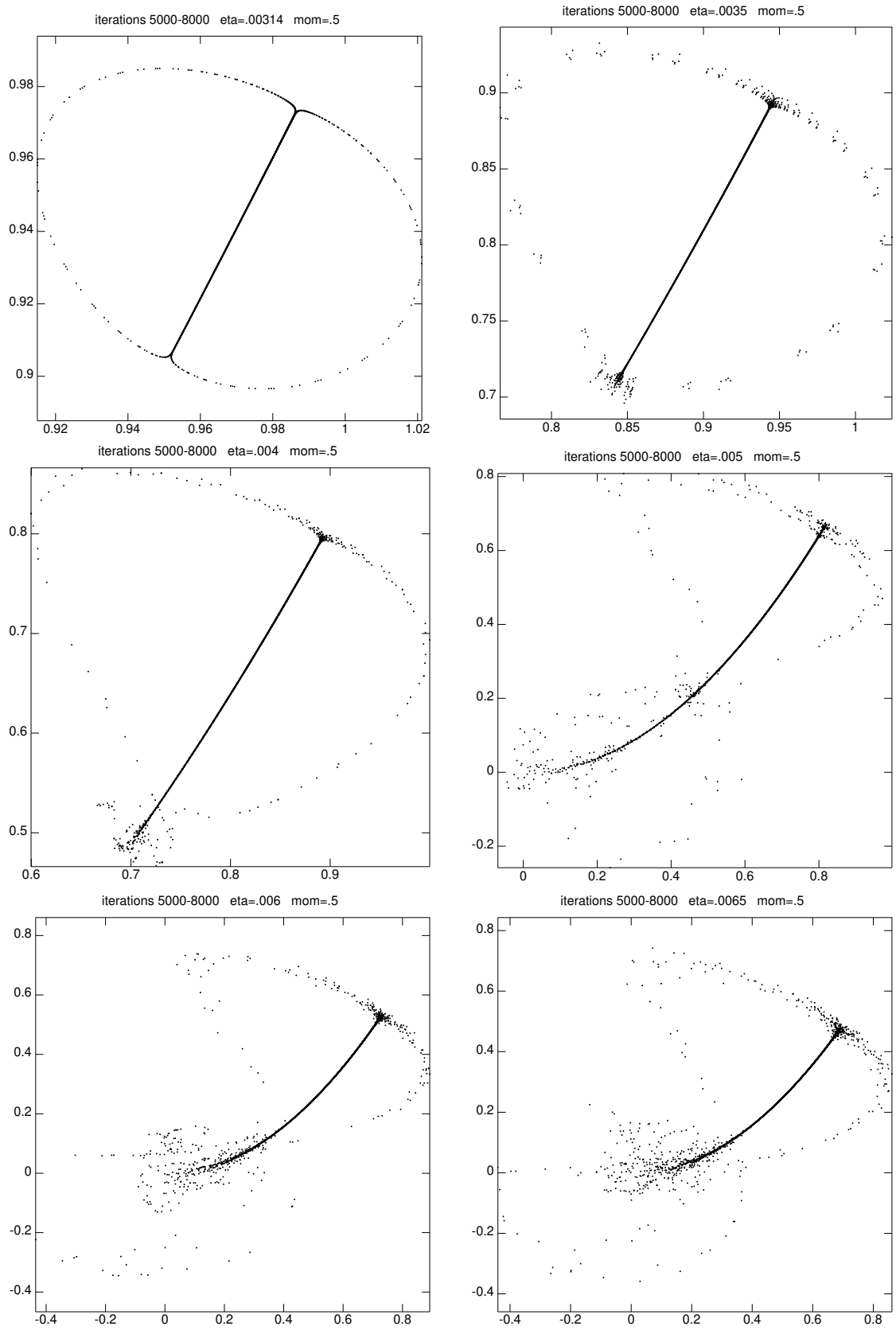


Figure 4.20: Attractors for gradient descent with momentum on Rosenbrock's function started at $w_1 = w_2 = 0$. Learning parameters: $\alpha = .5$, $\eta = .00314, \ldots, .0065$.

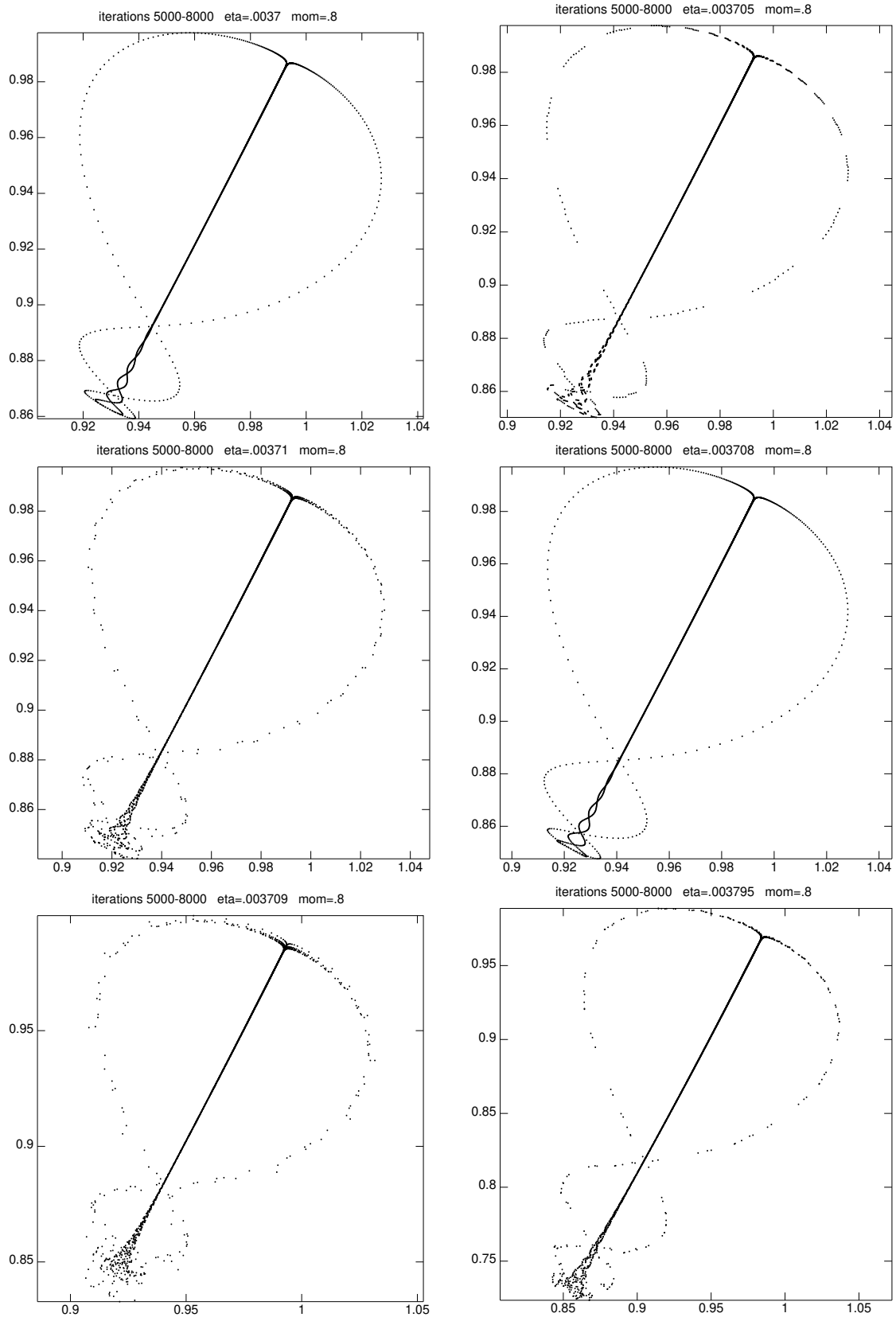


Figure 4.21: Attractors for gradient descent with momentum on Rosenbrock's function started at $w_1 = w_2 = 0$. Learning parameters: $\alpha = .8$, $\eta = .0037, \ldots, .003795$.

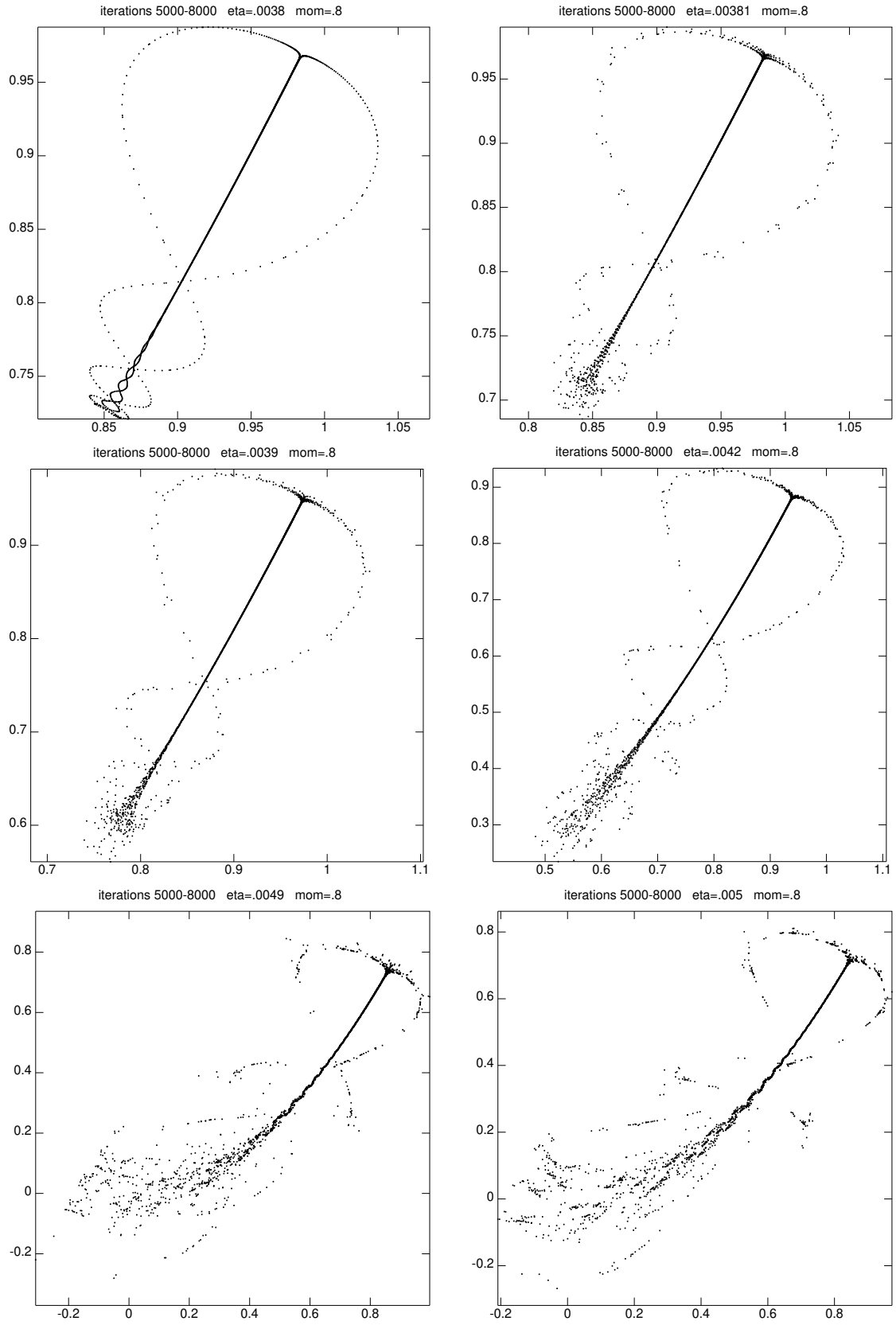


Figure 4.22: Attractors for gradient descent with momentum on Rosenbrock's function started at $w_1 = w_2 = 0$. Learning parameters: $\alpha = .8$, $\eta = .0038, \ldots, .005$.

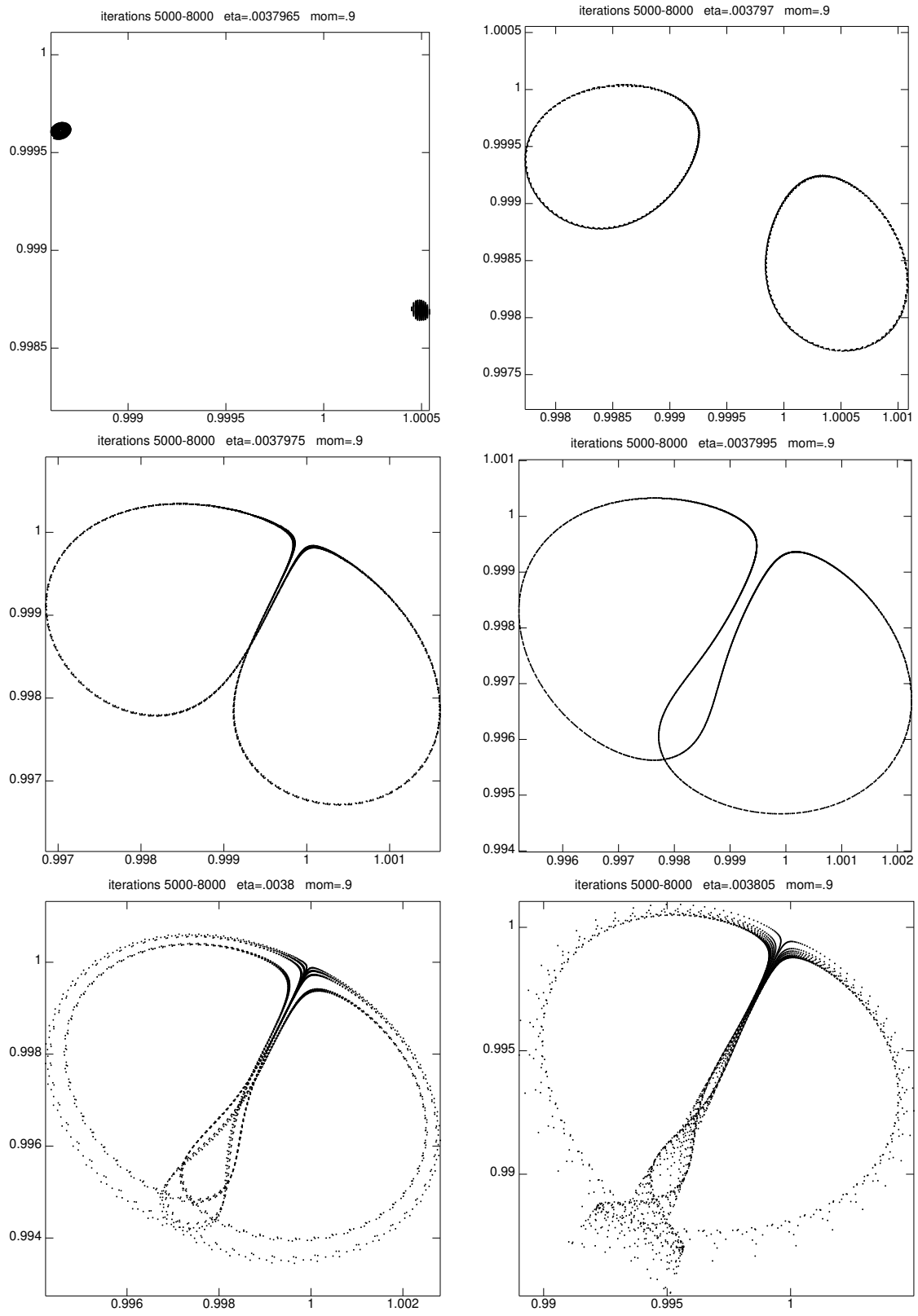


Figure 4.23: Attractors for gradient descent with momentum on Rosenbrock's function started at $w_1 = w_2 = 0$. Learning parameters: $\alpha = .9$, $\eta = .0037965, \ldots, .003805$.

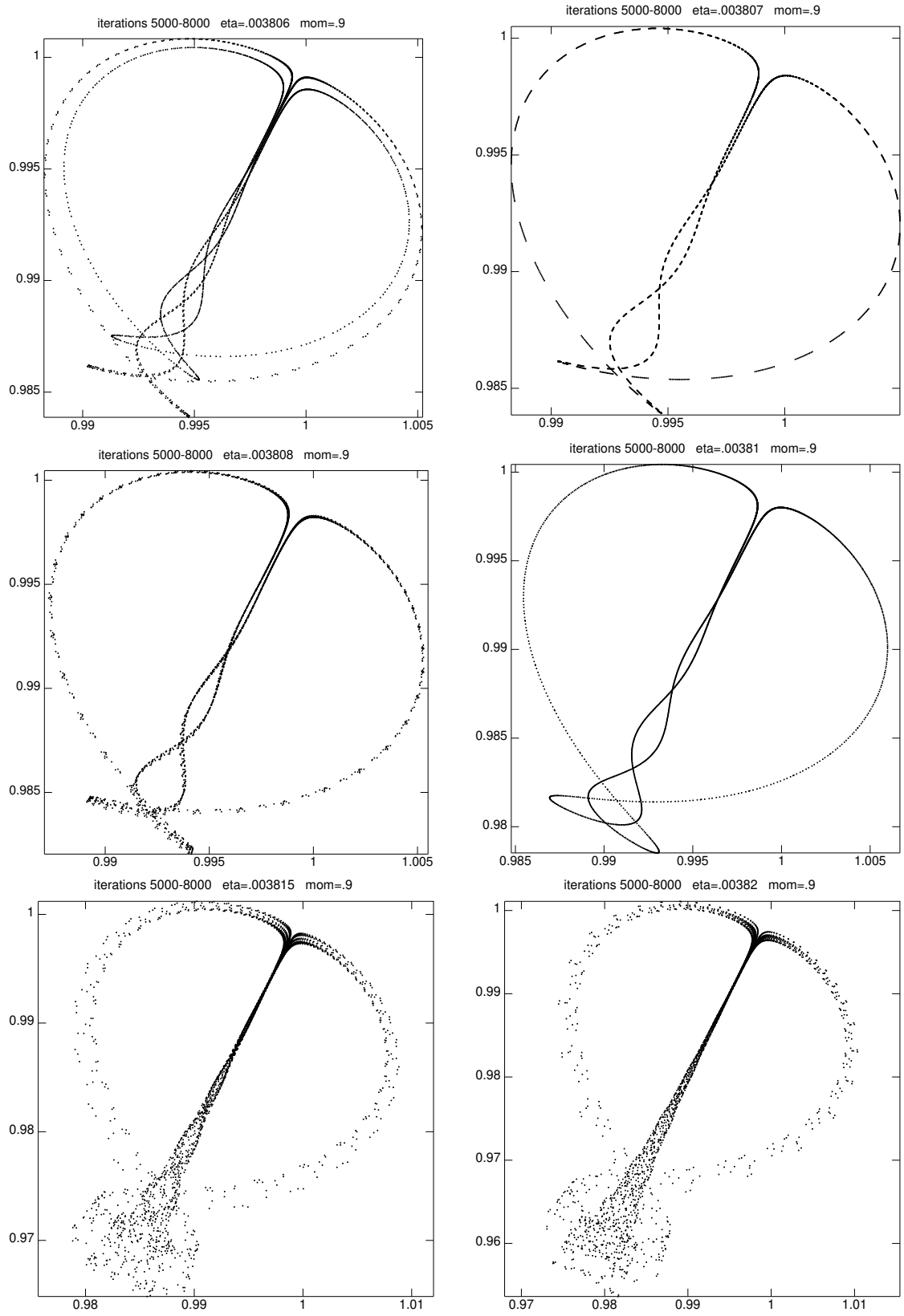


Figure 4.24: Attractors for gradient descent with momentum on Rosenbrock's function started at $w_1 = w_2 = 0$. Learning parameters: $\alpha = .9$, $\eta = .003806, \ldots, .00382$.

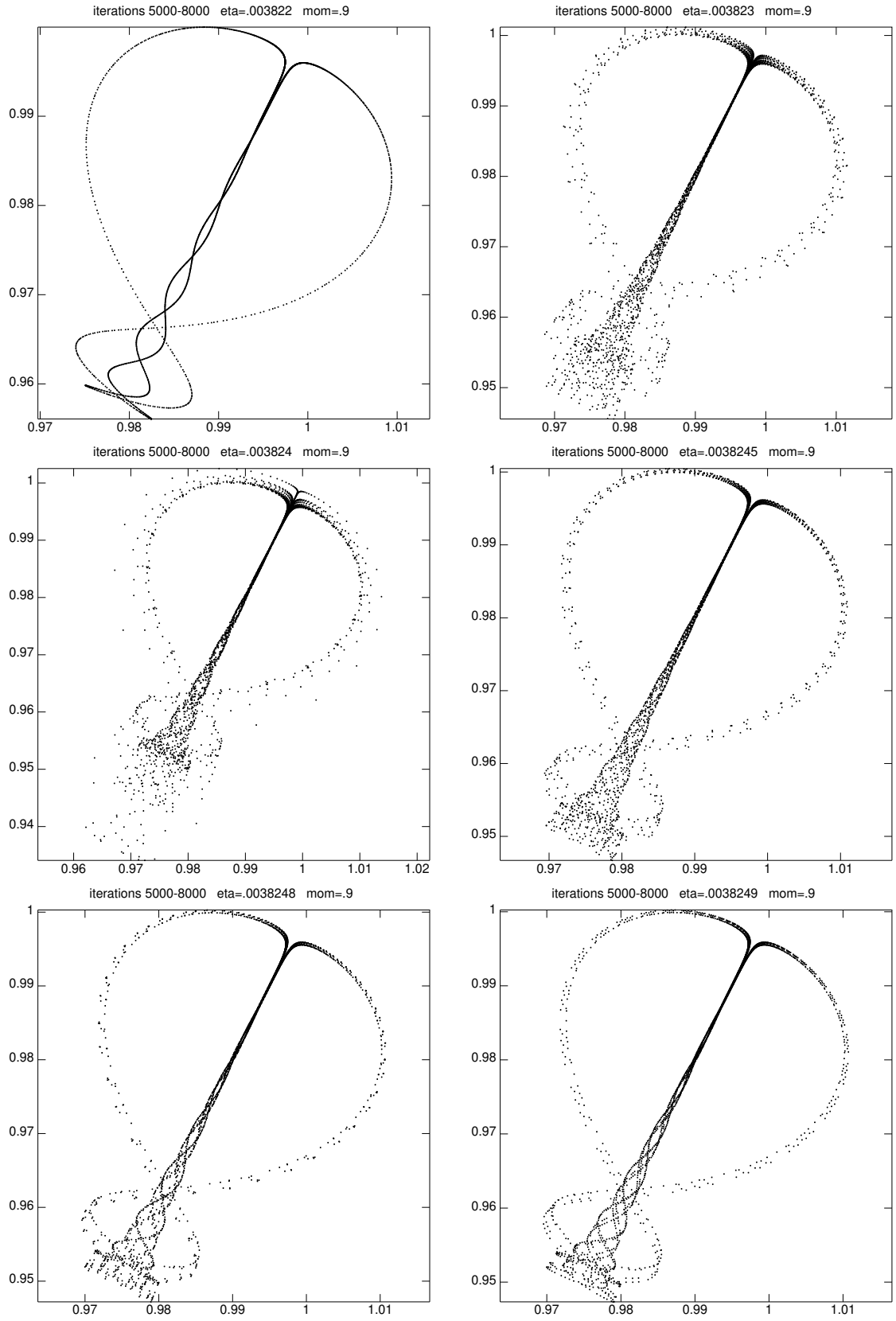


Figure 4.25: Attractors for gradient descent with momentum on Rosenbrock's function started at $w_1 = w_2 = 0$. Learning parameters: $\alpha = .9$, $\eta = .003822, \ldots, .0038249$.

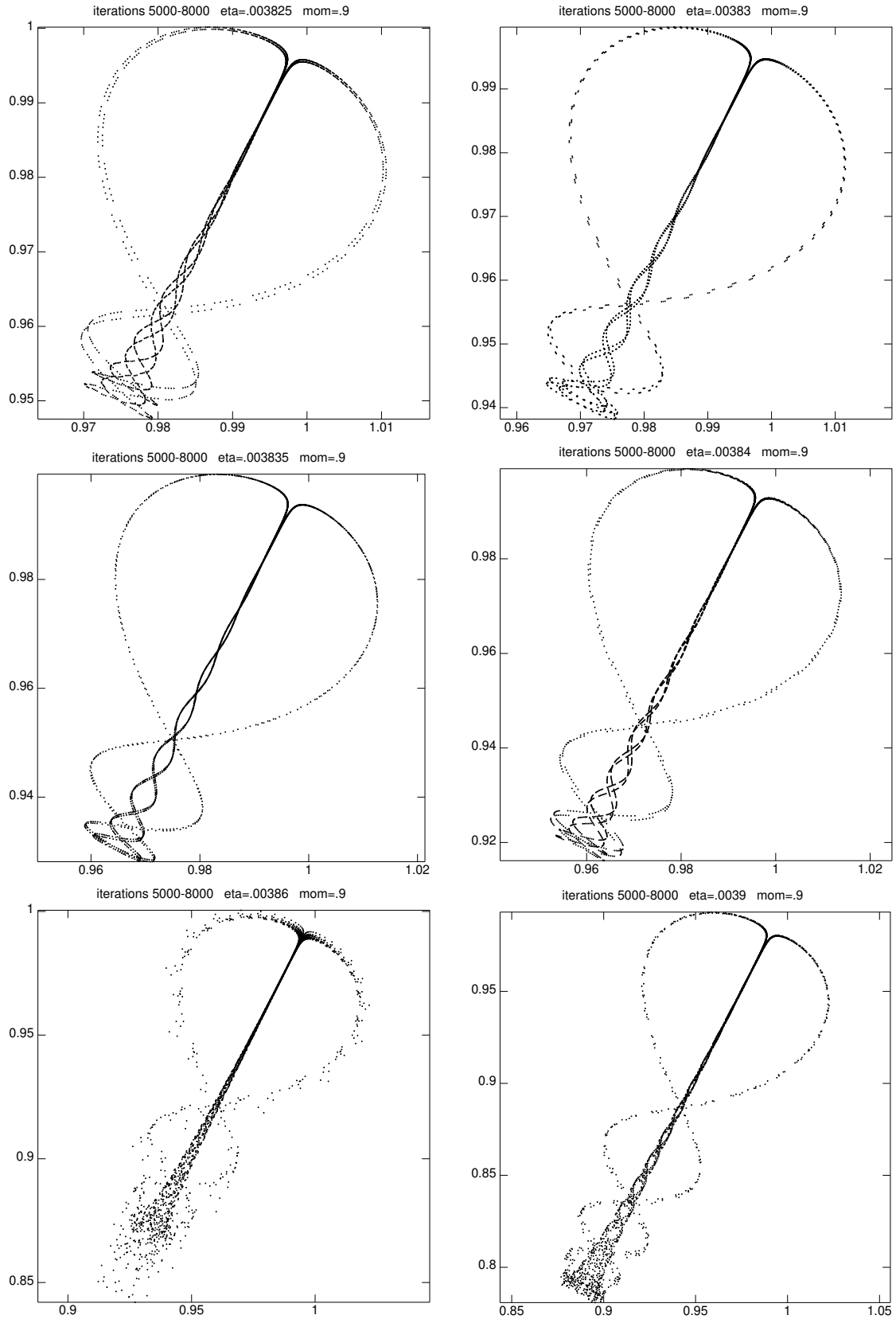


Figure 4.26: Attractors for gradient descent with momentum on Rosenbrock's function started at $w_1 = w_2 = 0$. Learning parameters: $\alpha = .9$, $\eta = .003825, \ldots, .0039$.

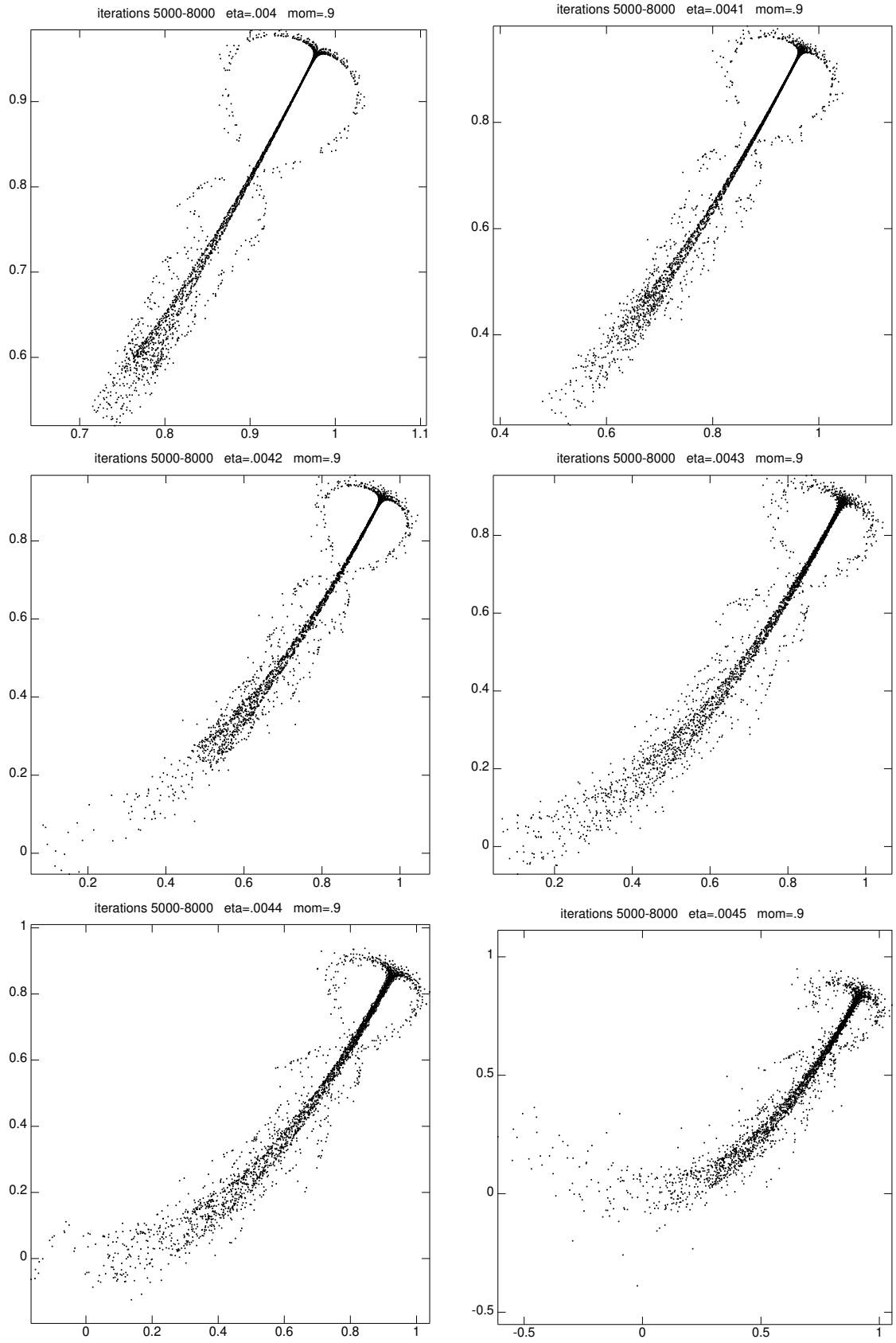


Figure 4.27: Attractors for gradient descent with momentum on Rosenbrock's function started at $w_1 = w_2 = 0$. Learning parameters: $\alpha = .9$, $\eta = .004, \ldots, .0045$.

Chapter 5

Fast Exact Multiplication by the Hessian

Capsule: Compute $\mathbf{Hv} = (\frac{\partial}{\partial r})\ \nabla_{\mathbf{w}}(\mathbf{w} + r\mathbf{v})|_{r=0}$ by forward propagation through whatever process is normally used to calculate $\nabla_{\mathbf{w}}$. This provides a numerically well conditioned technique for computing $\mathbf{Hv}$ with about the same amount of computation as required for a gradient calculation. It can be applied in both deterministic and stochastic gradient situations, to most any system which computes a gradient, and allows the Hessian, which is not in general sparse, to be treated using numerical techniques suitable for generalized sparse matrices.

Much of this chapter appeared as Pearlmutter (1994).

5.1 Introduction

Efficiently extracting second-order information from large neural networks is an important problem, because properties of the Hessian appear frequently—for instance, in the analysis of the convergence of learning algorithms (see chapter 4); in some techniques for predicting generalization rates in neural networks (MacKay, 1991; Moody, 1992); in techniques for enhancing generalization by weight elimination (LeCun, Denker, and Solla, 1990; Hassibi and Stork, 1993) or adjustment (Hochreiter and Schmidhuber, 1995); and in full second-order optimization methods (Watrous, 1987). As shown graphically in figure 5.1, the gradient gives little information about the local shape of the error surface, while the Hessian gives a great deal.

Bishop (1992), Werbos (1992), Buntine and Weigend (1994) calculate the full Hessian $\mathbf{H}$ (the matrix of second derivative terms $\partial^2 E/\partial w_i \partial w_j$ of the error E with

respect to the weights $\mathbf{w}$) of a backpropagation network, and MacKay (1991) calculates a reasonable estimate thereof—but even storing the full Hessian is impractical for large networks. Becker and LeCun (1989), LeCun *et al.* (1990) efficiently compute just the diagonal of the Hessian. This is useful when the trace of the Hessian is needed, or when a diagonal approximation is being made—but there is no reason to believe that the diagonal approximation is good in general, and it is reasonable to suppose that, as the system grows, the diagonal elements of the Hessian become less and less dominant. Further, the inverse of the diagonal approximation of the Hessian is known to be a poor approximation to the diagonal of the inverse Hessian.

Here we derive an efficient technique for calculating the product of an arbitrary vector $\mathbf{v}$ with the Hessian $\mathbf{H}$. This allows information to be extracted from the Hessian without ever calculating or storing the Hessian itself. A common use for an estimate of the Hessian is to take its product with various vectors. This takes $O(n^2)$ time when there are n weights. The technique we derive here finds this product in $O(n)$ time and space,[1] and does not make any approximations.

We first operate in a very general framework, to develop the basic technique. We then apply it to a series of more and more complicated systems, starting with a typical non-iterative gradient calculation algorithm, in particular a backpropagation network, and proceeding to some deterministic relaxation systems, and then to some stochastic systems, in particular a Boltzmann Machine and a weight perturbation system. Finally, we experiment with a technique for accelerating convergence by stiffening the principal eigenspace.

5.2 The Relation Between the Gradient and the Hessian

The basic technique is to note that the Hessian matrix appears in the Taylor expansion of the gradient about a point in weight space,

$$\nabla_{\mathbf{w}}(\mathbf{w} + \Delta\mathbf{w}) = \nabla_{\mathbf{w}}(\mathbf{w}) + \mathbf{H}\Delta\mathbf{w} + O(||\Delta\mathbf{w}||^2)$$

where $\mathbf{w}$ is a point in weight space, $\Delta\mathbf{w}$ is a perturbation of $\mathbf{w}$, $\nabla_{\mathbf{w}}$ is the gradient, the vector of partial derivatives $\partial E/\partial w_i$, and $\mathbf{H}$ is the Hessian, the matrix of second derivatives of E with respect to each pair of elements of $\mathbf{w}$ (shown geometrically in figure 5.2.) In chapter 4 this equation was used to analyze the convergence properties of some variants of gradient descent (Widrow *et al.*, 1976; LeCun *et al.*, 1991; Pearlmutter, 1992a), and to approximate the effect of deleting a weight from the network (LeCun *et al.*, 1990; Hassibi and Stork, 1993). Here we instead use it by choosing $\Delta\mathbf{w} = r\mathbf{v}$, where $\mathbf{v}$ is a vector and r is a small number. We wish to compute $\mathbf{H}\mathbf{v}$.

[1] Or $O(pn)$ time when, as is typical for supervised neural networks, the full gradient is the sum of p gradients, each for one single exemplar.

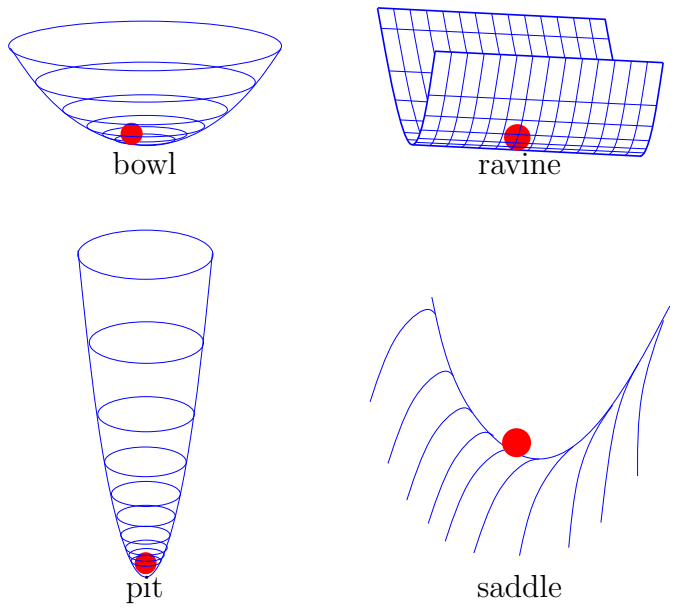


Figure 5.1: The gradient is the same; the difference is in the second-order structure.

Now we note that

$$\mathbf{H}(r\mathbf{v}) = r\mathbf{H}\mathbf{v} = \nabla_{\mathbf{w}}(\mathbf{w} + r\mathbf{v}) - \nabla_{\mathbf{w}}(\mathbf{w}) + O(r^2)$$

or, dividing by r,

$$\mathbf{H}\mathbf{v} = \frac{\nabla_{\mathbf{w}}(\mathbf{w} + r\mathbf{v}) - \nabla_{\mathbf{w}}(\mathbf{w})}{r} + O(r). \tag{5.1}$$

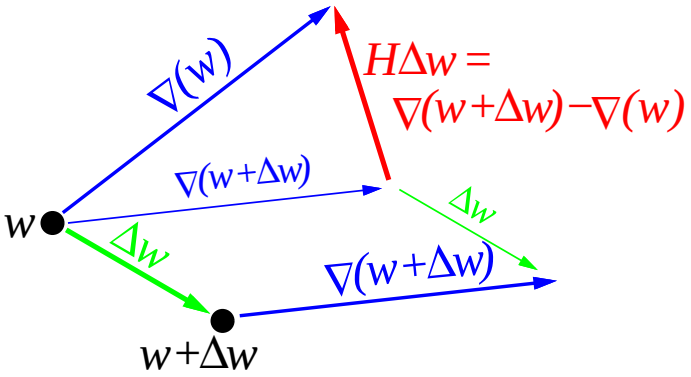


Figure 5.2: A geometric interpretation of the relation between the Hessian and the gradient.

This equation provides a simple approximation algorithm for finding $\mathbf{Hv}$ for any system whose gradient can be efficiently computed, in time about that required to compute the gradient (assuming that the gradient at $\mathbf{w}$ has already been computed.) Also, applying the technique requires minimal programming effort. This approximation was used to good effect in LeCun, Simard, and Pearlmutter (1993) and in many numerical analysis optimization routines, which use it to gradually build up an approximation to the inverse Hessian.

Unfortunately, this formula is succeptible to numeric and roundoff problems. The constant r must be small enough that the $O(r)$ term is insignificant. But as r becomes small, large numbers are added to tiny ones in $\mathbf{w}+r\mathbf{v}$, causing a loss of precision of $\mathbf{v}$. A similar loss of precision occurs in the subtraction of the original gradient from the perturbed one, because two nearly identical vectors are being subtracted to obtain the tiny difference between them.

5.3 The $\mathcal{R}\{\cdot\}$ Technique

Fortunately, there is a way to make an algorithm which exactly computes $\mathbf{Hv}$, rather than just approximating it, and simultaneously rid ourselves of these numeric difficulties. To do this, we first take the limit of equation (5.1) as $r \to 0$. The left hand side stays $\mathbf{Hv}$, while the right hand side matches the definition of a derivative, and thus

$$\mathbf{Hv} = \lim_{r\to 0} \frac{\nabla_{\mathbf{w}}(\mathbf{w}+r\mathbf{v}) - \nabla_{\mathbf{w}}(\mathbf{w})}{r} = \frac{\partial}{\partial r}\nabla_{\mathbf{w}}(\mathbf{w}+r\mathbf{v})\bigg|_{r=0} \tag{5.2}$$

As we shall see, there is a simple transformation to convert an algorithm that computes the gradient of the system into one that computes this new quantity. The key to this transformation is to define the operator

$$\mathcal{R}_{\mathbf{v}}\{f(\mathbf{w})\} \equiv \frac{\partial}{\partial r} f(\mathbf{w}+r\mathbf{v})\bigg|_{r=0} \tag{5.3}$$

so $\mathbf{Hv} = \mathcal{R}_{\mathbf{v}}\{\nabla_{\mathbf{w}}(\mathbf{w})\}$. (To avoid clutter we will usually write $\mathcal{R}\{\cdot\}$ instead of $\mathcal{R}_{\mathbf{v}}\{\cdot\}$.) We can then take all the equations of a procedure that calculates a gradient, *e.g.* the backpropagation procedure, and we can apply the $\mathcal{R}_{\mathbf{v}}\{\cdot\}$ operator to each equation. Because $\mathcal{R}\{\cdot\}$ is a differential operator, it obeys the usual rules for differential operators, such as:

$$\begin{aligned}
\mathcal{R}\{cf(\mathbf{w})\} &= c\mathcal{R}\{f(\mathbf{w})\} \\
\mathcal{R}\{f(\mathbf{w}) + g(\mathbf{w})\} &= \mathcal{R}\{f(\mathbf{w})\} + \mathcal{R}\{g(\mathbf{w})\} \\
\mathcal{R}\{f(\mathbf{w})g(\mathbf{w})\} &= \mathcal{R}\{f(\mathbf{w})\}\, g(\mathbf{w}) + f(\mathbf{w})\mathcal{R}\{g(\mathbf{w})\} \\
\mathcal{R}\{f(g(\mathbf{w}))\} &= f'(g(\mathbf{w}))\mathcal{R}\{g(\mathbf{w})\} \\
\mathcal{R}\left\{\frac{df(\mathbf{w})}{dt}\right\} &= \frac{d\mathcal{R}\{f(\mathbf{w})\}}{dt}
\end{aligned} \tag{5.4}$$

Also note that

$$\mathcal{R}\{\mathbf{w}\} = \mathbf{v}. \tag{5.5}$$

These rules are sufficient to derive, from the equations normally used to compute the gradient, a new set of equations about a new set of $\mathcal{R}$-variables. These new equations make use of variables from the original gradient calculation on their right hand sides. This can be thought of as an adjoint system to the gradient calculation, just as the gradient calculation of backpropagation can be thought of as an adjoint system to the forward calculation of the error measure. This new adjoint system computes the vector $\mathcal{R}\{\nabla_{\mathbf{w}}\}$, which is precisely the vector $\mathbf{Hv}$ which we desire.

5.4 Application of the $\mathcal{R}\{\cdot\}$ Technique to Various Networks

Let us utilize this new technique for transforming the equations that compute the gradient into equations that compute $\mathbf{Hv}$, the product of a vector $\mathbf{v}$ with the Hessian $\mathbf{H}$. We will, rather mechanically, derive appropriate algorithms for some standard sorts of neural networks that typify three broad classes of gradient calculation algorithms. These examples are intended to be illustrative, as the technique applies equally well to most other gradient calculation procedures, such as networks with weight sharing, weight decay, Sigma-Pi networks (Durbin and Rumelhart, 1989; Mel and Koch, 1990), RTRL (Robinson and Fallside, 1988; Williams and Zipser, 1989) and see section 3.3.2, the extended Kalman filter (Matthews, 1990; Williams, 1992), or even the linearity-based recurrent network gradient calculation technique of (Sun *et al.*, 1992).

Usually the error E is the sum of the errors for many patterns, $E = \sum_p E_p$. Therefore $\nabla_{\mathbf{w}}$ and $\mathbf{H}$ are sums over all the patterns, $\mathbf{H} = \sum_p \mathbf{H}_p$, and $\mathbf{Hv} = \sum_p \mathbf{H}_p\mathbf{v}$. As is usual, for clarity this outer sum over patterns is not shown except where necessary, and the gradient and $\mathbf{Hv}$ procedures are shown for only a single exemplar.

5.4.1 Simple Backpropagation Networks

Let us apply the above procedure to a simple backpropagation network, to derive the $\mathcal{R}\{\text{backprop}\}$ algorithm, a set of equations that can be used to efficiently calculate $\mathbf{Hv}$ for a backpropagation network. In press, I found that the $\mathcal{R}\{\text{backprop}\}$ algorithm was independently discovered a number of times. Werbos (1988a, eq. 14) derived it as a backpropagation process to calculate $\mathbf{Hv} = \nabla_{\mathbf{w}}(\mathbf{v} \cdot \nabla_{\mathbf{w}} E)$, where $\nabla_{\mathbf{w}} E$ is also calculated by backpropagation. That derivation is dual to the one given here, in that the direction of the equations is reversed, the backwards pass of the $\nabla_{\mathbf{w}} E$ algorithm becoming a forward pass in the $\mathbf{Hv}$ algorithm, while here the direction of the equations is unchanged. Another derivation is given in Møller (1993a). Also, the

procedure is known to the automatic differentiation community (Christianson, 1992; Kim, Nesterov, and Cherkassky, 1985).

For convenience, we will now change our notation for indexing the weights $\mathbf{w}$. Let $\mathbf{w}$ be the weights, now doubly indexed by their source and destination units' indices, as in w_{ij}, the weight from unit i to unit j. Because $\mathbf{v}$ is of the same dimension as $\mathbf{w}$, its elements will be similarly indexed. All sums over indices are limited to weights that exist in the network topology. As is usual, quantities which occur on the left sides of the equations are treated computationally as variables, and calculated in topological order, which is assumed to exist because the weights, regarded as a connection matrix, is zero-diagonal and can be put into triangular form (Werbos, 1974).

The forward computation of the network is[2]

$$\begin{aligned} x_i &= \sum_j w_{ji} y_j \\ y_i &= \sigma_i(x_i) + I_i \end{aligned} \tag{5.6}$$

where $\sigma_i(\cdot)$ is the nonlinearity of the ith unit, x_i is the total input to the ith unit, y_i is the output of the ith unit, and I_i is the external input (from outside the network) to the ith unit.

Let the error measure be $E = E(y)$, and its simple direct derivative with respect to y_i be $e_i = dE/dy_i$. We assume that e_i depends only on y_i, and not on any y_j for $j \neq i$. This is true of most common error measures, such as squared error or cross entropy (Hinton, 1987).[3] We can thus write $e_i(y_i)$ as a simple function. The backward pass is then

$$\begin{aligned} \frac{\partial E}{\partial y_i} &= e_i(y_i) + \sum_j w_{ij} \frac{\partial E}{\partial x_j} \\ \frac{\partial E}{\partial x_i} &= \sigma_i'(x_i) \frac{\partial E}{\partial y_i} \\ \frac{\partial E}{\partial w_{ij}} &= y_i \frac{\partial E}{\partial x_j} \end{aligned} \tag{5.7}$$

Applying $\mathcal{R}\{\cdot\}$ to the above equations gives

$$\begin{aligned} \mathcal{R}\{x_i\} &= \sum_j \left(w_{ji} \mathcal{R}\{y_j\} + v_{ji} y_j \right) \\ \mathcal{R}\{y_i\} &= \mathcal{R}\{x_i\} \, \sigma_i'(x_i) \end{aligned} \tag{5.8}$$

[2]This compact form of the backpropagation equations, due to Fernando Pineda, unifies the special cases of input units, hidden units, and output units. In the case of a unit i with no incoming weights, *i.e.* an input unit, it simplifies to $y_i = \sigma_i(0) + I_i$, allowing the value to be set entirely externally. For a hidden unit or output i, the term $I_i = 0$. In the corresponding equations for the backward pass (5.7) only the output units have nonzero direct error terms e_i, and since such output units have no outgoing weights, the situation for an output unit i simplifies to $\partial E/\partial y_i = e_i(y_i)$.

[3]If this assumption is violated then in equation (5.9) the $e_i'(y_i)\mathcal{R}\{y_i\}$ term generalizes to $\sum_j (\partial e_i/\partial y_j)\mathcal{R}\{y_j\}$.

for the forward pass, and, for the backward pass,

$$\begin{aligned}
\mathcal{R}\left\{\frac{\partial E}{\partial y_i}\right\} &= e_i'(y_i)\mathcal{R}\{y_i\} + \sum_j \left(w_{ij}\mathcal{R}\left\{\frac{\partial E}{\partial x_j}\right\} + v_{ij}\frac{\partial E}{\partial x_j}\right) \qquad (5.9)\\
\mathcal{R}\left\{\frac{\partial E}{\partial x_i}\right\} &= \sigma_i'(x_i)\mathcal{R}\left\{\frac{\partial E}{\partial y_i}\right\} + \mathcal{R}\{x_i\}\,\sigma_i''(x_i)\frac{\partial E}{\partial y_i}\\
\mathcal{R}\left\{\frac{\partial E}{\partial w_{ij}}\right\} &= y_i\mathcal{R}\left\{\frac{\partial E}{\partial x_j}\right\} + \mathcal{R}\{y_i\}\,\frac{\partial E}{\partial x_j}
\end{aligned}$$

The vector whose elements are $\mathcal{R}\{\partial E/\partial w_{ij}\}$ is just $\mathcal{R}\{\nabla_{\mathbf{w}}\} = \mathbf{Hv}$, the quantity we wish to compute.

For sum squared error $e_i(y_i) = y_i - d_i$ where d_i is the desired output for unit i, so $e_i'(y_i) = 1$. This simplifies (5.9) for simple output units to $\mathcal{R}\{\partial E/\partial y_i\} = \mathcal{R}\{y_i\}$. Note that, in the above equations, the topology of the neural network sometimes results in some $\mathcal{R}$-variables being guaranteed to be zero when $\mathbf{v}$ is sparse—in particular when $\mathbf{v} = (0\cdots0\ 1\ 0\cdots0)$, which can be used to compute a single desired column of the Hessian. In this situation, some of the computation is also shared between various columns.

5.4.2 Recurrent Backpropagation Networks

The recurrent backpropagation algorithm discussed in section 3.2.3 consists of a set of forward equations which relax to a solution for the gradient,

$$\begin{aligned}
x_i &= \sum_j w_{ji}y_j \qquad (5.10)\\
\frac{dy_i}{dt} &\propto -y_i + \sigma_i(x_i) + I_i\\
\frac{dz_i}{dt} &\propto -z_i + \sigma_i'(x_i)\sum_j (w_{ij}z_j) + e_i(y_i)\\
\frac{\partial E}{\partial w_{ij}} &= y_i z_j|_{t=\infty}
\end{aligned}$$

Adjoint equations for the calculation of $\mathbf{Hv}$ are obtained by applying the $\mathcal{R}\{\cdot\}$ operator, yielding

$$\begin{aligned}
\mathcal{R}\{x_i\} &= \sum_j (w_{ji}\mathcal{R}\{y_i\} + v_{ji}y_j) \qquad (5.11)\\
\frac{d\mathcal{R}\{y_i\}}{dt} &\propto -\mathcal{R}\{y_i\} + \sigma_i'(x_i)\mathcal{R}\{x_i\}\\
\frac{d\mathcal{R}\{z_i\}}{dt} &\propto -\mathcal{R}\{z_i\} + \sigma_i'(x_i)\sum_j (v_{ij}z_j + w_{ij}\mathcal{R}\{z_j\}) + \sigma_i''(x_i)\mathcal{R}\{x_i\}\sum_j (w_{ij}z_j) + e_i'(y_i)\mathcal{R}\{y_i\}\\
\mathcal{R}\left\{\frac{\partial E}{\partial w_{ij}}\right\} &= y_i\mathcal{R}\{z_j\} + \mathcal{R}\{y_i\}\,z_j|_{t=\infty}
\end{aligned}$$

These equations specify a relaxation process for computing $\mathbf{Hv}$. Just as the relaxation equations for computing $\nabla_{\mathbf{w}}$ are linear even though those for computing y and E are not, these new relaxation equations are linear.

5.4.3 Deterministic Boltzmann Machines

Deterministic (or Mean Field) Boltzmann Machines (reviewed in section 3.2.4) have relaxation equations, similar in spirit to those of recurrent backpropagation networks, so application of the $\mathcal{R}\{\cdot\}$ technique to them is precisely analogous to the procedure above. The gradient is calculated by

$$\begin{aligned} y_i &= \sigma(x_i/T) \\ x_i &= \sum_j w_{ji} y_j \\ p_{ij} &= \langle y_i y_j \rangle = \sum_\alpha P(\alpha) y_i^{(\alpha)} y_j^{(\alpha)} \\ \frac{\partial G}{\partial w_{ij}} &= (p_{ij}^+ - p_{ij}^-)/T \end{aligned} \tag{5.12}$$

where the weight matrix is symmetric, $w_{ij} = w_{ji}$ and zero diagonal, $w_{ii} = 0$, T is the temperature, G is the same error term used in stochastic Boltzmann Machines as described below, the $+$ and $-$ superscripts denote different environmental distributions, and $P(\alpha)$ is the probability of the pattern α over the visible units occurring in the environment, typically implicitly computed by summing over samples drawn from the environment.

We apply $\mathcal{R}\{\cdot\}$ and get

$$\begin{aligned} \mathcal{R}\{y_i\} &= \sigma'(x_i/T)\mathcal{R}\{x_i\}/T \\ \mathcal{R}\{x_i\} &= \sum_j \left(w_{ji}\mathcal{R}\{y_j\} + v_{ji} y_j\right) \\ \mathcal{R}\{p_{ij}\} &= \sum_\alpha P(\alpha)\left(\mathcal{R}\left\{y_i^{(\alpha)}\right\} y_j^{(\alpha)} + y_i^{(\alpha)}\mathcal{R}\left\{y_j^{(\alpha)}\right\}\right) \\ \mathcal{R}\left\{\frac{\partial G}{\partial w_{ij}}\right\} &= \left(\mathcal{R}\left\{p_{ij}^+\right\} - \mathcal{R}\left\{p_{ij}^-\right\}\right)/T. \end{aligned} \tag{5.13}$$

where $P(\alpha)$, the environmental probability of pattern α over the visible units is a constant because it is independent of the network.

5.4.4 Stochastic Boltzmann Machines

One might ask whether this technique can be used to derive a Hessian multiplication algorithm for a classic Boltzmann Machine (Ackley *et al.*, 1985), which is discrete

and stochastic, unlike its continuous and deterministic cousin to which application of $\mathcal{R}\{\cdot\}$ is simple. A classic Boltzmann Machine operates stochastically, with its binary unit states s_i taking on random values according to the probability

$$\begin{aligned} P(s_i = 1) &= p_i = \sigma(x_i/T) \\ x_i &= \sum_j w_{ji} s_j \end{aligned} \tag{5.14}$$

At equilibrium, the probability of a state α of all the units (not just the visible units) is related to its energy

$$E_\alpha = \sum_{i<j} s_i^\alpha s_j^\alpha w_{ij} \tag{5.15}$$

by $P(\alpha) = Z^{-1} \exp -E_\alpha/T$, where the partition function is $Z = \sum_\alpha \exp -E_\alpha/T$. The system's equilibrium statistics are sampled because, at equilibrium,

$$\frac{\partial G}{\partial w_{ij}} = \left(p_{ij}^+ - p_{ij}^-\right)/T \tag{5.16}$$

where $p_{ij} = \langle s_i s_j \rangle$, G is the asymmetric divergence, an information theoretic measure of the difference between the environmental distribution over the output units and that of the network, as used in Ackley *et al.* (1985), T is the temperature, and the $+$ and $-$ superscripts indicate the environmental distribution, $+$ for waking and $-$ for hallucinating.

Applying the $\mathcal{R}\{\cdot\}$ operator, we obtain

$$\mathcal{R}\left\{\frac{\partial G}{\partial w_{ij}}\right\} = \left(\mathcal{R}\left\{p_{ij}^+\right\} - \mathcal{R}\left\{p_{ij}^-\right\}\right)/T. \tag{5.17}$$

We shall soon find it useful if we define

$$D_\alpha = \mathcal{R}\{E_\alpha\} = \sum_{i<j} s_i^\alpha s_j^\alpha v_{ij} \tag{5.18}$$

$$q_{ij} = \langle s_i s_j D \rangle \tag{5.19}$$

(with the letter D chosen because it has the same relation to $\mathbf{v}$ that E has to $\mathbf{w}$) and to note that

$$\langle D \rangle = \sum_{i<j} p_{ij} v_{ij}. \tag{5.20}$$

With some calculus, we find $\mathcal{R}\{\exp -E_\alpha/T\} = -P(\alpha)\ Z\ D_\alpha/T$, and thus $\mathcal{R}\{Z\} = -Z \langle D \rangle /T$. Using these and the relation between the probability of a state and its energy, we have

$$\mathcal{R}\{P(\alpha)\} = P(\alpha)\left(\langle D \rangle - D_\alpha\right)/T \tag{5.21}$$

where the expression $P(\alpha)$ can not be treated as a constant because it is defined over all the units, not just the visible ones, and therefore depends on the weights. This

can be used to calculate

$$\begin{aligned}\mathcal{R}\{p_{ij}\} &= \sum_\alpha \mathcal{R}\{P(\alpha)\}\, s_i^\alpha s_j^\alpha \\ &= \sum_\alpha P(\alpha)\left(\langle D\rangle - D_\alpha\right) s_i^\alpha s_j^\alpha / T \\ &= \left(\langle s_i s_j\rangle \langle D\rangle - \langle s_i s_j D\rangle\right)/T \\ &= \left(p_{ij}\langle D\rangle - q_{ij}\right)/T. \end{aligned} \tag{5.22}$$

This beautiful formula[4] gives an efficient way to compute $\mathbf{Hv}$ for a Boltzmann Machine, or at least as efficient a way as is used to compute the gradient, simply by using sampling to estimate q_{ij}. This requires the additional calculation and broadcast of the single global quantity D, but is otherwise local.

The collection of statistics for the gradient is sometimes accelerated by using the equation

$$p_{ij} = \langle s_i\rangle \langle s_j | s_i = 1\rangle = \langle p_i\rangle \langle p_j | s_i = 1\rangle . \tag{5.23}$$

The analogous identity for accelerating the computation of q_{ij} is

$$q_{ij} = \langle p_i\rangle \langle s_j D | s_i = 1\rangle \tag{5.24}$$

or

$$q_{ij} = \langle p_i\rangle \langle p_j (D + (1-s_j)\Delta D_j) | s_i = 1\rangle \tag{5.25}$$

where $\Delta D_i = \sum_j s_j v_{ji}$ is defined by analogy with $\Delta E_i = E|_{s_i=1} - E|_{s_i=0} = \sum_j s_j w_{ji}$.

The derivation here was for the simplest sort of Boltzmann Machine, with binary units and only pairwise connections between the units. However, the technique is immediately applicable to higher-order Boltzmann Machines (Sejnowski, 1986), as well as to Boltzmann Machines with non-binary units (Movellan and McClelland, 1991).

5.4.5 Weight Perturbation

In weight perturbation (Jabri and Flower, 1991; Alspector, Meir, Yuhas, and Jayakumar, 1993; Flower and Jabri, 1993; Kirk, Kerns, Fleischer, and Barr, 1993; Cauwenberghs, 1993) the gradient $\nabla_{\mathbf{w}}$ is approximated using only the globally broadcast result of the computation of $E(\mathbf{w})$. This is done by adding a random zero-mean perturbation vector $\Delta\mathbf{w}$ to $\mathbf{w}$ repeatedly and approximating the resulting change in error by

$$E(\mathbf{w} + \Delta\mathbf{w}) = E(\mathbf{w}) + \Delta E = E(\mathbf{w}) + \nabla_{\mathbf{w}} \cdot \Delta\mathbf{w}.$$

[4]Equation (5.22) is similar in form to that of the gradient of the entropy (Geoff Hinton, personal communication.)

From the viewpoint of each individual weight w_i

$$\Delta E = \Delta w_i \frac{\partial E}{\partial w_i} + noise. \tag{5.26}$$

Because of the central limit theorem it is reasonable to make a least-squares estimate of $\partial E/\partial w_i$, which is $\partial E/\partial w_i = \langle \Delta w_i \Delta E \rangle / \langle \Delta w_i^2 \rangle$. The numerator is estimated from corresponding samples of Δw_i and ΔE, and the denominator from prior knowledge of the distribution of Δw_i. This requires only the global broadcast of ΔE, while each Δw_i can be generated and used locally.

It is hard to see a way to mechanically apply $\mathcal{R}\{\cdot\}$ to this procedure, but we can nonetheless derive a suitable procedure for estimating $\mathbf{Hv}$. We note that a better approximation for the change in error would be

$$E(\mathbf{w} + \Delta\mathbf{w}) = E(\mathbf{w}) + \nabla_{\mathbf{w}} \cdot \Delta\mathbf{w} + \tfrac{1}{2}\Delta\mathbf{w}^T \widehat{\mathbf{H}} \Delta\mathbf{w} \tag{5.27}$$

where $\widehat{\mathbf{H}}$ is an estimate of the Hessian $\mathbf{H}$. We wish to include in $\widehat{\mathbf{H}}$ only those properties of $\mathbf{H}$ which are relevent. Let us define $\mathbf{z} = \mathbf{Hv}$. If $\widehat{\mathbf{H}}$ is to be small in the least squares sense, but also $\widehat{\mathbf{H}}\mathbf{v} = \mathbf{z}$, then the best choice would then be $\widehat{\mathbf{H}} = \mathbf{z}\mathbf{v}^T/||\mathbf{v}||^2$, except that then $\widehat{\mathbf{H}}$ would not be symmetric, and therefore the error surface would not be well defined. Adding the symmetry requirement, which amounts to the added constraint $\mathbf{v}^T\widehat{\mathbf{H}} = \mathbf{z}^T$, the least squares $\widehat{\mathbf{H}}$ becomes

$$\widehat{\mathbf{H}} = \frac{1}{||\mathbf{v}||^2}\left(\mathbf{z}\mathbf{v}^T + \mathbf{v}\mathbf{z}^T - \frac{\mathbf{v}\cdot\mathbf{z}}{||\mathbf{v}||^2}\mathbf{v}\mathbf{v}^T\right). \tag{5.28}$$

Substituting this in and rearranging the terms, we find that, from the perspective of each weight,

$$\Delta E = \Delta w_i \frac{\partial E}{\partial w_i} + \frac{\Delta\mathbf{w}\cdot\mathbf{v}}{||\mathbf{v}||^2}\left(\Delta w_i - \frac{\Delta\mathbf{w}\cdot\mathbf{v}}{2||\mathbf{v}||^2} v_i\right) z_i + noise. \tag{5.29}$$

This allows both $\partial E/\partial w_i$ and z_i to be estimated in the same least-squares fashion as above, using only locally available values, v_i and Δw_i, and the globally broadcast ΔE, plus a new quantity which must be computed and globally broadcast, $\Delta\mathbf{w}\cdot\mathbf{v}$. The same technique applies equally well to other perturbative procedures, such as the unit perturbation of Flower and Jabri (1993), and a similar derivation can be used to find the diagonal elements of $\mathbf{H}$, without the need for any additional globally broadcast values.

5.5 Practical Applications

The $\mathcal{R}\{\cdot\}$ technique makes it possible to calculate $\mathbf{Hv}$ efficiently. This can be used in the center of many different iterative algorithms, in order to extract particular properties of $\mathbf{H}$. In essence, it allows $\mathbf{H}$ to be treated as a generalized sparse matrix.

5.5.1 Finding Eigenvalues and Eigenvectors

Standard variants of the power method allow one to

- Find the largest few eigenvalues of $\mathbf{H}$, and their eigenvectors.
- Find the smallest few eigenvalues of $\mathbf{H}$, and their eigenvectors.
- Sample $\mathbf{H}$'s eigenvalue spectrum, along with the corresponding eigenvectors.

The clever Skilling (1989a) algorithm estimates the eigenvalue spectrum of a generalized sparse matrix. It starts by choosing a random vector $\mathbf{v}_0$, calculating $\mathbf{v}_i = \mathbf{H}^i\mathbf{v}_0$ for $i = 1, \ldots, m$, using the dot products $\mathbf{v}_i \cdot \mathbf{v}_j$ as estimates of the moments of the eigenvalue spectrum, and using these moments to recover the shape of the eigenvalue spectrum. This algorithm is made applicable to the Hessian by the $\mathcal{R}\{\cdot\}$ technique, in both deterministic and, with minor modifications, stochastic gradient settings.

5.5.2 Multiplication by the Inverse Hessian

It is frequently necessary to find $\mathbf{x} = \mathbf{H}^{-1}\mathbf{b}$, which is the key calculation of all Newton's-method second-order numerical optimization techniques, and is also used in the Optimal Brain Surgeon technique, and in some techniques for predicting the generalization rate. The $\mathcal{R}\{\cdot\}$ technique does not directly solve this problem, but instead one can solve $\mathbf{Hx} = \mathbf{b}$ for $\mathbf{x}$ by minimizing $||\mathbf{Hx} - \mathbf{b}||^2$ using the conjugate-gradient method, thus exactly computing $\mathbf{x} = \mathbf{H}^{-1}\mathbf{b}$ in n iterations without calculating or storing $\mathbf{H}^{-1}$. This squares the condition number, but if $\mathbf{H}$ is known to be positive definite, one can instead minimize $\mathbf{x}^T\mathbf{Hx}/2 + \mathbf{x} \cdot \mathbf{b}$, which does not square the condition number (Press *et al.*, 1988, page 78). This application of fast exact multiplication by the Hessian, in particular $\mathcal{R}\{\text{backprop}\}$, was independently noted in Werbos (1988a).

5.5.3 Step Size and Line Search

Many optimization techniques repeatedly choose a direction $\mathbf{v}$, and then proceed along that direction some distance μ, which takes the system to the constrained minimum of $E(\mathbf{w} + \mu\mathbf{v})$. Finding the value for μ which minimizes E is called a line search, because it searches only along the line $\mathbf{w} + \mu\mathbf{v}$. There are many techniques for performing a line search. Some are approximate while others attempt to find an exact constrained minimum, and some use only the value of the error, while others also make use of the gradient.

In particular, the line search used within the Scaled Conjugate Gradient (SCG) optimization procedure, in both its deterministic (Møller, 1993b) and stochastic (Møller,

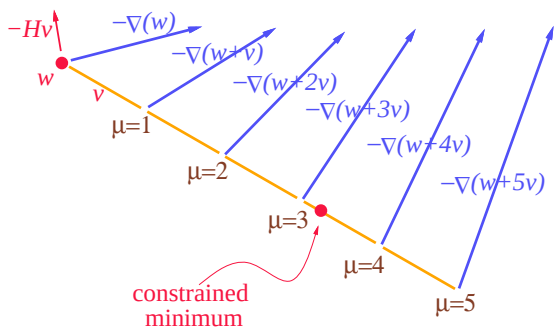


Figure 5.3: Geometric interpretation of approximate line search along direction $\mathbf{v}$.

1993c) incarnations, makes use of both first- and second-order information at $\mathbf{w}$ to determine how far to move. The first order information used is simply $\nabla_{\mathbf{w}}(\mathbf{w})$, while the second-order information is precisely $\mathbf{Hv}$, calculated with the one-sided finite difference approximation of equation (5.1). It can thus benefit immediately from the exact calculation of $\mathbf{Hv}$. In fact, the $\mathcal{R}\{\text{backprop}\}$ procedure was independently discovered for that application by Møller (1993a), and reviewed in comparitive studies by Jervis and Fitzgerald (1993).

The SCG line search proceeds as follows. Assuming that the error E is well approximated by a quadratic, then the product $\mathbf{Hv}$ and the gradient $\nabla_{\mathbf{w}}(\mathbf{w})$ predicts of the gradient at any point along the line $\mathbf{w} + \mu\mathbf{v}$ by

$$\nabla_{\mathbf{w}}(\mathbf{w} + \mu\mathbf{v}) = \nabla_{\mathbf{w}}(\mathbf{w}) + \mu\mathbf{Hv} + O(\mu^2). \tag{5.30}$$

Disregarding the $O(\mu^2)$ term, if we wish to choose μ to minimize the error, we take the dot product of $\nabla_{\mathbf{w}}(\mathbf{w} + \mu\mathbf{v})$ with $\mathbf{v}$ and set it equal to zero, as the gradient at the constrained minimum must be orthogonal to the space under consideration, as shown in figure fig:linesearch. This gives $\mathbf{v} \cdot \nabla_{\mathbf{w}}(\mathbf{w}) + \mu\mathbf{v}^T\mathbf{Hv} = 0$ or

$$\mu = -\frac{\mathbf{v} \cdot \nabla_{\mathbf{w}}(\mathbf{w})}{\mathbf{v}^T\mathbf{Hv}}. \tag{5.31}$$

Equation (5.30) then gives a prediction of the gradient at $\mathbf{w} + \mu\mathbf{v}$. To access the accuracy of the quadratic approximation we might wish to compare this with a gradient measurement taken at that point, or we might even preemptively take a step in that direction.

Divorced from the SCG algorithm, another application for this way of calculating μ is to eliminate the step size η of conventional gradient descent, which uses

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\nabla_{\mathbf{w}}(\mathbf{w}_t)$$

to gradually minimize E. Gradient descent suffers not only from a poor convergence rate, but also from the need to constantly tune η for rapid convergence as the minimization proceeds. The above simple line search suggests the use of $\eta = -\mu$ at each

step, or

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{||\nabla_{\mathbf{w}}(\mathbf{w}_t)||^2}{\nabla_{\mathbf{w}}(\mathbf{w}_t)^T \mathbf{H} \nabla_{\mathbf{w}}(\mathbf{w}_t)} \nabla_{\mathbf{w}}(\mathbf{w}_t). \tag{5.32}$$

The necessary modifications for gradient descent with momentum are trivial, as are the appropriate modifications for a stochastic gradient setting. Of course, this simple procedure needs to be augmented by mechanisms to check that $\mathbf{v}^T\mathbf{H}\mathbf{v} > 0$, and that the quadratic assumption is not inaccurate enough to cause failure to reduce E at each step.

5.5.4 Optimization of Stochastic Gradient Descent

The technique descibed in the previous section is, at least as stated, suitable only for deterministic gradient descent. In many systems, particularly large ones, deterministic gradient descent is impractical; only noisy estimates of the gradient are available. In joint work with colleagues at AT&T Bell Labs (LeCun *et al.*, 1993), the approximation technique of equation (5.1) enabled $\mathbf{H}$ to be treated as a generalized sparse matrix, and properties of $\mathbf{H}$ were extracted in order to accelerate the convergence of stochastic gradient descent.

Information accumulated online, in particular eigenvalues and eigenvectors of the principal eigenspace, was used to linearly transform the weight space in such a way that the ill-conditioned off-axis long narrow valleys in weight space, which slow down gradient descent, become well-conditioned circular bowls, as shown in figure 5.4. This work did not use an exact value for $\mathbf{H}\mathbf{v}$, but rather a stochastic unbiased estimate of the Hessian based on just a single exemplar at a time. Computations of the form $\mathbf{x}(t) = \mathbf{H}(t)\mathbf{v}$ were replaced with relaxations of the form $\mathbf{x}(t) = (1-\alpha)\mathbf{x}(t-1)+\alpha\widehat{\mathbf{H}}(t)\mathbf{v}$, where $0 < \alpha \ll 1$ determines the trade-off between steady-state noise and speed of convergence.

The key technique was to use the power method to compute the principal eigenvector with

$$\mathbf{v} \leftarrow (1-a)\mathbf{v} + \frac{a}{||\mathbf{v}||}\widehat{\mathbf{H}}\mathbf{v} \tag{5.33}$$

where a is a small constant and $\widehat{\mathbf{H}}$ an unbiased estimate of the Hessian $\mathbf{H}$. The vector $\mathbf{v}$ will converge to the princpal eigenvector of the true Hessian, and its length to the corresponding eigenvalue, at a rate determined jointly by a and the ratio between the principal and the next largest eigenvalues, and with a noise level determined by a. In practice the convergence has been found to be extremely rapid for networks of interest.

Different learning rates were then used in the space spanned by this principal eigenvector, and in the orthogonal remainder of the weight space.[5] The efficacy of this

[5]The technique could be generalized by maintaining not just one principal eigenvector, which evolves according to the above equation, but several of them, continuously orthogonalized, which

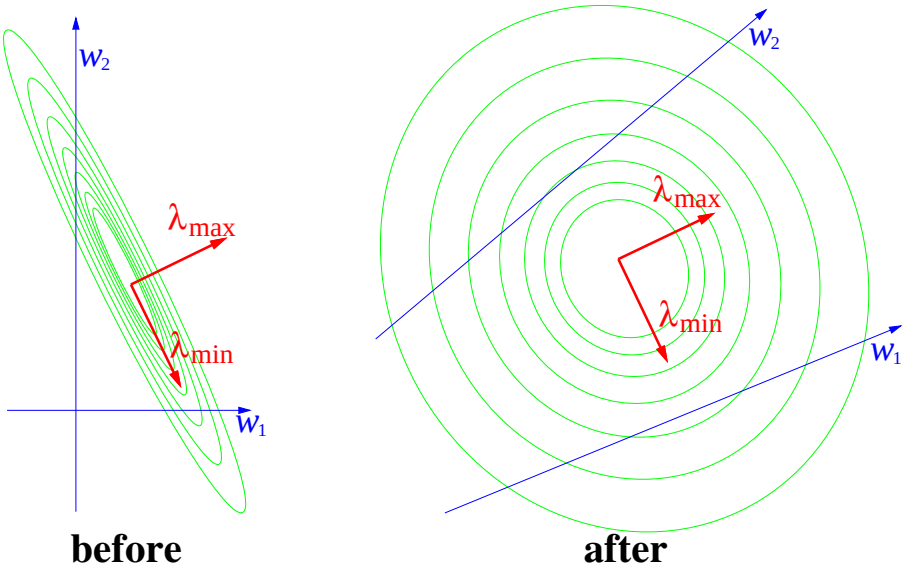


Figure 5.4: In the coordinate system defined by the weights, gradient descent is slow. By stretching the space in the direction of the principal eigenvector, gradient descent is accelerated. In the new coordinate system, the weights are no longer orthogonal, but become coupled.

approach was measured through experimentation on a very difficult two-dimensional problem, optimization of Rosenbrock's function from a starting point of $(w_1, w_2) = (-1.5, 1)$. Figure 5.5 shows that the global learning rate can be increased when the principal eigenspace is stiffened. This problem is ideal for the technique being considered, so if it cannot outperform the competition here, it certainly will not be able to do so elsewhere. Figure 5.6 shows the behavior of gradient descent with momentum on this problem. Figure 5.7 shows that, in the crucial regime of following a curving ravine, the new technique seems slightly superior to conjugate gradient. Note how conjugate gradient expends a great deal of effort keeping to the precise floor of the ravine, while the gradient-based technique is content to ride up the bank as the ravine curves.

In practice, although this technique seems to perform about as well as conjugate grdient methods on small low-dimensional problems particularly suited for it, it does

span a higher-dimensional principal eigenspace. Examination of the eiganvalue spectrum for our benchmark problems showed no large gaps, so there is no reason to believe that the speedup per tracked eigenvector would rise with the number of tracked eigenvectors. In fact, the measured spectra predict the reverse.

not appear to be competitive with the Sutton (1992a) incremental delta-bar-delta technique for adapting per-weight learning rates. When per-weight learning rate adaptation is employed, the principal eigenvalue of the Hessian is not much larger than the second largest eigenvalue, and so on. Such small separations obviously bode ill for the eigenstiffening technique in concert with incremental delta-bar-delta. Even without incremental delta-bar-delta, some simple experiments on the the NETTALK dataset using vanilla three-layer networks, with symmetric sigmoids and zero-mean inputs, the spread between the principal eigenvalue and its nearest competitor averaged about 20%—a small enough separation to make the overhead of the eigenstiffening technique outweigh its benefits.

Another way to view this is that the eigenstiffening technique proposed here can give a speedup of at most $\lambda_{\max}/\lambda_{\max-1}$. Typically this is not a very large number; perhaps two or three under exceptionally fortunate circumstances. Other advanced stochastic gradient methods can also obtain speedups in this ballpark, and with considerably less overhead and tuning. So we conclude that, at least without further advances in stochastic optimization algorithms, this technique is deserving of further research, but is not a serious alternative to current methods for production use.

In all fairness, it must also be pointed out that the basic idea upon which this stochastic gradient descent optimization scheme is based, namely that the direction of the principal eigenvector of the Hessian is the most difficult to optimize, is questionable. Although true in the batch setting, in the stochastic setting, in the terminal convergence regime, convergence is also limited by the noise. If the covariance matrix of the noise is nonspherical then the initial transients can be sped up by transforming the space to make the noise spherical, but this does not help with asymptotic convergence. It is sometimes conjectured that, in practice, stochastic gradient systems typically are run in the transient regime (that is, where the limitation on the learning rate η imposed by the principal eigenvalue of the Hessian is below the η dictated solely by the ratio between the distance to the minimum and the standard deviation of the noise). This of course makes assessing or comparing the performance of different optimization techniques very difficult, because their transient performance is (a) difficult to define, and (b) quite susceptible to tuning, initial conditions, and the initial values of automatically adjusted parameters of the optimization algorithm.

5.5.5 Second-Order Optimization Techniques

Optimization techniques like Levenberg-Marquart, Broyden-Fletcher-Goldfarb-Shanno, and SQB use approximations to the inverse Hessian. Many techniques for analyzing, estimating, and enhancing generalization in neural networks also make use of the Hessian or its inverse.

The conjugate gradient optimization takes n iterations, where one iteration in-

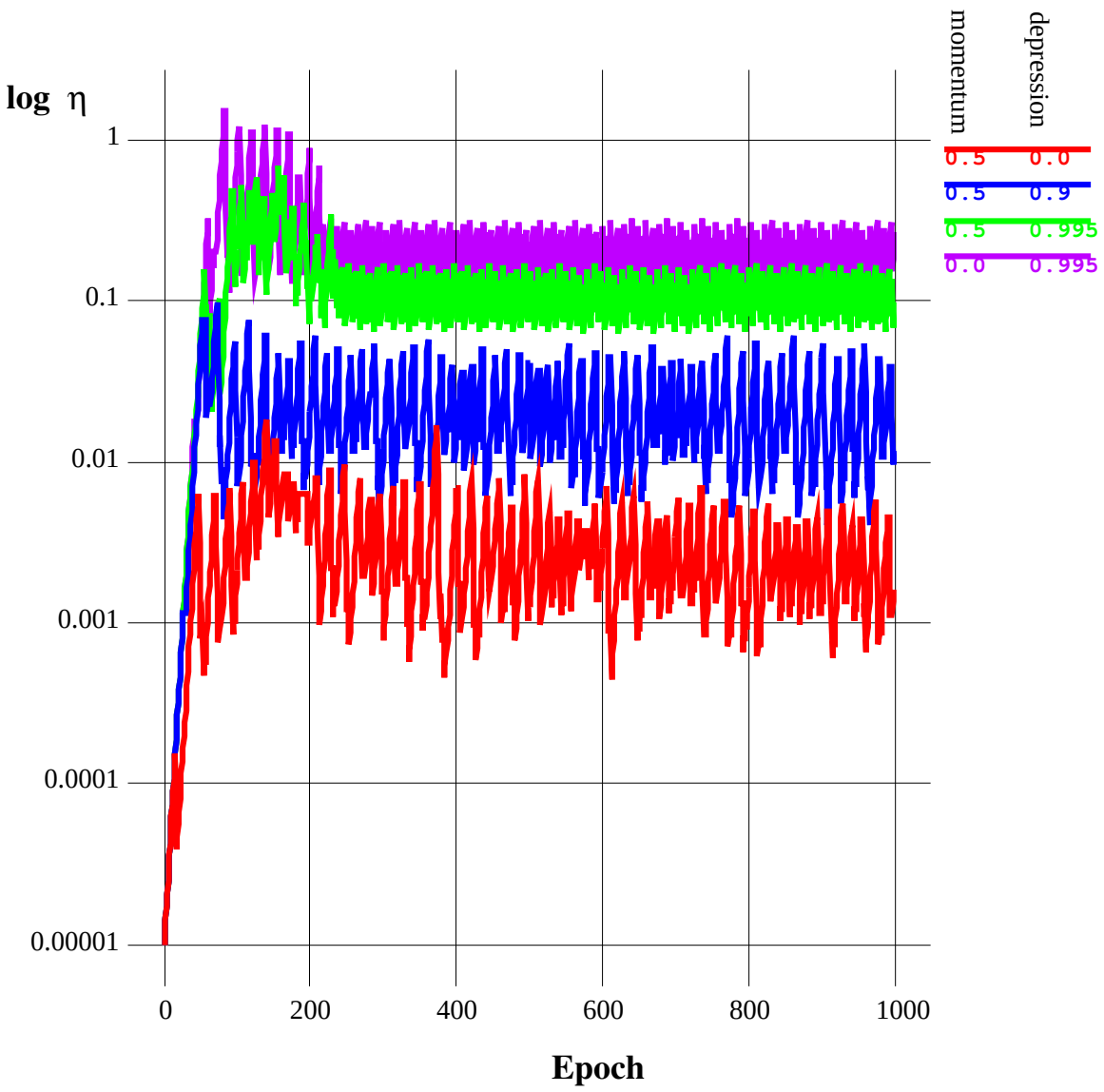


Figure 5.5: The depression parameter, which controls the extra stiffness of the system in the direction of the estimated principal eigenvector, allows the automatic controller to raise the learning rate η significantly. Note that the y axis is plotted on a logarithmic scale. The domain is a two dimensional optimization problem: minimization of Rosenbrocks function.

volves one multiplication by the Hessian, and is thus $O(n)$.[6] This gives an $O(n^2)$-

[6]In actuality, when we are not too concerned with having an exact solution, it is possible to make do with many fewer iterations, and it is also sometimes possible to make do with stochastic techniques. Also, when the problem is quadratic, the exact solution (rather than an approximation) is obtained in exactly n steps.

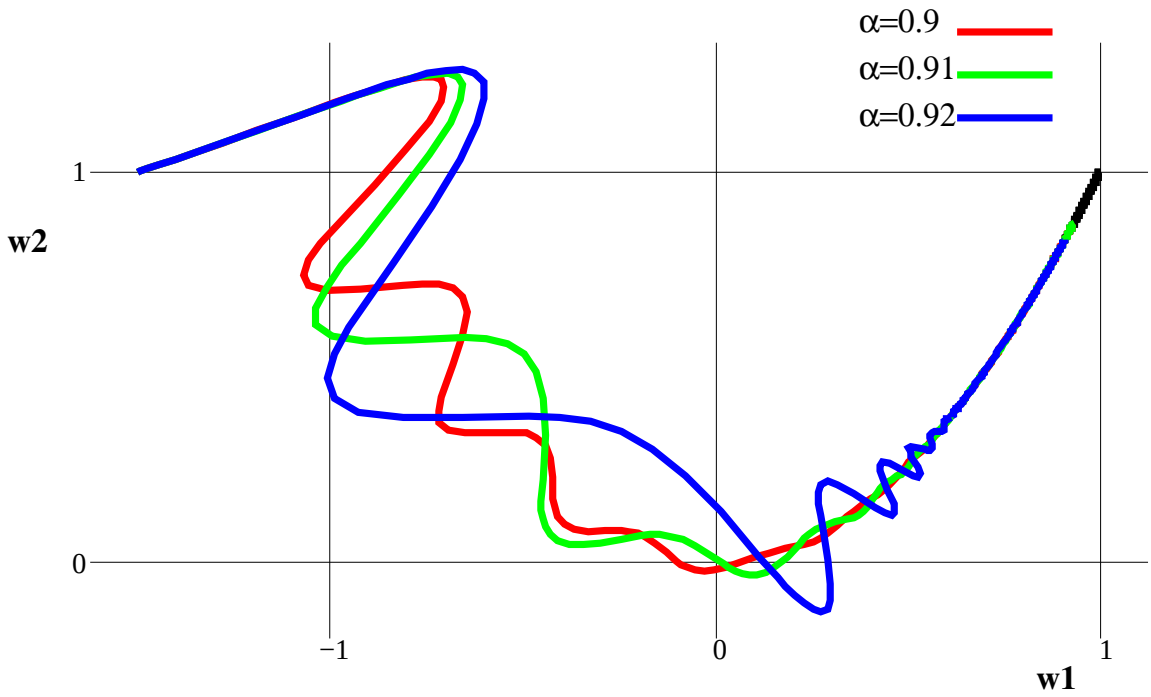


Figure 5.6: This graph illustrates the oscillatory effect of near-optimal values of momentum ($\alpha = 0.9, 0.91, 0.92$) on gradient descent with momentum optimization of Rosenbrock's function. Lower values of α would supress the oscillations, but convergence would be slower. Higher values of momentum would lead to even wilder oscillations, enough to slow or even halt convergence, or to escape from the basin of attraction.

time, $O(n)$-space technique for multiplying an arbitrary vector by the inverse Hessian. The same order of time that would be required even if the inverse Hessian itself were available for free, except that just storing the inverse Hessian requires $O(n^2)$-space.

5.6 Summary and Conclusion

Second-order information about the error is of great practical and theoretical importance. It allows sophisticated optimization techniques to be applied, appears in many theories of generalization, and is used in sophisticated weight pruning procedures. Unfortunately, the Hessian matrix $\mathbf{H}$, whose elements are the second derivative terms $\partial^2 E / \partial w_i \partial w_j$, is unwieldy. We have derived the $\mathcal{R}\{\cdot\}$ technique, which directly computes $\mathbf{Hv}$, the product of the Hessian with a vector. The technique is

- *exact:* no approximations are made.

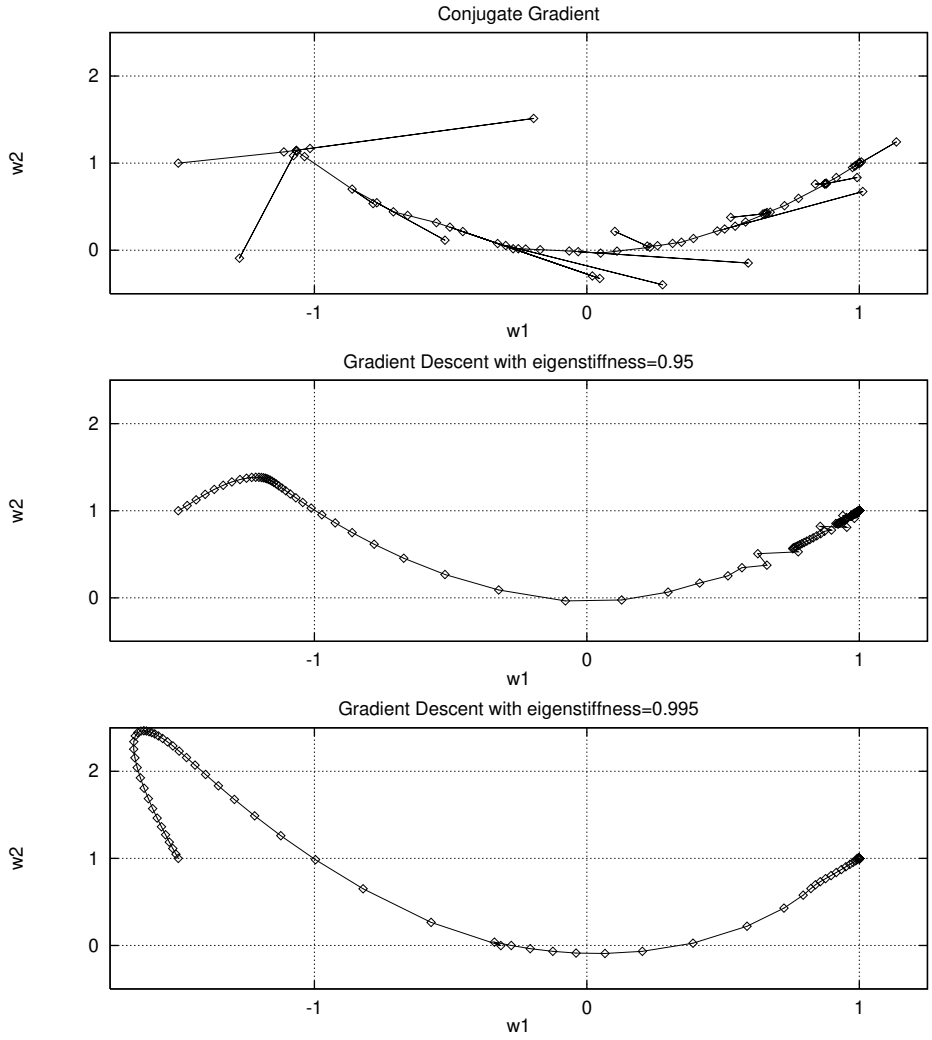


Figure 5.7: The weight-space trajectory of a standard tuned conjugate gradient technique *vs.* gradient descent with automatic stiffening of the principal eigenspace on Rosenbrock's function. Top panel: conjugate gradient technique. Bottom two panels: gradient descent with automatic stiffening of the principal eigenspace. No momentum was used. The global learning rate η was adjusted automatically, by a standard variant of the delta-bar-delta rule. The learning rate η was initialized with a value five orders of magnitude below the optimal, which accounts for the slow startup times. Each point in the graphs represents a gradient evaluation. Note that more evaluations are made by conjugate gradient in the critical regime along the bottom curve of the ravine. This regime is typical of most of the time spent in optimization of backpropagation networks.

- *numerically accurate:* there is no drastic loss of precision.

- *efficient:* it takes about the same amount of computation as a gradient calculation.
- *flexible:* it applies to all existing gradient calculation procedures.
- *robust:* if the gradient calculation gives an unbiased estimate of $\nabla_{\mathbf{w}}$, then our procedure gives an analogous unbiased estimate of $\mathbf{Hv}$.

Procedures that result from the applications of the $\mathcal{R}\{\cdot\}$ technique are about as local, parallel, and efficient as the original untransformed gradient calculation. The technique applies naturally to backpropagation networks, recurrent networks, relaxation networks, Boltzmann Machines, and perturbative methods. This new class of algorithms for efficiently multiplying vectors by the Hessian can facilitate the construction of stochastic optimization and pruning algorithms that make explicit use of second-order structure, but are nonetheless efficient in both space and time.

Chapter 6

Final Words

6.1 Potential Applications: Recurrent and Temporally Continuous Networks

In this thesis we have only touched on a few of many possible applications of the theoretical results and algorithms. Most chapters contain their own sections discussing potential applications and directions for future research. We will not repeat that material here. Instead, we will tie up, or at least describe, some loose ends. In particular, this chapter will discuss some potential application for the networks described in chapter 3 in control domains, or using control architectures applied to other domains.

These musings will concentrate on temporally continuous domains, because the networks of chapter 3 are temporally continuous, which makes it difficult for them to represent clocked phenomena. This can be thought of as a bias in the *a priori* expectations of the learning algorithm, and insofar as this bias is correct it should speed learning and enhance generalization. For an example of this effect, Williams and Zipser report difficulty in training a clocked recurrent network to exhibit sinusoidal motion without teacher forcing (Williams and Zipser, 1988). In contrast, an otherwise architecturally similar, but temporally continuous, network was easily trained to exhibit sinusoidal motion (see chapter 3).

These applications are different in character from the straightforward way in which neural networks are typically applied to real-world domains, in that they involve using the basic network architecture as components of larger systems, rather than as black boxes that solve a problem in isolation.

6.1.1 Generative Continuous Backpropagation

Generative techniques (Levin, 1991) are a way to run a connectionist network backwards, generating inputs that are locally optimal for producing the observed output.

It is analogous to the way a hidden Markov model is run in performance mode to produce the input that most probably produced the observed output, and philosophically in the vein of the motor theory of speech recognition (Lane, 1965; Studdert-Kennedy, Liberman, Harris, and Cooper, 1970; Mattingly and Studdert-Kennedy, 1991).

Consider a network of the sort discussed in chapter 3. The network has initial states $y_i(t_0)$ and driving inputs $I_i(t)$. Let the set of cases we wish to teach the network be indexed by α, where each case consists of initial values and driving inputs for a subset of the nodes in the network, along with desired outputs for some of the nodes. Further, let the inputs for all the cases be divided into two disjoint sets, A and B, where the inputs to the units in case A are given by the task, but the inputs from set B are permitted to change as learning occurs.

Learning a generative model involves doing gradient descent not only in the internal parameters of the network, but also on I_i^α and $y_i^\alpha(t_0)$ for $i \in B$. But I_i^α is a function, so in order to use gradient descent to optimize it we use a variational expression,

$$\Delta I_i = -\epsilon \frac{\delta E}{\delta I_i}.$$

If we follow this naive procedure, we would expect the learning to use the driving inputs to pull the units up and down quickly in order to perform the task, making the magnitudes of both I_i and its derivative large. (This is because optimal control under these circumstances is bang-bang control, and therefore this holds regardless of the relative dimensionalities of the input and output.) In order to prevent the network from encapsulating so much information in I_i it is natural to use a regularization term, so we add a term like

$$\int \sum_{i \in B} \left(\frac{dI_i(t)}{dt} \right)^2 dt$$

to the original E. The precise form of this term depends on the task, and details on appropriate regularizers are readily found in the computer vision literature. (In general, it would seem appropriate for the regularizer to penalize a function I according to how much information it carries.) A comprehensive bibliography is available from Szeliski (1988).

In performance mode, we have only the output and neither the A nor the B components of the input. Therefore, we do gradient descent on all the inputs while holding the network's internal parameters constant. Because we are doing gradient descent on the A portion of the inputs, we will require an additional regularization term to prevent these from becoming too choppy. In general, the regularization term for the A subset of the inputs need bear no relationship to that for the B subset.

6.1.2 Signal Processing and Sensor Integration

Signal processing is a very natural domain from an architectural viewpoint; a network receives inputs from sensors and its outputs are trained to be the values of the underlying variables. Gradient descent is used to optimize the network's transfer function into a (locally) optimal filter. The catch is that the values of the underlying variables must be known in order to train a network to generate them. The straightforward way to accomplish this is to simply measure them with accurate sensors, as in figure 6.1.

This same architecture can also do sensor integration, since sensor integration can simply be regarded as having more inputs, each of which is related to the desired output in a more subtle and noisy fashion.

There are two problems with this architecture. First, the underlying variables are not always known. Second, there is a distinction between training mode and performance mode, since there would be no point in using such a network if the more accurate sensors were always available. As new sorts of sensors became available, or as the sensors in use degraded, costly off-line retraining would be required.

One approach to overcoming these problems is to use a different set of target values for the network. The reason that the underlying variables are important is presumably that they contain all the information relevent to the future and none of the transient noise which will soon dissipate. In order to learn underlying variables, we can require the network to predict its input values for some time in the future, as in figure 6.2.

The major free variable in this architecture is the amount of time in the future that the network is to predict. This value should be long enough that any transient sensor noise has washed out of the signal, but short enough that even the smallest actual signal in the input is relevent. Of course, without detailed knowledge of the sensor and process characteristics, it is impossible to determine if these constraints can be met simultaniously.

Rather than using a fixed prediction period, we could have the network predict the sensor readings at a number of times in the future, at the expense of more computation resulting from a larger network.

If the nonlinear networks of figure 6.2 are replaced by simple linear networks, then the result of the optimization process is a factor analysis—which could be computed directly using standard techniques. It might be a good idea to use factor analysis to extract all the available linear structure, and preloading the network with initial weights encapsulating this information, before attempting to discover nonlinear structure in the time series.

The squared prediction error being minimized in either the linear or nonlinear predictors discussed above can be viewed as giving lower bounds on the mutual information given by the structure extracted from the history about the future. It

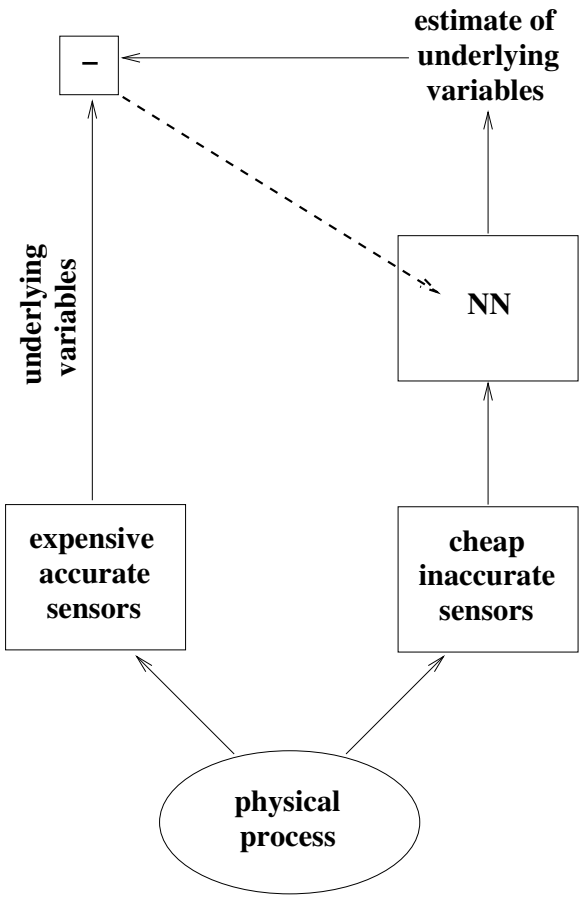


Figure 6.1: A network being trained to perform signal processing. The dashed line indicates propagation of error signals, used during training. The network's inputs are readings from inexpensive but inaccurate sensors. The output is trained to estimate the corresponding readings from expensive sensors, which are available at training time, but not in the field. An instance where this would be appropriate is in monitoring a wastewater treatment plant, where variables like opacity, conductivity, and pH are readily available, but the underlying variables, like bacteria concentration, biological oxygen levels, and the like, are quite expensive to monitor, and hence are available only in pilot or research plants.

would therefore be tempting to extend this approach, to enable very high level features to be extracted, by instead learning to extract features from the time series that act as low-dimensional witnesses to mutual information between the past and the future, as encapsulated by these features themselves having high entropy but also being predicatable (having high conditional information from) both the past and the future. Architectures of this sort have been explored both in the temporal domain (Schmidhuber, 1992c, 1992a) and in the spatial domain (Becker and Hinton, 1992,

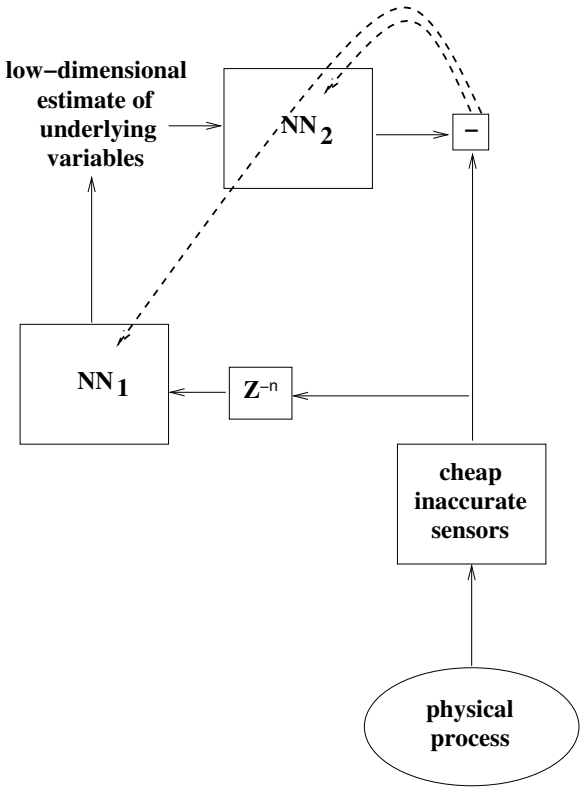


Figure 6.2: A network being trained to perform signal processing. Dashed lines indicate propagation of error signals, used during training. The network's task is to use current sensor readings to produce a low-dimensional description of the current state, where this description is valuable for predicting future sensor reading. The value of the description for predicting future sensor readings (the Z^{-n} indicates a delay of n time steps) is assayed by training a second network to use the reduced description for prediction.

1993).

6.2 Potential Applications: Fast Exact Multiplication by the Hessian

The $R\{\cdot\}$ family of techniques described in chapter 5 allows the calculation of the product of the Hessian with a vector in about the same amount of time as a gradient calculation. The gradient gives the slope of the error surface at a point, but the combination of the gradient and the product Hv predicts the values of the gradient along the line spanned by v. As described in the chapter, there are many potential

applications of this technique, ranging from diagnosis, the simple characterization of the local properties of the error surface, to faster optimization techniques, to application to predicting and enhancing generalization. Preliminary experiments seem to indicate that faster optimization will not be easily achieved using this family of techniques. However, applications to generalization are being explored by a number of groups, with apparent success.

Bibliography

AANN*90 (1990). *Applications of Artificial Neural Networks*, No. 1294 in APIE Proceedings Series, Orlando, Florida.

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for Boltzmann Machines. *Cognitive Science*, *9*, 147–169.

Albesano, D., Gemello, R., and Mana, F. (1992). Word Recognition with Recurrent Network Automata. In IJCNN92'Baltimore (1992), pp. 308–313.

Alexander, S. T. (1986). *Adaptive Signal Processing*. Springer-Verlag.

Allen, R. B. and Alspector, J. (1989). Learning of stable states in stochastic asymmetric networks. Tech. rep. TM-ARH-015240, Bell Communications Research, Morristown, NJ.

Almeida, L. B. (1987). A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In Caudill and Butler (1987), pp. 609–618.

Alspector, J., Meir, R., Yuhas, B., and Jayakumar, A. (1993). A Parallel Gradient Descent Method for Learning in Analog VLSI Neural Networks. In NIPS*92 (1993), pp. 836–844.

An, C. H., Atkeson, C. G., and Hollerbach, J. M. (1988a). Model-Based Control of a DD Arm, Part I: Building Models. In *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*, pp. 1374–1379 Philadelphia. IEEE Press.

An, C. H., Atkeson, C. G., and Hollerbach, J. M. (1988b). *Model-Based Control of a Robot Manipulator*. Artificial Intelligence. MIT Press.

Anderson, B. D. O. and Moore, J. B. (1979). *Optimal Filtering*. Prentice-Hall.

Angluin, D. (1987). Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, *75*, 87–106.

Atiya, A. F. (1988). Learning on a General Network. In NIPS*87 (1988), pp. 22–30.

Atkeson, C. G. and Reinkensmeyer, D. J. (1988). Using Associative Content-Addressable Memories to Control Robots. In *IEEE Conference on Decision and Control*, pp. 792–797 Austin, TX.

Atkeson, C. G. and Reinkensmeyer, D. J. (1990). *Using Associative Content-Addressable Memories to Control Robots*, chap. 11, pp. 254–285. In Miller *et al.*(1990).

Back, A. and Tsoi, A. (1991). FIR and IIR Synapses, a New Neural Network Architecture for Time Series Modelling. *Neural Computation*, *3*(3), 337–350.

Baird, B. (1990). A Learning Rule for CAM Storage of Continuous Periodic Sequences. In IJCNN90 II (1990), pp. 493–498.

Baird, B. and Eeckman, F. (1991). CAM Storage of Analog Patterns and Continuous Sequences with $3N^2$ Weights. In NIPS*90 (1991), pp. 91–97.

Baldi, P. and Pineda, F. (1991). Contrastive Learning and Neural Oscillations. *Neural Computation*, *3*(4), 526–545.

Ballard, D. H. and Brown, C. M. (1982). *Computer Vision*. Printice-Hall.

Becker, S. and Hinton, G. (1992). A self-organizing neural network that discovers surfaces in random-dot stereograms. *Nature*, *355*, 161–163.

Becker, S. and Hinton, G. (1993). Learning Mixture Models of Spatial Coherence. *Neural Computation*, *5*(2), 267–277.

Becker, S. and LeCun, Y. (1989). Improving the Convergence of Back-Propagation Learning with Second Order Methods. In CMSS-88 (1989), pp. 29–37. Also published as Technical Report CRG-TR-88-5, Department of Computer Science, University of Toronto.

Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press.

Bellman, R. E. (1973). *Methods of Nonlinear Analysis: Volume II*. Academic Press.

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

Bishop, C. (1992). Exact Calculation of the Hessian Matrix for the Multilayer Perceptron. *Neural Computation*, *4*(4), 494–501.

Blom, J. G., Sanz-Serna, J. M., and Verwer, J. G. (1986). *On Simple Moving Grid Methods for One-Dimensional Evolutionary Partial Differential Equations*. Stichting Mathematisch Centrum, Amsterdam, The Netherlands.

Bodenhausen, U. (1990). Learning Internal Representations of Pattern Sequences in a Neural Network with Adaptive Time-Delays. In IJCNN90 II (1990).

Brady, M., Hollerbach, J., Johnson, T., Lozano-Perez, T., and Mason, M. (1982). *Robot Motion: Planning and Control*. MIT Press.

Bryson, Jr., A. E. (1962). A Steepest Ascent Method for solving Optimum Programming Problems. *Journal of Applied Mechanics*, *29*(2), 247.

Buntine, W. L. and Weigend, A. S. (1994). Computing Second Derivatives in Feed-Forward Networks: a Review. *IEEE Transactions on Neural Networks*, *5*, 480–488.

Caudill, M. and Butler, C. (Eds.)., ICNN87 (1987). *IEEE First International Conference on Neural Networks*, San Diego, CA.

Cauwenberghs, G. (1993). A Fast Stochastic Error-Descent Algorithm for Supervised Learning and Optimization. In NIPS*92 (1993), pp. 244–251.

Chauvin, Y. and Rumelhart, D. E. (Eds.). (1995). *Back-propagation: Theory, Architectures and Applications*. Lawrence Erlbaum Associates.

Christianson, B. (1992). Automatic Hessians by Reverse Accumulation. *IMA Journal of Numerical Analysis*, *12*, 135–150.

Cleeremans, A., Servan-Schreiber, D., and McClelland, J. (1989). Finite State Automata and Simple Recurrent Networks. *Neural Computation*, *1*(3), 372–381.

CMSS-88 (1989). *Proceedings of the 1988 Connectionist Models Summer School*. Morgan Kaufmann.

Cohen, M. A. and Grossberg, S. (1983). Stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE Transactions on Systems, Man and Cybernetics*, *13*, 815–826.

Cottrell, G. W. and Small, S. (1983). A connectionist scheme for modelling word sense disambiguation. *Cognition and Brain Theory*, *6*, 89–120.

Craig, J. J. (1986). *Introduction to Robotics: Mechanics and Control*. Addison-Wesley.

Crutchfield, J. P. and McNamara, B. S. (1987). Equations of motion from a data series. *Complex Systems*, *1*, 417–452.

Dabis, H. S. and Moir, T. J. (1991). Least mean squares as a control system. *International Journal of Control*, *54*(2), 321–335.

Das, S., Giles, C. L., and Sun, G.-Z. (1993). Using Prior Knowledge in a NNPDA to Learn Context-Free Languages. In NIPS*92 (1993), pp. 65–72.

Das, S. and Mozer, M. C. (1994). A Unified Gradient-Descent/Clustering Architecture for Finite State Machine Induction. In NIPS*93 (1994).

Davis, L. S. and Rosenfeld, A. (1981). Cooperating processes for low-level vision: A survey. *Artificial Intelligence*, *3*, 245–264.

Day, S. P. and Davenport, M. R. (1993). Continuous-Time Temporal Backpropagation with Adaptable Time Delays. *IEEE Transactions on Neural Networks*, *4*(2), 348–354. Ftp archive.cis.ohio-state.edu: /pub/neuroprose/day.temporal.ps.Z.

de Vries, B. and Principe, J. (1992). The Gamma Model—a New Neural Network for Temporal Processing. *Neural Networks*, *5*(4), 565–576.

de Vries, B. and Principe, J. C. (1991). A Theory for Neural Networks with Time Delays. In NIPS*90 (1991), pp. 162–168.

Dennis, Jr., J. E. and Schnabel, R. B. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall.

Doya, K., Boyle, M. E. T., and Selverston, A. I. (1993). Mapping between Neural and Physical Activities of the Lobster Gastric Mill System. In NIPS*92 (1993), pp. 913–920.

Doya, K., Selverston, A. I., and Rowat, P. F. (1994). A Hodgkin-Huxley Type Neuron Model that Learns Slow Non-Spike Oscillation. In NIPS*93 (1994).

Dreyfus, S. E. (1965). *Dynamic Programming and the Calculus of Variations*, Vol. 21 of *Mathematics in science and engineering*. Academic Press.

Du, L.-M., Hou, Z.-Q., and Li, Q.-H. (1992). Optimum Block-Adaptive Learning Algorithm for Error Back-Propagation. *IEEE Transactions on Signal Processing*, *40*(12), 3032–3042.

Durbin, R. and Rumelhart, D. E. (1989). Product Units: A Computationally Powerful and Biologically Plausible Extension to backpropagation Networks. *Complex Systems*, *1*, 133.

Elman, J. L. (1988). Finding Structure in Time. Tech. rep. CRL-8801, Center for Research in Language, UCSD.

Elman, J. (1990). Finding Structure in Time. *Cognitive Science*, *14*, 179–211.

Fahlman, S. E. (1988). An Emphirical Study of Learning Speed in Back-Propagation Networks. Tech. rep. CMU-CS-88-162, Carnegie Mellon University School of Computer Science, Pittsburgh, PA.

Fang, Y. and Sejnowski, T. J. (1990). Faster Learning for Dynamic Recurrent Backpropagation. *Neural Computation*, *2*(3), 270–273.

Flower, B. and Jabri, M. (1993). Summed Weight Neuron Perturbation: An $O(n)$ Improvement over Weight Perturbation. In NIPS*92 (1993), pp. 212–219.

Freeman, W. J. (1987). Simulation of Chaotic EEG Patterns with a Dynamic Model of the Olfactory System. *Biological Cybernetics*, *56*, 139.

Galland, C. C. and Hinton, G. E. (1989). Deterministic Boltzmann Learning in Networks With Asymmetric Connectivity. Tech. rep. CRG-TR-89-6, University of Toronto Department of Computer Science.

Gelb, A. *et al.* (Eds.). (1974). *Applied Optimal Estimation*. MIT Press.

Gherrity, M. (1989). A Learning Algorithm for Analog, Fully Recurrent Neural Networks. In IJCNN89 (1989), pp. 643–644.

Giles, C. L., Miller, C. B., Chen, D., Sun, G. Z., Chen, H. H., and Lee, Y. C. (1992). Extracting and Learning an Unknown Grammar with Recurrent Neural Networks. In NIPS*91 (1992), pp. 317–324.

Gluck, M. A., Glauthier, P. T., and Sutton, R. S. (1992). Adaptation of Cue-Specific Learning Rates in Network Models of Human Category Learning. In *Proceedings*

of the Fourteenth Annual Meeting of the Cognitive Science Society Bloomington, IN.

Goldberg, K. Y. and Pearlmutter, B. A. (1988). Using a Neural Network to Learn the Dynamics of the CMU Direct-Drive Arm II. Tech. rep. CMU-CS-88-160, Carnegie Mellon University.

Goldberg, K. Y. and Pearlmutter, B. A. (1989). Using a Neural Network to Learn the Dynamics of the CMU Direct-Drive Arm II. In NIPS*88 (1989), pp. 356–363.

Gori, M., Bengio, Y., and de Mori, R. (1989). BPS: A Learning Algorithm for Capturing the Dynamic Nature of Speech. In IJCNN89 (1989), pp. 417–423.

Gorman, P. R. and Sejnowski, T. J. (1988). Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets. *Neural Networks*, *1*(1), 75–89.

Grossman, T., Meir, R., and Domany, E. (1989). Learning by Choice of Internal Representations. *Complex Systems*, *2*, 555–575.

Hassibi, B. and Stork, D. G. (1993). Second Order Derivatives for Network Pruning: Optimal Brain Surgeon. In NIPS*92 (1993), pp. 164–171.

Hihi, S. E. and Bengio, Y. (1996). Hierarchical recurrent neural networks for long-term dependencies. In *Advances in Neural Information Processing Systems 8*. MIT Press. To appear.

Hinton, G. E. (1976). Using relaxation to find a puppet. In *Proceedings of the A.I.S.B. Summer Conference* University of Edinburgh.

Hinton, G. E. (1977). *Relaxation and its role in vision.* Ph.D. thesis, University of Edinburgh. Described in (Ballard and Brown, 1982, pages 408–430).

Hinton, G. E. (1986). Learning Distributed Representations of Concepts. In *Proceedings of the Eighth Annual Cognitive Science Conference*. Lawrence Erlbaum Associates.

Hinton, G. E. (1987). Connectionist Learning Procedures. Tech. rep. CMU-CS-87-115, Carnegie Mellon University, Pittsburgh, PA 15213. Also published in the AI Journal.

Hinton, G. E. (1989). Deterministic Boltzmann learning performs steepest descent in weight-space. *Neural Computation*, *1*(1), 143–150.

Hinton, G. E. and Lang, K. J. (1985). Shape recognition and illusory conjunctions. In *the Ninth International Joint Conference on Artificial Intelligence*, Vol. 1, pp. 252–259 Los Angeles. Morgan Kaufmann.

Hinton, G. E. and Sejnowski, T. J. (1983). Optimal Perceptual Inference. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 448–453 Washington DC. IEEE Computer Society.

Hochreiter, J. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diplomarbeit, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

Hochreiter, S. and Schmidhuber, J. H. (1995). Flat Minimum Search Finds Simple Nets. In NIPS*94 (1995). Ftp informatik.tu-muenchen.de:/pub/fki/fki-200-94.ps.gz.

Hollerbach, J. M. (1980). A Recursive Lagrangian Formulation of Manipulator Dynamics and a Comparitive Study of Dynamics Formulation Complexity. *IEEE Transactions on Systems, Man and Cybernetics*, *10*(11), 730–736.

Hopfield, J. J. and Tank, D. W. (1985). 'Neural' Computation of Decisions in Optimization Problems. *Biological Cybernetics*, *52*, 141–152.

Howard, R. (1960). *Dynamic Programming and Karkhov Processes*. MIT Press.

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

IJCNN89 (1989). *International Joint Conference on Neural Networks*, Washington DC. IEEE Press.

IJCNN90 II (1990). *International Joint Conference on Neural Networks*, San Diego, CA. IEEE Press.

IJCNN92'Baltimore (1992). *International Joint Conference on Neural Networks*, Baltimore, MD. IEEE Press.

Jabri, M. and Flower, B. (1991). Weight Perturbation: An Optimal Architecture and Learning Technique for Analog VLSI Feedforward and Recurrent Multilayer Networks. *Neural Computation*, *3*(4), 546–565.

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural Networks*, *1*(4), 295–307.

Jacobson, D. H. (1968). New Second Order and First Order Algorithm for Determining Optimal Control: A Differential Dynamic Programming Approach. *Journal of Optimization Theory and Applications*, *2*.

Jervis, T. T. and Fitzgerald, W. J. (1993). Optimization Schemes for Neural Networks. Tech. rep. CUED/F-INFENG/TR 144, Cambridge University Engineering Department, Cambridge, England.

Jordan, M. I. (1989). Generic Constraints on Underspecified Target Trajectories. In IJCNN89 (1989).

Jordan, M. I. and Jacobs, R. A. (1990). Learning to Control an Unstable System with Forward Modeling. In NIPS*89 (1990).

Jordan, M. I. (1986). Attractor Dynamics and Parallelism in a Connectionist Sequential Machine. In *Proceedings of Ninth Annual Conference of the Cognitive Science Society*, pp. 531–546. Lawrence Erlbaum Associates.

Juang, B., Kung, S., and Camm, C. A. (Eds.). (1991). *Neural Networks for Signal Processing: Proceedings of the 1991 IEEE Workshop*. IEEE Press.

Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems. *Trans. ASME Journal of Basic Engineering*, *82*(1), 35–45.

Karjala, T. W., Himmelblau, D. M., and Miikkulainen, R. (1992). Data Rectification using Recurrent (Elman) Neural Networks. In IJCNN92'Baltimore (1992), pp. 901–905.

Kawato, M., Setoyama, T., and Suzuki, R. (1988). Feedback Error Learning of Movement by Multi-Layer Neural Networks. In *Proceedings of the International Neural Networks Society First Annual Meeting*, p. 342.

Kilian, J. and Siegelmann, H. T. (1993). Computability With The Classical Sigmoid. In *Sixth Annual ACM Workshop on Computational Learning Theory*, pp. 137–143 Santa Cruz, CA.

Kim, K. V., Nesterov, Y. E., and Cherkassky, B. V. (1985). An algorithm for fast differentiations and its applications. In *Abstracts of the 12th IFIP Conference on System Modeling and Optimization*, pp. 181–182 Budapest, Hungary.

Kirk, D. B., Kerns, D., Fleischer, K., and Barr, A. H. (1993). Analog VLSI Implementation of Gradient Descent. In NIPS*92 (1993), pp. 789–796.

Kirkpatrick, S., Gelatt, Jr., C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*, 671–680.

Kolen, J. F. (1994). Fool's Gold: Extracting Finite State Machines from Recurrent Network Dynamics. In NIPS*93 (1994). Ftp archive.cis.ohio-state.edu: /pub/neuroprose/kolen.foolsgold.ps.Z.

Kuhn, G. (1987). A First Look at Phonetic Discrimination using Connectionist Models with Recurrent Links. Scimp working paper 82018, Institute for Defense Analysis, Princeton, New Jersey.

Kuperstein, M. and Wang, J. (1990). Neural Controller for Adaptive Movements with Unforseen Payloads. *IEEE Transactions on Neural Networks*, *1*(1), 137–142.

Lane, H. L. (1965). The motor theory of speech perception: A critical review. *Psychology Review*, *72*, 275–309.

Lang, K. and Hinton, G. (1988). The Development of the Time-Delay Neural Network Architecture for Speech Recognition. Tech. rep. CMU-CS-88-152, Department of Computer Science, Carnegie Mellon University.

Lang, K. J. (1992). Random DFA's can be Approximately Learned from Sparse Uniform Examples. In *Fifth Annual ACM Workshop on Computational Learning Theory*, pp. 45–52 Pittsburgh, PA.

Lang, K. J. and Hinton, G. E. (1990). Dimensionality Reduction and Prior Knowledge in E-set Recognition. In NIPS*89 (1990), pp. 178–185.

Lang, K. J., Hinton, G. E., and Waibel, A. (1990). A Time-Delay Neural Network Architecture for Isolated Word Recognition. *Neural Networks*, *3*(1), 23–43.

Lapedes, A. and Farber, R. (1987). Nonlinear Signal Processing Using Neural Networks: Prediction and System Modelling. Tech. rep. LA-UR-87-2662, Theoretical Division, Los Alamos National Laboratory, Los Alamos, NM.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, *1*(4), 541–551.

LeCun, Y. (1985). Une Procédure d'Apprentissage pour Réseau à Seuil Assymétrique. In *Cognitiva 85: A la Frontière de l'Intelligence Artificielle des Sciences de la Connaissance des Neurosciences*, pp. 599–604 Paris 1985. CESTA, Paris.

LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal Brain Damage. In NIPS*89 (1990), pp. 598–605.

LeCun, Y., Kanter, I., and Solla, S. A. (1991). Second Order Properties of Error Surfaces: Learning Time and Generalization. In NIPS*90 (1991), pp. 918–924.

LeCun, Y., Simard, P. Y., and Pearlmutter, B. A. (1993). Automatic Learning Rate Maximization by On-Line Estimation of the Hessian's Eigenvectors. In NIPS*92 (1993), pp. 156–163.

Levin, E. (1991). Modeling Time Varying Systems Using Hidden Control Neural Architecture. In NIPS*90 (1991), pp. 147–154.

Lockery, S. R., Fang, Y., and Sejnowski, T. J. (1990). A Dynamic Neural Network Model of Sensorimotor Transformations in the Leech. *Neural Computation*, *2*(3), 274–282.

Lockery, S. R. and Kristan, Jr., W. B. (1990a). Distributed Processing of Sensory Information in the Leech I: Input-Output Relations of the Local Bending Relex. *Journal of Neuroscience*, *10*(6), 1811–1815.

Lockery, S. R. and Kristan, Jr., W. B. (1990b). Distributed Processing of Sensory Information in the Leech II: Identification of Interneurons Contributing to the Local Bending Reflex. *Journal of Neuroscience*, *10*(6), 1816–1829.

Lockery, S. R. and Sejnowski, T. J. (1993). The Computational Leech. *Trends in Neuroscience*, *16*(7), 283–290.

Lockery, S. R., Wittenberg, G., Kristan, Jr., W. B., Qian, N., and Sejnowski, T. J. (1990). Neural network analysis of distributed representations of sensory information in the leech. In NIPS*89 (1990), pp. 28–35.

MacKay, D. J. C. (1991). A Practical Bayesian Framework for Back-Prop Networks. *Neural Computation*, *4*(3), 448–472.

Mahra, R. K. (1970). On the Identification of Variances and Adaptive Kalman Filtering. *IEEE Transactions on Automatic Control*, *AC-15*(2), 175–184.

Marr, D. and Poggio, T. (1976). Cooperative computation of stereo disparity. *Science*, *194*, 283–287.

Marr, D. Palm, G. and Poggio, T. (1978). Analysis of a cooperative stereo algorithm. *Biological Cybernetics*, *28*, 223–229.

Matthews, M. B. (1990). Neural network nonlinear adaptive filtering using the extended Kalman filter algorithm. In *Proceedings of the International Neural Networks Conference*, Vol. 1, pp. 115–119 Paris, France.

Mattingly, I. G. and Studdert-Kennedy, M. (Eds.). (1991). *Modularity and the Motor Theory of Speech Perception: Proceedings of a Conference to Honor Alvin M. Liberman*. Lawrence Erlbaum Associates.

Maxwell, T., Giles, C. L., Lee, Y. C., and Chen, H. H. (1986). Nonlinear Dynamics of Artificial Neural Systems. In Snowbird-86 (1986), pp. 299–304.

McClelland, J. L., Rumelhart, D. E., and Hinton, G. E. (1986). The appeal of parallel distributed processing. In Rumelhart *et al.*(1986a).

Mel, B. W. and Koch, C. (1990). Sigma-Pi Learning: On Radial Basis Functions and Cortical Associative Learning. In NIPS*89 (1990), pp. 474–481.

Miller, III, W. T., Sutton, R. S., and Werbos, P. J. (Eds.). (1990). *Neural Networks for Control*. MIT Press.

Møller, M. (1993a). Exact Calculation of the Product of the Hessian Matrix of Feed-Forward Network Error Functions and a Vector in $O(n)$ Time. Daimi PB-432, Computer Science Department, Aarhus University, Denmark.

Møller, M. (1993b). A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning. *Neural Networks*, *6*(4), 525–533.

Møller, M. (1993c). Supervised Learning on Large Redundant Training Sets. *International Journal of Neural Systems*, *4*(1), 15–25.

Moody, J. E. (1992). The *Effective* Number of Parameters: an Analysis of Generalization and Regularization in Nonlinear Learning Systems. In NIPS*91 (1992), pp. 847–854.

Movellan, J. R. and McClelland, J. L. (1991). Learning Continuous Probability Distributions with the Contrastive Hebbian Algorithm. Tech. rep. PDP.CNS.91.2, Carnegie Mellon University Dept. of Psychology, Pittsburgh, PA.

Mozer, M. C. (1989). A Focused Backpropagation Algorithm for Temporal Pattern Recognition. *Complex Systems*, *3*(4), 349–381.

Mozer, M. C. (1992). Induction of Multiscale Temporal Structure. In NIPS*91 (1992), pp. 275–282.

Mozer, M. C. and Das, S. (1993). A Connectionist Symbol Manipulator that Discovers the Structure of Context-Free Languages. In NIPS*92 (1993), pp. 863–870.

Narendra, K. S. and Parthasarathy, K. (1991). Gradient Methods for the Optimization of Dynamical Systems Containing Neural Networks. *IEEE Transactions on Neural Networks*, *2*(2), 252–262.

Narendra, K. and Parthasarathy, K. (1990). Identification and Control of Dynamical Systems Using Neural Networks. *IEEE Transactions on Neural Networks*, *1*, 4–27.

Nerrand, O., Roussel-Ragot, P., L. Personnaz, G. D., and Marcos, S. (1993). Neural Networks and Non-linear Adaptive Filtering: Unifying Concepts and New Algorithms. *Neural Computation*, *5*(2), 165–197.

Nguyen, M. and Cottrell, G. (1993). A technique for adapting to speech rate. In Kamm, C., Kuhn, G., Yoon, B., Chellappa, R., and Kung, S. (Eds.), *Neural Networks for Signal Processing 3*. IEEE Press.

NIPS*87 (1988). *Neural Information Processing Systems*, New York, New York. American Institute of Physics.

NIPS*88 (1989). *Advances in Neural Information Processing Systems I*. Morgan Kaufmann.

NIPS*89 (1990). *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann.

NIPS*90 (1991). *Advances in Neural Information Processing Systems 3*. Morgan Kaufmann.

NIPS*91 (1992). *Advances in Neural Information Processing Systems 4*. Morgan Kaufmann.

NIPS*92 (1993). *Advances in Neural Information Processing Systems 5*. Morgan Kaufmann.

NIPS*93 (1994). *Advances in Neural Information Processing Systems 6*. Morgan Kaufmann.

NIPS*94 (1995). *Advances in Neural Information Processing Systems 7*. MIT Press.

Norris, B. J., Coleman, M. J., and Nusbaum, M. P. (1994). Distinct responses of electrically-coupled pacemaker neurons to activation of a modulatory projection neuron. *Society for Neuroscience Abstracts*, *20*(18.6), 23.

Nowlan, S. J. and Hinton, G. E. (1992). Adaptive Soft Weight Tying using Gaussian Mixtures. In NIPS*91 (1992), pp. 993–1000.

Nowlan, S. J. (1988). Gain Variation in Recurrent Error Propagation Networks. *Complex Systems*, *2*(3), 305–320.

Ottaway, M. B., Simard, P. Y., and Ballard, D. H. (1989). Fixed Point Analysis for Recurrent Neural Networks. In NIPS*88 (1989).

Parker, D. B. (1985). Learning-Logic. Tech. rep. TR-47, MIT Center for Research in Computational Economics and Management Science.

Parker, D. B. (1987). Optimal Algorithms for Adaptive Networks: Second Order Back Propagation, Second Order Direct Propagation, and Second Order Hebbian Learning. In Caudill and Butler (1987), pp. 593–600.

Pearlmutter, B. (1989a). Learning state space trajectories in recurrent neural networks. *Neural Computation*, *1*(2), 263–269.

Pearlmutter, B. A. (1989b). Learning State Space Trajectories in Recurrent Neural Networks. In CMSS-88 (1989), pp. 113–117.

Pearlmutter, B. A. (1990a). Dynamic recurrent neural networks. Tech. rep. CMU-CS-90-196, Carnegie Mellon University School of Computer Science, Pittsburgh, PA.

Pearlmutter, B. A. (1990b). Two New Learning Procedures for Recurrent Networks. *Neural Network Review*, *3*(3), 99–101.

Pearlmutter, B. A. (1992a). Gradient Descent: Second-Order Momentum and Saturating Error. In NIPS*91 (1992), pp. 887–894.

Pearlmutter, B. A. (1992b). Temporally Continuous vs. Clocked Networks. In *Neural Networks in Robotics*, pp. 237–252. Kluwer Academic Publishers.

Pearlmutter, B. A. (1994). Fast Exact Multiplication by the Hessian. *Neural Computation*, *6*(1), 147–160.

Pearlmutter, B. A. (1995). Gradient Calculation for Dynamic Recurrent Neural Networks: a Survey. *IEEE Transactions on Neural Networks*, *6*(5), 1212–1228.

Peterson, C. and Anderson, J. R. (1987a). A Mean Field Theory Learning Algorithm for Neural Nets. *Complex Systems*, *1*.

Peterson, C. and Anderson, J. R. (1987b). A Mean Field Theory Learning Algorithm for Neural Networks. Tech. rep. EI-259-87, MCC.

Pineda, F. (1987). Generalization of Back-Propagation to Recurrent Neural Networks. *Physical Review Letters*, *19*(59), 2229–2232.

Poddar, P. and Unnikrishnan, K. (1991). Nonlinear Prediction of Speech Signals Using Memory Neuron Networks. In Juang *et al.*(1991), pp. 395–404.

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Verrerling, W. T. (1988). *Numerical Recipes in C*. Cambridge University Press.

Qian, N. and Sejnowski, T. J. (1989). Learning to solve random-dot stereograms of dense and transparent surfaces with recurrent backpropagation. In CMSS-88 (1989), pp. 435–443.

Raibert, M. H. and Horn, B. K. P. (1978). Manipulator Control using Configuration Space Method. *Industrial Robot*, *5*, 69–73.

Renals, S. and Rohwer, R. (1990). A Study of Network Dynamics. *Journal of Statistical Physics*, *58*, 825–848.

Robinson, A. J. and Fallside, F. (1988). Static and Dynamic Error Propagation Networks with Application to Speech Coding. In NIPS*87 (1988), pp. 632–641.

Rohwer, R. (1990). The "Moving Targets" Training Algorithm. In NIPS*89 (1990), pp. 558–565.

Rowat, P. F. and Selverston, A. I. (1991). Learning Algorithms for Oscillatory Networks with Gap Junctions and Membrane Currents. *Network: Computation in Neural Systems*, *2*(1), 17–42.

Rumelhart, D. E., McClelland, J. L., and the PDP research group. (Eds.). (1986a). *Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1: Foundations*. MIT Press.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b). Learning Internal Representations by Error Propagation. In Rumelhart *et al.*(1986a).

Schmidhuber, J. H. (1992a). Learning Factorial Codes By Predictability Minimization. *Neural Computation*, *4*(6), 863–879.

Schmidhuber, J. H. (1992b). A Fixed Size Storage $O(n^3)$ Time Complexity Learning Algorithm for Fully Recurrent Continually Running Networks. *Neural Computation*, *4*(2), 243–248.

Schmidhuber, J. H. (1992c). Learning Complex, Extended Sequences using the Principle of History Compression. *Neural Computation*, *4*(2), 234–242.

Schmidhuber, J. H. (1992d). Learning Unambiguous Reduced Sequence Descriptions. In NIPS*91 (1992), pp. 291–298.

Sejnowski, T. J. (1986). Higher-Order Boltzmann Machines. In Snowbird-86 (1986), pp. 398–403.

Sejnowski, T., Kienker, P., and Hinton, G. (1986). Learning Symmetry Groups with Hidden Units: Beyond the Perceptron. *Physica D*, *22*, 260–275.

Shynk, J. J. and Roy, S. (1988). The LMS algorithm with momentum updating. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 2651–2654.

Siegelmann, H. T. and Sontag, E. D. (1991). Turing Computability with Neural Networks. *Applied Mathematics Letters*, *4*(6), 77–80.

Siegelmann, H. T. and Sontag, E. D. (1993). Analog Computation via Neural Networks. In *The Second Israel Symposium on Theory of Computing and Systems* Natanya, Israel. To appear in *Theoretical Computer Science*.

Silva, F. M. and Almeida, L. B. (1990). Acceleration Techniques for the Backpropagation Algorithm. In Almeida, L. B. and Wellekens, C. J. (Eds.), *Proceedings of the 1990 EURASIP Workshop on Neural Networks*. Springer-Verlag. (Lecture Notes in Computer Science series).

Simard, P. Y., Rayzs, J. P., and Victorri, B. (1991). Shaping the State Space Landscape in Recurrent Networks. In NIPS*90 (1991), pp. 105–112.

Skarda, C. A. and Freeman, W. J. (1987). How brains make chaos in order to make sense of the world. *Brain and Behavioral Science*, *10*.

Skilling, J. (1989a). The Eigenvalues of Mega-dimensional Matrices. In *Maximum Entropy and Bayesian Methods* (1989b), pp. 455–466.

Skilling, J. (Ed.). (1989b). *Maximum Entropy and Bayesian Methods*. Kluwer Academic Publishers.

Snowbird-86 (1986). *Snowbird Conference on Neural Networks for Computing*, No. 151 in AIP conference proceedings. American Institute of Physics.

Studdert-Kennedy, M., Liberman, A. M., Harris, K. S., and Cooper, F. S. (1970). Motor Theory of Speech Perception: A Reply to Lane's Critical Review. *Psychological Review*, *77*, 234–249.

Sun, G.-Z., Chen, H.-H., and Lee, Y.-C. (1992). Green's Function Method for Fast On-line Learning Algorithm of Recurrent Neural Networks. In NIPS*91 (1992), pp. 333–340.

Sutton, R. S. (1992a). Adapting Bias by Gradient Descent: An Incremental Version of Delta-Bar-Delta. In *Proceedings of the National Conference on Artificial Intelligence AAAI-92*.

Sutton, R. S. (1992b). Gain Adaptation Beats Least Squares?. In *Seventh Yale Workshop on Adaptive and Learning Systems*, pp. 161–166.

Szeliski, R. (1986). Cooperative Algorithms for Solving Random-Dot Stereograms. Tech. rep. CMU-CS-86-133, Carnegie Mellon University.

Szeliski, R. S. (1988). *Bayesian Modeling of Uncertainty in Low-Level Vision*. Ph.D. thesis, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA. Available as technical report CMU-CS-88-169.

Tank, D. W. and Hopfield, J. J. (1987a). Concentrating Information in Time: Analog Neural Networks with Applications to Speech Recognition Problems. In Caudill and Butler (1987), pp. 455–468.

Tank, D. W. and Hopfield, J. J. (1987b). Neural Computation by Time Compression. *Proceedings of the National Academy of Sciences*, *84*, 1896–1900.

Tesauro, G., He, Y., and Ahmad, S. (1989). Asymptotic Convergence of Backpropagation. *Neural Computation*, *1*(3), 382–391.

Tollenaere, T. (1990). SuperSAB: Fast Adaptive Back Propagation with Good Scaling Properties. *Neural Networks*, *3*(5), 561–573.

Touretzky, D. and Pomerleau, D. (1989). What's Hidden in the Hidden Layers?. *BYTE*, 227–233.

Tsung, F.-S. and Cottrell, G. (1995). Phase space learning. In NIPS*94 (1995).

Tuğay, M. A. and Tanik, Y. (1989). Properties of the momentum LMS algorithm. *Signal Processing*, *18*(2), 117–127.

Tzirkel-Hancock, E. and Fallside, F. (1991). A Direct Control Method for a Class of Nonlinear Systems using Neural Networks. Tech. rep. CUED/F-INFENG/TR.65, Cambridge University Engineering Department, Cambridge, England.

Uchiyama, T., Shimohara, K., and Tokunaga, Y. (1989). A Modified Leaky Integrator Network for Temporal Pattern Recognition. In IJCNN89 (1989), pp. 469–475.

VanLandingham, H. F. (1985). *Introduction to Digital Control Systems*. Macmillan, New York.

Vogl, T. P., Mangis, J. K., Zigler, A. K., Zink, W. T., and Alkon, D. L. (1988). Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, *59*, 257–263.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. (1989). Phoneme Recognition using Time-Delay Networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *37*(3), 328–339.

Waltz, D. L. and Pollack, J. B. (1985). Massively parallel parsing: A strongly interactive model of natural language interpretation. *Cognitive Science*, *9*, 51–74.

Watanabe, N., Nagata, S., and Asakawa, K. (1988). Mobile Robot Control by Neural Networks and their Associated Learning Algorithms. Tech. rep., Fujitsu, Artificial Intelligence Laboratory, Kawasaki 211, Japan.

Watrous, R. L. (1988). *Speech Recognition Using Connectionist Networks*. Ph.D. thesis, University of Pennsylvania.

Watrous, R. (1987). Learning Algorithms for Connectionist Networks: Applied Gradient Methods of Nonlinear Optimization. In Caudill and Butler (1987), pp. 619–627.

Watrous, R. L. and Kuhn, G. M. (1992). Induction of Finite-State Automata Using Second-Order Recurrent Networks. In NIPS*91 (1992), pp. 309–316.

Watrous, R. L., Laedendorf, B., and Kuhn, G. M. (1990). Complete Gradient Optimization of a Recurrent Network Applied to BDG Descrimination. *Journal of the Acoustical Society of America*, *87*(3), 1301–1309.

Weigend, A. S., Rumelhart, D. E., and Huberman, B. A. (1991). Generalization by Weight-Elimination with Application to Forecasting. In NIPS*90 (1991), pp. 875–882.

Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph.D. thesis, Harvard University.

Werbos, P. J. (1982). Applications of Advances in Nonlinear Sensitivity Analysis. In Drenick, R. F. and Kozin, F. (Eds.), *System Modeling and Optimization: Proceedings of the 10th IFIP Conference*, No. 38 in Lecture Notes in Control and Information Sciences (New York, Aug 31–Sep 4, 1981). Springer-Verlag.

Werbos, P. J. (1988a). Backpropagation: Past and Future. In *IEEE International Conference on Neural Networks*, Vol. I, pp. 343–353 San Diego, CA.

Werbos, P. J. (1988b). Generalization of Backpropagation with Application to a Recurrent Gas Market Model. *Neural Networks*, *1*, 339–356.

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

Werbos, P. J. (1992). Neural Networks, System Identification, and Control in the Chemical Process Industries. In White, D. A. and Sofge, D. A. (Eds.), *Handbook of Intelligent Control—Neural, Fuzzy, and Adaptive approaches*, chap. 10, pp. 283–356. Van Norstrand Reinhold. see section 10.7.

Widrow, B. and Hoff, M. (1960). Adaptive Switching Circuits. In *Western Electronic Show and Convention, Convention Record*, Vol. 4, pp. 96–104. Institute of Radio Engineers (now IEEE).

Widrow, B., McCool, J. M., Larimore, M. G., and Johnson, Jr., C. R. (1976). Stationary and Nonstationary Learning Characteristics of the LMS Adaptive Filter. *Proceedings of the IEEE*, *64*, 1151–1162.

Widrow, B. and Stearns, S. D. (1985). *Adaptive Signal Processing*. Prentice-Hall signal processing series. Prentice-Hall.

Wilde, D. J. and Beightler, C. S. (1967). *Foundations of Optimization*. Prentice-Hall.

Williams, R. J. (1992). Some observations on the use of the extended Kalman filter as a recurrent network learning algorithm. Tech. rep. NU-CCS-92-1, College of Computer Science, Northeastern University, Boston, MA.

Williams, R. J. and Zipser, D. (1990). Gradient-based learning algorithms for recurrent connectionist networks. Tech. rep. NU-CCS-90-9, College of Computer Science, Northeastern University, Boston, MA. Also published as Williams and Zipser (1995).

Williams, R. J. and Zipser, D. (1995). Gradient-based learning algorithms for recurrent networks and their computational complexity. In Chauvin and Rumelhart (1995). Also published as Williams and Zipser (1990).

Williams, R. J. (1992). Training Recurrent Networks Using the Extended Kalman Filter. In IJCNN92'Baltimore (1992), pp. 241–250.

Williams, R. J. and Peng, J. (1990). An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories. *Neural Computation*, *2*(4), 490–501.

Williams, R. J. and Zipser, D. (1988). A learning algorithm for continually running fully recurrent neural networks. Tech. rep. ICS Report 8805, UCSD, La Jolla, CA 92093.

Williams, R. J. and Zipser, D. (1989). A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, *1*(2), 270–280.

Zipser, D. (1990). Subgrouping Reduces Complexity and Speeds Up Learning in Recurrent Networks. In NIPS*89 (1990), pp. 638–641.