5-2010

Neural Networks and the Natural Gradient

Michael R. Bastian
Utah State University

NEURAL NETWORKS AND THE NATURAL GRADIENT

by

Michael R. Bastian

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

Approved:

Dr. Jacob H. Gunther Major Professor	Dr. Todd K. Moon Committee Member
Dr. YangQuan Chen Committee Member	Dr. Wei Ren Committee Member
Dr. Donald Cooley Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2009

Abstract

Neural Networks and the Natural Gradient

by

Michael R. Bastian, Doctor of Philosophy

Utah State University, 2009

Major Professor: Dr. Jacob H. Gunther
Department: Electrical and Computer Engineering

Neural network training algorithms have always suffered from the problem of local minima. The advent of natural gradient algorithms promised to overcome this shortcoming by finding better local minima. However, they require additional training parameters and computational overhead. By using a new formulation for the natural gradient, an algorithm is described that uses less memory and processing time than previous algorithms with comparable performance.

(153 pages)

Ad Majorem Dei Gloriam

Acknowledgments

I would have not been able to write this dissertation without the help of Dr. Jacob H. Gunther and Dr. Todd K. Moon. They were both excellent mentors and have vast knowledge of our discipline. Without their guidance I would not have been able to develop my research.

My wife has been very patient with me during the last six years while I completed coursework and developed the ideas in this dissertation. Her encouragement has helped me complete it. My children have also helped me by giving me time to write.

I would like to thank my other committee members for their valuable input during the progress of my research. The faculty and staff of the Electrical and Computer Engineering Deparment have been an excellent resource. I thank you all.

The United States Air Force is my employer and has been very supportive of my efforts and has allowed me time off to work on my dissertation. The financial support of the USAF has also allowed me to present my work at two conferences.

Finally, writing this dissertation would have been much more difficult without the LaTeX Companion [1].

Michael R. Bastian

Contents

List of Tables

List of Figures

Notation

a	A scalar value
$f(x)$	A real-valued function of x
X	A random variable
$\mathbf{x}$	A vector
$\langle \mathbf{x}, \mathbf{y} \rangle$	The inner product of $\mathbf{x}$ and $\mathbf{y}$
$\|\mathbf{x}\|$	The norm of $\mathbf{x}$
$\mathbf{A}$	A matrix
$\mathbf{X}$	A random vector
$\mathbf{F}(\mathbf{x})$	A real vector-valued function of the vector $\mathbf{x}$
$\mathbf{A}^T$	The transpose of the matrix $\mathbf{A}$
$\mathrm{tr}(\mathbf{A})$	Trace of the matrix $\mathbf{A}$
$E\{X\}$	The expected value of of the random variable X

Acronyms

ANGL	Adaptive Natural Gradient Learning (Multilayer Perceptron)
BFGS	Broyden, Fletcher, et al. Optimization Algorithm
BSS	Blind Source Separation
CDF	Cumulative Distribution Function
FIM	Fisher Information Matrix
ICA	Independent Component Analysis
LM	Levenburg-Marquardt Optimization Algorithm
LR	Learning Rate
LSM	Line Search Method
MAP	Maximum A Posteriori
ML	Maximum Likelihood
MLP	Multilayer Perceptron
NG	Natural Gradient
OGL	Ordinary Gradient Learning (Multilayer Perceptron)
PDF	Probability Density Function
RBF	Radial Basis Function
RVM	Relevance Vector Machine
SVM	Support Vector Machine
XOR	Exclusive-OR

Chapter 1

Introduction to Artificial Neural Networks

This dissertation is about training artificial neural networks with natural gradient algorithms. Knowledge of the basics of neural networks is necessary to understand the developments in this dissertation. Thus, this first chapter is a gentle introduction to the field of artificial neural networks and machine learning.

1.1 Historical Development

The discipline of artificial intelligence began with Huxley's hypothesis that animal behavior could be explained with automata [2]. This caused William James to ask the question, "Are we automata?" five years later [3]. However, these two essays are purely philosophical. The first scientific development linking human ideas and eletrical impulses in the brain was reported by McColluch and Pitts [4]. Further evidence was found by Hebb [5]. These ideas combined with the development of Turing Machines [6–8] laid the physical and mathematical foundations for the development of neural networks.

1.2 Biological Background

The neuron is the fundamental cell of the human nervous system [9, 10]. It is the basis upon which neural networks were designed. A neuron is composed of the dendrites, cell body (soma), and axon (see fig. 1.1). The dendrites receive electrochemical signals and transfer them to the cell body. The cell body processes these signals and then sends its own signal to other neurons via the axon. The axon terminates in the synaptic cleft. The synaptic cleft is a collection of synapses that transmit signals from the axon terminal to the dendrites of other neurons. The brain cells are all interconnected with each other. This interconnected architecture is the foundation of artificial neural networks [11].

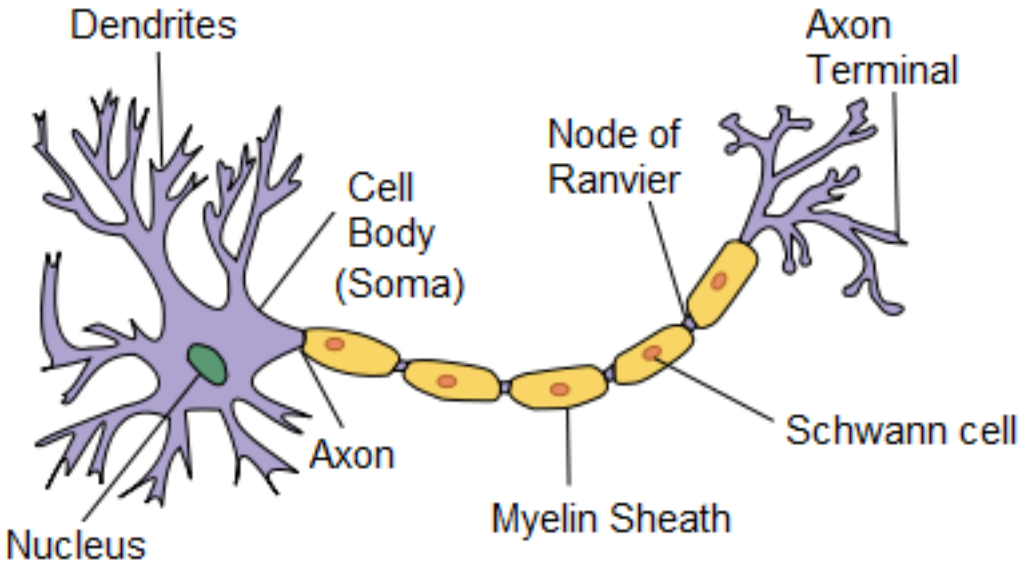

Fig. 1.1: The basic anatomy of the neuron. The soma is the body of the neuron cell. The dendrites receive signals through the synapses. The axon connects to other synapses.

Neurons react to the stimuli they receive from the synapses connected to their dendrites. When a neuron is near its unexcited state and the impulses increase or decrease in voltage, the electrical response from the neuron is nearly linear to its input. However, there is a threshold in its response that it does not exceed. Its response gradually tapers off as the magnitude of the voltage increases. When it reaches this limit the neuron is in a saturated state. This trait is mimicked by artificial neurons. The biological inspiration of artificial neural networks is the interconnected collection of neurons with limited responses to stimuli [4, 5].

1.3 Basic Computational Structure

Artificial neural networks are composed of artificial neurons that mimic the behavior of a neuron in a limited way [12]. They mimic the saturation behavior of the human neuron in the brain. Artificial neurons are functions of several variables with two horizontal asymptotes. A simple definition of an artificial neuron is a function that maps vectors in a vector space V to the interval $[0, 1]$. The interval can actually be any interval of the real line, but $[0, 1]$ and $[-1, 1]$ are frequently chosen.

1.3.1 Parameters of an Artificial Neuron

An artificial neuron has a weight vector $\mathbf{w} = (w_1, w_2, \ldots, w_m)$ and a threshold or bias b. The weights are analogous to the strength of the dendritic connections. The bias is considered the value that must be surpassed by the inputs before an artificial neuron will become active (i.e., greater than zero). However, it is usually written as an additive term.

The *activation* of a neuron is the sum of the inner product of the weight vector with an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_m)$ and the bias:

$$a = \sum_{i=1}^{m} w_i x_i + b. \tag{1.1}$$

The weights and the bias define a hyperplane in the vector space V with dimension n. The activation of the neuron is the distance of the input vector from this hyperplane.

1.3.2 Sigmoid Functions

The saturation property is achieved by "squashing" the activation of a neuron with a sigmoid transfer function like

$$f(t) = \frac{1}{1 + e^{-t}}. \tag{1.2}$$

This limits the output between 0 and 1. These functions are called sigmoid functions because of their "S"-shape (see fig. 1.2). The function in (1.2) is the logistic function. It is the CDF of the Logistic Distribution. Other sigmoid functions are the arctangent, hyperbolic tangent, and the error function [13].

Another property of the sigmoid function is that their derivatives can be written as quadratic functions of their values. For example, the derivative of the logistic function in (1.2) is

$$f'(t) = \frac{e^{-t}}{(1 + e^{-t})^2} = f(t)[1 - f(t)]. \tag{1.3}$$

1.3.3 Linear Classifiers

Suppose that there are two finite sets of vectors, X_0 and X_1, in a vector space V. Any hyperplane will split V into two distinct subsets, V_0 and V_1. If $X_0 \subset V_0$ and $X_1 \subset V_1$, then

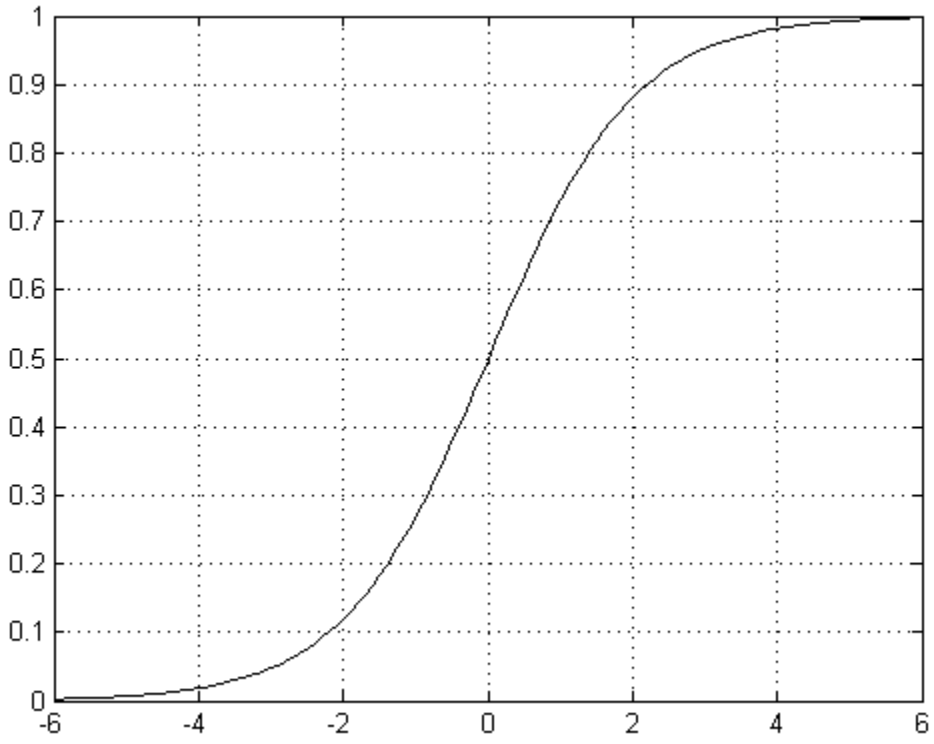

Fig. 1.2: The logistic curve.

define the probability that a random vector $\mathbf{x}$ is in X_1 to be the logistic distribution

$$P[\mathbf{x} \in X_1] = \frac{1}{1 + \exp\big(-\big[\sum_{i=1}^{n} w_i x_i + b\big]\big)}. \tag{1.4}$$

An artificial neuron is really just a logistic function whose argument is the output of a linear classifier [12]. The probability that $\mathbf{x} \in X_1$ increases with the positive distance of the vector $\mathbf{x}$ above the hyperplane. Likewise, this probability decreases with the negative distance $\mathbf{x}$ is below the hyperplane. When the vector $\mathbf{x}$ rests on the hyperplane (i.e., the distance is zero), then the probability $P[\mathbf{x} \in X_1] = 0.5$. The distance from the hyperplane is $d = \langle \mathbf{x}, \mathbf{w} \rangle + b$ where $\mathbf{w}$ is the normal vector of the hyperplane and b is the distance from the origin measured in the direction of the normal vector.

An example linear classifier is given in fig. 1.3. The positive examples of the class in X_1 are 100 random vectors from a normal distribution with $\mu_1 = (1.5, 0.5)$ and $\sigma^2 = 0.01$. The negative examples are 100 random vectors from a similar normal distribution with $\mu_0 = (0.5, -0.5)$. The decision boundary is the line $x_1 + x_2 = 1$.

The neuron parameters are the normal vector and the distance from the origin. Hence, $\mathbf{w} = (1, 1)$ and $b = -1$. The vector μ_1 is exactly 1 unit away from the decision boundary. Likewise, the vector μ_0 is also exactly 1 unit away from the boundary. Thus,

$$P[\mu_1 \in X_1] = f(1) = \frac{1}{1 + e^{-1}} \approx 0.7311,$$

and

$$P[\mu_0 \in X_1] = f(-1) = \frac{1}{1 + e} \approx 0.2689.$$

1.3.4 Training an Artificial Neuron

An artificial neuron can be trained to classify vectors by giving it both positive and negative examples. The training algorithm will then determine the optimal hyperplane with which to divide the vector space. The neuron is considered to have "learned" the concept by adjusting its weights and bias.

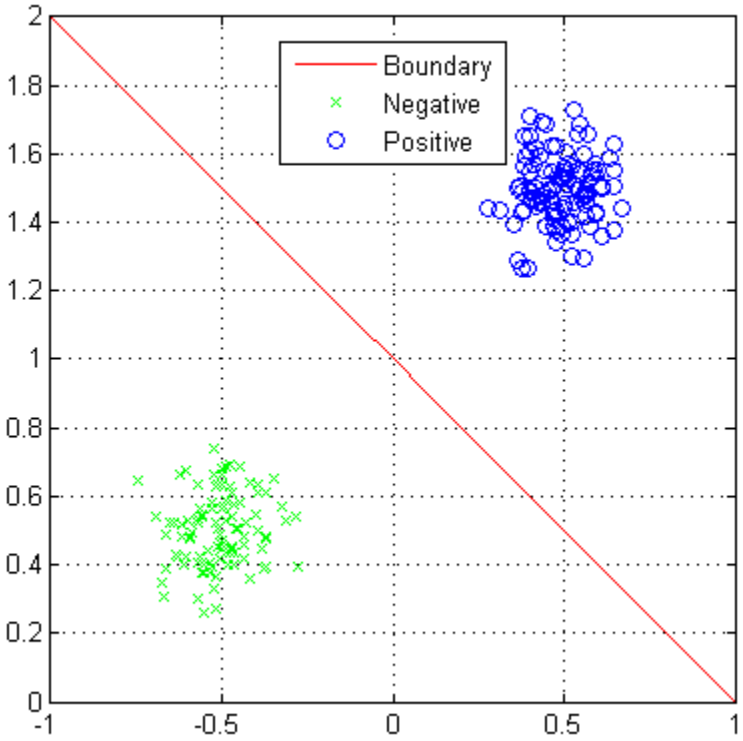

Fig. 1.3: The blue circles are the positive examples and the green x's are the negative examples. The red line is the decision boundary of the linear classifier.

The Training Set

The training set is really two sets: The training examples $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ and the training targets $Y = \{y_1, y_2, \ldots, y_n\}$, where each $y_k \in \{0, 1\}$ is the desired output of the neuron. The indices match such that y_k is the desired output for example $\mathbf{x}_k$. Hence, the positive examples belong to the set $X_1 = \{\mathbf{x}_k | y_k = 1\}$ and the negative examples belong to the set $X_0 = \{\mathbf{x}_k | y_k = 0\}$.

Probability of Correct Classification

For a given random vector $\mathbf{x}$ and the parameters $\mathbf{w}$ and b, the probability of a positive classification is

$$P[y = 1|\mathbf{x}, \mathbf{w}, b] = f(\langle \mathbf{x}, \mathbf{w} \rangle + b), \tag{1.5}$$

where $f(\cdot)$ is the logistic sigmoid function [12]. Similarly, the probability of a correct classification is

$$p(y|\mathbf{x}, \mathbf{w}, b) = [f(\langle \mathbf{w}, \mathbf{x} \rangle + b)]^y [1 - f(\langle \mathbf{w}, \mathbf{x} \rangle + b)]^{1-y}. \tag{1.6}$$

The Objective Function

The goal of a training algorithm is maximize the correct classification of all the training examples. Formally, that means determine parameters $\mathbf{w}$ and b such that

$$p(y|\mathbf{w}, b) = \prod_{k=1}^{n} [f(\langle \mathbf{w}, \mathbf{x}_k \rangle + b)]^{y_k} [1 - f(\langle \mathbf{w}, \mathbf{x}_k \rangle + b)]^{1-y_k} \tag{1.7}$$

is maximized [14]. An equivalent problem is to minimize the log-likelihood

$$\ell(\mathbf{w}, b) = \log p(y|\mathbf{w}, b) \tag{1.8}$$

$$= \sum_{k=1}^{n} y_k \log(f(\langle \mathbf{w}, \mathbf{x}_k \rangle + b)) + (1 - y_k) \log(1 - f(\langle \mathbf{w}, \mathbf{x}_k \rangle + b)). \tag{1.9}$$

The Gradient

A gradient ascent algorithm will determine the optimal values of $\mathbf{w}$ and b. Let $d_k = \langle \mathbf{w}, \mathbf{x}_k \rangle + b$. The derivative of the log-likelihood function ℓ with respect to the vector $\mathbf{w}$ is

$$\frac{\partial \ell}{\partial \mathbf{w}} = \sum_{k=1}^{n} \left[\frac{y_k f'(d_k)}{f(d_k)} - \frac{(1-y_k) f'(d_k)}{1-f(d_k)} \right] \mathbf{x}_k \tag{1.10}$$

$$= \sum_{k=1}^{n} \left[y_k (1 - f(d_k)) - (1-y_k) f(d_k) \right] \mathbf{x}_k \tag{1.11}$$

$$= \sum_{k=1}^{n} [y_k - f(d_k)] \mathbf{x}_k. \tag{1.12}$$

Similarly, the derivative of the log-likelihood function with respect to the bias is

$$\frac{\partial \ell}{\partial b} = \sum_{k=1}^{n} [y_k - f(d_k)]. \tag{1.13}$$

The gradient of the neuron is the difference between its output and the designated target multiplied by its input vector.

The Learning Rule

The final piece of the algorithm is the learning rule. A learning rule defines how the weights and bias are changed. In a gradient descent algorithm, the learning rule is to subtract a small multiple η of the gradient vector from the weights and bias. By distributing the minus sign into the formula for the gradient, the learning rule is

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \sum_{k=1}^{n} [y_k - f(\langle \mathbf{w}_t, \mathbf{x}_k \rangle + b_t)] \mathbf{x}_k, \tag{1.14}$$

$$b_{t+1} = b_t - \eta_t \sum_{k=1}^{n} [y_k - f(\langle \mathbf{w}_t, \mathbf{x}_k \rangle + b_t)]. \tag{1.15}$$

The variable η is called the *learning rate*. It is the step size for the algorithm. It is determined empirically for each problem. The subscript t for the weights, bias and learning rate η denotes the current step of the algorithm. The initial weight vector $\mathbf{w}_0$ and bias b_0 are chosen randomly from a uniformly distributed random vector and random variable

respectively. A common range is $[-0.1, 0.1]^m$ for $\mathbf{w}_0$ and $[-0.1, 0.1]$ for b_0.

In order for this method of finding the optimal weights and bias to be a true algorithm, there must be stopping criteria with which the algorithm halts with its best solution. A good solution for a neuron would be one in which most of the training examples were classified correctly. When the sum of squares of the errors is less than a predetermined tolerance, then it can be assumed that most of the training examples were classified correctly. Sum of squares and the sum of absolute values guard against large errors cancelling each other out in the sum. As an example, if there is a requirement for 90% accuracy (i.e., the output is 0.9 or greater for each positive example and 0.1 or less for each negative example), then the maximum error for each training example would be 0.1. Because there are n training examples a good stopping criteria would be

$$\frac{1}{n}\sum_{k=1}^{n}[y_k - f(\langle \mathbf{w}_t, \mathbf{x}_k\rangle + b_t)]^2 < 0.01. \tag{1.16}$$

1.3.5 Online and Batch Training

There are two ways in which the learning rule may be applied. In batch training the error is calculated for all the patterns in the training set and then applied to the weights and bias via the learning rule as in (1.14) and (1.15). In online training the learning rule is applied after each training example. Hence, given that $k = t \mod m$, the online learning rule for the neuron would be

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t[y_k - f(\langle \mathbf{w}_t, \mathbf{x}_k\rangle + b_t)]\mathbf{x}_k, \tag{1.17}$$

$$b_{t+1} = b_t - \eta_t[y_k - f(\langle \mathbf{w}_t, \mathbf{x}_k\rangle + b_t)]. \tag{1.18}$$

There are two terms used quite frequently in online learning applications: *learning cycle* and *learning epoch*. A learning cycle is one step through an algorithm for one training example. A learning epoch is one full trip through all the training examples (i.e., n learning cycles where n is the number of training examples). In a batch algorithm a learning cycle and a learning epoch are essentially the same concept.

There has been some investigation into the merits of both online and batch training [14–19]. However, batch training will be the focus in this work because of the computational overhead online training imposes on more advanced training algorithms.

1.3.6 Avoiding Overtraining

Overtraining occurs when a training algorithm has been run through an excessive number of learning epochs. Essentially, when a nearly optimal solution is found, the weights and bias tend to increase in value with each succeeding training epoch. The neuron comes closer and closer to responding with one for each positive example and zero with each negative example. This is undesirable because the transfer is supposed to be linear in an interval of its domain. The weight value is essentially the slope. As it increases, so does the slope. The interval of which it is linear shrinks. The neuron is now overconfident in its classification.

In fig. 1.4 four sigmoid curves are shown. Each is the plot of $f(t)$, $f(2t)$, $f(3t)$, and $f(6t)$, respectively. As the weights and bias increase the curve tightens. With a sufficiently large factor the sigmoid begins to approximate a step function.

Suppose that an artificial neuron was being trained on a computer with floating-point numbers, then there is a number $\epsilon > 0$ such that $1 + \epsilon = 1$ because of limited precision. For example, with double precision floating-point arithmetic, $\epsilon = 2.2204 \times 10^{-16}$. Let $t_{\max}$ represent the value such that $f(t_{\max}) = 1$; then $t_{\max} = -\log(\epsilon) = 36.0437$. The neuron mimics the behavior of a saturated transistor with stimuli of this magnitude.

A trivial method to avoid overtraining is to stop once the training algorithm has converged to a nearly optimal solution. However, defining convergence and "nearly optimal" depends heavily on the training data used. Early stopping has the same trait as the halting problem from computer science: How long is long enough?

Adding a prior distribution on the neural parameters $\mathbf{w}$ and b that penalizes larger parameters with smaller probabilities causes the algorithm to essentially stop when the parameters become large enough, but no larger [12].

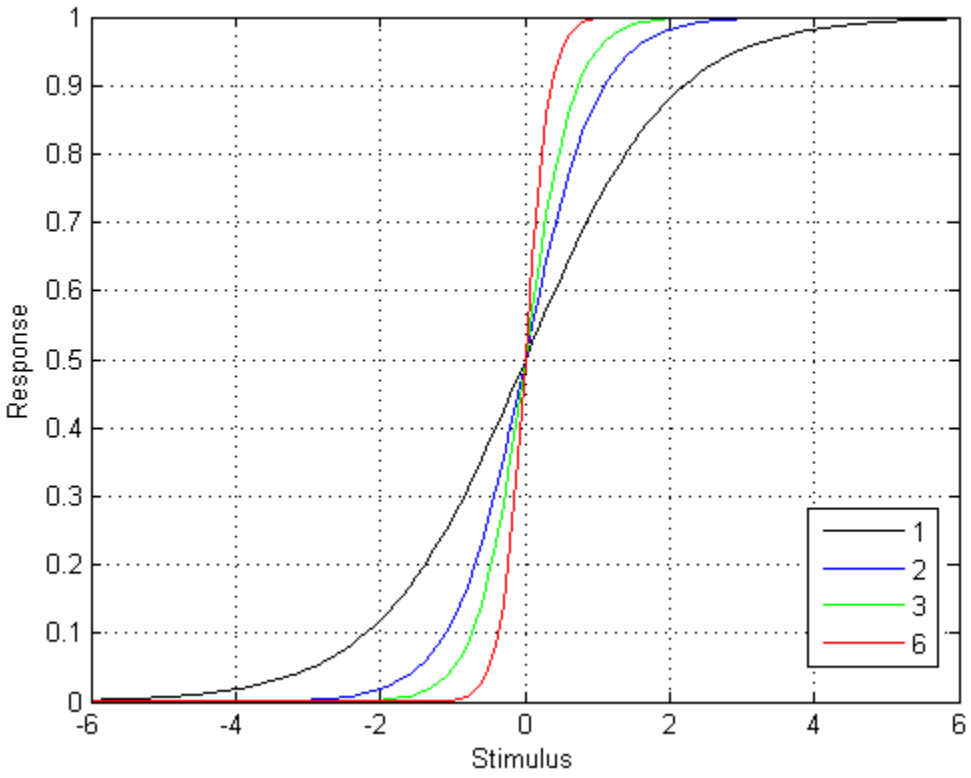

Fig. 1.4: The effect of overtraining is shown as the gain of the input to a sigmoid function increases. The slope of the line increases and the curvature at the saturation level also increases.

A Gaussian distribution with density function $p(\mathbf{w}, b) = \exp(-\frac{1}{2}\alpha[\|\mathbf{w}\|^2 + b^2])$ is an adequate choice. The hyperparameter α controls how big the parameters become. The relationship between α and the standard deviation σ of the Gaussian curve is $\alpha = \frac{1}{\sigma^2}$. The modified probability of correct classification would be

$$p(y|x, \mathbf{w}, b) = \exp\left(-\frac{1}{2}\alpha\big[\|\mathbf{w}\|^2+b^2\big]\right) \prod_{k=1}^{n} \big[f(\langle \mathbf{w}, \mathbf{x}_k\rangle+b)\big]^{y_k} \big[1-f(\langle \mathbf{w}, \mathbf{x}_k\rangle+b)\big]^{1-y_k}. \quad (1.19)$$

The log-likelihood function would then become

$$\ell(\mathbf{w}, b) = -\frac{1}{2}\alpha\big[\|\mathbf{w}\|^2 + b^2\big] + \sum_{k=1}^{n} \big[y_k \log(f(\langle \mathbf{w}, \mathbf{x}_k\rangle + b)) + (1 - y_k) \log(1 - f(\langle \mathbf{w}, \mathbf{x}_k\rangle + b))\big]. \quad (1.20)$$

The partial derivative of the weights is now

$$\frac{\partial \ell}{\partial \mathbf{w}} = \sum_{k=1}^{n} \big[y_k - f(\langle \mathbf{w}, \mathbf{x}_k\rangle + b)\big]\mathbf{x}_k - \alpha\mathbf{w}. \quad (1.21)$$

The partial derivative of the bias is now

$$\frac{\partial \ell}{\partial b} = \sum_{k=1}^{n} \big[y_k - f(\langle \mathbf{w}, \mathbf{x}_k\rangle + b)\big] - \alpha b. \quad (1.22)$$

The effect of this change to the training algorithm is that when the total error is nearly equal to the hyperparameter α, then the algorithm reaches a steady state where $\mathbf{w}$ and b do not change very much.

1.3.7 Newton's Method

To find the minimum of a concave (up) function, $g(x)$, then the value $x_{\min}$ must satisfy the requirements that $g'(x_{\min}) = 0$ and $g''(x_{\min}) > 0$. Newton's method [20] to find this value is

$$x_{t+1} = x_t - \frac{g'(x_t)}{g''(x_t)}. \quad (1.23)$$

When x is a vector, then the *Hessian matrix*, $\nabla^2 g(x)$, is inverted and then multiplied with the gradient vector

$$x_{t+1} = x_t - \left[\nabla^2 g(x_t)\right]^{-1} \nabla g(x_t). \tag{1.24}$$

When a vector $x_{\min}$ is the minimum of a function $g(x)$, then its gradient is the zero vector and the Hessian matrix is positive-definite (i.e., all its eigenvalues are positive [21]). An artificial neuron's Hessian matrix is [12]

$$\nabla^2 H = \begin{bmatrix} \frac{\partial^2 \varphi}{\partial \mathbf{w}^2} & \frac{\partial^2 \varphi}{\partial \mathbf{w} \partial b} \\ \left(\frac{\partial^2 \varphi}{\partial b \partial \mathbf{w}}\right)^T & \frac{\partial^2 \varphi}{\partial b^2} \end{bmatrix}, \tag{1.25}$$

$$\frac{\partial^2 \varphi}{\partial \mathbf{w}^2} = \sum_{k=1}^{n} f(\langle \mathbf{w}, \mathbf{x}_k \rangle + b) \left[1 - f(\langle \mathbf{w}, \mathbf{x}_k \rangle + b)\right] \mathbf{x}_k \mathbf{x}_k^T + \alpha \mathbf{I}, \tag{1.26}$$

$$\frac{\partial^2 \varphi}{\partial \mathbf{w} \partial b} = \sum_{k=1}^{n} f(\langle \mathbf{w}, \mathbf{x}_k \rangle + b) \left[1 - f(\langle \mathbf{w}, \mathbf{x}_k \rangle + b)\right] \mathbf{x}_k, \tag{1.27}$$

$$\frac{\partial^2 \varphi}{\partial b^2} = \sum_{k=1}^{n} f(\langle \mathbf{w}, \mathbf{x}_k \rangle + b) \left[1 - f(\langle \mathbf{w}, \mathbf{x}_k \rangle + b)\right] + \alpha. \tag{1.28}$$

The Newton's Method learning rule is

$$\begin{bmatrix} \mathbf{w}_{t+1} \\ b_{t+1} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_t \\ b_t \end{bmatrix} - \eta_t \left[\nabla^2 H_t\right]^{-1} \nabla H_t, \tag{1.29}$$

where the subscript t indicates that the gradient and Hessian matrix are calculated with the weight vector $\mathbf{w}_t$ and bias b_t.

When one or more of the eigenvalues of the Hessian matrix are very close to zero, then inverting the matrix will lead to numerical instability because it is similar to dividing by zero. This inevitably happens when training a neuron. As the training algorithm proceeds, the outputs come closer to being either 0 or 1. The closer the outputs come to these limits, the closer the derivative in (1.25) comes to 0 (see fig. 1.5). When all the outputs have been memorized by the neuron, then the sum-squared error will be small and so will the gradient and the Hessian matrix. Hence, the more extreme cases occur when the neuron has been

overtrained. Of course, these cases are avoided by using (1.20) as the objective function.

Suppose that for every input pattern $\mathbf{x}_k$, the output of the network is either 0 or 1. Assume that each y_k is either 0 or 1. Now, using the block inversion of $\nabla^2 H$, the changes to $\mathbf{w}$ and b will be

$$\delta b = \frac{\left(\frac{\partial^2\varphi}{\partial b\partial \mathbf{w}}\right)^T \left[\frac{\partial^2\varphi}{\partial \mathbf{w}^2}\right]^{-1} \frac{\partial\varphi}{\partial \mathbf{w}} - \frac{\partial\varphi}{\partial b}}{\left(\frac{\partial^2\varphi}{\partial b\partial \mathbf{w}}\right)^T \left[\frac{\partial^2\varphi}{\partial \mathbf{w}^2}\right]^{-1} \frac{\partial^2\varphi}{\partial \mathbf{w}\partial b} - \frac{\partial^2\varphi}{\partial b^2}}, \tag{1.30}$$

$$\delta \mathbf{w} = \left[\frac{\partial^2\varphi}{\partial \mathbf{w}^2}\right]^{-1} \frac{\partial\varphi}{\partial \mathbf{w}} + \left[\frac{\partial^2\varphi}{\partial \mathbf{w}^2}\right]^{-1} \frac{\partial^2\varphi}{\partial \mathbf{w}\partial b}\delta b. \tag{1.31}$$

In this case,

$$\frac{\partial\varphi}{\partial \mathbf{w}} = \alpha\mathbf{w}, \qquad \frac{\partial\varphi}{\partial b} = \alpha b, \tag{1.32}$$

$$\frac{\partial^2\varphi}{\partial \mathbf{w}^2} = \alpha\mathbf{I}, \qquad \frac{\partial^2\varphi}{\partial b^2} = \alpha, \tag{1.33}$$

$$\left[\frac{\partial^2\varphi}{\partial \mathbf{w}^2}\right]^{-1} = \alpha^{-1}\mathbf{I}, \qquad \frac{\partial^2\varphi}{\partial \mathbf{w}\partial b} = 0. \tag{1.34}$$

Hence, $\delta\mathbf{w} = \mathbf{w}$ and $\delta b = b$. Thus, each step would move the parameters one step closer to zero. The algorithm starts backing away from its overtrained parameters.

The hyperparameter α now has a dual purpose: It prevents the weight vector from getting too large and it prevents the Hessian matrix from becoming nearly singular. When training with Newton's Method the hyperparameter α could be chosen based on the eigenvalues of $\nabla^2 H$ [22]. Let λ_1 be the smallest eigenvalue of $\nabla^2 H$. If $\lambda_1 \geq 0$, then α should be chosen such that the ratio

$$\frac{\lambda_{m+1} + \alpha}{\lambda_1 + \alpha} > 0, \tag{1.35}$$

where m is the dimension of $\mathbf{w}$ and λ_{m+1} is the largest eigenvalue of $\nabla^2 H$. The ratio in (1.35) should not be too large. Hence, in early learning cycles α could be effectively zero when the Hessian matrix is full rank and well-conditioned. As the algorithm proceeds, α could increase and thus hinder the further increase of the parameters. If $\lambda_1 < 0$, then $\alpha = -\lambda_1 + \mu$ where $\mu > \lambda_1$ is chosen so that α is positive and (1.35) is not very large.

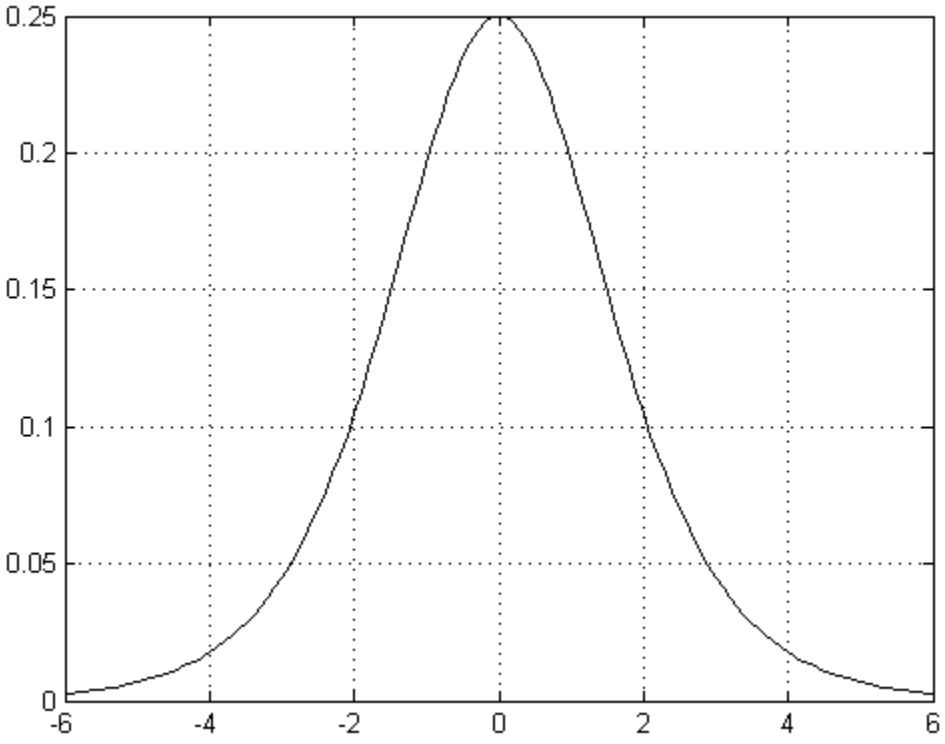

Fig. 1.5: The derivative of the logistic function.

1.3.8 Annealed Learning Rates

Using a constant learning rate is usually not a good idea [16]. As the learning algorithm progresses, the errors will decrease in magnitude. Hence, the gradient will be smaller, too. Experiments have shown that reducing the learning rate over time brings the algorithm to convergence faster and also avoids instabilities that occur because the step size is too big.

A decaying learning rate, similar to the multiplicative inverse curve

$$\eta_t = \min\left(\eta_0, \frac{\tau}{t}\right), \tag{1.36}$$

works very well (see fig. 1.6). The initial learning rate, η_0, is the default step size used until the decaying learning rate is smaller. When used with the minimum function this avoids a divide by zero. The τ factor allows the curvature to be adjusted.

Choosing the values of η_0 and τ based on empirical results allows the fine tuning many learning algorithms require. The initial learning rate, η_0, should be chosen so that the objective function decreases at a good rate without causing any instability. Most learning algorithms hit a plateau after a few hundred epochs. The parameter τ is chosen so that the learning rate decay begins in this plateau region. This should help the learning algorithm escape the plateau without overtraining.

1.3.9 Criticism of Neurons and Linear Classifiers

The harshest criticisms of artificial neurons and other linear classifiers stem from their inherent limitation: They can only learn linearly separable problems. This is a deficiency of the model.

Neurons are also overly optimistic [12] because the distance of a pattern from the hyperplane is used to calculate the probability. The neuron really has no concept of a "center." Hence, when a pattern is far away from any of the training patterns, the probability of a correct classification should be smaller than it really is.

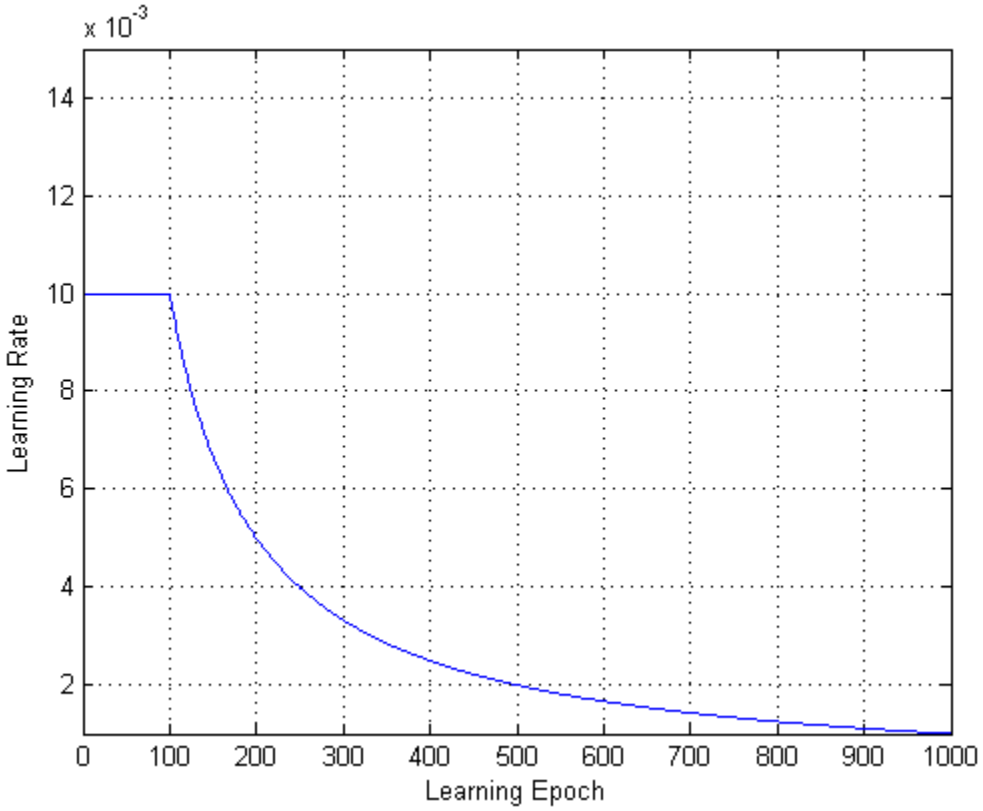

Fig. 1.6: The learning rate is flat for the first hundred epochs and then decays with the inverse of the learning epoch index.

1.4 Multilayer Perceptrons

One way to overcome the neuron's limitations is to assemble them to work both in parallel and serial connections. This section will cover feed-forward (i.e., no feedback) networks with multiple layers of neurons.

1.4.1 The Linear Perceptron

A linear perceptron [23–25] is a bank of artificial neurons with linear activation functions. Another description of a linear perceptron is an affine transformation that maps one set of vectors $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ to another set of vectors $Y = \{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n\}$. Its parameters are a weight matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$ to offset any translation between the two sets. Each row of $\mathbf{W}$ is the weight vector of a neuron and each component of $\mathbf{b}$ is its respective bias. The objective function of a linear perceptron is

$$J(\mathbf{W}, \mathbf{b}) = \frac{1}{2} \sum_{k=1}^{n} \|\mathbf{y}_k - (\mathbf{W}\mathbf{x}_k + \mathbf{b})\|^2. \tag{1.37}$$

This is a least-squares problem with a matrix parameter. Expanding (1.37) yields

$$\begin{aligned} J(\mathbf{W}, \mathbf{b}) &= \frac{1}{2} \sum_{k=1}^{n} \|\mathbf{y}_k - (\mathbf{W}\mathbf{x}_k + \mathbf{b})\|^2 && (1.38) \\ &= \frac{1}{2} \sum_{k=1}^{n} \|\mathbf{y}_k\|^2 - 2\langle \mathbf{y}_k, \mathbf{W}\mathbf{x}_k + \mathbf{b} \rangle + \|\mathbf{W}\mathbf{x}_k + \mathbf{b}\|^2 && (1.39) \\ &= \frac{1}{2} \sum_{k=1}^{n} \|\mathbf{y}_k\|^2 - 2\langle \mathbf{y}_k, \mathbf{W}\mathbf{x}_k + \mathbf{b} \rangle + \|\mathbf{W}\mathbf{x}_k\|^2 + 2\langle \mathbf{W}\mathbf{x}_k, \mathbf{b} \rangle + \|\mathbf{b}\|^2 && (1.40) \\ &= \frac{1}{2} \sum_{k=1}^{n} \mathbf{y}_k^T \mathbf{y}_k - 2\mathbf{y}_k^T (\mathbf{W}\mathbf{x}_k + \mathbf{b}) + \mathbf{x}_k^T \mathbf{W}^T \mathbf{W}\mathbf{x}_k + 2\mathbf{x}_k^T \mathbf{W}^T \mathbf{b} + \mathbf{b}^T \mathbf{b}. && (1.41) \end{aligned}$$

This sets up the derivation of the gradient as

$$\begin{aligned} \frac{\partial J}{\partial \mathbf{W}} &= \sum_{k=1}^{n} -\mathbf{y}_k \mathbf{x}_k^T + \mathbf{W}\mathbf{x}_k \mathbf{x}_k^T + \mathbf{b}\mathbf{x}_k^T && (1.42) \\ &= -\sum_{k=1}^{n} \big[\mathbf{y}_k - (\mathbf{W}\mathbf{x}_k + \mathbf{b})\big] \mathbf{x}_k^T, && (1.43) \end{aligned}$$

and

$$\frac{\partial J}{\partial \mathbf{b}} = -\sum_{k=1}^{n} \left[\mathbf{y}_k - (\mathbf{W}\mathbf{x}_k + \mathbf{b})\right]. \tag{1.44}$$

The solution is found by setting the gradient to zero and isolating the variables

$$\hat{\mathbf{b}} = \sum_{k=1}^{n} \left(\mathbf{y}_k - \hat{\mathbf{W}}\mathbf{x}_k\right), \tag{1.45}$$

$$\hat{\mathbf{W}} = \left[\sum_{k=1}^{n} \left(\mathbf{y}_k - \hat{\mathbf{b}}\right) \mathbf{x}_k^T\right] \left[\sum_{k=1}^{n} \mathbf{x}_k \mathbf{x}_k^T\right]^{-1}. \tag{1.46}$$

Solving a least-squares problem like this is more difficult than solving a linear system. The validity of the solution depends on the training set, especially the input patterns $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$. If the number of training examples is less than the dimension of the vector space from which these vectors are drawn, then the inverse matrix on the right-hand side of (1.46) does not exist. A rank deficient matrix will also result if any of the input vectors are linearly dependent. The initial guesses of $\hat{\mathbf{W}}$ and $\hat{\mathbf{b}}$ will also affect the solution. One place to start would be $\hat{\mathbf{W}} = \mathbf{I}$ and $\hat{\mathbf{b}} = 0$.

1.4.2 Nonlinear Perceptrons

Like neurons, nonlinear perceptrons have transfer functions on their outputs. A neuron has only one output and its transfer function is either the logistic function $f(t) = \frac{1}{1+\exp(-t)}$ or the hyperbolic tangent $\tanh(t)$.

For classification problems the outputs of a perceptron are the relative probabilities of an input pattern $\mathbf{x}$ belonging to a class. These probabilities are not independent. Therefore they should sum to one. The output map for a perceptron classifier is the *soft-max* function. This function is a sigmoid function, but each output depends on the relative value of the activations of all the other outputs. Let the vector $\mathbf{a} = (a_1, a_2, \ldots, a_m)$ be the linear activations of the outputs of a perceptron (i.e., $\mathbf{a} = \mathbf{W}\mathbf{x} + \mathbf{b}$), then the soft-max output for unit i is

$$f_i(\mathbf{a}) = \frac{\exp(a_i)}{\sum_{j=1}^{m} \exp(a_j)}. \tag{1.47}$$

Clearly, $\sum_{i=1}^{m} f_i(\mathbf{a}) = 1$, hence the outputs will always sum to one. The components of the Jacobian matrix for this map are

$$\frac{\partial f_i}{\partial a_j} = \begin{cases} f_i(1 - f_i) & i = j \\ -f_i f_j & i \neq j. \end{cases} \tag{1.48}$$

Let $\mathbf{F}(\mathbf{a}) = (f_1(\mathbf{a}), f_2(\mathbf{a}), \ldots, f_m(\mathbf{a}))$ be the output map for a perceptron and let $\mathbf{F}'(\mathbf{a})$ be the $m \times m$ Jacobian matrix of $\mathbf{F}$. The objective function for a perceptron with a nonlinear output map is

$$J(\mathbf{W}, \mathbf{b}) = \frac{1}{2} \sum_{k=1}^{n} \|\mathbf{y}_k - \mathbf{F}(\mathbf{W}\mathbf{x}_k + \mathbf{b})\|^2. \tag{1.49}$$

The gradient is then defined to be

$$\frac{\partial J}{\partial \mathbf{W}} = -\sum_{k=1}^{n} \left[\mathbf{F}'(\mathbf{W}\mathbf{x}_k + \mathbf{b})\right]^T (\mathbf{y}_k - \mathbf{F}(\mathbf{W}\mathbf{x}_k + \mathbf{b}))\, \mathbf{x}_k^T, \tag{1.50}$$

$$\frac{\partial J}{\partial \mathbf{b}} = -\sum_{k=1}^{n} \left[\mathbf{F}'(\mathbf{W}\mathbf{x}_k + \mathbf{b})\right]^T (\mathbf{y}_k - \mathbf{F}(\mathbf{W}\mathbf{x}_k + \mathbf{b})). \tag{1.51}$$

1.4.3 Perceptron Composition

Affine transformations such as the linear perceptron are closed under composition. Unfortunately, this is viewed as problematic because combining two linear perceptrons is just another linear perceptron. Let $\Phi_1(\mathbf{x}) = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$ and $\Phi_2(\mathbf{x}) = \mathbf{W}_2\mathbf{x} + \mathbf{b}_2$ be affine transformations, then their composition would be

$$(\Phi_1 \circ \Phi_2)(\mathbf{x}) = \mathbf{W}_1(\mathbf{W}_2\mathbf{x} + \mathbf{b}_2) + \mathbf{b}_1 = \mathbf{W}_1\mathbf{W}_2\mathbf{x} + \mathbf{W}_1\mathbf{b}_2 + \mathbf{b}_1. \tag{1.52}$$

This composition is an affine transformation $\phi_3(\mathbf{x}) = \mathbf{W}_3\mathbf{x} + \mathbf{b}_3$ where

$$\mathbf{W}_3 = \mathbf{W}_1\mathbf{W}_2, \tag{1.53}$$

$$\mathbf{b}_3 = \mathbf{W}_1\mathbf{b}_2 + \mathbf{b}_1. \tag{1.54}$$

When modelling the human brain with its networks of neurons connected through synapses,

the perceptron model does not fit very well because of this composition rule. However, the transfer function helps to change the model in a way such that it fits better.

Combining nonlinear perceptrons fits the model of the brain better. The outputs of the first perceptron are called *hidden units*. The transfer function limits the range of the outputs of these units. However, there still are some issues worth noting [26]. Let $\Phi_1(\mathbf{x}) = \mathbf{F}_1(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$ and $\Phi_2(\mathbf{x}) = \mathbf{F}_2(\mathbf{W}_2\mathbf{x} + \mathbf{b}_2)$. Their composition is

$$(\Phi_1 \circ \Phi_2)(\mathbf{x}) = \mathbf{F}_1(\mathbf{W}_1\mathbf{F}_2(\mathbf{W}_2\mathbf{x} + \mathbf{b}_2) + \mathbf{b}_1). \tag{1.55}$$

Their composition is still a function, but it is more complex. The parameters of Φ_1 and Φ_2 stay separate. However, note that if the rows of $\mathbf{W}_2$ are permuted along with the components of $\mathbf{b}_2$ and the columns of $\mathbf{W}_1$ are permuted the same way, then multilayer perceptron computes the same function

$$\mathbf{F}_1(\mathbf{W}_1\mathbf{P}^T\mathbf{F}_2(\mathbf{P}\mathbf{W}_2\mathbf{x} + \mathbf{P}\mathbf{b}_2) + \mathbf{b}_1) = \mathbf{F}_1(\mathbf{W}_1\mathbf{F}_2(\mathbf{W}_2\mathbf{x} + \mathbf{b}_2) + \mathbf{b}_1). \tag{1.56}$$

If the number of hidden units is k, then there are $k!$ such permutations. Two neural networks are congruent if they compute the same function. So for every neural network with k hidden units, there are $k!$ congruent networks. Also, if the function $\mathbf{F}_2$ is odd and $\mathbf{W}_1$ and $\mathbf{W}_2$ can both be multiplied by -1, and it will be equivalent to the original network. Let the matrix $\mathbf{S}$ be a diagonal matrix with either $+1$ or -1 on the main diagonal. If a network has k hidden units, then there are 2^k such matrices possible.

1.4.4 Learning the XOR Function

The architecture of neural networks consists of many simple computational nodes combined in a synergistic manner. Rumelhart and McClelland developed the method by which this is accomplished with the perceptron model [27]. By applying a transfer function to each layer and using the chain rule to derive the learning rule, a multilayer perceptron can be trained in a similar fashion to a perceptron with just one weight matrix and bias vector.

Of course, the question must be asked: What can be achieved with multiple layers that cannot be achieved with a single layer? The problem used to illustrate this is the XOR (i.e., eXclusive-OR) function

$$y = \psi(x_1, x_2) = \begin{cases} -1 & x_1 = x_2 \\ +1 & x_1 \neq x_2, \end{cases} \tag{1.57}$$

where $(x_1, x_2) \in \{-1, +1\} \times \{-1, +1\}$.

A single neuron cannot learn this function because there is not a line that will separate the positive examples $X_1 = \{(-1, +1), (+1, -1)\}$ from negative examples $X_0 = \{(-1, -1), (+1, +1)\}$. There are, of course, two lines that can separate them: $x_1 - x_2 = 1$ and $x_1 - x_2 = 1$ (see fig. 1.7). Both positive examples are a unit distance from both lines. Furthermore, the first and second negative examples are each a distance of 1 from the first and second lines, respectively. However, $(-1, -1)$ is a distance of -3 from the second line and $(1, 1)$ is also a distance of -3 from the first line. The resulting transformation is

$$(-1, -1) \mapsto (-1, -1), \tag{1.58}$$

$$(-1, +1) \mapsto (-3, +1), \tag{1.59}$$

$$(+1, -1) \mapsto (+1, -3), \tag{1.60}$$

$$(+1, +1) \mapsto (-1, -1), \tag{1.61}$$

as shown in fig. 1.8.

The perceptron weight matrix and bias vector are these two line equations stacked on top of each other and multiplied by two

$$\mathbf{W}_1 = \begin{bmatrix} +2 & -2 \\ -2 & +2 \end{bmatrix}, \qquad \mathbf{b}_1 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}. \tag{1.62}$$

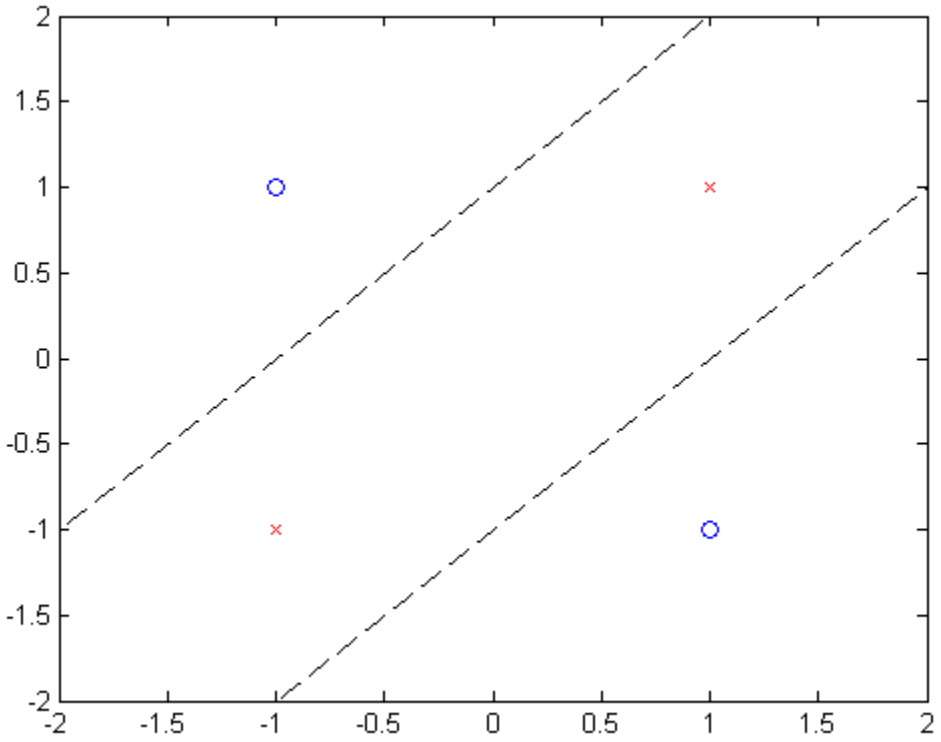

Fig. 1.7: The points of XOR problem space and their separating lines $x_1 - x_2 - 1 = 0$ and $x_2 - x_1 - 1 = 0$.

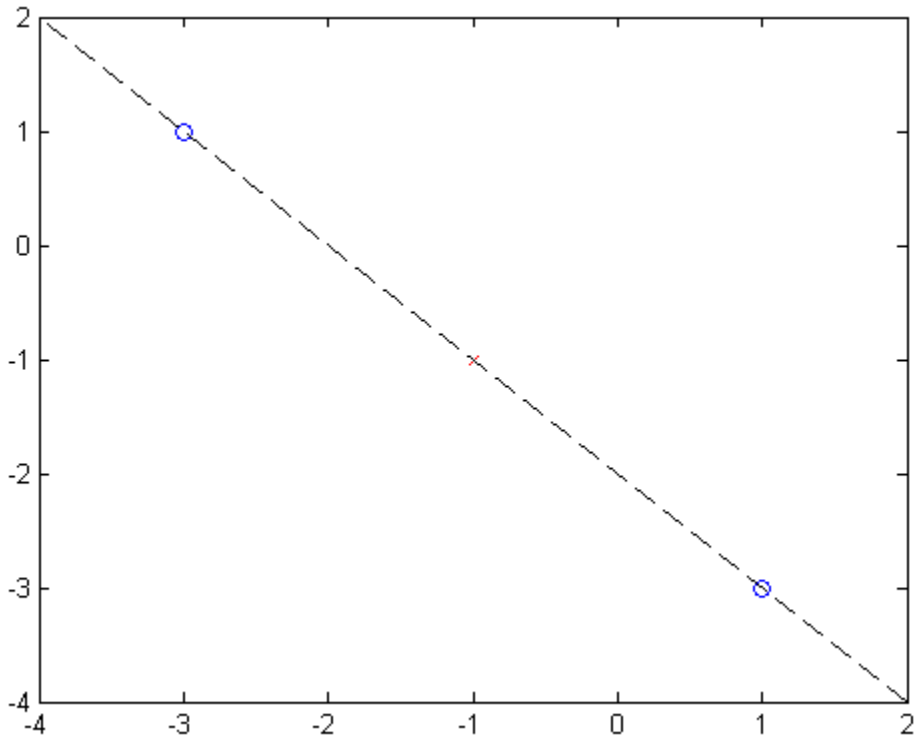

Fig. 1.8: The first affine transformation maps both $(-1,-1)$ and $(+1,+1)$ to $(-1,-1)$. The mapping of all the training points lie on the same line: $x_1 + x_2 = -2$.

By using the hyperbolic tangent function to "squash" the vector components, the four examples are mapped by the perceptron to

$$(-2, -2) \mapsto (-0.96, -0.96), \tag{1.63}$$

$$(-6, +2) \mapsto (-1.00, +0.96), \tag{1.64}$$

$$(+2, -6) \mapsto (+0.96, -1.00), \tag{1.65}$$

$$(-2, -2) \mapsto (-0.96, -0.96). \tag{1.66}$$

These vectors are used as inputs to the second layer that use the line $x_1 + x_2 = -1$ to separate the points further (see fig. 1.9). These coefficients of the line, multiplied by two, then give the output of the multilayer perceptron

$$(-0.96, -0.96) \mapsto \tanh(-1.84) = -0.95, \tag{1.67}$$

$$(-1.00, +0.96) \mapsto \tanh(+1.92) = +0.96, \tag{1.68}$$

$$(+0.96, -1.00) \mapsto \tanh(+1.92) = +0.96, \tag{1.69}$$

$$(-0.96, -0.96) \mapsto \tanh(-1.84) = -0.95. \tag{1.70}$$

Hence, a double-layer perceptron can compute a function that cannot be computed by a single neuron. The output of the network as both inputs are varied between -1 and 1 is shown in fig. 1.10.

If these two perceptrons were composed without the transfer functions, the resulting transformation would be

$$\begin{bmatrix} 2 & 2 \end{bmatrix} \begin{bmatrix} 2 & -2 \\ -2 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 2 & 2 \end{bmatrix} \begin{bmatrix} -2 \\ -2 \end{bmatrix} + 2 = -2. \tag{1.71}$$

It would map all points to the value -2. Thus, the transfer functions not only limit the output values of the neurons, but also warp the transformations to allow further separations.

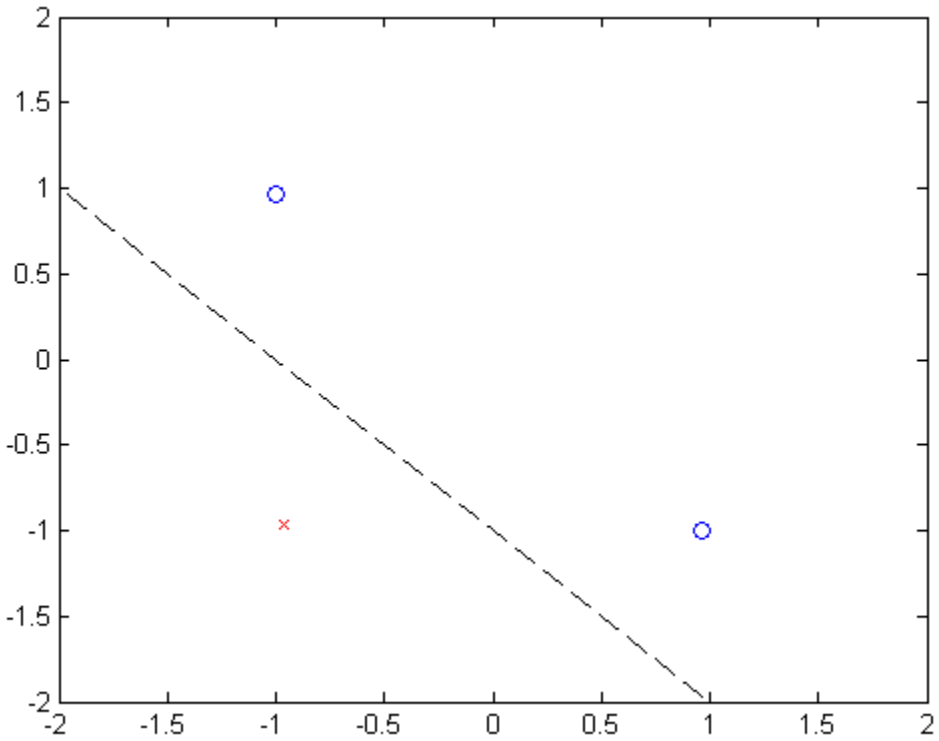

Fig. 1.9: The transfer function warps the points so that the line $x_1 + x_2 = -1$ separates the positive example from the negative ones.

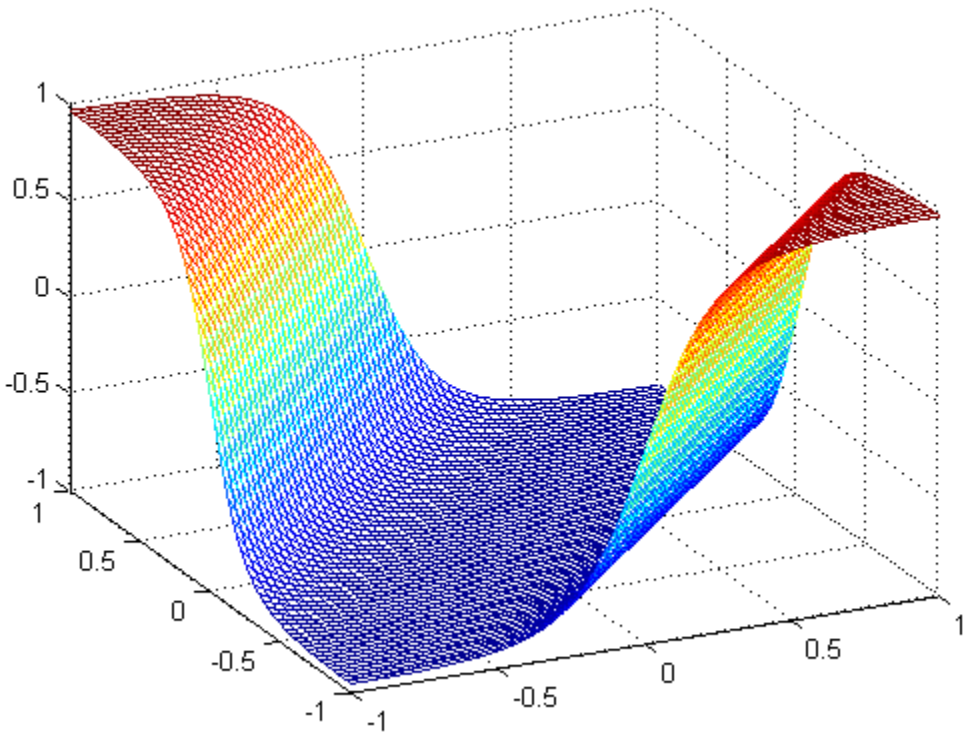

Fig. 1.10: The plot of the output of a double-layer perceptron that computes the XOR function.

1.4.5 The Backpropagation of Error

This numerical method of training multilayer perceptrons was first introduced by the PDP group [27]. The algorithm is quite complicated so to simplify notation matrices and vectors will be used [28] instead of the standard derivation using sums and indices.

It is easiest to first show how two neurons in series are trained together with this algorithm. Then the model is expanded to be two layers of neurons (i.e., perceptrons) in series. The output of this simple network for input x is

$$h(x) = f(w_2 f(w_1 x + b_1) + b_2), \tag{1.72}$$

where $f(x)$ is the logistic function in (1.2). Given N points on the plane,

$$X = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\},$$

the sum-squared objective function is

$$J(w_1, b_1, w_2, b_2) = \frac{1}{2} \sum_{n=1}^{N} \big(y_n - f(w_2 f(w_1 x_n + b_1) + b_2)\big)^2. \tag{1.73}$$

Gradient descent search will be used (1.14, 1.15). The key to the backpropagation algorithm is the application of the chain rule to (1.72). The partial derivatives of J are

$$\frac{\partial J}{\partial w_2} = \sum_{n=1}^{N} -\big(y_n - h(x)\big)h(x)\big(1 - h(x)\big) f(w_1 x_n + b_1), \tag{1.74}$$

$$\frac{\partial J}{\partial b_2} = \sum_{n=1}^{N} -\big(y_n - h(x)\big)h(x)\big(1 - h(x)\big), \tag{1.75}$$

$$\frac{\partial J}{\partial w_1} = \sum_{n=1}^{N} -\big(y_n - h(x)\big)h(x)\big(1 - h(x)\big) w_2 f(w_1 x_n + b_1)\big(1 - f(w_1 x_n + b_1)\big) x_n, \tag{1.76}$$

$$\frac{\partial J}{\partial b_1} = \sum_{n=1}^{N} -\big(y_n - h(x)\big)h(x)\big(1 - h(x)\big) w_2 f(w_1 x_n + b_1)\big(1 - f(w_1 x_n + b_1)\big). \tag{1.77}$$

In general, for K neurons in series, $h_i(x) = f(w_i h_{i-1}(x) + b_i)$, $h_0(x) = x$, the derivative of

weight w_i is the product of the derivative of each neuron and its weights for $j > i$ and the derivative of neuron i multiplied with the output of neuron $i-1$. That product would then be

$$\frac{\partial J}{\partial w_i} = \sum_{n=1}^{N} -\big(y_n - h(x)\big) \prod_{j=i+1}^{K} \big[h_j(x)\big(1-h_j(x)\big)w_j\big] h_i(x_n)\big(1-h_i(x_n)\big)h_{i-1}(x_n). \quad (1.78)$$

For the case of an arbitrary number of inputs, outputs, and hidden nodes, matrices are used in place of the single weights. The order of operations must be changed, too. Thus, in the general case the backpropagation of the error is

$$\frac{\partial J}{\partial \mathbf{W}_i} = \sum_{n=1}^{N} -[\mathbf{F}_i'(\mathbf{W}_i\mathbf{z}_n + \mathbf{b}_i)]^T \prod_{j=i+1}^{K} \mathbf{W}_j^T[\mathbf{F}_j'(\mathbf{W}_j\mathbf{z}_n + \mathbf{b}_j)]^T \mathbf{e}_n\mathbf{z}_n^T, \quad (1.79)$$

where

$$\mathbf{e}_n = \mathbf{y}_n - \mathbf{F}_K(\mathbf{W}_K\mathbf{z}_n + \mathbf{b}_K),$$

and

$$\mathbf{z}_n = \mathbf{F}_{i-1}(\mathbf{W}_{i-1}\cdots\mathbf{F}_1(\mathbf{W}_1\mathbf{x}_n + \mathbf{b}_1) + \mathbf{b}_{i-1}).$$

The dots represent the outputs of the lower layers being fed into the higher layers as inputs.

The important thing to remember about backpropagation is that the gradient for any layer is the outer product of the output error vector and the input to the layer. This rank 1 matrix is multiplied on the left by the Jacobian matrix of the output layer activation map, then the output layer weight matrix, then the Jacobian matrix of the highest hidden layer's activation map and then its weight matrix and so on until the Jacobian matrix of the activation map of the current layer is reached.

1.5 Radial Basis Function Networks

Radial basis function networks [29] are double-layer perceptrons in which the output layer has a linear activation function and the hidden units are Gaussian functions. If there

are m hidden nodes, then the output of the network from input $\mathbf{x}$ is

$$\rho(\mathbf{x}) = \sum_{i=1}^{m} w_i \exp(-\beta \|\mathbf{x} - \mathbf{c}_i\|^2). \tag{1.80}$$

The vectors $\{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_m\}$ are the centroids of the network. The parameter β is a shape parameter usually set *a priori* to scale the outputs of the Gaussian functions. A radial basis function network is said to be *normalized* when

$$\rho(\mathbf{x}) = \frac{\sum_{i=1}^{m} w_i \exp(-\beta \|\mathbf{x} - \mathbf{c}_i\|^2)}{\sum_{i=1}^{m} \exp(-\beta \|\mathbf{x} - \mathbf{c}_i\|^2)}. \tag{1.81}$$

Radial basis function networks are similar to Gaussian mixture models. However, there is no requirement that each basis function be a Gaussian probability density function or that the network compute a probability density function.

1.5.1 An Example Radial Basis Function Network

A radial basis function network used as a classifier for the example problem in sec. 1.3.3 would have $\mathbf{c}_1 = (1.5, 0.5)$, $\mathbf{c}_2 = (0.5, -0.5)$, $\beta = 50$, $w_1 = 1$, and $w_2 = -1$. This network would have the same decision boundary as the neuron. However, it would be less confident about its answers when $\mathbf{x}$ is far away from either $\mathbf{c}_1$ or $\mathbf{c}_2$. The neuron only measures the distance of $\mathbf{x}$ from the decision boundary hyperplane. The radial basis functions measure the distance of $\mathbf{x}$ from each of the m centroids. The shape parameter β was calculated from the variance of the example's data source. The variance is $\sigma^2 = 0.01$. Then $\beta = \frac{1}{2\sigma^2} = 50$.

1.5.2 Training a Radial Basis Function Network

The most rudimentary algorithm for training a radial basis function simply uses each of the training input patterns $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ as the center of a radial basis function. Thus, a training set with n patterns will generate a network with n hidden nodes using this method. To determine the n output weights, solve the linear system: $\mathbf{Gw} = \mathbf{y}$ where

$g_{ij} = \exp(-\beta\|\mathbf{x}_i - \mathbf{x}_j\|^2)$. In more detail, this linear system is

$$\begin{bmatrix} 1 & \exp(-\beta\|\mathbf{x}_1 - \mathbf{x}_2\|^2) & \cdots & \exp(-\beta\|\mathbf{x}_1 - \mathbf{x}_n\|^2) \\ \exp(-\beta\|\mathbf{x}_2 - \mathbf{x}_1\|^2) & 1 & \cdots & \exp(-\beta\|\mathbf{x}_2 - \mathbf{x}_n\|^2) \\ \vdots & \vdots & \ddots & \vdots \\ \exp(-\beta\|\mathbf{x}_n - \mathbf{x}_1\|^2) & \exp(-\beta\|\mathbf{x}_n - \mathbf{x}_2\|^2) & \cdots & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}. \tag{1.82}$$

For large data sets this method is impractical. In such a case it is better to use a *clustering algorithm* like k-means [30]. Randomly select m vectors from the training set inputs $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ as the initial centroids $\{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_m\}$. For each training vector $\mathbf{x}_i$ find the closest centroid $\mathbf{c}_j$ to it. Add all the training vectors together that are closest to centroid $\mathbf{c}_j$. Divide this vector-sum by the number of matched vectors to calculate the new centroid. Follow this procedure once or twice again to refine the centroids. The linear system is now overdetermined with $g_{ij} = \exp(-\beta\|\mathbf{x}_i - \mathbf{c}_j\|^2)$. Use the pseudo-inverse of $\mathbf{G}$ to find the weights.

In a classification problem with prelabeled training vectors, a centroid is calculated for each class. For more than two classes a separate weight vector will be necessary for each class. The output training vectors' components will be either one or zero. If the training vector is in the class associated with the weight vector then the component should be one; otherwise it is zero.

For example, Fisher's iris data set [31,32] has three classes: *Iris setosa*, *Iris versicolor*, and *Iris virginica*. They are labelled 1, 2, and 3, respectively, and the training examples are partitioned into three sets X_1, X_2, and X_3. The three weight vectors would each be a column in a matrix that is multiplied on the left by $\mathbf{G}$. The classifications would appear in the three columns of the matrix $\mathbf{Y}$. The matrix $\mathbf{Y}$ would be defined as

$$y_{ij} = \begin{cases} 1 & \mathbf{x}_i \in X_j, j \in 1, 2, 3 \\ 0 & \text{otherwise.} \end{cases} \tag{1.83}$$

The weight matrix $\mathbf{W}$ would be the solution to the generalized linear system $\mathbf{GW} = \mathbf{Y}$.

The shaping parameter β could be replaced by computing the covariance matrices $\mathbf{R}_1$, $\mathbf{R}_2$, and $\mathbf{R}_3$. Then the elements of the matrix $\mathbf{G}$ would be

$$g_{ij} = \exp\left(-\frac{1}{2}\left[\mathbf{x}_i - \mathbf{c}_j\right]^T \mathbf{R}_j^{-1}\left[\mathbf{x}_i - \mathbf{c}_j\right]\right). \tag{1.84}$$

Thus this radial basis function network would decorrelate the data with respect to each class.

1.6 Conclusion

This chapter was an introduction to neural networks and machine learning. The concept of the artificial neuron was introduced. Two neural network architectures, multilayer perceptrons and radial basis function networks, were described. The next chapter will discuss the theoretical aspects of natural gradient and its application to training multilayer perceptrons.

Chapter 2

Simplified Natural Gradient

This chapter explains the theoretical development of the Simplified Natural Gradient. The theoretical innovation is retaining the algebraic structure of the neural network parameters. These parameters are matrices. Their fundamental action is transforming the input vectors to output vectors between two layers of nodes. It is a common practice to reform these matrix parameters as vectors to facilitate the use of advanced optimization algorithms. However, keeping the structure leads to a simplification of the Adaptive Natural Gradient [33–35].

The Simplified Natural Gradient also uses a regularization technique [12] to prevent its weight matrices from becoming too large as described in sec. 1.3.6. It also uses an annealed learning rate [16] as described in sec. 1.3.8. Before these innovations and their results are discussed in detail, the natural gradient needs to be properly introduced.

2.1 Natural Gradient

A natural gradient algorithm is an algorithm that performs the optimization of a parameter in a Riemannian (i.e., not Euclidean) space. In a Euclidean space the shortest distance between two points is a straight line. However, in a Riemannian space the shortest distance between two points is a curve.

Imagine the sphere with unit radius as a two-dimensional manifold S^2. It is the set of all points in $\mathbb{R}^3$ with unit distance from the origin. Each point on the sphere can be given coordinates with two angles, θ and ϕ. The Euclidean distance between any two points on the sphere, $\mathbf{x}_1$ and $\mathbf{x}_2$, is the measure of a line segment between the two points in $\mathbb{R}^3$. However, only the endpoints of this line segment are part of the sphere S^2. When measuring their distance on the sphere, then the line segment is actually an arc on the sphere. Such

an arc is called a *geodesic*.

Looking at a typical map of the Earth such as the one drawn by Gerardus Mercator in 1569 (see fig. 2.1), it would seem that the shortest distance between two points on that map would be along a straight line between them. However, it is not. All geodesics on a sphere are arcs along a *great circle*. A great circle on S^2 is a circle of radius r with its center at the origin. Hence, the equator and the meridians on the globe are great circles. However, the lines of latitude are *small circles* whose radii are less than r. This is why airline routes veer to the north in the Northern Hemisphere and veer to the south in the Southern Hemisphere; these routes follow great circles.

The goal of a natural gradient algorithm is to follow the curvature of the manifold to find the best solution of a parameter by measuring the direction of steepest descent according to the Riemannian geometry of the parameter space. This direction of descent is natural in the sense that the algorithm follows the natural flow of the geometry.

These geometries typically come about when a parameter is constrained in some way. For example, the two-dimensional sphere S^2 is the set of all vectors $\{\mathbf{x} : \|\mathbf{x}\| = 1\}$. Constraining a parameter to the unit sphere is easily handled by using Lagrange multipliers when solving an optimization in closed form. However, for more complex problems, more sophistication may be required.

2.1.1 A Probabilistic Model

All machine learning algorithms have to deal with uncertainty. There are numerous variations in nature. There is usually some degree of error in most measurements. Sometimes classification may be mislabelled. Regardless of the source or the reason, machine learning algorithms compute statistics about the training data and use that information to determine the optimal parameters for the given problem.

Suppose there is a learning model that is a parameterized function, $\mathbf{F} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^k$, where the first argument is the parameter $\mathbf{w}$ of the function and the second is an input $\mathbf{x}$ to be learned from training data. Let $\mathbf{X}$ be a random training vector with an unknown

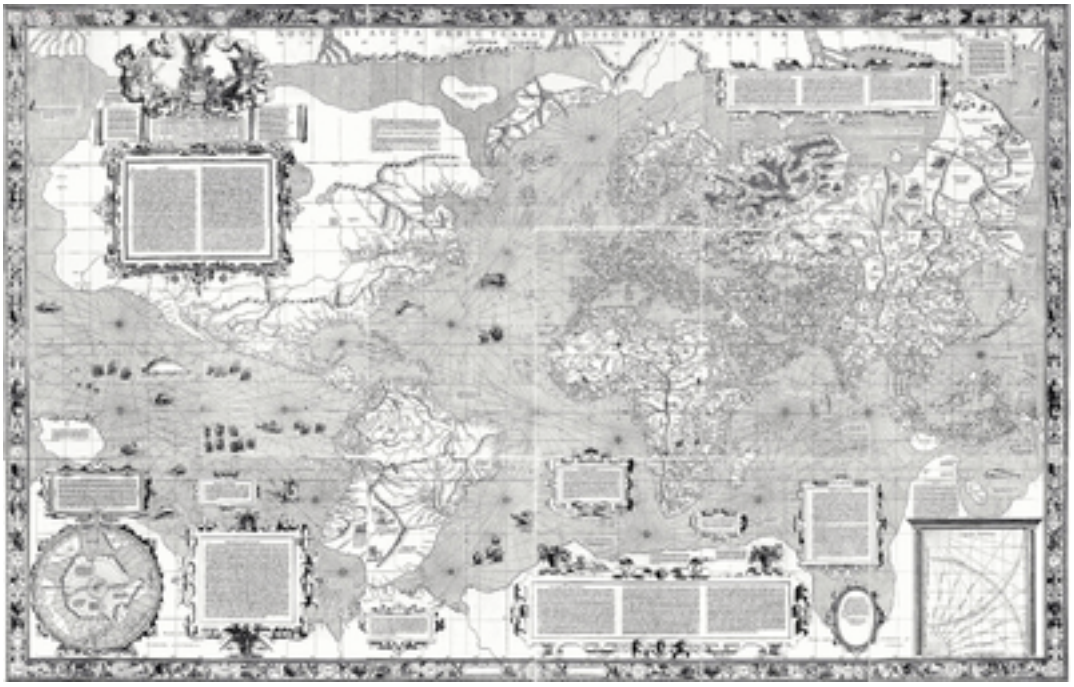

Fig. 2.1: The map of the Earth drawn by Gerardus Mercator in 1569.

density function, $p_{\mathbf{X}}(\mathbf{x})$. Let $\mathbf{Y} = \mathbf{F}(\mathbf{w}, \mathbf{X}) + \zeta$, be a random target vector where $\zeta \sim \mathcal{N}(0, \mathbf{I})$ represents a possible error in the training set. One possible model is

$$p(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N | \mathbf{w}, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N) = (2\pi)^{-\frac{Nk}{2}} \exp\left(-\frac{1}{2} \sum_{i=1}^{N} \|\mathbf{y}_i - \mathbf{F}(\mathbf{w}, \mathbf{x}_i)\|^2 \right), \quad (2.1)$$

where N is the number of training patterns, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ are N samples of $\mathbf{X}$ and $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N$ are N samples of $\mathbf{Y}$. The errors ζ_i are not directly observable. The utility of this model is derived from the fact that it is a Gaussian distribution that is also the exponent of the neural network objective function.[1]

2.1.2 Sufficient Statistics

A *sufficient statistic* is a function of samples from a random variable with a probability distribution with an underlying parameter such that another statistic from the same samples does not reveal further information about the underlying parameter [37]. Formally, this

[1] "Essentially, all models are wrong, but some are useful." [36]

means that the probability distribution with the parameter is statistically independent of the probability distribution given the sufficient statistic [38]. Given a sample x of a random variable X, let $T(x)$ be a statistic derived from that sample. Then $T(x)$ is sufficient for θ if $P(X = x|T(x) = t, \theta) = P(X = x|T(x) = t)$.

There is a way to determine if a statistic is sufficient without proving the previous probability identity. The Fisher-Neyman factorization theorem states that a statistic $T(x)$ is sufficient if and only if there are functions c and a such that

$$f_{X|\Theta}(x|\theta) = c(x)a(T(x), \theta). \tag{2.2}$$

If the probability density function of X given the parameter θ can be factored into a function of the sample(s) times a function of the parameter and the statistic, then the statistic is sufficient.

For example, let X be a random variable that maps to the set $\{0, 1\}$ with a Bernoulli distribution with parameter $0 < p < 1$. Given a sequence of samples $x_1, x_2, \ldots, x_N$ of X, the joint probability mass function of these samples is

$$\begin{aligned} f_X(x_1, x_2, \ldots, x_N|p) &= p^{x_1}(1-p)^{1-x_1}p^{x_2}(1-p)^{1-x_2}\cdots p^{x_N}(1-p)^{1-x_N} \\ &= p^{x_1+x_2+\cdots+x_N}(1-p)^{N-(x_1+x_2+\cdots x_N)}. \end{aligned} \tag{2.3}$$

Let $T(x_1, x_2, \ldots, x_N) = x_1 + x_2 + \cdots + x_N$, then

$$f_X(x_1, x_2, \ldots, x_N|p) = p^{T(x_1,x_2,\ldots,x_N)}(1-p)^{N-T(x_1,x_2,\ldots,x_N)}. \tag{2.4}$$

By choosing $c(x) = 1$ and $a(t, p) = p^t(1-p)^{N-t}$, the probability mass function of X can be factored. Thus, $T(x)$ is sufficient for p.

2.1.3 The Exponential Family

There is a special family of probability distributions called the *exponential family*. Each distribution in the family has a PDF that can be factored as

$$f_{X|\Theta}(x|\theta) = c(x)\exp(\eta(\theta)T(x) - A(\theta)). \tag{2.5}$$

Hence, it is relatively straightforward to find a sufficient statistic for a parameter if the underlying distribution is a member of the exponential family. For example, the Bernoulli distribution with parameter p is a member of the exponential family. Choose $c(x) = 1$ and

$$a(T(x_1, x_2, \cdots, x_N), p) = \exp((\log p - \log(1-p))T(x_1, x_2, \ldots, x_N) + N\log(1-p)). \tag{2.6}$$

Thus, $\eta(p) = \log p - \log(1-p)$ and $A(p) = -N\log(1-p)$.

There are exponential families for random vectors whose distributions have multiple parameters. There are also multiple statistics. Their general form is

$$f_{\mathbf{X}|\Theta}(\mathbf{x}|\theta_1, \theta_2, \ldots, \theta_d) = c(\mathbf{x})\exp\left(\sum_{i=1}^{r}\eta_i(\theta_1, \theta_2, \ldots, \theta_d)T_i(\mathbf{x}) - A(\theta_1, \theta_2, \ldots, \theta_d)\right). \tag{2.7}$$

2.1.4 Optimal Parameter Estimation

If the goal is to find the optimal parameter, $\hat{\theta}$, then the log-likelihood function provides a mechanism through which the parameters can be estimated in terms of the sufficient statistic. The log-likelihood function is the natural logarithm of the probability density function

$$\begin{aligned}\ell(x, \theta) &= \log(f_{X|\Theta}(x|\theta)) \\ &= \log(c(x)) + \eta(\theta)T(x) - A(\theta).\end{aligned} \tag{2.8}$$

To find the optimal parameter, take the derivative of the log-likelihood function

$$\frac{\partial \ell}{\partial \theta} = \frac{\partial \eta}{\partial \theta}T(x) - \frac{\partial A}{\partial \theta}. \tag{2.9}$$

This is called the score function. Like many other optimization problem, the optimal parameter is the one for which the derivative of the function vanishes. In estimation problems, the derivative is the score function, so the best parameter has a score of zero. For example, the score function of the Bernoulli distribution is

$$\frac{\partial \ell}{\partial p} = \frac{T(x_1, x_2, \ldots, x_N)}{p(1-p)} - \frac{N}{1-p} = 0. \tag{2.10}$$

Solving for p gives the optimal parameter estimate, $\hat{p} = \frac{1}{N} T(x_1, x_2, \ldots, x_N) = \frac{1}{N} \sum_{i=1}^{N} x_i$. This is the maximum likelihood (ML) estimate of p. The score function of the PDF in (2.1) is the vector-valued function

$$\frac{\partial \ell}{\partial \mathbf{w}} = \sum_{i=1}^{N} \left(\frac{\partial \mathbf{F}}{\partial \mathbf{w}} \right)^T (\mathbf{y}_i - \mathbf{F}(\mathbf{w}, \mathbf{x}_i)), \tag{2.11}$$

where $\frac{\partial \mathbf{F}}{\partial \mathbf{w}} = \left[\frac{\partial f_i}{\partial w_j} \right]$ is the Jacobian matrix of $\mathbf{F}$. The optimal parameter cannot be found directly by solving (2.11). A gradient descent algorithm is commonly used.

2.1.5 The Fisher Information

The performance of a gradient descent algorithm may be improved significantly by using Newton's method [22]. Newton's method divides the first derivative of the function by its second derivative. The score function of a random variable from a distribution with a single parameter is

$$\frac{\partial \ell}{\partial \theta} = \frac{1}{f_{X|\Theta}(x|\theta)} \frac{\partial f_{X|\Theta}}{\partial \theta}. \tag{2.12}$$

Hence, the second derivative is

$$\frac{\partial^2 \ell}{\partial \theta^2} = -\frac{1}{\left(f_{X|\Theta}(x|\theta)\right)^2} \left[\frac{\partial f_{X|\Theta}}{\partial \theta} \right]^2 + \frac{1}{f_{X|\Theta}(x|\theta)} \frac{\partial^2 f_{X|\Theta}}{\partial \theta^2}. \tag{2.13}$$

For a log-likelihood function with multiple parameters, its gradient vector is multiplied by the inverse of its Hessian matrix. This works well for points in the parameter space close to a minimum. However, when one or more of the eigenvalues of the Hessian matrix approach

zero, then Newton's method will fail because of division by a number close to zero.

With maximum likelihood estimation, the Fisher information $G(\theta)$ is used instead of the Hessian matrix. It is the expected value of the second derivative of the log-likelihood function

$$G(\theta) = E\left\{\frac{\partial^2 \ell}{\partial \theta^2}\middle|\theta\right\} = -\int_{-\infty}^{\infty}\left[\frac{\partial \ell}{\partial \theta}\right]^2 f_{X|\Theta}(x|\theta)dx + \int_{-\infty}^{\infty}\frac{\partial^2 f_{X|\Theta}}{\partial \theta^2}dx. \tag{2.14}$$

When the second term of (2.14) is required to be zero as a regularity constraint, then the formal definition of the Fisher information is

$$G(\theta) = E\left\{\left[\frac{\partial \ell}{\partial \theta}\right]^2\middle|\theta\right\}. \tag{2.15}$$

Since the mean of the score is zero, it is also the variance of the score [39,40]. For example, the Fisher information of the Bernoulli distribution with parameter p is

$$\begin{aligned} G(p) &= -E\left\{\frac{\partial^2 \ell}{\partial p^2}\middle|p\right\} \\ &= E\left\{\frac{(1-2p)T(x_1,x_2,\ldots,x_N)}{p^2(1-p)^2}\middle|p\right\} + \frac{N}{(1-p)^2} \\ &= \frac{Np(1-2p)}{p^2(1-p)^2} + \frac{N}{(1-p)^2} \\ &= \frac{N}{p(1-p)}. \end{aligned} \tag{2.16}$$

Dividing the score function of the Bernoulli distribution by the Fisher information simplifies the terms

$$\frac{1}{G(p)}\frac{\partial \ell}{\partial p} = \frac{1}{N}T(x_1,x_2,\ldots,x_n) - p = 0. \tag{2.17}$$

When the parameter is a vector, then the Fisher information is a matrix. The matrix is the expected value of the *outer product* of the score function with itself

$$\mathbf{G}(\theta_1,\theta_2,\ldots,\theta_d) = E\left\{\nabla_\theta \ell (\nabla_\theta \ell)^T | \theta_1,\theta_2,\ldots,\theta_d\right\}. \tag{2.18}$$

Except for a few pathological cases, the Fisher information matrix is positive definite.

2.1.6 The Geometry of Exponential Families

For an exponential family there is an affine geometry on its parameters [41, 42]. The points in this geometry are probability measures. For each point p in the set of all probability measures P there is a tangent space T_pP. The tangent vectors are the random variables defined with the probability measure p. Random variables are functions that map events to the real numbers. If the event space is defined as the real numbers, then the random variables will also transform one probability measure into another. The action of a random variable X on a probability measure p is $p + X = \exp(X)p$. The action follows the rules of addition. If Y is another random variable, then $(p + X) + Y = \exp(Y)\exp(X)p = \exp(Y + X)p = p + (X + Y)$.

A coordinate system can be chosen for this affine geometry. These coordinates are the parameters for the probability measures in P. Suppose that the vector components $(\theta^1, \ldots, \theta^r)$ are the coordinates for a probability measure p, then there are r random variables $X_1, \ldots, X_r$ that are sufficient statistics for $\theta^1, \ldots, \theta^r$. Since P is an exponential family,

$$p = \exp(\theta^1 X_1 + \cdots + \theta^r X_r - K). \tag{2.19}$$

Coordinates such as these are the canonical coordinates. Any coordinate system can be transformed into a canonical coordinate system by selecting the appropriate functions.

There is a log-likelihood function $\ell(p)$ associated with each probability measure p in P. It is the natural logarithm of p

$$\ell(p) = \theta^1 X_1 + \cdots + \theta^r X_r - K. \tag{2.20}$$

Furthermore, the score functions are

$$\frac{\partial \ell}{\partial \theta^i} = X_i. \tag{2.21}$$

Thus, the score functions form a basis for the tangent space T_pP. An inner product can be defined on T_pP. For two random variables Y and Z, the inner product is defined as

$$\langle Y, Z \rangle = E_p[YZ], \tag{2.22}$$

where E_p is the expectation operator using the probability measure p. The elements of the Fisher information matrix are

$$g_{ij} = E_p[X_i X_j] = \langle X_i, X_j \rangle. \tag{2.23}$$

The matrix of the inner products of the basis vectors of a tangent space is also the Riemannian metric for a metric space. Thus, for an exponential family used as a metric space, the metric is the Fisher information matrix.

2.1.7 Direction of Steepest Descent

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, its gradient ∇f is the direction of positive increase or ascent of the function f in $\mathbb{R}^n$. Clearly, $-\nabla f$ would be the direction of descent. Thus, any vector $v \in \mathbb{R}^n$, is a descent direction if $\langle v, \nabla f \rangle < 0$. Let $h(v) = \langle v, \nabla f \rangle$ be a linear functional. Clearly, $h(v)$ reaches its minimum when $v = -\nabla f$ since $h(-\nabla f) = -\|\nabla f\|^2$.

For any probability measure p, the gradient of its log-likelihood function is likewise a tangent vector. Thus, the gradient $\nabla(\log p) = \sum_{i=1}^{r} \sigma^i X_i$ where σ^i is the score of p on parameter i. Similarly, a random variable V is a descent direction if $\langle V, \nabla(\log p) \rangle < 0$. Thus, if $V = \sum_{i=1}^{r} \tau^i X_i$, then the inner product is

$$\langle V, \nabla(\log p) \rangle = \sum_{i=1}^{r} \sum_{j=1}^{r} g_{ij} \sigma^i \tau^j. \tag{2.24}$$

The steepest descent direction will always be

$$V = -G^{-1} \nabla(\log p), \tag{2.25}$$

where G is the Fisher information matrix.

2.2 Amari's Adaptive Natural Gradient for Multilayer Perceptrons

The Adaptive Natural Gradient Learning (ANGL) algorithm for multilayer perceptrons [34, 35] is a Newton-type algorithm that uses the Fisher information matrix instead of the Hessian matrix. It is an online learning algorithm that performs an autoregressive estimate of the Fisher information matrix with a learning rate ϵ_t

$$\mathbf{G}_{t+1} = (1-\epsilon_t)\mathbf{G}_t + \epsilon_t \left(\frac{\partial \mathbf{F}(\mathbf{w}, \mathbf{x}_i)}{\partial \mathbf{w}}\right)^T (\mathbf{y}_i - \mathbf{F}(\mathbf{w}, \mathbf{x}_i))(\mathbf{y}_i - \mathbf{F}(\mathbf{w}, \mathbf{x}_i))^T \left(\frac{\partial \mathbf{F}(\mathbf{w}, \mathbf{x}_i)}{\partial \mathbf{w}}\right). \quad (2.26)$$

Instead of inverting the Fisher information after each pattern, the inverse is updated as a rank-1 update using the matrix inversion lemma [40]

$$\mathbf{G}_{t+1}^{-1} = (1+\epsilon_t)\mathbf{G}_t^{-1} - \epsilon_t \frac{\mathbf{G}_t^{-1}\left(\frac{\partial \mathbf{F}(\mathbf{w}, \mathbf{x}_i)}{\partial \mathbf{w}}\right)^T (\mathbf{y}_i - \mathbf{F}(\mathbf{w}, \mathbf{x}_i))(\mathbf{y}_i - \mathbf{F}(\mathbf{w}, \mathbf{x}_i))^T \left(\frac{\partial \mathbf{F}(\mathbf{w}, \mathbf{x}_i)}{\partial \mathbf{w}}\right)\mathbf{G}_t^{-1}}{1 + (\mathbf{y}_i - \mathbf{F}(\mathbf{w}, \mathbf{x}_i))^T \left(\frac{\partial \mathbf{F}(\mathbf{w}, \mathbf{x}_i)}{\partial \mathbf{w}}\right)\mathbf{G}_t^{-1}\left(\frac{\partial \mathbf{F}(\mathbf{w}, \mathbf{x}_i)}{\partial \mathbf{w}}\right)^T (\mathbf{y}_i - \mathbf{F}(\mathbf{w}, \mathbf{x}_i))}. \quad (2.27)$$

Amari shows that the best adaptation rate is $\epsilon_t = \frac{1}{t}$. It must be noted that the Adaptive Natural Gradient Learning algorithm uses the common convention of ordering all the parameters into one large vector. Let the homomorphism ϕ denote this operation on the parameters of a multilayer perceptron. Then for the parameters, $\mathbf{W}_1$, $\mathbf{b}_1$, etc., there is a vector $\mathbf{w}$ such that

$$\phi(\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \dots) = \mathbf{w}. \quad (2.28)$$

Typically, the columns of each matrix are stacked on top of each other with the bias vectors interleaved between the matrices. Regardless, this has the effect of making the Fisher information matrix very large. However, maintaining this structure may yield other benefits.

2.3 Algebraic Structure of Multilayer Perceptrons

As was discussed in sec. 1.4.5, each layer of a multilayer perceptron is an affine transformation with a transfer function that restricts the output of each layer to either the

interval $[0, 1]$ or the interval $[-1, 1]$. Understanding how the derivatives work is necessary to developing any kind of gradient descent algorithm. Let $\mathbf{W}$ be the weight matrix for a single-layer perceptron with $\mathbf{b}$ as its bias and $\mathbf{F}$ as its transfer function. Let $\mathbf{x}$ and $\mathbf{y}$ be a generic input-output pair from a suitable training set. The squared error of this pair is $J(\mathbf{W}, \mathbf{b}) = \frac{1}{2}\|\mathbf{y} - \mathbf{F}(\mathbf{W}\mathbf{x} + \mathbf{b})\|^2$.

2.3.1 Directional Derivative

This section describes in detail how the derivatives of the function J are computed. This will lead into developing the Fisher information matrix and the natural gradient of a multilayer perceptron. The first step is to find the directional derivative of J in the direction of steepest descent.

Given a weight matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$, assume that the matrix $\mathbf{V}$ and the vector $\mathbf{c}$ are the steepest descent direction from $(\mathbf{W}, \mathbf{b})$ toward the optimal solution. Let $\gamma(t) = \mathbf{W} + t\mathbf{V}$ be the line through the space of possible weight matrices and $\theta(t) = \mathbf{b} + t\mathbf{c}$ be the line through the space of possible bias vectors that follows the steepest descent direction. Then their derivatives are $\dot{\gamma}(t) = \mathbf{V}$ and $\dot{\theta}(t) = \mathbf{c}$. By composing J with the tuple $(\gamma(t), \theta(t))$ the real-valued function of the single parameter, t, is defined, $J(\gamma(t), \theta(t))$. The derivative of this function is

$$\frac{d}{dt}(J(\gamma(t), \theta(t))) = \lim_{t \to 0} \frac{J(\mathbf{W} + t\mathbf{V}, \mathbf{b} + t\mathbf{c}) - J(\mathbf{W}, \mathbf{b})}{t}. \tag{2.29}$$

There is a problem with (2.29). The variable t cannot be directly factored out of the expression $J(\mathbf{W} + t\mathbf{V}, \mathbf{b} + t\mathbf{c})$. However, since the transfer function $\mathbf{F}$ is differentiable, its Jacobian matrix can be used in a first-order Taylor series of J. This first-order Taylor series of J expanded about $\mathbf{W}$ is

$$J(\mathbf{W} + t\mathbf{V}, \mathbf{b}) = J(\mathbf{W}, \mathbf{b}) + \left\langle \frac{\partial J}{\partial \mathbf{W}}, t\mathbf{V} \right\rangle, \tag{2.30}$$

and the first-order Taylor series of J expanded about $\mathbf{b}$ is

$$J(\mathbf{W}, \mathbf{b} + t\mathbf{c}) = J(\mathbf{W}, \mathbf{b}) + \left\langle \frac{\partial J}{\partial \mathbf{b}}, t\mathbf{c} \right\rangle. \tag{2.31}$$

Therefore, using (2.29), (2.30), and (2.31), the directional derivative of J in the direction of $(\mathbf{V}, \mathbf{c})$ is

$$\frac{dJ}{dt} = \frac{d}{dt} J(\gamma(t), \theta t) = \left\langle \frac{\partial J}{\partial \mathbf{W}}, \mathbf{V} \right\rangle + \left\langle \frac{\partial J}{\partial \mathbf{b}}, \mathbf{c} \right\rangle. \tag{2.32}$$

The next step is to find the partial derivatives $\frac{\partial J}{\partial \mathbf{W}}$ and $\frac{\partial J}{\partial \mathbf{b}}$. If t, $\mathbf{V}$ and $\mathbf{c}$ can be isolated, then t will vanish in the limit, the matrix in the inner product with $\mathbf{V}$ will be $\frac{\partial J}{\partial \mathbf{W}}$ and the vector in the inner product with $\mathbf{c}$ will be $\frac{\partial J}{\partial \mathbf{b}}$. This is accomplished by using the first-order Taylor series of $\mathbf{F}$. Expanding $\mathbf{F}$ about $\mathbf{W}$ is

$$\mathbf{F}((\mathbf{W} + t\mathbf{V})\mathbf{x} + \mathbf{b}) = \mathbf{F}(\mathbf{W}\mathbf{x} + \mathbf{b}) + t\mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})\mathbf{V}\mathbf{x}, \tag{2.33}$$

where $\mathbf{F}'(\mathbf{z}) = \left[\frac{\partial f_i}{\partial z_j}\right]$ is the Jacobian matrix of $\mathbf{F}$. Similarly, the first-order Taylor series expansion of $\mathbf{F}$ about $\mathbf{b}$ is

$$\mathbf{F}(\mathbf{W}\mathbf{x} + \mathbf{b} + t\mathbf{c}) = \mathbf{F}(\mathbf{W}\mathbf{x} + \mathbf{b}) + t\mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})\mathbf{c}. \tag{2.34}$$

The composition of the objective function J with the curve $\gamma(t)$ is

$$J(\gamma(t), \mathbf{b}) = J(\mathbf{W} + t\mathbf{V}, \mathbf{b}) = \frac{1}{2}\|\mathbf{y} - \mathbf{F}((\mathbf{W} + t\mathbf{V})\mathbf{x} + \mathbf{b})\|^2. \tag{2.35}$$

The partial derivative with respect to $\mathbf{W}$ can be found by differentiating with respect to t. Thus, the derivative is

$$\begin{aligned}
\frac{d}{dt}J(\mathbf{W}+t\mathbf{V},\mathbf{b}) &= -\left\langle \mathbf{y}, \frac{d}{dt}\mathbf{F}((\mathbf{W}+t\mathbf{V})\mathbf{x}+\mathbf{b})\right\rangle + \frac{1}{2}\frac{d}{dt}\left\|\mathbf{F}((\mathbf{W}+t\mathbf{V})\mathbf{x}+\mathbf{b})\right\|^2 \\
&= -\left\langle \mathbf{y}, \mathbf{F}'(\mathbf{W}\mathbf{x}+\mathbf{b})\mathbf{V}\mathbf{x}\right\rangle + \left\langle \mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b}), \mathbf{F}'(\mathbf{W}\mathbf{x}+\mathbf{b})\mathbf{V}\mathbf{x}\right\rangle \\
&= -\left\langle \mathbf{V}\mathbf{x}, \mathbf{F}'(\mathbf{W}\mathbf{x}+\mathbf{b})^T(\mathbf{y}-\mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b}))\right\rangle \\
&= -\mathbf{x}^T\mathbf{V}^T\mathbf{F}'(\mathbf{W}\mathbf{x}+\mathbf{b})^T(\mathbf{y}-\mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b})) \\
&= -\operatorname{tr}\left(\mathbf{V}^T\mathbf{F}'(\mathbf{W}\mathbf{x}+\mathbf{b})^T(\mathbf{y}-\mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b}))\mathbf{x}^T\right) \\
&= -\left\langle \mathbf{V}, \mathbf{F}'(\mathbf{W}\mathbf{x}+\mathbf{b})^T(\mathbf{y}-\mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b}))\mathbf{x}^T\right\rangle,
\end{aligned} \tag{2.36}$$

where the last inner product in (2.36) is the Matrix Inner Product (i.e., $\langle A, B\rangle = \operatorname{tr}(A^TB)$). Similarly, the composition of J with $\theta(t)$ is

$$J(\mathbf{W},\theta(t)) = J(\mathbf{W},\mathbf{b}+t\mathbf{c}) = \frac{1}{2}\|\mathbf{y}-\mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b}+t\mathbf{c})\|^2. \tag{2.37}$$

It's derivative with respect to t is

$$\begin{aligned}
\frac{d}{dt}J(\mathbf{W},\mathbf{b}+t\mathbf{c}) &= -\left\langle \mathbf{y}, \frac{d}{dt}\mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b}+t\mathbf{c})\right\rangle + \frac{1}{2}\frac{d}{dt}\left\|\mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b}+t\mathbf{c})\right\|^2 \\
&= -\left\langle \mathbf{y}, \mathbf{F}'(\mathbf{W}\mathbf{x}+\mathbf{b})\mathbf{c}\right\rangle + \left\langle \mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b}), \mathbf{F}'(\mathbf{W}\mathbf{x}+\mathbf{b})\mathbf{c}\right\rangle \\
&= -\left\langle \mathbf{c}, \mathbf{F}'(\mathbf{W}\mathbf{x}+\mathbf{b})^T(\mathbf{y}-\mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b}))\right\rangle.
\end{aligned} \tag{2.38}$$

Therefore, the partial derivatives with repect to the weight matrix $\mathbf{W}$ and the bias vector $\mathbf{b}$ are

$$\frac{\partial J}{\partial \mathbf{W}} = -\mathbf{F}'(\mathbf{W}\mathbf{x}+\mathbf{b})^T(\mathbf{y}-\mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b}))\mathbf{x}^T, \tag{2.39}$$

$$\frac{\partial J}{\partial \mathbf{b}} = -\mathbf{F}'(\mathbf{W}\mathbf{x}+\mathbf{b})^T(\mathbf{y}-\mathbf{F}(\mathbf{W}\mathbf{x}+\mathbf{b})). \tag{2.40}$$

Please note that the dimensions of both partial derivatives are those of the parameters $\mathbf{W}$ and $\mathbf{b}$, respectively. Typically, in almost all neural network training algorithms, all the

parameters are stacked into one large vector. However, these parameters are used as affine transformations. Therefore, their algebraic structure is preserved.

2.3.2 The Fisher Information Matrix of a Single-Layer Perceptron

Let $\mathbf{e} = \mathbf{y} - \mathbf{F}(\mathbf{W}\mathbf{x} + \mathbf{b})$ be the $\mathcal{N}(0, \mathbf{I})$ random variable that is the error of the perceptron for $\mathbf{x}$ and $\mathbf{y}$. The Fisher information matrix using this model is then

$$\begin{aligned}
\mathbf{G}(\mathbf{W}, \mathbf{b}) &= E\left\{ \nabla J (\nabla J)^T \middle| \mathbf{W}, \mathbf{b}, \mathbf{x} \right\} \\
&= E\left\{ \left[\frac{\partial J}{\partial \mathbf{W}} \frac{\partial J}{\partial \mathbf{b}} \right] \left[\frac{\partial J}{\partial \mathbf{W}} \frac{\partial J}{\partial \mathbf{b}} \right]^T \middle| \mathbf{W}, \mathbf{b}, \mathbf{x} \right\} \\
&= E\left\{ \frac{\partial J}{\partial \mathbf{W}} \left(\frac{\partial J}{\partial \mathbf{W}} \right)^T + \frac{\partial J}{\partial \mathbf{b}} \left(\frac{\partial J}{\partial \mathbf{b}} \right)^T \middle| \mathbf{W}, \mathbf{b}, \mathbf{x} \right\} \\
&= E\left\{ \mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})^T \mathbf{e} \mathbf{x}^T \left(\mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})^T \mathbf{e} \mathbf{x}^T \right)^T \middle| \mathbf{W}, \mathbf{b}, \mathbf{x} \right\} \\
&\quad + E\left\{ \mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})^T \mathbf{e} \left(\mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})^T \mathbf{e} \right)^T \middle| \mathbf{W}, \mathbf{b}, \mathbf{x} \right\} \\
&= \left(\mathbf{x}^T \mathbf{x} + 1 \right) \mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})^T E\left\{ \mathbf{e}\mathbf{e}^T \middle| \mathbf{W}, \mathbf{b}, \mathbf{x} \right\} \mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b}) \\
&= \left(\mathbf{x}^T \mathbf{x} + 1 \right) \mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})^T \mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b}).
\end{aligned} \tag{2.41}$$

This version of the Fisher information matrix is more compact than that used in (2.26). The gradient used in the ANGL is a vector with a component for every weight matrix element and bias vector component. The outer product of that gradient with itself is a large, rank-1 matrix. In contrast, the gradient computed in (2.39) is the same size as $\mathbf{W}$ and in (2.40) the gradient has the same dimension as $\mathbf{b}$. Therefore, the Fisher information matrix in (2.41) is square with the number of rows and number of columns both equal to the number of outputs of the neural network. Now the gradient of J is

$$\nabla_{\mathbf{W}} J = -\frac{\left(\mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})^T \mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b}) \right)^{-1} \mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})^T \mathbf{e} \mathbf{x}^T}{\mathbf{x}^T \mathbf{x} + 1}, \tag{2.42}$$

$$\nabla_{\mathbf{b}} J = -\frac{\left(\mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})^T \mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b}) \right)^{-1} \mathbf{F}'(\mathbf{W}\mathbf{x} + \mathbf{b})^T \mathbf{e}}{\mathbf{x}^T \mathbf{x} + 1}. \tag{2.43}$$

Hence, in this example, the natural gradient is the product the pseudo-inverse of the

Jacobian matrix of $\mathbf{F}$ and the outer product of the errors and the inputs all divided by the squared length of the input vectors offset by one. Given that the Jacobian matrices of most transfer functions used in neural networks are square and diagonal,[2] the pseudo-inverse is really just the inverse of a diagonal matrix.

When there is more than one training example (a very desirable situation), then the computation of the Fisher information matrix is more complicated. For online learning, (2.42) and (2.43) can be used on each individual training pattern. Assuming that the Jacobian matrix is square, real-symmetric and invertible, the natural gradient simplifies to

$$\nabla_{\mathbf{W}} J_k = -\frac{\mathbf{F}'(\mathbf{W}\mathbf{x}_k + \mathbf{b})^{-1}\mathbf{e}_k\mathbf{x}_k^T}{\mathbf{x}_k^T\mathbf{x}_k + 1}, \tag{2.44}$$

$$\nabla_{\mathbf{b}} J_k = -\frac{\mathbf{F}'(\mathbf{W}\mathbf{x}_k + \mathbf{b})^{-1}\mathbf{e}_k}{\mathbf{x}_k^T\mathbf{x}_k + 1}. \tag{2.45}$$

One problem with this formulation is that as the weights become saturated, the Jacobian matrix becomes nearly-singular. Even more problematic is its implicit reliance on the accuracy of the model.

A more robust approach starts from the objective function,

$$J(\mathbf{W}, \mathbf{b}) = \frac{1}{2}\sum_{i=1}^{N} \|\mathbf{y}_i - \mathbf{F}(\mathbf{W}\mathbf{x}_i + \mathbf{b})\|^2, \tag{2.46}$$

that measures the sum-squared error. Using the derivatives calculated in (2.39) and (2.40), the partial derivatives of the sum-squared error are

$$\frac{\partial J}{\partial \mathbf{W}} = -\sum_{i=1}^{N} \mathbf{F}'(\mathbf{W}\mathbf{x}_i + \mathbf{b})^T(\mathbf{y}_i - \mathbf{F}(\mathbf{W}\mathbf{x}_i + \mathbf{b}))\mathbf{x}_i^T, \tag{2.47}$$

$$\frac{\partial J}{\partial \mathbf{b}} = -\sum_{i=1}^{N} \mathbf{F}'(\mathbf{W}\mathbf{x}_i + \mathbf{b})^T(\mathbf{y}_i - \mathbf{F}(\mathbf{W}\mathbf{x}_i + \mathbf{b})). \tag{2.48}$$

[2] One exception to this is the soft-max function. Its Jacobian matrix is square and real-symmetric.

Let $\mathbf{e}_i = \mathbf{y}_i - \mathbf{F}(\mathbf{W}\mathbf{x}_i + \mathbf{b})$ be the error of training example i. The unbiased estimate of the Fisher information matrix is

$$\hat{\mathbf{G}} = \frac{1}{N-1} \sum_{i=1}^{N} \left(\mathbf{x}_i^T \mathbf{x}_i + 1\right) \mathbf{F}'(\mathbf{W}\mathbf{x}_i + \mathbf{b})^T \mathbf{e}_i \mathbf{e}_i^T \mathbf{F}'(\mathbf{W}\mathbf{x}_i + \mathbf{b}). \tag{2.49}$$

This model will be able to adjust to skewed data sets and noisy data by estimating the mean and variance of the error and using them in its estimate of the Fisher information matrix.

2.3.3 The Fisher Information Matrix of a Multilayer Perceptron

So far, the focus has been the Fisher information matrix of a single-layer network. This section will generalize the model to multilayer perceptrons. An L-layer feed-forward network is functionally an alternating sequence of matrices and nonlinear transfer functions. The output of layer $l \in \{1, 2, \ldots, L\}$ depends on its current input, the output of layer $l-1$, along with the parameters $\mathbf{W}_l$ and $\mathbf{b}_l$. These are combined into a vector that the nonlinear transfer function, $\mathbf{F}_l$, squashes. There is a recurrence relation between the outputs of succeeding layers in the network. Let $\varphi_0(\mathbf{x}_n) = \mathbf{x}_n$ represent the inputs to the network. Then each layer l has the output $\varphi_l(\mathbf{x}_n) = \mathbf{F}_l(\mathbf{W}_l \varphi_{l-1}(\mathbf{x}_n) + \mathbf{b}_l)$. This allows the objective to be written simply as

$$J(\mathbf{W}_1, \ldots, \mathbf{W}_L, \mathbf{b}_1, \ldots, \mathbf{b}_L) = \frac{1}{2} \sum_{i=1}^{N} \|\mathbf{y}_i - \varphi_L(\mathbf{x}_i)\|^2. \tag{2.50}$$

Using (2.39), (2.40), and the chain rule, the partial derivatives of the objective function J with respect to the output layer weights, output layer bias, last hidden layer weights, and

last hidden layer bias, respectively, are

$$\frac{\partial J}{\partial \mathbf{W}_L} = -\sum_{i=1}^{N} \varphi'_L(\mathbf{x}_n)^T(\mathbf{y}_n - \varphi_L(\mathbf{x}_n))\varphi_{L-1}(\mathbf{x}_n)^T, \tag{2.51}$$

$$\frac{\partial J}{\partial \mathbf{b}_L} = -\sum_{i=1}^{N} \varphi'_L(\mathbf{x}_n)^T(\mathbf{y}_n - \varphi_L(\mathbf{x}_n)), \tag{2.52}$$

$$\frac{\partial J}{\partial \mathbf{W}_{L-1}} = -\sum_{i=1}^{N} \varphi'_{L-1}(\mathbf{x}_n)^T\mathbf{W}_L^T\varphi'_L(\mathbf{x}_n)^T(\mathbf{y}_n - \varphi_L(\mathbf{x}_n))\varphi_{L-2}(\mathbf{x}_n)^T, \tag{2.53}$$

$$\frac{\partial J}{\partial \mathbf{b}_{L-1}} = -\sum_{i=1}^{N} \varphi'_{L-1}(\mathbf{x}_n)^T\mathbf{W}_L^T\varphi'_L(\mathbf{x}_n)^T(\mathbf{y}_n - \varphi_L(\mathbf{x}_n)). \tag{2.54}$$

Just as the output of each layer l of the network depends not only on its own weights and biases, but on the weights and biases of the previous layers, the backpropagation of the error depends on the weights of successive layers. This is best expressed as a recurrent relation as

$$\zeta_L(\mathbf{x}_n) = \varphi'_L(\mathbf{x}_n), \tag{2.55}$$

$$\zeta_{L-1}(\mathbf{x}_n) = \zeta_L(\mathbf{x}_n)\mathbf{W}_L\varphi'_{L-1}(\mathbf{x}_n), \tag{2.56}$$

$$\zeta_l(\mathbf{x}_n) = \zeta_{l+1}(\mathbf{x}_n)\mathbf{W}_{l+1}\varphi'_l(\mathbf{x}_n). \tag{2.57}$$

The image of the map ζ_l is a backpropagation transformation. This will transform the error vector such that the change in a weight matrix depends on the strength of each connection [27]. By substituting these formulae into the equations of the partial derivatives, the equations can be written as

$$\frac{\partial J}{\partial \mathbf{W}_l} = -\sum_{i=1}^{N} \zeta_l(\mathbf{x}_n)^T(\mathbf{y}_n - \varphi_L(\mathbf{x}_n))\varphi_{l-1}(\mathbf{x}_n)^T, \tag{2.58}$$

$$\frac{\partial J}{\partial \mathbf{b}_l} = -\sum_{i=1}^{N} \zeta_l(\mathbf{x}_n)^T(\mathbf{y}_n - \varphi_L(\mathbf{x}_n)). \tag{2.59}$$

The Fisher information matrix of a multilayer perceptron can be conceived as an $L \times L$ block matrix with each matrix in the diagonal being the Fisher information matrix of a

layer's parameters. Thus, the Fisher information matrix for the network parameters of layer l would be

$$\hat{\mathbf{G}}_l = \frac{1}{N-1}\sum_{i=1}^{N}\left(\varphi_{l-1}(\mathbf{x}_i)^T\varphi_{l-1}(\mathbf{x}_i)+1\right)\zeta_l(\mathbf{x}_i)^T\mathbf{e}_i\mathbf{e}_i^T\zeta_l(\mathbf{x}_i). \tag{2.60}$$

Inverting this matrix involves only inverting the blocks along the main diagonal. The learning rule for each layer would then be

$$\mathbf{W}_l(t+1) = \mathbf{W}_l(t) + \eta(t)\hat{\mathbf{G}}_l^{-1}(t)\frac{\partial J}{\partial \mathbf{W}_l}, \tag{2.61}$$

$$\mathbf{b}_l(t+1) = \mathbf{b}_l(t) + \eta(t)\hat{\mathbf{G}}_l^{-1}(t)\frac{\partial J}{\partial \mathbf{b}_l}, \tag{2.62}$$

where t represents the current learning epoch. This is the Simplified Natural Gradient learning rule.

2.4 Simplified Natural Gradient Example

This section shows how the Fisher information matrix would be calculated for the neural network described in sec. 1.4.4. The input vectors of the XOR problem are

$$\mathbf{x}_0 = \begin{bmatrix}-1\\-1\end{bmatrix}, \qquad \mathbf{x}_1 = \begin{bmatrix}-1\\1\end{bmatrix}, \qquad \mathbf{x}_2 = \begin{bmatrix}1\\-1\end{bmatrix}, \qquad \mathbf{x}_3 = \begin{bmatrix}1\\1\end{bmatrix}. \tag{2.63}$$

The corresponding outputs of the XOR problem are

$$y_0 = -1, \qquad y_1 = 1, \qquad y_2 = 1, \qquad y_3 = -1. \tag{2.64}$$

The transfer functions, weight matrices and bias thresholds as determined in sec. 1.4.4 are

$$\mathbf{F}_1(\mathbf{a}_1) = \tanh(\mathbf{a}_1), \qquad \mathbf{W}_1 = \begin{bmatrix}2 & -2\\-2 & 2\end{bmatrix}, \qquad \mathbf{b}_1 = \begin{bmatrix}-2\\-2\end{bmatrix}, \tag{2.65}$$

$$f_2(a_2) = \tanh(a_2), \qquad \mathbf{W}_2 = \begin{bmatrix}1 & 1\end{bmatrix}, \qquad b_2 = 1. \tag{2.66}$$

Thus, the outputs of the hidden layer are

$$\mathbf{F}_1(\mathbf{W}_1\mathbf{x}_0 + \mathbf{b}_1) = \begin{bmatrix} -0.9640 \\ -0.9640 \end{bmatrix}, \qquad \mathbf{F}_1(\mathbf{W}_1\mathbf{x}_1 + \mathbf{b}_1) = \begin{bmatrix} -1.0000 \\ 0.9640 \end{bmatrix}, \tag{2.67}$$

$$\mathbf{F}_1(\mathbf{W}_1\mathbf{x}_3 + \mathbf{b}_1) = \begin{bmatrix} -0.9640 \\ -0.9640 \end{bmatrix}, \qquad \mathbf{F}_1(\mathbf{W}_1\mathbf{x}_2 + \mathbf{b}_1) = \begin{bmatrix} 0.9640 \\ -1.0000 \end{bmatrix}. \tag{2.68}$$

The final outputs of the two-layer perceptron are then

$$f_2(\mathbf{W}_2\mathbf{F}_1(\mathbf{W}_1\mathbf{x}_0 + \mathbf{b}_1) + b_2) = -0.7297, \quad f_2(\mathbf{W}_2\mathbf{F}_1(\mathbf{W}_1\mathbf{x}_1 + \mathbf{b}_1) + b_2) = 0.7461, \tag{2.69}$$

$$f_2(\mathbf{W}_2\mathbf{F}_1(\mathbf{W}_1\mathbf{x}_3 + \mathbf{b}_1) + b_2) = -0.7297, \quad f_2(\mathbf{W}_2\mathbf{F}_1(\mathbf{W}_1\mathbf{x}_2 + \mathbf{b}_1) + b_2) = 0.7461. \tag{2.70}$$

These outputs are used to compute the errors of each training pattern

$$e_0 = -0.2703, \qquad e_1 = 0.2539, \qquad e_2 = 0.2539, \qquad e_3 = -0.2703. \tag{2.71}$$

The estimated variance of the error is then computed to be

$$E\left\{e^2\right\} = 0.2751. \tag{2.72}$$

The partial derivatives of the objective function are then computed as

$$\frac{\partial J}{\partial \mathbf{W}_1} = \begin{bmatrix} 0.0080 & -0.0080 \\ -0.0080 & 0.0080 \end{bmatrix}, \qquad \frac{\partial J}{\partial \mathbf{b}_1} = \begin{bmatrix} -0.0099 \\ -0.0099 \end{bmatrix}, \tag{2.73}$$

$$\frac{\partial J}{\partial \mathbf{W}_2} = \begin{bmatrix} 0.2396 & 0.2396 \end{bmatrix}, \qquad \frac{\partial J}{\partial \mathbf{b}_2} = -0.0276. \tag{2.74}$$

This gives the estimates of the Fisher information of each layer as

$$\mathbf{G}_1 = \begin{bmatrix} 0.2174 & -0.1500 \\ -0.1500 & 0.2174 \end{bmatrix} \times 10^{-3}, \tag{2.75}$$

$$g_2 = 0.0550. \tag{2.76}$$

The inverse of $\mathbf{G}_1$ is

$$\mathbf{G}_1^{-1} = \begin{bmatrix} 8.7793 & 6.0575 \\ 6.0575 & 8.7793 \end{bmatrix} \times 10^{3}, \tag{2.77}$$

$$\frac{1}{g_2} = 18.1818. \tag{2.78}$$

These are obviously large values. Using the inverse of the Fisher information matrix in the place of the Hessian in a Newton-type algorithm has the same problems when the errors get small: The inverse becomes large and that drives the values of the weights to become large. This can lead to severe overtraining. However, if the same mechanism for avoiding overtraining is used as described in sec. 1.3.6, then these effects can be mitigated.

2.5 Adding a Prior Distribution with a Hyperparameter

The prior distribution on the weight matrix of level l, $\mathbf{W}_l$, has the PDF

$$p_{\mathbf{W}_l}(\mathbf{W}_l) = \exp\left(-\frac{1}{2}\alpha\|\mathbf{W}_l\|^2\right). \tag{2.79}$$

The mean value of $\mathbf{W}_l$ is $E\{\mathbf{W}_l\} = 0$. The variance of its norm $\mathrm{tr}(\mathbf{W}_l\mathbf{W}_l^T)$ is

$$E\left\{\mathrm{tr}\left(\mathbf{W}_l\mathbf{W}_l^T\right)\right\} = \frac{1}{\alpha}. \tag{2.80}$$

By using the modified partial-derivative (1.21) in the Fisher information matrix (2.60), then the definition of the estimate of the Fisher information matrix is modified by adding $\alpha\mathbf{I}$ to it. For example, if $\alpha = 0.1$, then the inverse of the modified Fisher information matrix in

(2.75) would be

$$(\mathbf{G}_1 + \alpha\mathbf{I})^{-1} = \begin{bmatrix} 9.9783 & 0.0149 \\ 0.0149 & 9.9783 \end{bmatrix}, \tag{2.81}$$

$$\frac{1}{g_2 + \alpha} = 6.4516. \tag{2.82}$$

The regularization of the gradient values carries through to the definition of the Fisher information matrix. The modified Fisher information matrix definition is

$$\hat{\mathbf{G}}_l = \frac{1}{N-1}\sum_{i=1}^{N}\left(\varphi_{l-1}(\mathbf{x}_i)^T\varphi_{l-1}(\mathbf{x}_i) + 1\right)\zeta_l(\mathbf{x}_i)^T\mathbf{e}_i\mathbf{e}_i^T\zeta_l(\mathbf{x}_i) + \alpha\mathbf{I}. \tag{2.83}$$

To explain this definition let $\mathbf{G}_\alpha$ be the Fisher information matrix computed with the augmented gradient as compared to the Fisher information matrix $\mathbf{G}$ computed with just the gradient ∇J. Thus, the modified Fisher information matrix is computed as

$$\begin{aligned} \mathbf{G}_\alpha &= E\left\{(\nabla J - \alpha\mathbf{W})(\nabla J - \alpha\mathbf{W})^T\right\}, \\ &= E\left\{\nabla J(\nabla J)^T - \alpha\left((\nabla J)\mathbf{W}^T + \mathbf{W}(\nabla J)^T\right) + \alpha^2\mathbf{W}\mathbf{W}^T\right\}, \\ &= E\left\{\nabla J(\nabla J)^T\right\} - \alpha E\left\{(\nabla J)\mathbf{W}^T + \mathbf{W}(\nabla J)^T\right\} + \alpha^2 E\left\{\mathbf{W}\mathbf{W}^T\right\}, \\ &= \mathbf{G} + \alpha\mathbf{I}. \end{aligned} \tag{2.84}$$

The following question arises: Why is $E\{\nabla J\mathbf{W}^T\} = 0$? In the case of the neuron, it can be shown to be zero simply by substituting $z = y - f(\mathbf{w}^T\mathbf{x} + b)$ and rearranging the order

of operations in the expectation integral

$$
\begin{aligned}
E\{\nabla J \mathbf{w}^T\} &= \int f'(\mathbf{w}^T\mathbf{x}+b)[y - f(\mathbf{w}^T\mathbf{x}+b)]\mathbf{x}\mathbf{w}^T \cdot \\
&\qquad \exp\left(-\frac{1}{2}[y - f(\mathbf{w}^T\mathbf{x}+b)]^2 - \frac{1}{2}\alpha\|\mathbf{w}\|^2\right) dy\, d\mathbf{w}\, d\mathbf{x}, \\
&= \int z f'(\mathbf{w}^T\mathbf{x}+b)\mathbf{x}\mathbf{w}^T \exp\left(-\frac{1}{2}\left[z^2 + \alpha\|\mathbf{w}\|^2\right]\right) dz\, d\mathbf{w}\, d\mathbf{x} \\
&= \int_{-\infty}^{\infty} z \exp\left(-\frac{1}{2}z^2\right) dz \int f'(\mathbf{w}^T\mathbf{x}+b)\mathbf{x}\mathbf{w}^T \exp\left(-\frac{1}{2}\alpha\|\mathbf{w}\|^2\right) d\mathbf{w}\, d\mathbf{x}, \\
&= 0.
\end{aligned}
\tag{2.85}
$$

The statistical model assumes that $y = f(\mathbf{w}^T\mathbf{x} + b) + z$ for any $\mathbf{x}$, y or $\mathbf{w}$. The random variable z is normally distributed, $z \sim \mathcal{N}(0, 1)$. Hence, $Ez = 0$. It is also assumed that z is independent of $\mathbf{x}$ and $\mathbf{w}$. This may seem to be an incorrect assumption at first as z is computed from y, $\mathbf{x}$, and $\mathbf{w}$. However, the network is correlating the input $\mathbf{x}$ with y by changing $\mathbf{w}$. That implies that z would be decorrelated. These are really only mathematical arguments to justify an algorithm design decision based on empirical results. These cross terms are very small and do not significantly impact the performance of the algorithm. Just as the cross terms of the neuron's Fisher information matrix are determined to be zero, so are the cross terms of the multilayer perceptron's Fisher information matrix.

2.6 Conclusion

Natural gradient is a gradient-descent technique that uses the Fisher information matrix of the parameter estimator as the Riemannian metric of a manifold. With neural networks, this allows a learning algorithm to follow the manifold to find better parameters to solve problems.

This chapter has described the Simplified Natural Gradient Learning Algorithm and its differences with ordinary gradient descent and Amari's Adaptive Natural Gradient. The next chapter will discuss the empirical results from experiments carried out to compare these algorithms.

Chapter 3

Experimental Results

The following experiments compare the performance of the Ordinary Gradient Learning (OGL) algorithm, the Adaptive Natural Gradient Learning (ANGL) algorithm, and the Simplified Natural Graident Learning (SNGL) algorithm described in this thesis for training multilayer perceptrons (MLPs).

The first experiment teaches a network to learn the XOR (eXclusive OR) function. It is not linearly separable and the properties of its parameter space (as an MLP learning this function) have been studied in depth [14, 26, 35, 43]. Remember that a simple neuron can only learn linearly separable functions. This is because a neuron finds the hyperplane with which to divide its inputs. If a problem is not linearly separable, then a multilayer perceptron must be used. XOR is an excellent example because there are four points on the plane for which there is no line that can separate them according to their labeling.

The second experiment teaches a network to encode a 5-bit message using a Low-Density Parity Check (LDPC) code. The encoding function adds 5 parity check bits to the message bits. It is similar to XOR but has a much larger parameter space.

The next experiment teaches a network to recognize three different species of Iris: *Iris setosa*, *Iris versicolor*, and *Iris virginica*. The experiment uses Fisher's Iris data [31].

The Mackey-Glass chaotic time series prediction problem tests a neural network training algorithm's ability to predict data. If $x(t)$ is the Mackey-Glass series for the index t, then the problem is to predict $x(t+6)$ given $x(t)$, $x(t-6)$, $x(t-12)$, and $x(t-18)$.

Finally, Nonlinear Dimension Reduction will be used to generate interesting data sets to train an MLP to compute a nonlinear dimension-reducing map. Nonlinear Dimension Reduction algorithms transform data embedded on a curved surface in a higher-dimensional manifold. The algorithms work on finite sets by maximizing the distance between distant

points while holding the distance between close points to be constant. This unfolds the surface onto a lower-dimensional Euclidean space. However, the algorithm does not produce the map from the surface to the Euclidean space. This is an excellent problem for a Machine Learning algorithm because sufficient data available for both training and testing.

3.1 Effective Learning Rate

Neural network training algorithms are very sensitive to the learning rate. In the OGL algorithm, the step-size for each learning epoch is $\eta\|\nabla J\|$. The natural gradient algorithms have a step-size of $\eta\|G^{-1}\nabla J\|$. An interesting point of comparison is the relative step-sizes of these algorithms. Since the Fisher information matrix is positive-definite in most cases, then all its eigenvalues are positive. Therefore, if λ is the largest eigenvalue of G^{-1}, then $\lambda\|\nabla J\| \geq \|G^{-1}\nabla J\|$. For the Simplified Natural Gradient Learning algorithm, the largest eigenvalue of G^{-1} is the multiplicative inverse of the smallest eigenvalue of G. Thus, for SNGL, the effective learning rate is $\frac{\eta}{\lambda_1}$ where λ_1 is the smallest eigenvalue of G. For ANGL, the effective learning rate is the product of the nominal learning rate η and the largest eigenvalue of the estimate of G^{-1}.

3.2 Exclusive OR Problem (XOR)

The XOR Problem was used as an example in secs. 1.4.4 and 2.4. In review, the network is to be trained to compute the XOR function. Again, this function is not linearly separable (see fig. 3.1). The inputs and outputs are

$$\mathbf{x}_0 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \qquad \mathbf{x}_1 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \qquad \mathbf{x}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \qquad \mathbf{x}_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \tag{3.1}$$

$$y_0 = -1, \qquad y_1 = 1, \qquad y_2 = 1, \qquad y_3 = -1, \tag{3.2}$$

respectively. The architecture is a two-layer perceptron with hyperbolic tangent transfer functions between both the hidden units and the output units. Thus, the error for each

pattern is

$$\epsilon_n = y_n - \tanh\left(\mathbf{W}_2 \tanh\left(\mathbf{W}_1 \mathbf{x}_n + \mathbf{b}_1\right) + b_2\right). \tag{3.3}$$

There are two hidden units and each layer has a bias. Hence, $\mathbf{W}_1$ is a 2-by-2 matrix and $\mathbf{W}_2$ is a 1-by-2 matrix. The bias vector $\mathbf{b}_1$ is a 2-vector and b_2 is a scalar. The source code for these experiments is in appendix A.

The first part of this experiment is to inspect the eigenvalues of the Fisher information matrix. Next, a performance comparison will be measured between the three algorithms OGL, SNGL, and ANGL using the sum-squared error metric:

$$J(\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, b_2) = \frac{1}{2} \sum_{n=0}^{3} \left(y_n - \tanh\left(\mathbf{W}_2 \tanh\left(\mathbf{W}_1 \mathbf{x}_n + \mathbf{b}_1\right) + b_2\right)\right)^2. \tag{3.4}$$

Finally, the effective learning rate of each algorithm will be investigated.

3.2.1 The Eigenvalues of the Fisher Information Matrix

Figure 3.2 shows the eigenvalues of the Fisher information matrix for each learning epoch of the SNGL algorithm. For this algorithm the effective learning rate is its learning rate divided by the largest eigenvalue of the Fisher information matrix. The eigenvalues λ_1 and λ_2 belong to the 2-by-2 Fisher information matrix of the first layer of connection weights and λ_3 is actually just the scalar value g_2 because the Fisher information matrix of the second layer is a 1-by-1 matrix.

These eigenvalues only have significant values during the first ten learning epochs. In later epochs, the Fisher information matrix is very close to $\alpha\mathbf{I}$ because the gradient matrices are so small.

3.2.2 Performance Comparison

Figure 3.3 shows the sum-squared error computed by (3.4) of each learning epoch for OGL, SNGL, and ANGL. Note that the epochs in which the biggest improvements are achieved by the SNGL algorithm are the first ten for which the eigenvalues of the Fisher

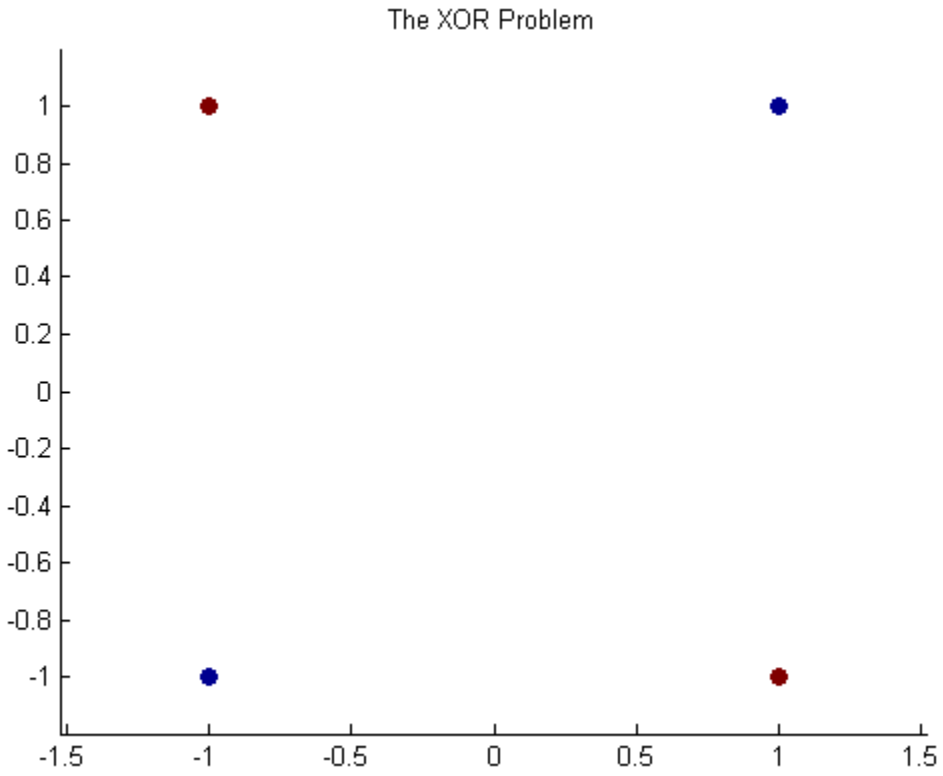

Fig. 3.1: This plot shows the labeling of the XOR problem. The red correspond to an output value of 1 and the blue to a value of -1.

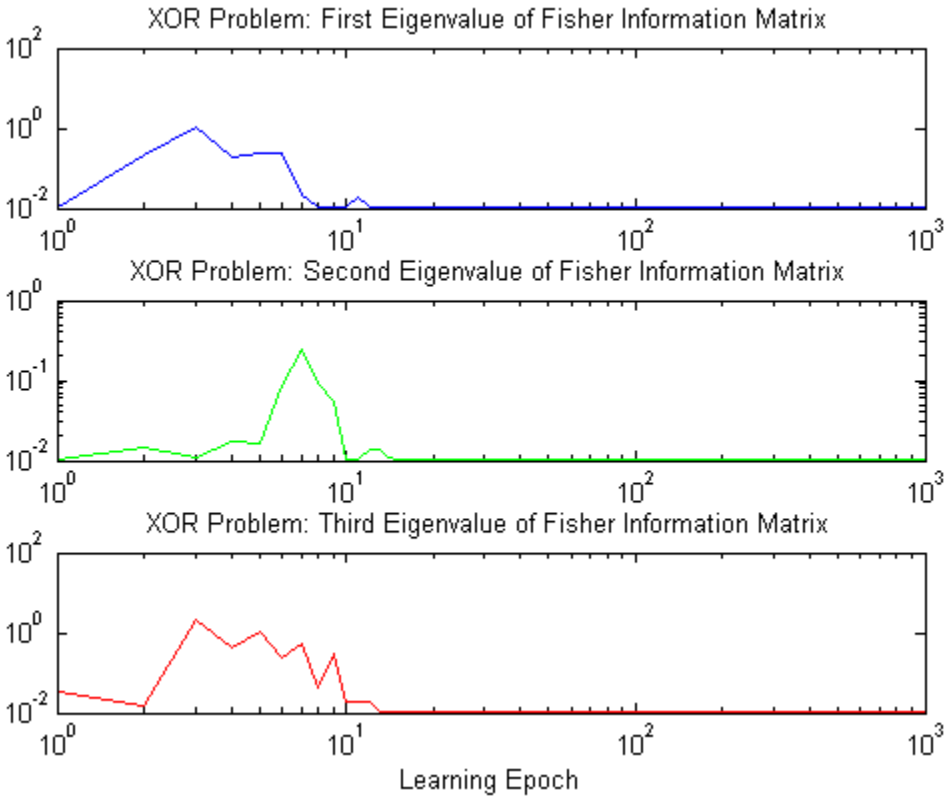

Fig. 3.2: This plot shows the eigenvalues of the Fisher information matrix for each learning epoch. The Fisher information is a metric for the variance of the parameters around the current estimate. High variance means that there more information to be gleaned from the data into the parameters.

information matrix are significant in fig. 3.2. Both the OGL and ANGL algorithms were stuck in plateaus for the first two hundred learning epochs.

Table 3.1 shows the parameters used in the three learning algorithms and some of the results of the experiment. The nominal learning rate is the actual step size η chosen in the learning rules of the learning algorithms. The initial learning rate is the effective learning rate used in the first learning epoch.

The learning rate is annealed in the SNGL algorithm as described in sec. 1.3.8. This is done to counteract the learning rate approaching $\frac{\eta}{\alpha}$ when the Fisher information matrix gets close to $\alpha\mathbf{I}$ in the later learning epochs. The OGL algorithm's performance did not improve when using an annealed learning rate.

In the ANGL algorithm, a fixed learning rate is used. However, the adaptation rate used in the autoregressive estimator of the inverse of the Fisher information matrix given in (2.27) is chosen to be $\frac{1}{t}$ in [34, 35]. In the implementation used in this experiment, the adaptation rate is chosen to be $\min\left(0.1, \frac{1}{t}\right)$ so that the algorithm actually converges.

Equations (3.5-3.10) show the final values of the parameters computed by the algorithms. The ANGL algorithm computes slightly larger parameters because including the prior distribution on the weights causes the Fisher information matrix to approach infinity and fail. As the algorithm continues to run the absolute values of the parameters increase without bound. However, the ratio between any two parameters is relatively constant. Thus, it is overtraining in the later epochs. The final values of the parameters in the OGL algorithm are

$$\mathbf{W}_1 = \begin{bmatrix} 1.7280 & 1.7280 \\ 1.7280 & 1.7280 \end{bmatrix}, \qquad \mathbf{b}_1 = \begin{bmatrix} -1.6535 \\ 1.6535 \end{bmatrix}, \tag{3.5}$$

$$\mathbf{W}_2 = \begin{bmatrix} -1.7377 & 1.7377 \end{bmatrix}, \qquad b_2 = -1.5822. \tag{3.6}$$

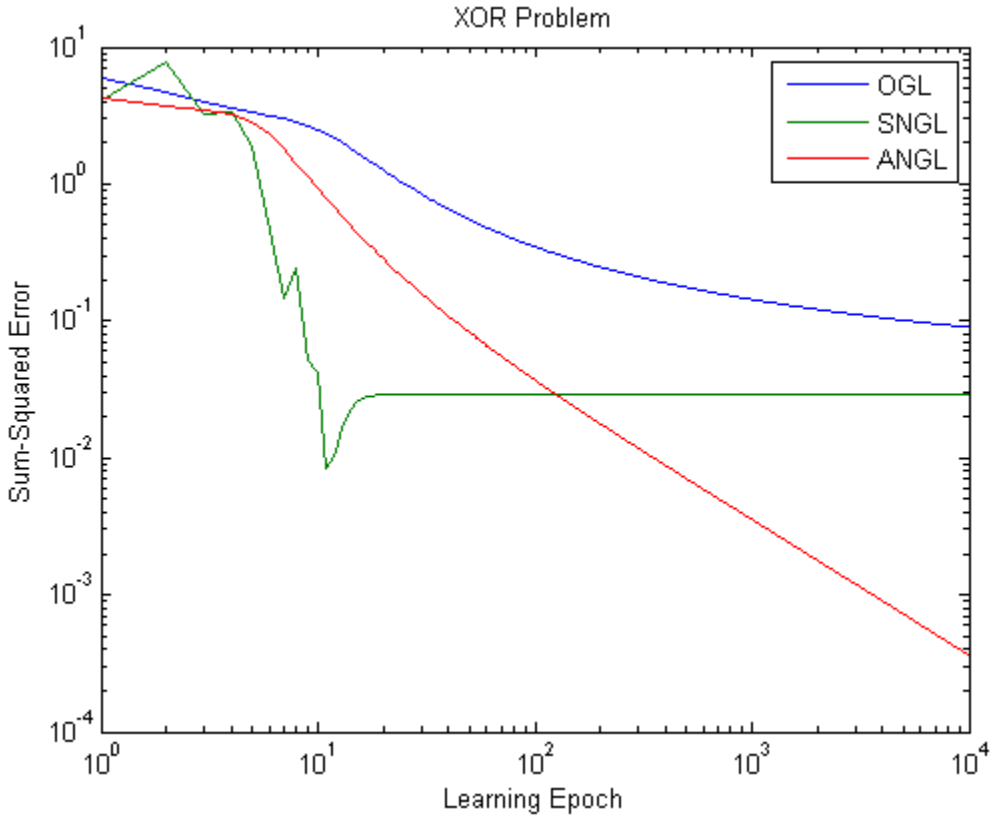

Fig. 3.3: The sum-squared error curves of OGL, ANGL, and SNGL with effective learning rate $\eta = 0.25$ and SNGL modified to use an annealed learning rate.

Table 3.1: The results of and parameters used for the OGL, SNGL, and ANGL algorithms on the XOR problem.

XOR	OGL	SNGL	ANGL
Hidden Units	2	2	2
Nominal Learning Rate	0.25	0.25	0.01
Initial Learning Rate	0.25	6.5351	0.2574
Annealed Learning Rate	Yes	Yes	No
Prior Probability on Parameters	Yes	Yes	No
Hyperparameter of Prior Distribution	0.01	0.01	N/A
Fisher Information Matrix Size	N/A	(1, 2)	9
Adaptation Rate	N/A	N/A	$\min\left(0.1, \frac{1}{t}\right)$
Number of Extra Parameters	0	5	81
Final Learning Rate	1e-4	0.0097	0.1157
Final Sum-Squared Error	0.0902	0.0289	3.6627e-4
Learning Epoch when SSE < 0.0289	10,000	11	127

The final values of the parameters in the SNGL algorithm are

$$\mathbf{W}_1 = \begin{bmatrix} 1.7280 & -1.7280 \\ -1.7280 & 1.7280 \end{bmatrix}, \qquad \mathbf{b}_1 = \begin{bmatrix} -1.6536 \\ -1.6536 \end{bmatrix}, \tag{3.7}$$

$$\mathbf{W}_2 = \begin{bmatrix} 1.7377 & 1.7377 \end{bmatrix}, \qquad b_2 = 1.5822. \tag{3.8}$$

The final values of the parameters in the ANGL algorithm are

$$\mathbf{W}_1 = \begin{bmatrix} -2.9141 & -2.9142 \\ -2.9108 & -2.9107 \end{bmatrix}, \qquad \mathbf{b}_1 = \begin{bmatrix} -2.8228 \\ 2.8251 \end{bmatrix}, \tag{3.9}$$

$$\mathbf{W}_2 = \begin{bmatrix} 2.7112 & -2.7116 \end{bmatrix}, \qquad b_2 = -2.6101. \tag{3.10}$$

One interesting side note is worth discussing. Because the hyperbolic tangent function is an odd function, negating column i in $\mathbf{W}_2$ and negating row i in $\mathbf{W}_1$ and element i in $\mathbf{b}_1$ does not change the function that the network computes. Furthermore, permuting the columns of $\mathbf{W}_2$ and applying the same permutation to the rows of $\mathbf{W}_1$ and the elements of $\mathbf{b}_1$ also does not change the function that the network computes. Therefore, the parameters in eqs. (3.5-3.8) are equivalent.

Figure 3.4 shows the effective learning rates for all three algorithms. It shows some interesting behavior of the SNGL algorithm. The SNGL algorithm takes one large step in the first epoch followed by several small steps. Then it takes a few more big steps and then it slowly backs off as the effects of the prior distribution hold it back from overtraining. Essentially, SNGL has a variable learning rate for each layer in the network. This is where the real power of the Simplified Natural Gradient Learning algorithm comes into play. The steps taken are governed by the Fisher information of the parameters in each layer. The ANGL algorithm also performs very well. Again, the prior distribution is not used in the ANGL algorithm because it causes instability in the estimate of the Fisher information matrix's inverse. Note that its effective learning rate increases in later epochs.

Figure 3.5 shows the effects of decreasing the hyperparameter of the prior distribution of

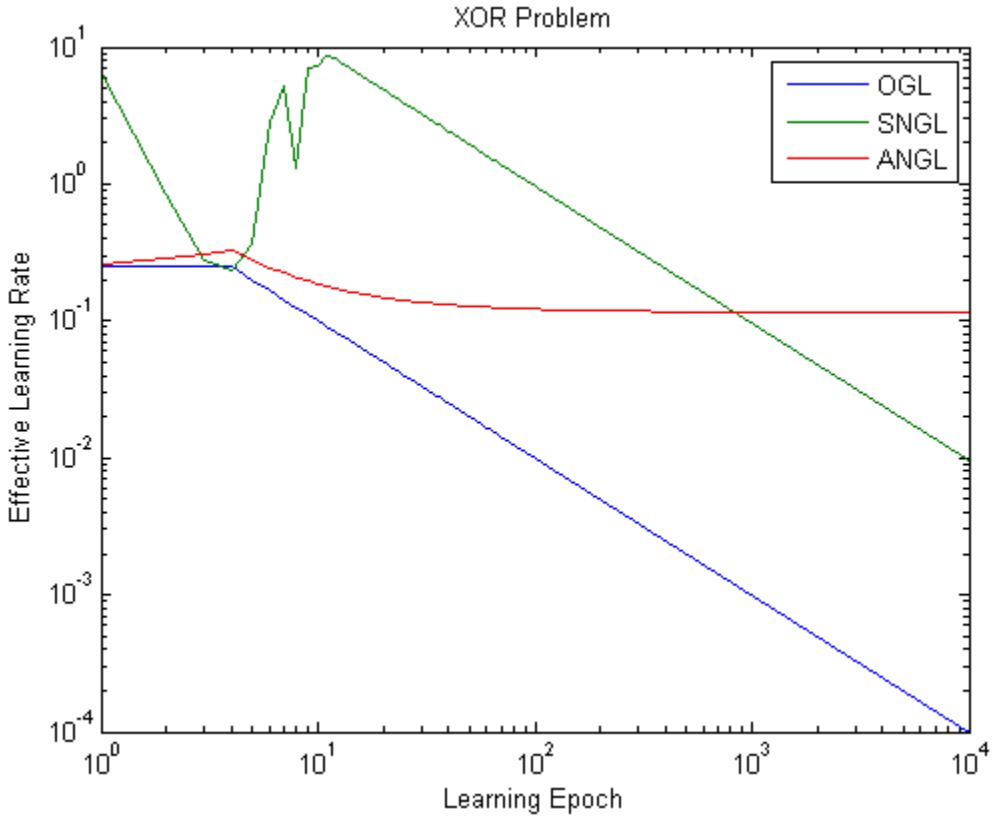

Fig. 3.4: The effective learning rates of the OGL, SNGL, and ANGL algorithms for the XOR problem.

the weights. The weights get slightly larger and the performance of the algorithm improves. However, the learning rate must be reduced to compensate for the larger eigenvalues of the Fisher information matrix. The first experiment was run with the learning rate $\eta = 0.25$ and the hyperparameter $\alpha = 0.01$. When the hyperparameter is decreased to $\alpha = 0.001$, then the learning rate is decreased to $\eta = 0.01$. Otherwise, the stepsize is too big and the learning algorithm never converges. There is, therefore, a tradeoff between speed of convergence for SNGL and the accuracy required of the resulting network.

The XOR problem is fairly easy with a small training set. A more complex problem will reveal more information on how well SNGL performs compared to OGL and ANGL.

3.3 Low-Density Parity Check (LDPC) Code

Encoding data with a Low-Density Parity Check (LDPC) code [44] could be considered the XOR problem's big brother. An LDPC is computed by multiplying (0,1)-vectors with a (0,1)-matrix over $GF(2)$ to get a longer (0,1)-vector. Remember that in $GF(2)$ addition is the XOR operation and multiplication is the AND operation.

Formally, a Low-Density Parity Check code is a code in which extra bits that are parity checks are added to the message. A matrix is used to represent the transformation of the bits over $GF(2)$. For example, if the message to be encoded was $\mathbf{m} = (0, 0, 1, 0, 1)^T$, and the encoding matrix was

$$\mathbf{G} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.11}$$

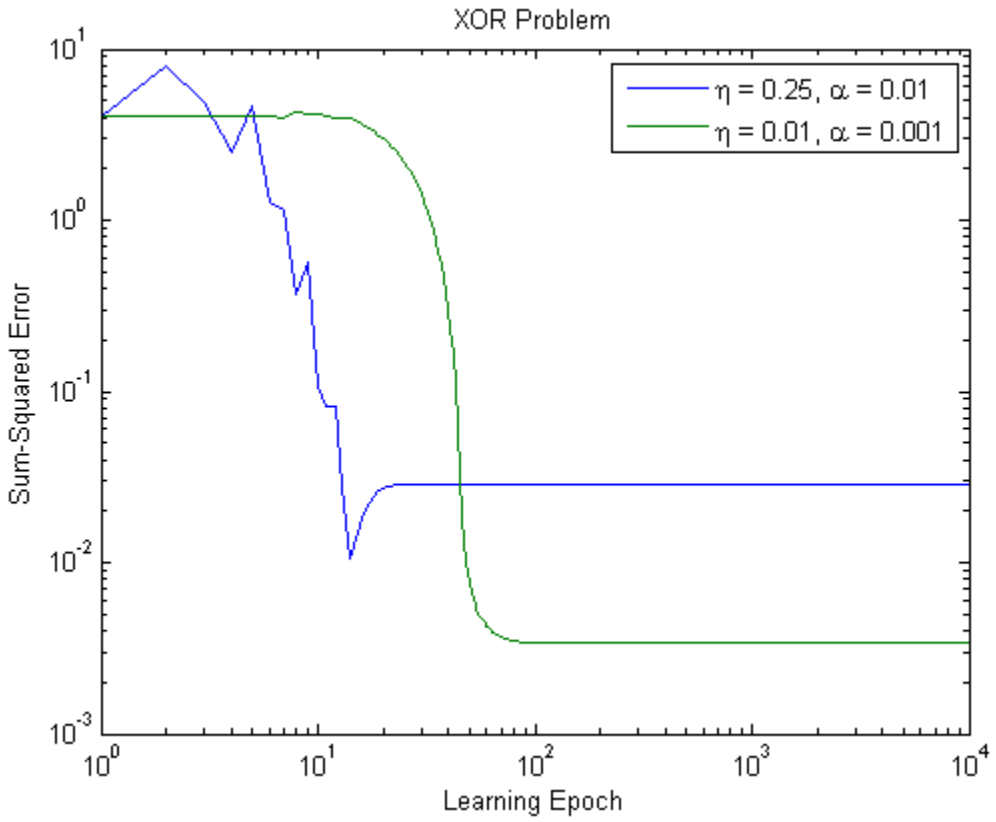

Fig. 3.5: The SSE of the SNGL algorithm for the XOR problem with the two values for learning rate and hyperparameters.

then the encoded message $\mathbf{c}$ would be

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}. \tag{3.12}$$

This code has a rate of 1/2 because the code words are twice as big as the message words.

The encoding function is the matrix $\mathbf{G}$ in (3.11). The training set consists of 32 training patterns. The inputs are the 32 possible messages and the outputs are the 32 10-bit code words corresponding to each message. Thus, the network has 5 input units and 10 output units. The multilayer perceptron will also have 10 hidden units. During the training the bits are changed according to a bit error rate of 0.01. This will help the network deal with bit errors.

First, the performance of the three algorithms will be compared. This will be followed by a comparison of their effective learning rates. Finally, an experiment in which the effective learning rate of the SNGL and ANGL algorithms is used in the OGL algorithm will be discussed. The source code for these experiments is in appendix B.

3.3.1 Performance Comparison

Figure 3.6 shows the sum-squared error of the three algorithms when learning this LDPC encoding function. As can be seen, the performance of the Simplified Natural Gradient Learning algorithm is comparable to that of the Ordinary Gradient Learning algorithm,

but it reaches convergence in 16% of the time.

Table 3.2 lists the parameters used in all three algorithms and some of the performance metrics. The ANGL algorithm achieves a much lower MSSE. Using the prior probability by adding the current parameters multiplied by the hyperparameter α caused the components of the Fisher information matrix to approach infinity. Therefore, it was omitted and the weights of the network were allowed to grow without constraint. Thus, the ANGL overtrained the network significantly.

3.3.2 The Effective Learning Rate

Figure 3.7 shows the effective learning rates for these three algorithms. For OGL the effective learning rate is just the learning rate chosen for the algorithm. For ANGL it is the learning rate multiplied by the smallest eigenvalue of the estimated Fisher information matrix. For SNGL it is the inverse of the largest eigenvalue of the Fisher information matrix multiplied by the learning rate.

The effective learning rate for the SNGL algorithm is small for the first 100 learning epochs. Between the hundredth and the first thousandth epoch, the learning rate becomes quite large, for a step size, becoming larger than one for a few epochs. When the algorithm reaches convergence, then the learning rate begins to decrease due to annealing. The LDPC function is more complicated than XOR and this is reflected in the number of learning epochs required to reach the three phases of learning for SNGL.

3.3.3 Substituting Learning Rates

One might argue that SNGL and ANGL perform better than OGL because they both use higher learning rates. However, as fig. 3.7 shows, both of these algorithms have smaller learning rates at first and the learning rates do not increase until progress is made towards good parameter values.

Figure 3.8 shows the performance of OGL with the effective learning rates of ANGL and SNGL in addition to OGL with a flat learning rate along with ANGL and SNGL. The plot shows that using the effective learning rate of ANGL improves the performance

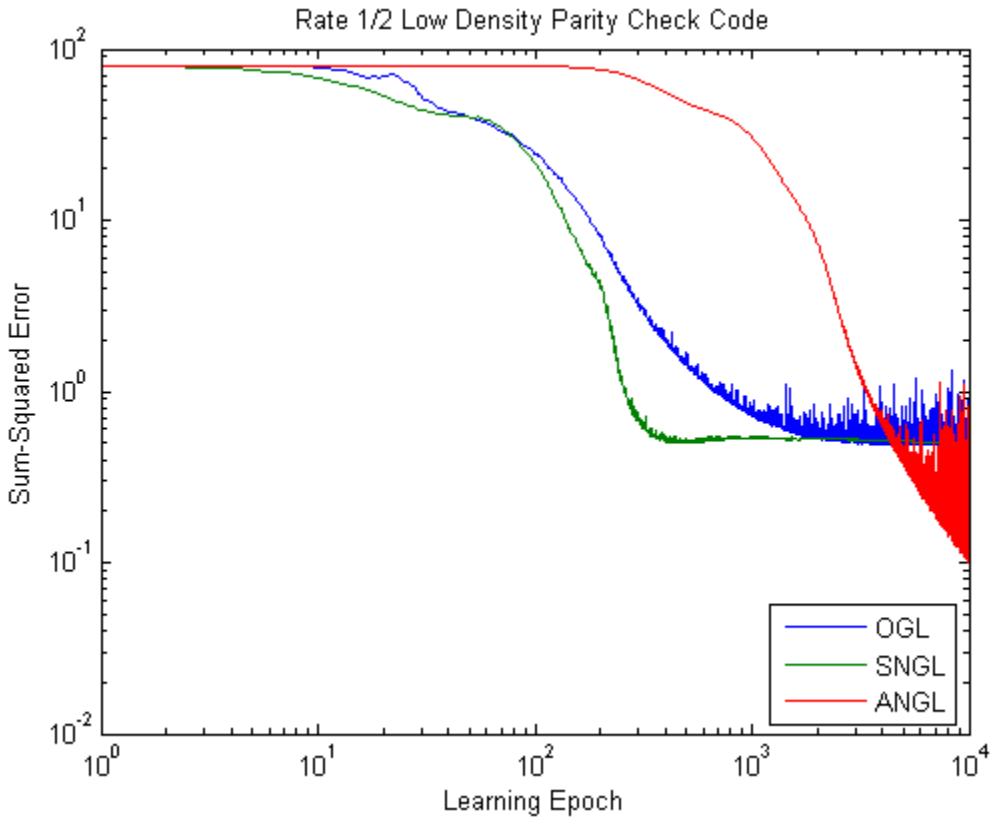

Fig. 3.6: Sum-squared error of the Ordinary Gradient Learning (OGL), Simplified Natural Gradient Learning (SNGL), and Adaptive Natural Gradient Learning (ANGL) algorithms when learning the rate 1/2 (10,5) low-density parity check encoding function with $\mathbf{G}$ as its generator matrix.

Table 3.2: LDPC experiment results.

LDPC	OGL	SNGL	ANGL
Hidden Units	10	10	10
Initial Learning Rate	0.2	0.1835	0.01
Annealed Learning Rate	No	Yes	No
Prior Probability on Parameters	Yes	Yes	No
Hyperparameter of Prior Distribution	0.001	0.001	N/A
Fisher Information Matrix Size	N/A	(10, 10)	170
Adaptation Rate	N/A	N/A	$\min\left(1e-3, \frac{1}{t}\right)$
Number of Extra Parameters	0	200	28,900
Final Learning Rate	0.2	0.0161	0.2368
Minimum Sum-Squared Error	0.4893	0.4869	0.0991
Learning Epoch when SSE < 0.5	2997	477	4398

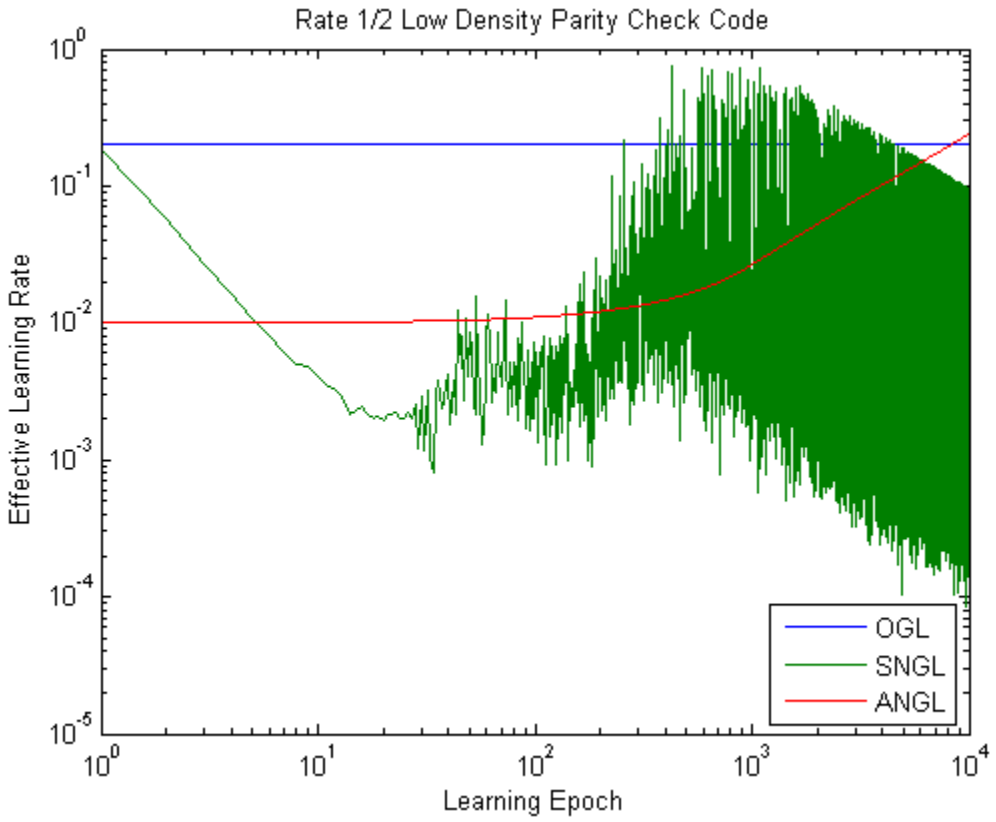

Fig. 3.7: Effective learning rate of the OGL, SNGL, and ANGL algorithms when learning the LDPC encoding function.

of OGL. However, the performance of OGL does not improve with the SNGL algorithm's effective learning rate. For this problem, the Fisher information matrix used in SNGL is a block-diagonal matrix with two 10-by-10 blocks on the main diagonal. In the ANGL algorithm, it is a 61-by-61 matrix. These matrices are both positive definite and can be decomposed into the product of three matrices [21]

$$\mathbf{G} = \mathbf{V}^T\mathbf{D}\mathbf{V}. \tag{3.13}$$

The matrix $\mathbf{V}$ is a real orthognal matrix. The matrix $\mathbf{D}$ is a diagonal matrix and has the eigenvalues of $\mathbf{G}$ as its diagonal elements. The columns of $\mathbf{V}$ are the eigenvectors of $\mathbf{G}$ and all are unit vectors representing directions. The eigenvalues represent how much information is associated with each direction. Using the maximum eigenvalue to determine an effective learning rate is good for demonstration purposes. However, it will not improve performance of the algorithm because the directional information is missing.

These first two examples both teach a network to learn a boolean function. Each unit in the network models a bit in a computer. The next example problem will involve classification where each output is a probability.

3.4 The Iris Species Identification Problem

Fisher's Iris classification data [31] has four inputs and three classes. The inputs are the petal length, petal width, sepal length, and sepal width measured in millimeters. The three possible classes are *Iris setosa*, *Iris versicolor*, and *Iris virgnica*. This data set is interesting because it has two misclassifications. Park used this data set [35] to compare the performance of the ANGL algorithm with that of the OGL algorithm. The network architecture is four input nodes, four hidden nodes, and three output nodes. The transfer function for the hidden nodes is the hyperbolic tangent. Each of the three outputs will be the estimated probability that the given pattern belongs to *Iris setosa*, *Iris versicolor*, or *Iris virginica*.

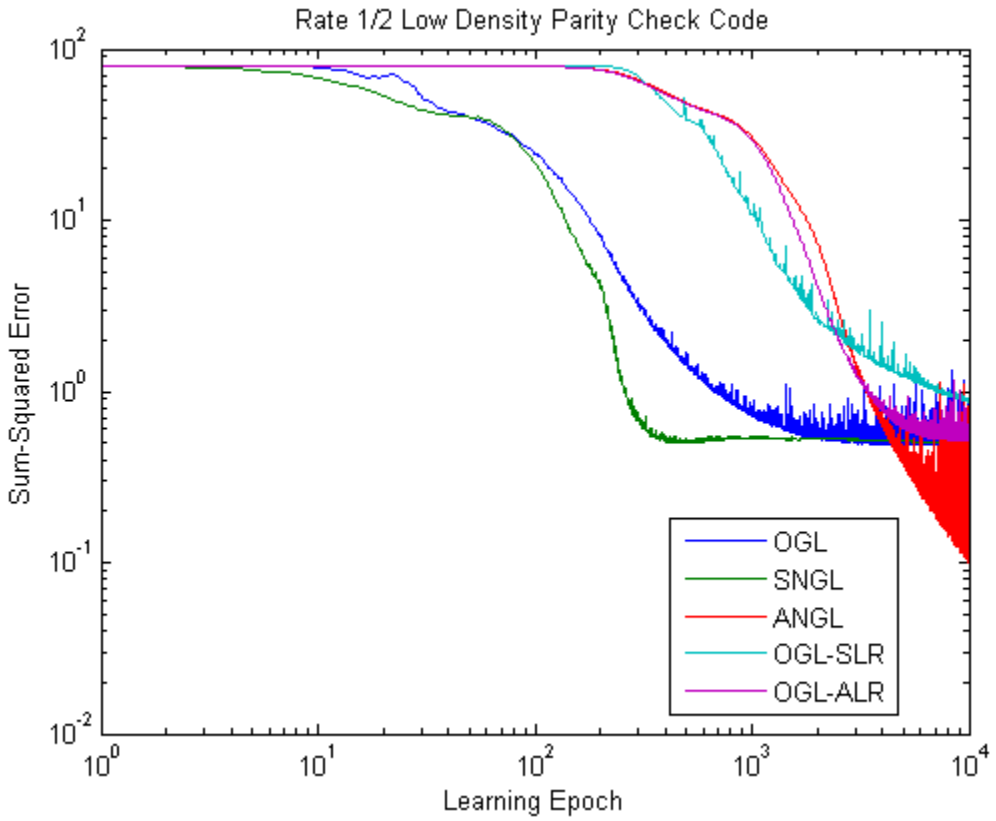

Fig. 3.8: Sum-squared error of the OGL, SNGL, and ANGL algorithms compared to the OGL algorithm with the effective LR of the ANGL and SNGL algorithms.

3.4.1 Probabilistic Model

Thus, the three outputs must always add to one. Therefore, the *soft-max* function will be used

$$f_i(\mathbf{a}) = \frac{\exp(a_i)}{\exp(a_1) + \exp(a_2) + \exp(a_3)}. \tag{3.14}$$

The vector $\mathbf{a}$ in (3.14) represents the activations of the output units that are the parameters to its soft-max transfer function.

The components of the Jacobian matrix for this function are

$$\frac{\partial f_i}{\partial a_j} = \begin{cases} f_i(1 - f_i) & i = j \\ -f_i f_j & i \neq j. \end{cases} \tag{3.15}$$

This function is a multivariable generalization of the logistic function in (1.2). The probabilistic model will be

$$p(\mathbf{y}|\mathbf{x}) = [f_1(\mathbf{A}(\mathbf{x}))]^{y_1} [f_2(\mathbf{A}(\mathbf{x}))]^{y_2} [f_3(\mathbf{A}(\mathbf{x}))]^{y_3}. \tag{3.16}$$

The activation vector is determined by $\mathbf{A}(\mathbf{x}) = \mathbf{W}_2 \tanh(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$. The components of $\mathbf{y} = (y_1, y_2, y_3)$ are each either zero or one with one and only one component equal to one for each pattern. The likelihood of a given pattern is

$$\ell(\mathbf{x}, \mathbf{y}) = y_1 \log(f_1(\mathbf{A}(\mathbf{x}))) + y_2 \log(f_2(\mathbf{A}(\mathbf{x}))) + y_3 \log(f_3(\mathbf{A}(\mathbf{x}))). \tag{3.17}$$

Using the Jacobian matrix in (3.15), the derivative of the likelihood function with respect to the activation vector $\mathbf{a}$ is

$$\frac{\partial \ell}{\partial \mathbf{a}} = \begin{bmatrix} f_1(1 - f_1) & -f_1 f_2 & -f_1 f_3 \\ -f_1 f_2 & f_2(1 - f_2) & -f_2 f_3 \\ -f_1 f_3 & -f_2 f_3 & f_3(1 - f_3) \end{bmatrix} \begin{bmatrix} \frac{y_1}{f_1} \\ \frac{y_2}{f_2} \\ \frac{y_3}{f_3} \end{bmatrix} = \begin{bmatrix} y_1 - f_1 \\ y_2 - f_2 \\ y_3 - f_3 \end{bmatrix}. \tag{3.18}$$

The score function is the gradient of the likelihod function. Thus, for the likelihood function in (3.17), the score functions are

$$\frac{\partial \ell}{\partial \mathbf{W}_2} = \begin{bmatrix} y_1 - f_1 \\ y_2 - f_2 \\ y_3 - f_3 \end{bmatrix} \mathbf{A}(\mathbf{x})^T, \qquad \frac{\partial \ell}{\partial \mathbf{b}_2} = \begin{bmatrix} y_1 - f_1 \\ y_2 - f_2 \\ y_3 - f_3 \end{bmatrix}, \tag{3.19}$$

$$\frac{\partial \ell}{\partial \mathbf{W}_1} = \tanh'(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\mathbf{W}_2^T \begin{bmatrix} y_1 - f_1 \\ y_2 - f_2 \\ y_3 - f_3 \end{bmatrix} \mathbf{x}^T, \tag{3.20}$$

$$\frac{\partial \ell}{\partial \mathbf{b}_1} = \tanh'(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\mathbf{W}_2^T \begin{bmatrix} y_1 - f_1 \\ y_2 - f_2 \\ y_3 - f_3 \end{bmatrix}, \tag{3.21}$$

where $\tanh'(\mathbf{z}) = \mathbf{I} - \mathrm{diag}(\mathbf{z})^2$.

This problem can be difficult to learn, even when very small random weights are chosen. The difficulty comes because of the large numbers in the measurements. One approach can be to translate and scale the measurements such that the mean measurements are zero and their covariance matrix is the identity matrix. This is done by calculating the average of each measurement from all the training patterns. Subtract the mean from each training pattern. Use the inverse of the Cholesky decomposition [45] of the unbiased estimate of the covariance [40] of the measurements to rescale the centered measurements. Multiplying the centered measurements by its inverse is equivalent to dividing by the standard deviation of a random variable.

There are 150 examples in the Iris data set. There are 50 examples of each of the three species. To mimic the experiment performed by Park [35], 30 of each species were randomly selected to create a training set of 90 examples. The other 60 examples will be used as a test set. The source code for these experiments is in appendix C.

3.4.2 Performance Comparison

Figure 3.9 shows the sum-squared error of each learning epoch corresponding to the three MLP training algorithms OGL, SNGL, and ANGL. The most striking feature of this plot is when the ANGL algorithm starts plunging just before the thousandth epoch. Again, this is because the ANGL algorithm does not have any mechanism to constrain the values of the weight matrices. It does a very good job at overtraining the network for this problem. It reaches the same SSE as the other two algorithms after 1000 epochs. The OGL algorithm converges near the 3000th epoch while the NGL algorithm hits the same SSE near the 500th epoch. Thus, NGL converges faster than OGL with comparable results.

Figure 3.10 shows the number of misclassified test cases. After 300 epochs, for all three algorithms, the number of misclassified examples fluctuates between one and three. There are two misclassified examples in Fisher's iris data. Thus, one, two, or three errors on the test set is considered good performance.

Figure 3.11 shows the effective learning rates of each of the algorithms. Notice that the SNGL algorithm's learning rate starts small and rises radically in the beginning like it did with the XOR and LDPC problems. After that, it decreases monotonically.

Table 3.3 shows the results of the experiments. Note that even though the sum-squared error of the ANGL algorithm is so much smaller than those of the OGL and SNGL algorithms, the number of test set errors is the same in both the ANGL and SNGL algorithms. The OGL algorithm had three test set errors, so its performance is comparable.

Thus far, the examples have been static functions. The next problem will require a network to predict the next element of a series.

3.5 Mackey-Glass Time Series Prediction Problem

The Mackey-Glass time series (see fig. 3.12) is a chaotic series given by the recurrence relation

$$x(t+1) = (1-b)x(t) + \frac{ax(t-\tau)}{1+x(t-\tau)^{10}}, \tag{3.22}$$

with $a = 0.2$, $b = 0.1$, and $\tau = 17$. The initial conditions are chosen to be $x(t) = 0.9$ for

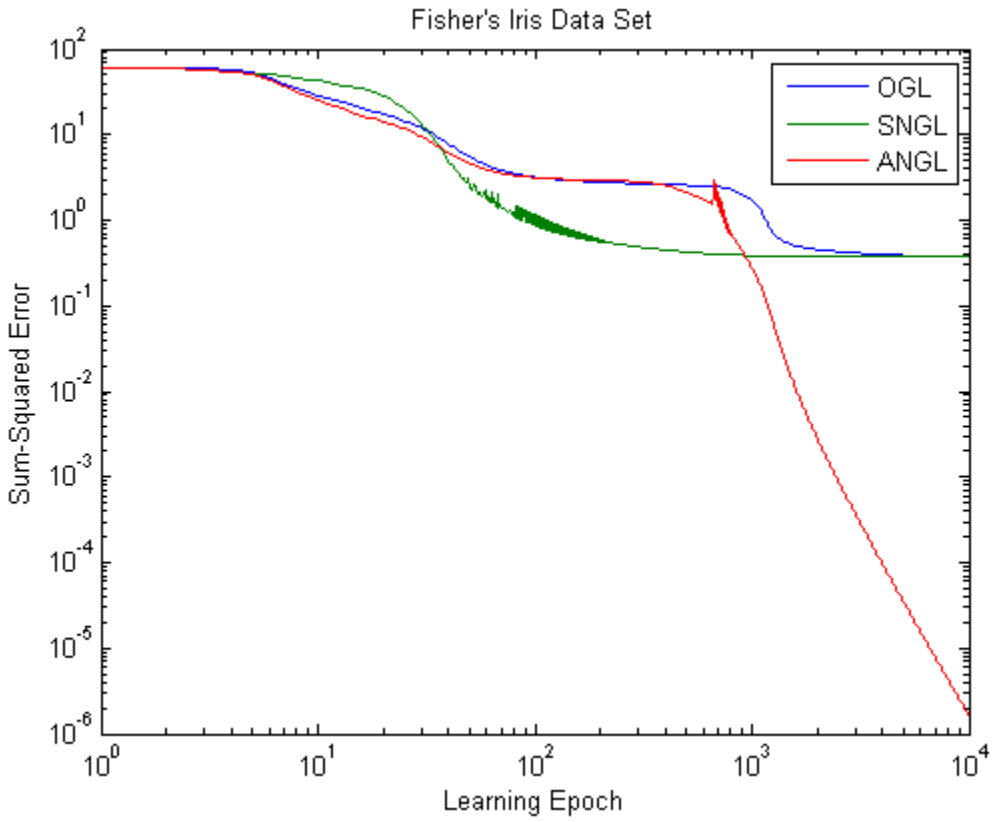

Fig. 3.9: Sum-squared error of the OGL, ANGL, and SNGL algorithms on Fisher's iris data.

Table 3.3: Fisher's iris data experiment results.

Fisher's Iris Data	OGL	SNGL	ANGL
Hidden Units	4	4	4
Initial Learning Rate	0.01	0.0098	0.01
Annealed Learning Rate	No	Yes	No
Prior Probability on Parameters	Yes	Yes	No
Hyperparameter of Prior Distribution	0.1	0.1	N/A
Fisher Information Matrix Size	N/A	(3,4)	35
Adaptation Rate	N/A	N/A	0.001
Number of Extra Parameters	0	25	1,225
Final Learning Rate	0.01	4.8057e-4	0.2010
Minimum Sum-Squared Error	0.3885	0.3762	1.6431e-6
Test Set Classification Errors	3	2	2

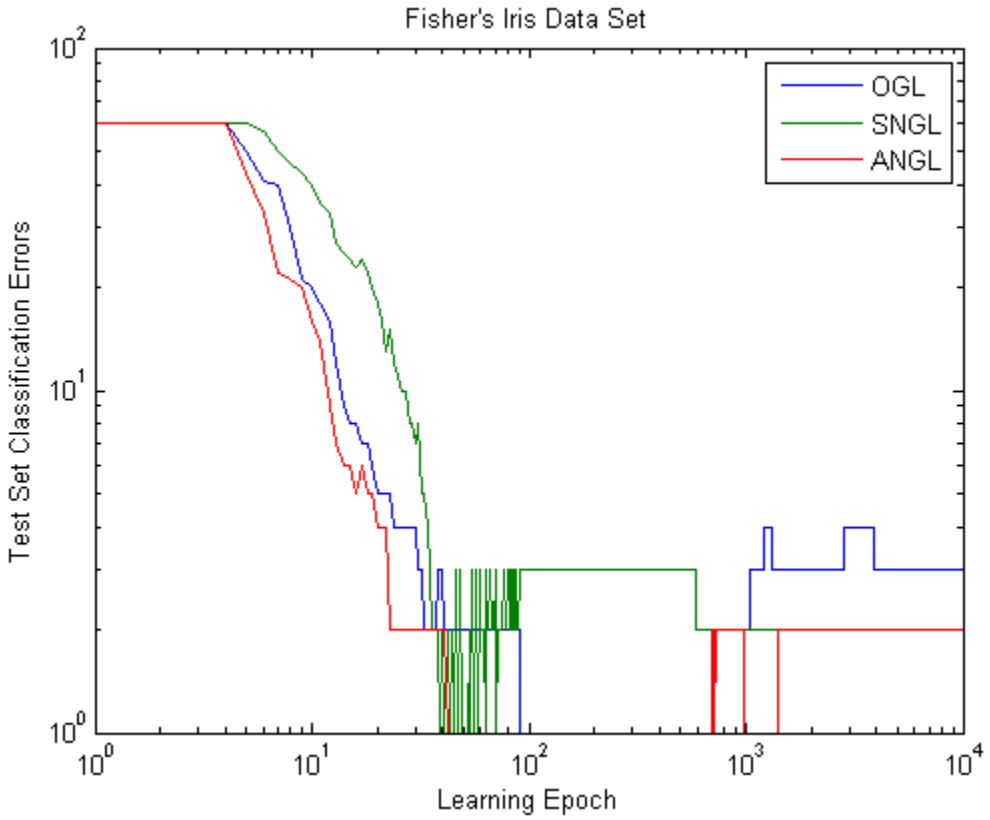

Fig. 3.10: The number of misclassified test case of the OGL, ANGL, and SNGL algorithms on Fisher's iris data.

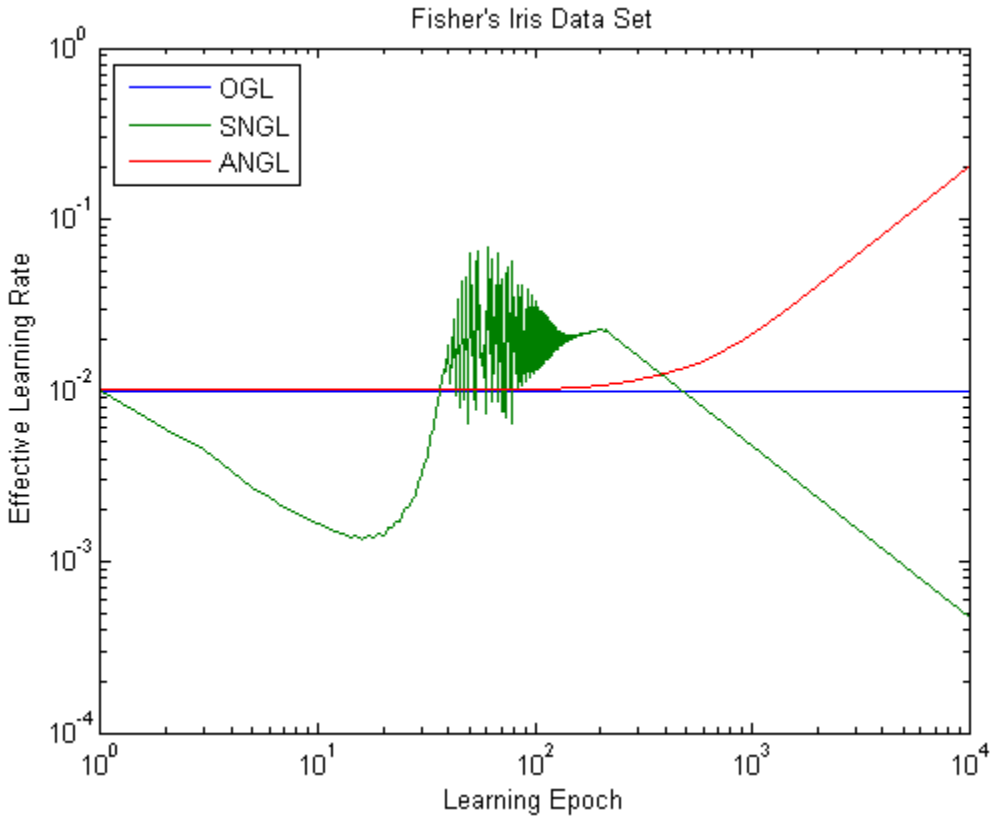

Fig. 3.11: The effective learning rate of the OGL, ANGL, and SNGL algorithms on Fisher's iris data.

$1 \leq t \leq \tau$. The network will try to predict $x(t+6)$ given the observations $x(t)$, $x(t-6)$, $x(t-12)$, and $x(t-18)$. The training set is elements 200 through 700 and the test set is elements 5000 through 5500.

3.5.1 Network Architecture

The network architecture has four inputs as described above. There are ten hidden units with a hyperbolic tangent transfer function. The outputs have linear transfer functions. Thus, the error function is

$$J(\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sum_{t=200}^{700} \left(x(t+6) - \mathbf{W}_2 \tanh\left(\mathbf{W}_1 \xi_t + \mathbf{b}_1\right) - b_2\right)^2, \tag{3.23}$$

where

$$\xi_t = \begin{bmatrix} x(t) \\ x(t-6) \\ x(t-12) \\ x(t-18) \end{bmatrix}. \tag{3.24}$$

In this experiment $\mathbf{W}_1$ is a 10-by-5 matrix, $\mathbf{b}_1$ is a 10-tuple, $\mathbf{W}_2$ is a 1-by-11 matrix, and b_2 is a scalar. The source code for these experiments is in appendix D.

3.5.2 Performance Comparison

Figure 3.13 shows the sum-squared error of the three algorithms. It takes a long time for a network to learn this problem. The OGL algorithm gets stuck in a plateau after epoch 100. It finally begins to leave it after epoch 10,000 but doesn't make much progress after 100,000 epochs. The ANGL algorithm breaks out of the plateau region near learning epoch 3,000. It jumps around between epochs 300 and 400 and then again between 5,000 and 6,000. After epoch 6,000, it plunges down signficiantly. The SNGL performs significantly worse until epoch 300. After epoch 300, the algorithm bounces around erratically with an overall downward trend. By epoch 25,000 the SNGL algorithm is starting to converge. While ANGL and SNGL have comparable performance in the later learning epochs, they

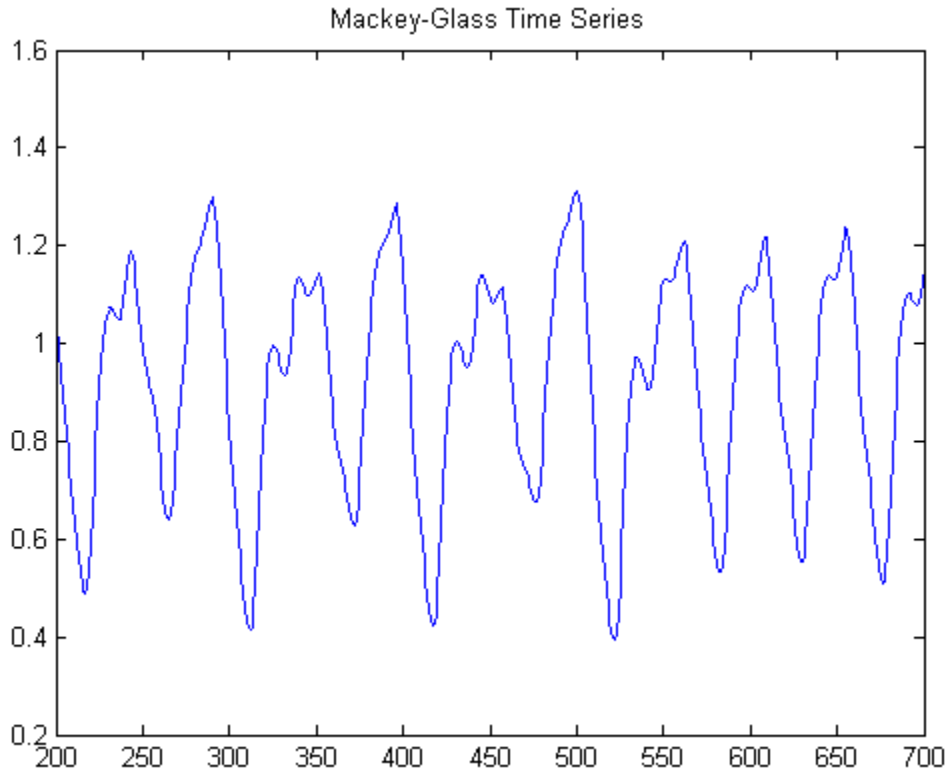

Fig. 3.12: The training set of the Mackey-Glass time series prediction problem.

both significantly outperform OGL in this experiment.

Figure 3.14 shows the effective learning rates of the three algorithms. The learning rate was annealed for OGL because the algorithm began to act erratically in the late learning epochs because it was taking steps that were too large. The ANGL algorithm's learning rate increased slightly near the end as it began to converge to a better answer. The learning rate for SNGL begins extremely small and remains so for the first 300 learning epochs. This means that there was a lot of information at each point in the parameter space and small steps were necessary to find good parameters. After epoch 300, the learning rate fluctuates violently with a steadily decreasing envelope due to annealing that finally closes around the 90,000th epoch.

Table 3.4 shows the parameters for the three learning algorithms. Note how small the learning rate is at the beginning for the SNGL algorithm. This simply implies that there is a lot of information and small steps should be taken. The smallest steps are taken in the direction with the most information and the largest steps are taken in the direction with the least.

3.6 Nonlinear Dimension Reduction

There are several algorithms recently developed to map a non-Euclidean manifold existing in a higher-dimensional Euclidean space into a Euclidean space of a lower dimension [46–52]. For example, suppose that a set points lies on a circle with each point equally spaced. This parametric curve can be unfolded into a line segment. Let $t_n = \frac{n}{N}$ for $n = 0, 1, 2, \ldots N$, then the points on the circle can be parameterized as $x_1(t) = \cos(2\pi t)$ and $x_2(t) = \sin(2\pi t)$. The source code for these experiments is in appendix E.

3.6.1 Flattening a Semicircle

The parametric curve is mapped to the real line by minimizing the distance between points in both the higher-dimensional and lower-dimensional Euclidean spaces. The outputs of these algorithms are the coordinates of the points in the lower-dimensional Euclidean

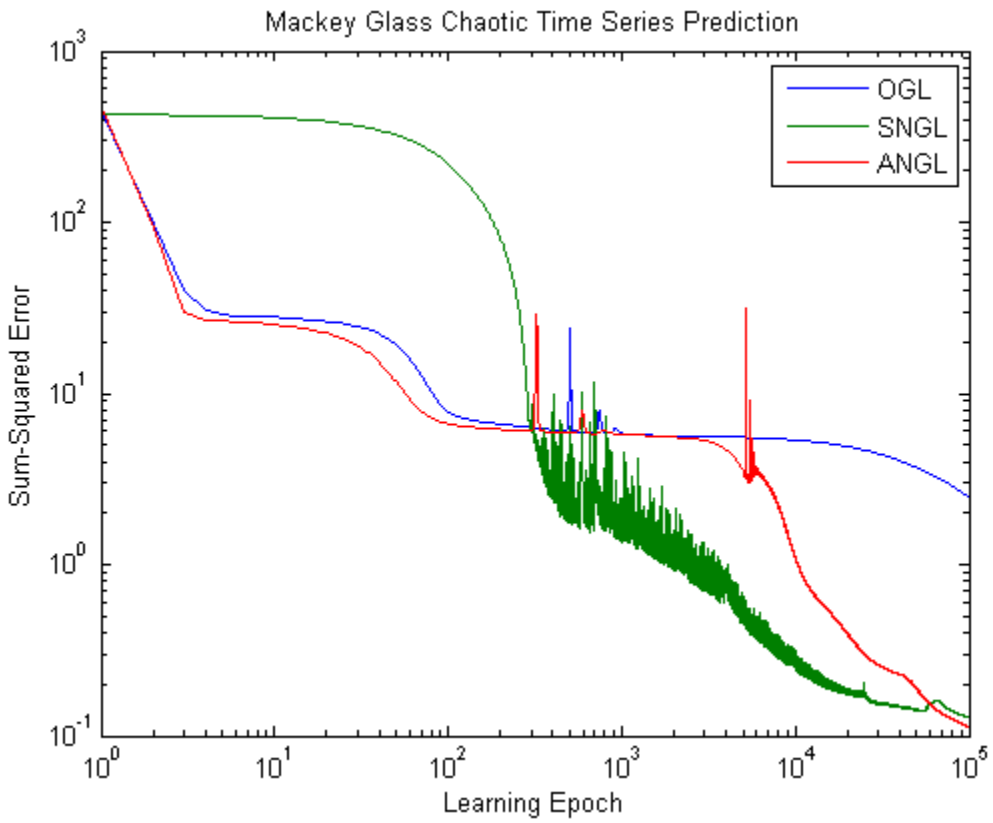

Fig. 3.13: The sum-squared error of the OGL, ANGL, and SNGL algorithms on the Mackey-Glass time series prediction problem.

Table 3.4: Mackey-Glass experiment results.

Mackey-Glass	OGL	SNGL	ANGL
Hidden Units	10	10	10
Nominal Learning Rate	1e-3	1e-3	1e-3
Initial Learning Rate	1e-3	2.3137e-6	1e-3
Annealed Learning Rate	Yes	Yes	No
Prior Probability on Parameters	Yes	Yes	No
Hyperparameter of Prior Distribution	0.01	1e-3	N/A
Fisher Information Matrix Size	N/A	(1,10)	61
Adaptation Rate	N/A	N/A	1e-5
Number of Extra Parameters	0	101	3,721
Final Learning Rate	1e-5	5.7206e-4	0.0023
Minimum Sum-Squared Error	2.4944	0.1283	0.1127

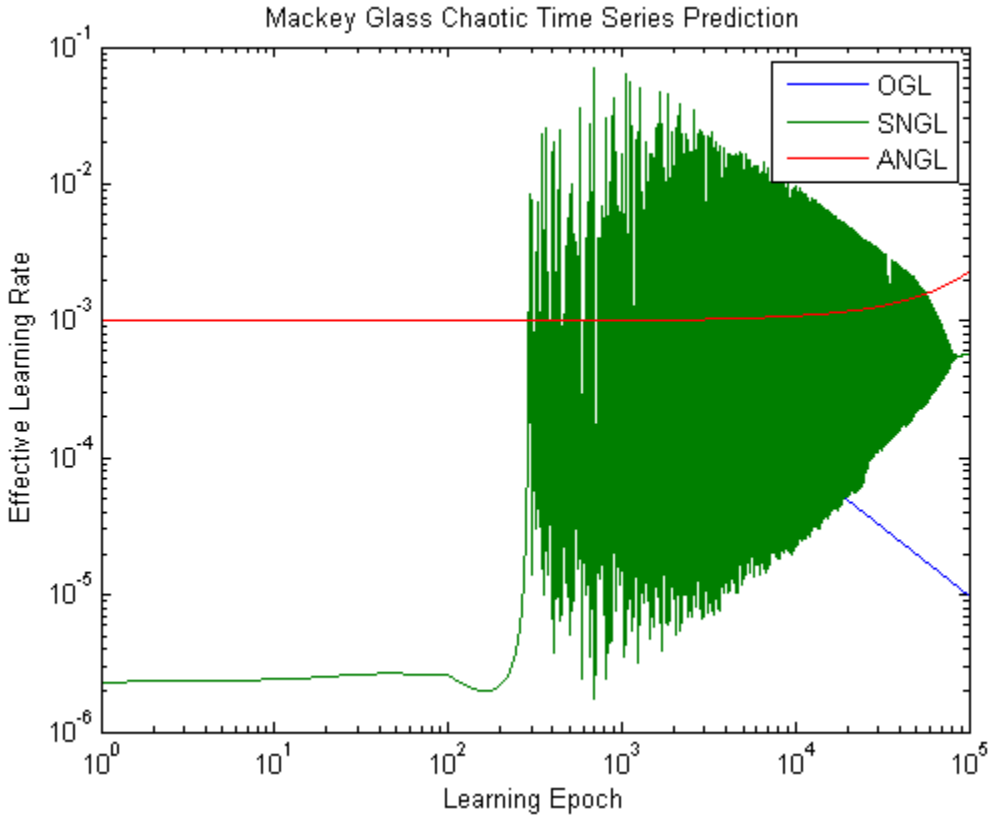

Fig. 3.14: The effective learning rate of the OGL, ANGL, and SNGL algorithms on the Mackey-Glass time series prediction problem.

space and the one-to-one correspondence between these points and the points in the higher-dimensional space. However, the nonlinear map between these two spaces is still not known. A neural network should be able to find an approximation of this map.

For example, a neural network can learn the map of a noisy semicircle between the angles $\frac{\pi}{6}$ and $\frac{11\pi}{6}$ given the following inputs

$$\mathbf{x}_n = \begin{bmatrix} \cos\left(\frac{5\pi n}{3N} + \frac{\pi}{6}\right) \\ \sin\left(\frac{5\pi n}{3N} + \frac{\pi}{6}\right) \end{bmatrix} + \mathbf{z}_n, \tag{3.25}$$

where $N = 100$, $\mathbf{z}_n \sim \mathcal{N}(0, \sigma^2\mathbf{I})$, $\sigma = 0.1$, and by using t_n (as described above) as the output. Figure 3.15 shows a plot of the data set.

Network Architecture

The network architecture for this problem is similar to the network architecture used in the Mackey-Glass problem. For this network, there are two inputs and eight hidden nodes. Table 3.5 shows the results of the experiment and the parameters used in the OGL, SNGL, and ANGL algorithms.

Performance Comparison

The sum-squared error plots are given in fig. 3.16. There are multiple plateaus for all three algorithms. The OGL and ANGL algorithms both reach a plateau near epoch 20 and start to leave it near epoch 50. They reach a second plateau near epoch 150. The ANGL algorithm leaves the its second plateau near epoch 2,000. It flattens out again near 4,000.

The performance of SNGL lags compared to the other two algorithms for the first hundred learning epochs. Again, this is because the SNGL algorithm starts with a small learning rate at the beginning as can be seen in fig. 3.17. The effective learning rate of SNGL increases significantly near epoch 100. There is a corresponding decrease in the SSE near this epoch. The learning rate rises and falls dramatically with a closing envelope due to the annealing of the learning rate. The learning rate begins to converge near epoch 6,000.

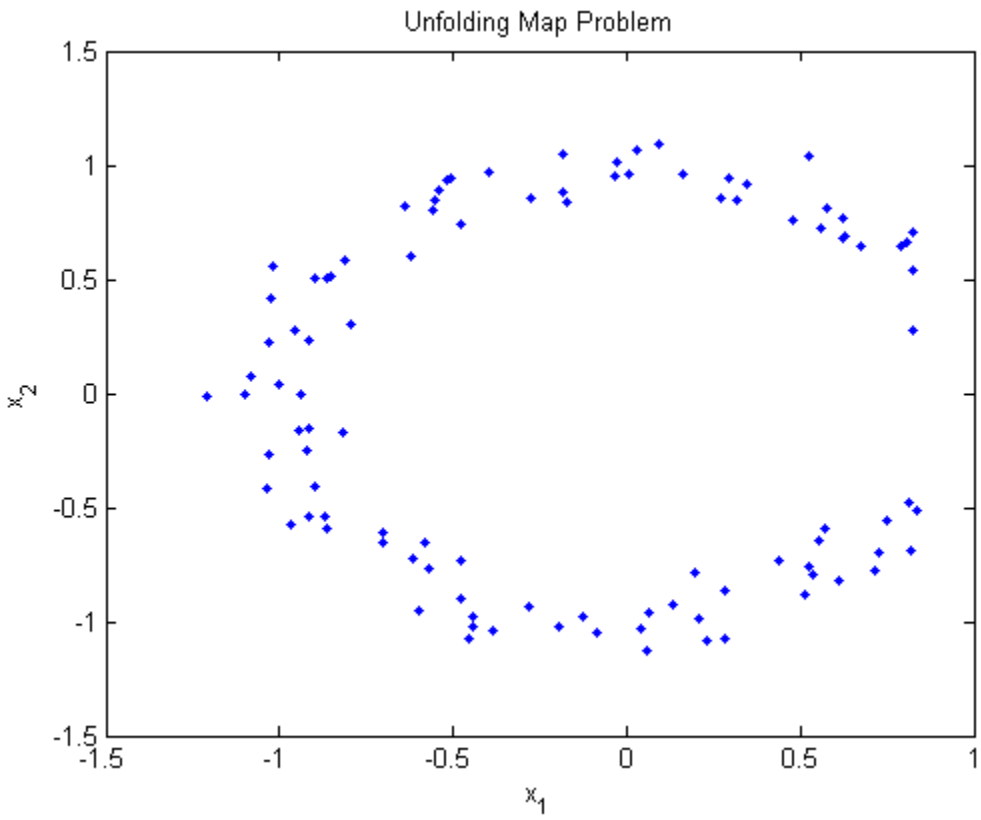

Fig. 3.15: An instance of 100 points on the semicircle between $\frac{\pi}{6}$ and $\frac{11\pi}{6}$ that has been corrupted by additive white Gaussian noise with $\sigma = 0.1$.

Table 3.5: Nonlinear dimension reduction experiment results.

Nonlinear Dimension Reduction	OGL	SNGL	ANGL
Hidden Units	8	8	8
Nominal Learning Rate	0.001	0.001	0.001
Initial Learning Rate	0.001	2.9440e-5	0.001
Annealed Learning Rate	Yes	Yes	No
Prior Probability on Parameters	Yes	Yes	No
Hyperparameter of Prior Distribution	0.01	0.01	N/A
Fisher Information Matrix Size	N/A	(1,8)	33
Adaptation Rate	N/A	N/A	1e-4
Number of Extra Parameters	0	65	1,089
Final Learning Rate	1e-4	0.0086	0.0027
Minimum Sum-Squared Error	0.8938	0.0759	0.4113

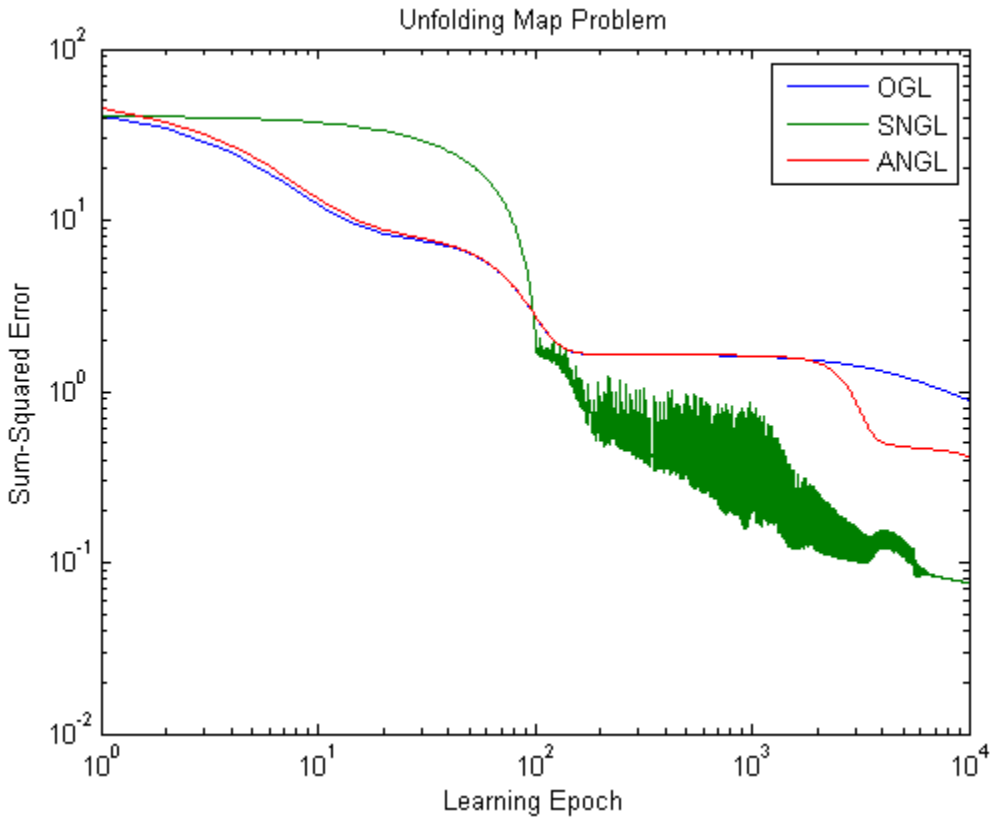

Fig. 3.16: The sum-squared error of the for the OGL, SNGL, and ANGL algorithms learning the noisy semicircle unfolding problem.

There is a corresponding convergence in the SSE, too. In this experiment, SNGL performed significantly better than OGL or ANGL.

3.6.2 Locally Linear Embedding

Locally Linear Embedding (LLE) [47] is an interesting approach because it uses linear models to map nonlinear manifolds into a lower-dimensional space. The details of the algorithm are beyond the scope of this work. However, a neural network can be trained to learn the unknown mapping.

Figure 3.18 shows the S-curve [46, 47]. It is an interesting data set. Table 3.6 has the network parameters and the experiment's results.

Figure 3.19 shows the SSE of each learning epoch of OGL, SNGL, and ANGL. All three make very little progress during the first 50 epochs. Significant improvements are achieved about epoch 400. The SNGL algorithm begins its fluctuations after epoch 300. There is a decreasing envelope due to annealing the learning rate of SNGL. The SSE converges near epoch 70,000. Both OGL and ANGL leave their second plateau near epoch 1,000. They reach a third plateau near epoch 10,000. Then ANGL's SSE jumps up and then descends further. It takes another jump near epoch 30,000 and fluctuates until epoch 50,000. It remains steady until epoch 80,000 and then begins fluctuating again. The performance of SNGL and ANGL is comparable. OGL performs worse as it never escapes its third plateau.

Figure 3.20 shows the effective learning rates of OGL, SNGL, and ANGL in this experiment. The learning rate of OGL is held constant at 10^{-5} as this achieved the best performance. This learning rate was also used on both SNGL and ANGL. The effective learning rate of SNGL begins to decrease after the fifth epoch and continues to do so until the hundredth. It increases dramatically between epoch 100 and 300. It then fluctuates for the rest of its execution. The envelope of the fluctuation begins to steadily decrease near epoch 70,000. This corresponds to where the SSE of the algorithm converges. The learning rate for ANGL begins to increase near epoch 1,000. The slope increases after epoch 6,000 and then decreases slightly after epoch 10,000. This corresponds to the sudden jump and then steady decrease of the SSE of the algorithm.

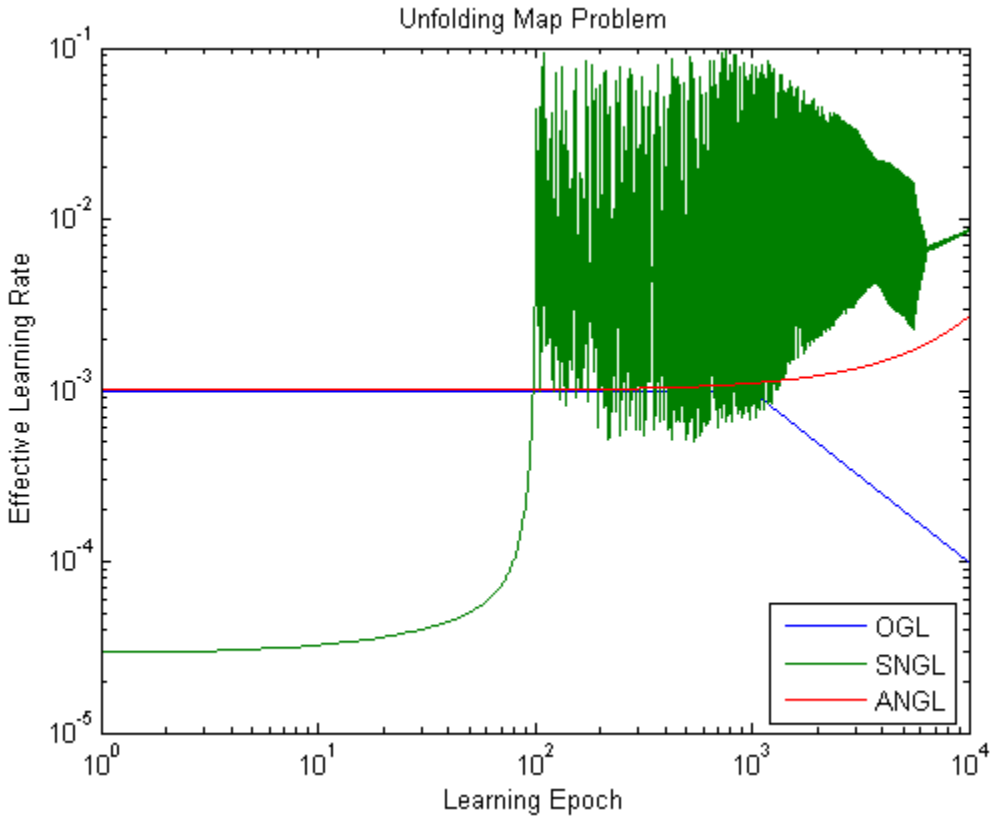

Fig. 3.17: The effective learning rate of the OGL, SNGL, and ANGL algorithms on the noisy semicircle unfolding Problem.

Table 3.6: Locally linear embedding results.

LLE	OGL	SNGL	ANGL
Hidden Units	5	5	5
Nominal Learning Rate	1e-5	1e-3	1e-5
Initial Learning Rate	1e-5	9.87e-6	1.0001e-5
Annealed Learning Rate	No	Yes	No
Prior Probability on Parameters	Yes	Yes	No
Hyperparameter of Prior Distribution	0.01	0.01	N/A
Fisher Information Matrix Size	N/A	(2,5)	32
Adaptation Rate	N/A	N/A	1e-4
Number of Extra Parameters	0	29	1,024
Final Learning Rate	1e-5	6.79e-7	1.17e-4
Minimum Sum-Squared Error	10.29	5.93	4.75
Minimum Learning Epoch	100,000	99,964	86,023

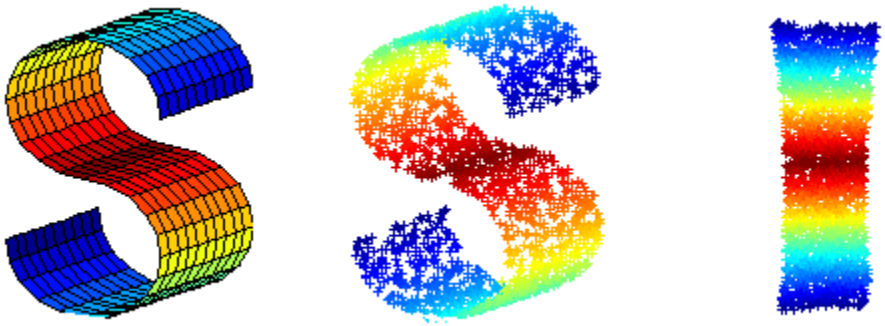

Fig. 3.18: The S-curve manifold, a set of sampled data and its image under a locally linear embedding map.

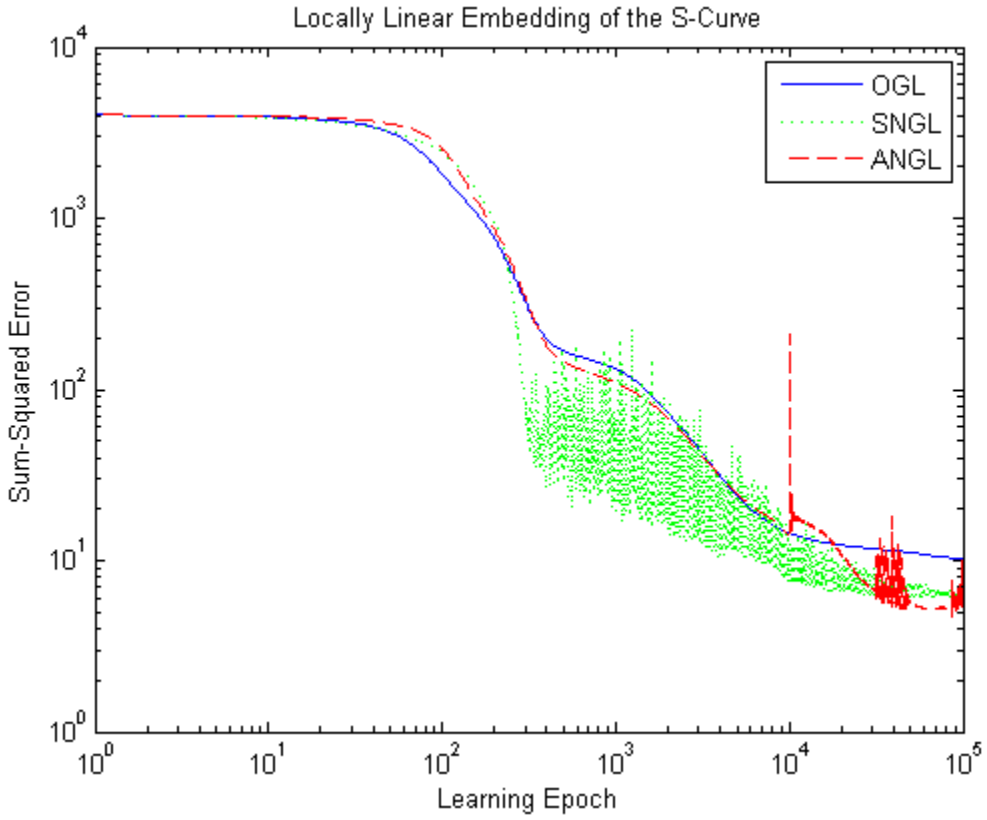

Fig. 3.19: The sum-squared error of the for the OGL, SNGL, and ANGL algorithms on the S-curve problem.

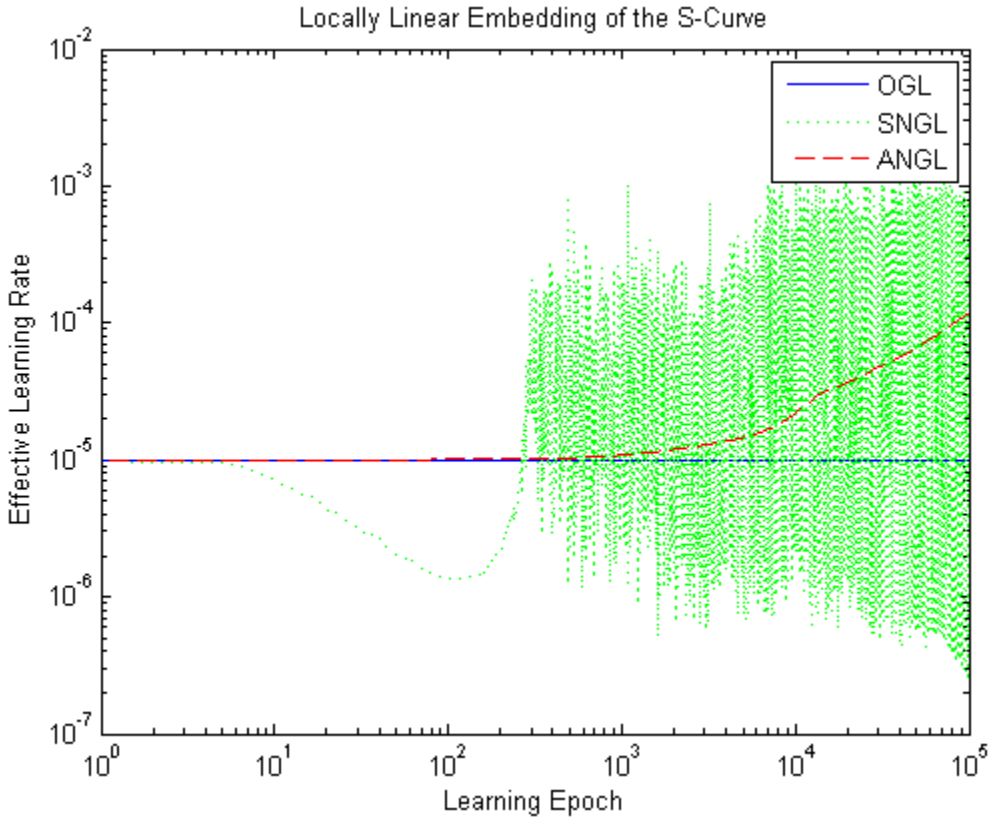

Fig. 3.20: The effective learning rate of the OGL, SNGL, and ANGL algorithms on the noisy semicircle unfolding problem.

3.7 Computational Complexity Comparison

Each of the learning algorithms use the same structure in their implementations. They all iterate through the training set calculating the errors from the current parameters. They then use those errors to calculate the parameters to be used in the next iteration. Hence, this section discusses the complexity of the learning rules of SNGL and ANGL assuming that the complexity of OGL is one.

Table 3.7 shows both complexities of SNGL and ANGL. The SNGL algorithm uses a smaller Fisher information matrix. However, it inverts its estimate of the Fisher information matrix. Matrix inversion has a computational complexity of $O(n^3)$ where n is the number of rows or columns of the square matrix. Its spatial complexity in terms of its parameters is $O(n^2)$. The ANGL algorithm estimates the inverse of a larger matrix. Mutliplying a vector by a square matrix has complexity $O(n^2)$. Thus, its complexity in both space and time are nearly the same. The last two columns of Table 3.7 shows the percentage of the space and time complexity of SNGL compared to those of ANGL. Clearly, SNGL uses signficantly less memory than ANGL. Its computational complexity is also significantly less than ANGL even though it inverts the Fisher information matrix.

The savings in parametric complexity matters because real-world problems in machine learning may have a large number of inputs. The networks used also have a large number of hidden nodes. Using the ANGL requires a lot of memory. This handicap limits the types

Table 3.7: Comparison between the computational complexity of the SNGL and ANGL algorithms in both space and time.

Algorithm	SNGL		ANGL	Percentage	
Complexity	Space	Time	Both	Space	Time
XOR	5	9	81	6%	11%
LDPC	200	2,000	28,900	1%	7%
IRIS	25	91	1,225	2%	7%
MG	101	1,001	3,721	3%	27%
SCIRC	65	513	1,089	6%	47%
SCURVE	29	133	1,024	3%	13%

of problems with which ANGL could be used. With SNGL, the memory requirements will still impose limits but not nearly to the same extent as with ANGL.

The computational complexity savings shows that inverting a block-diagonal Fisher information matrix in SNGL is cheaper than multiplying out the much larger inverse of the Fisher information matrix in ANGL.

3.8 Comparison with Other Optimization Algorithms

There are two general categories of optimization algorithms: Line Search and Trust Region [22]. Both methods determine the correct step size to take in the parameter space. Both categories will be discussed at a high level.

3.8.1 Line Search

The line search algorithm computes a descent direction and solves a one-dimensional optimization in that direction. A descent direction $\mathbf{P}$ is defined as a direction vector (or, in the case of neural networks, a matrix) such that

$$\langle \mathbf{P}, \nabla J \rangle < 0, \tag{3.26}$$

where ∇J is the gradient of the objective function. The goal of a line search algorithm is to minimize the function J in the direction of $\mathbf{P}$. Thus, the one-dimensional minimization problem can be written as a First-Order Taylor Series expansion of J about $\mathbf{W}$,

$$J(\mathbf{W} + \alpha\mathbf{P}) = J(\mathbf{W}) + \alpha\langle \mathbf{P}, \nabla J(\mathbf{W}) \rangle. \tag{3.27}$$

Hence, a descent direction reduces the error. Differentiating (3.27) with respect to alpha and assigning it to zero simply restates the goal of a vanishing gradient. From this point of view, there is no specific benefit to the line search method compared to ordinary gradient descent because the value of α vanishes and is thus irrelevant.

The Second-Order Taylor Series expansion of J about $\mathbf{W}$ is a quadratic function of α:

$$\mu(\alpha) = J(\mathbf{W}) + \alpha\langle \mathbf{P}, \nabla J(\mathbf{W})\rangle + \frac{1}{2}\alpha^2 \operatorname{tr}\left(\mathbf{P}^T \nabla^2 J(\mathbf{W})\mathbf{P}\right). \tag{3.28}$$

In this case, the derivative of (3.28) is a linear equation whose solution is

$$\alpha = -\frac{\operatorname{tr}\left(\mathbf{P}^T \nabla J(\mathbf{W})\right)}{\operatorname{tr}\left(\mathbf{P}^T \nabla^2 J(\mathbf{W})\mathbf{P}\right)}. \tag{3.29}$$

Let λ_1 be the largest eigenvalue of the Hessian matrix $\nabla^2 J$. If the Hessian matrix is positive-definite, then it is true that $\operatorname{tr}\left(\mathbf{P}^T \nabla^2 J \mathbf{P}\right) \leq \lambda_1 \operatorname{tr}\left(\mathbf{P}^T \mathbf{P}\right)$. By choosing $\mathbf{P} = -\nabla J$ as the descent direction, then $\alpha \geq \frac{1}{\lambda_1}$. Remember that the effective learning rate of the Simplified Natural Gradient Learning algorithm is $\frac{\eta}{\lambda_1}$ where η is the learning rate chosen *a priori*. If, instead, the Hessian matrix is invertible and the descent direction is chosen to be

$$\mathbf{P} = -\left[\nabla^2 J\right]^{-1} \nabla J, \tag{3.30}$$

then $\alpha = 1$.

An alternative method to computing the step size is numerically solving for it by using (3.29) iteratively. For example, an initial parameter $\mathbf{W}_0$ is chosen with descent direction $\mathbf{P}_0 = -\nabla J(\mathbf{W}_0)$. Then α_0 is computed using (3.29). The new parameter is $\mathbf{W}_1 = \mathbf{W}_0 + \alpha_0 \mathbf{P}_0$. The algorithm continues forward using $\mathbf{P}_0$ until α vanishes or the maximum number of iterations is exceeded.

Figure 3.21 shows the performance comparison of the Line Search Method (LSM) using the Fisher information matrix defined in (2.83) with the other algorithms. The performance of LSM lags behind SNGL to some extent, but generally performs in a similar fashion. The LSM algorithm performed only one iteration of the line search for each update. Thus, the learning rule is

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \frac{\operatorname{tr}\left(\nabla J(\mathbf{W}_t)^T \nabla J(\mathbf{W}_t)\right)}{\operatorname{tr}\left(\nabla J(\mathbf{W}_t)^T \mathbf{G}_t \nabla J(\mathbf{W}_t)\right)}. \tag{3.31}$$

Figure 3.22 shows the effective learning rates of the Line Search Method compared to

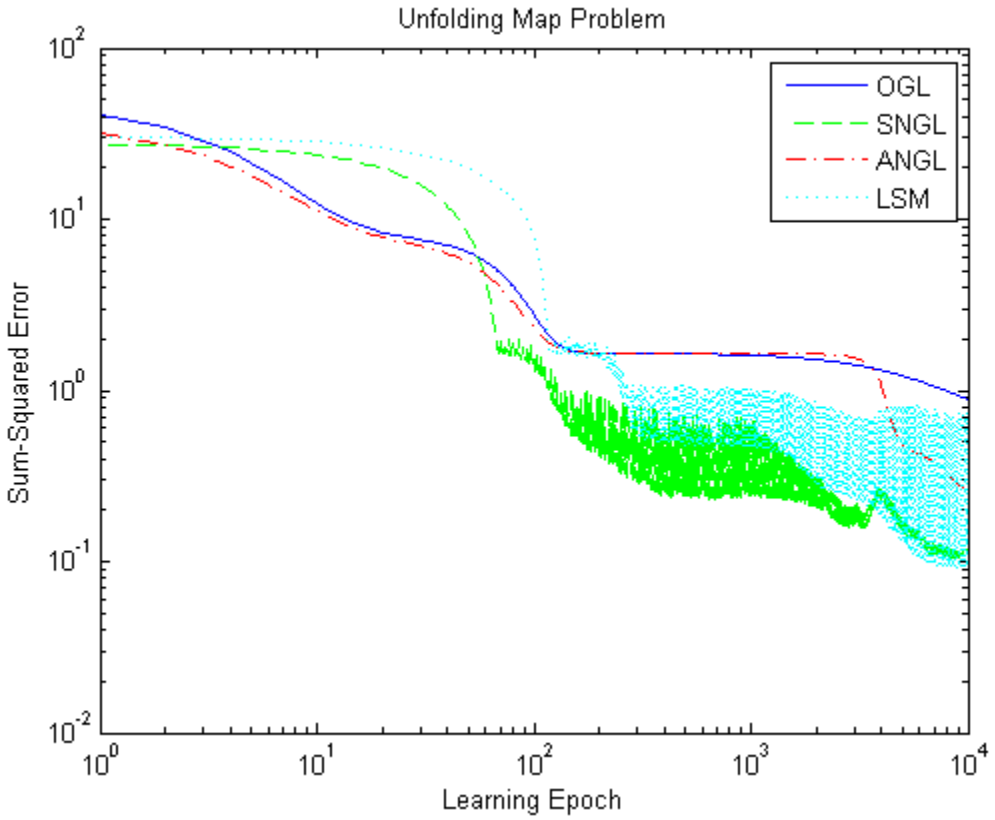

Fig. 3.21: The sum-squared error of the for the OGL, SNGL, ANGL, and Line Search Method (LSM) algorithms on the noisy semicircle unfolding problem.

the other algorithms. LSM and SNGL have similar curves. LSM will eventually converge on its learning rate like SNGL, but requires more time in terms of learning epochs.

There are more sophisticated algorithms that use the Line Search Method (e.g., Conjugate Gradient and BFGS). These algorithms are fairly complicated and building matrix versions would require significant time. In order to provide a more broad and general comparison it is better to discuss trust region methods.

3.8.2 Trust Region

A Trust Region Method determines regions where the Hessian matrix is positive-definite. The Hessian matrix is used in a Newton-type method inside the trust region. Outside the trust region, ordinary gradient descent is used. Thus, a Trust Region Method computes its descent direction as a linear combination of $\mathbf{P}_D = -\nabla J$ and $\mathbf{P}_U = -\mathbf{G}^{-1}\nabla J$. The descent direction is a function of the parameter τ such that

$$\mathbf{P}(\tau) = \begin{cases} \mathbf{P}_U & \tau \leq 0 \\ (1-\tau)\mathbf{P}_U + \tau\mathbf{P}_D & 0 < \tau < 1, \\ \mathbf{P}_D & \tau \geq 1. \end{cases} \tag{3.32}$$

The parameter τ is determined by maximizing

$$\sigma(\tau) = \|(1-\tau)\mathbf{P}_U + \tau\mathbf{P}_D\|. \tag{3.33}$$

Differentiating $\sigma(\tau)$ and solving for τ yields,

$$\sigma'(\tau) = 0 \tag{3.34}$$

$$\frac{(1-\tau)\operatorname{tr}\left(\mathbf{P}_U^T\mathbf{P}_U\right) + (\tau - \frac{1}{2})\operatorname{tr}\left(\mathbf{P}_U^T\mathbf{P}_D\right) - \tau\operatorname{tr}\left(\mathbf{P}_D^T\mathbf{P}_D\right)}{\|(1-\tau)\mathbf{P}_U + \tau\mathbf{P}_D\|} = 0 \tag{3.35}$$

$$\tau = \frac{\operatorname{tr}\left(\mathbf{P}_U^T\mathbf{P}_U\right) - \frac{1}{2}\operatorname{tr}\left(\mathbf{P}_U^T\mathbf{P}_D\right)}{\operatorname{tr}\left(\mathbf{P}_U^T\mathbf{P}_U\right) - \operatorname{tr}\left(\mathbf{P}_U^T\mathbf{P}_D\right) + \operatorname{tr}\left(\mathbf{P}_D^T\mathbf{P}_D\right)}. \tag{3.36}$$

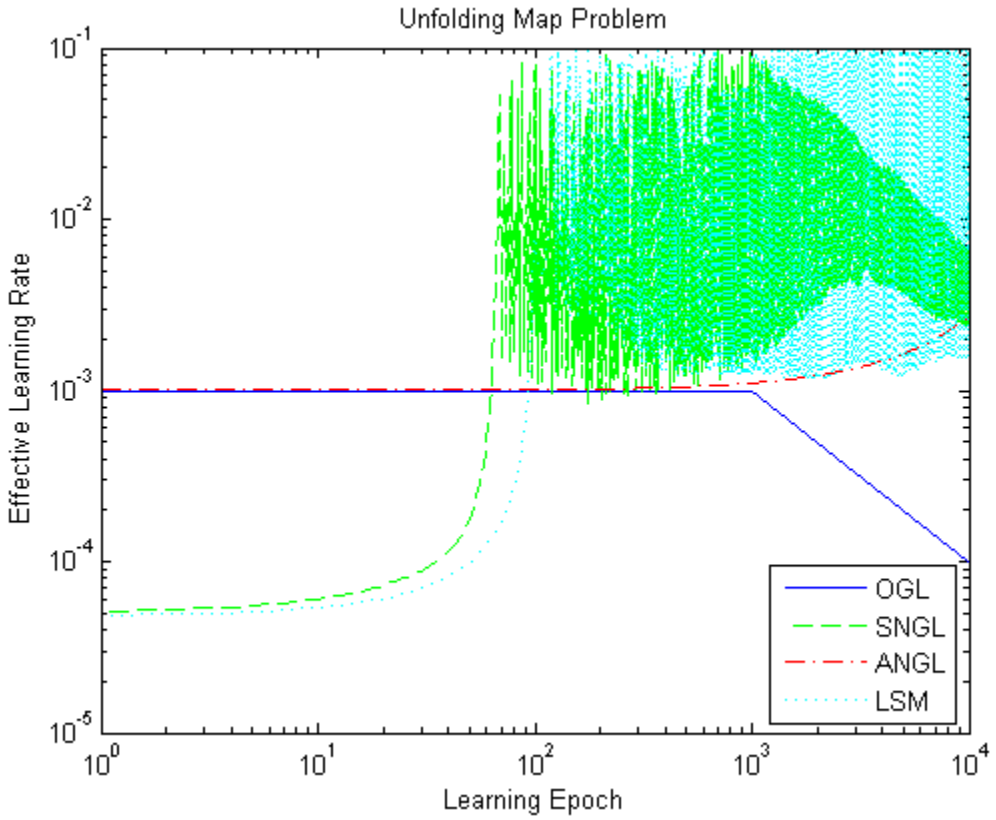

Fig. 3.22: The effective learning rate of the OGL, SNGL, ANGL, and Line Search Method (LSM) algorithms on the noisy semicircle unfolding problem.

When $\mathbf{P}_U$ and $\mathbf{P}_D$ are orthogonal to each other, then τ is

$$\tau = \frac{\|\mathbf{P}_U\|^2}{\|\mathbf{P}_U\|^2 + \|\mathbf{P}_D\|^2}. \tag{3.37}$$

Also, when $\mathbf{P}_D = \beta \mathbf{P}_U$, then τ is

$$\tau = \frac{1 - \frac{\beta}{2}}{1 - \beta + \beta^2}. \tag{3.38}$$

Figure 3.23 shows the performance of the Trust Region Method using the Fisher information matrix compared to the Line Search Method and the other algorithms discussed in this chapter. It starts out taking slightly larger steps than SNGL but not as steep as OGL and ANGL. It hits a plateau at learning epoch 100 with all the other algorihms. It maintains a smooth performance like OGL and ANGL. It finds the next lower plateau before ANGL. Before epoch 100, the Trust Region Method behaves similarly to SNGL. That is consistent with τ being small. After epoch 100, it behaves similarly to OGL. This is consistent with τ being close to unity.

Figure 3.24 shows that value of the parameter τ during the course of the Trust Region Method algorithm. Note that it starts out small and then increases during the first 100 learning epochs. When it becomes unity, the Fisher information does not contain any useful information and only multiplies the gradient by $\frac{1}{\alpha}$ where α is the hyperparameter of the prior distribution. Hence, the parameter τ approaches unity because the trust region has grown significantly. Remember that the inverse Fisher information matrix is used in the determination of the size of the trust region. Thus, when the Fisher information matrix is nearly equal to $\alpha\mathbf{I}$, then the trust radius is $\frac{1}{\alpha}$ for almost all the nodes in each layer.

3.9 Normalizing the Error Gradient

Normalizing the gradient is frequently done in many optimization algorithms [22]. This is done so that fixed size steps are taken in the course of the algorithms as described above. Using a normalized gradient in neural network training to compare algorithms on a level

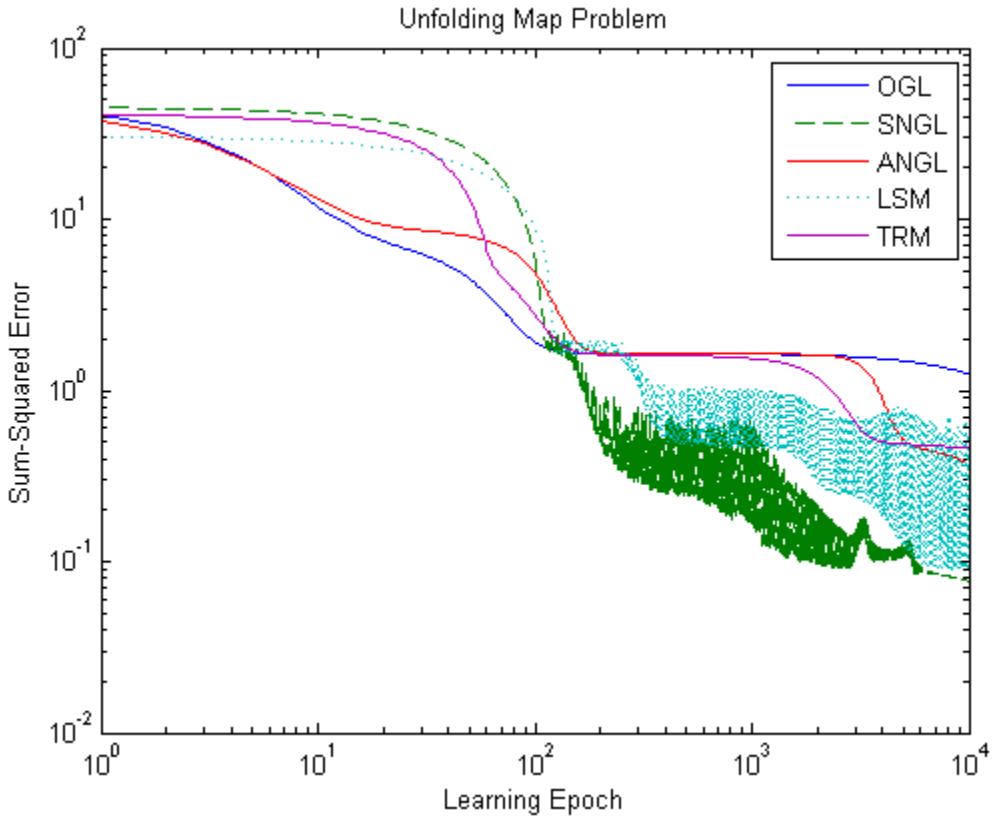

Fig. 3.23: The sum-squared error of the for the OGL, SNGL, ANGL, Line Search Method (LSM), and Trust Region Method (TRM) algorithms on the noisy semicircle unfolding problem.

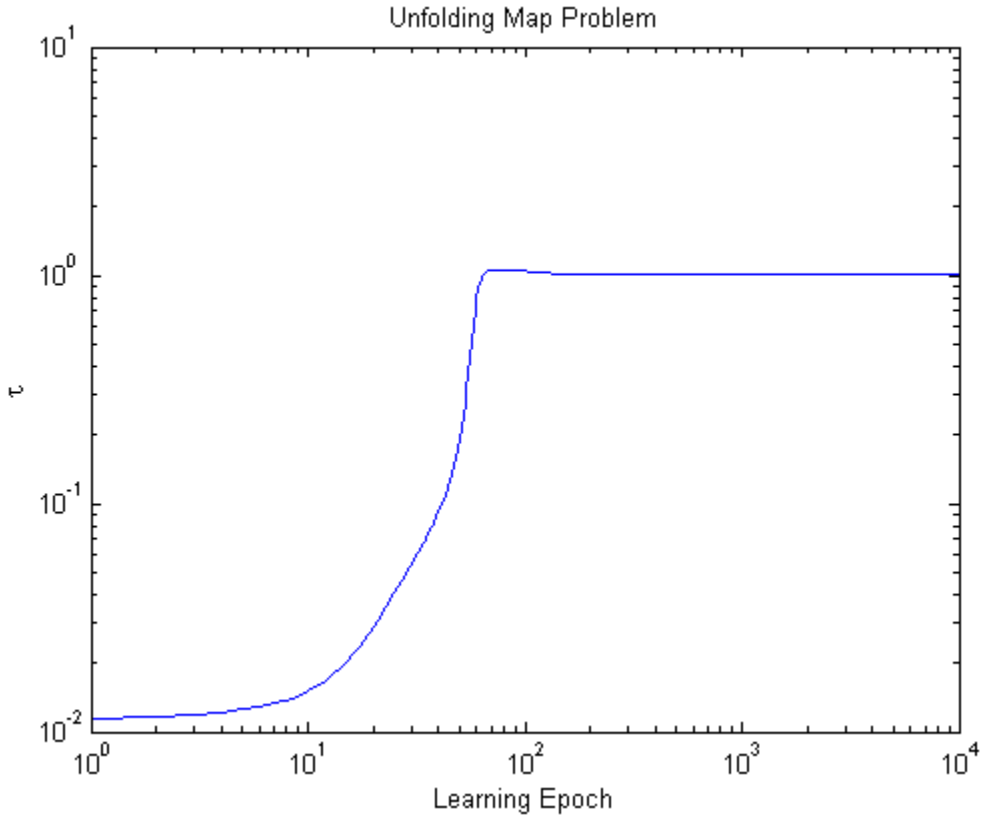

Fig. 3.24: The value of the parameter τ used in the trust region method algorithm on the noisy semicircle unfolding problem.

playing field has a problem: The effect of the normalization on the prior distribution of the connection weight parameters. The purpose of this prior distribution is to mitigate overtraining. Normalizing the gradient before subtracting the scaled-down version of the current weights negates the effect of the prior. This is because the gradient is always the same length (i.e., one). Normalizing the gradient after the subtraction amplifies the noise in the data set when the gradient gets small. Thus, normalizing the gradient of the error function should be done without subtracting the current weights multiplied by the hyperparameter α.

The three learning algorithms OGL, ANGL, and SNGL performed similarly with normalized gradients. What can be concluded from these results is that improving the performance of neural network training algorithms relies more on determining when to take larger or smaller steps than on determining the right direction.

3.10 Conclusion

A natural gradient algorithm uses the Fisher information matrix of its current parameters in a Newton-type gradient descent method by replacing the Hessian matrix. The power of natural gradient algorithms is their efficiency [33]. However, there are different ways in which to compute the Fisher information matrix. The Simplified Natural Gradient Learning (SNGL) algorithm uses a smaller Fisher information matrix that is inverted in a batch training algorithm. Its performance is compared to the Ordinary Gradient Learning (OGL) algorithm and the Adaptive Natural Gradient Learing (ANGL) algorithm. The SNGL algorithm performs well. The improvement comes from using a relatively low learning rate in the beginning and a much higher rate when a better solution is found.

Chapter 4

Summary and Future Work

This chapter summarizes the ideas discussed in this dissertation and describes how it might be applied or extended in the future by others. In a letter to Robert Hooke, Isaac Newton wrote, "If I have seen further it is only by standing on the shoulders of giants" [53]. Standing on the shoulders of giants is an excellent description of this work.

4.1 Research Summary

This dissertation describes a simplification to a complex learning algorithm for multilayer perceptrons. Ordinary gradient descent learning works quite well for many problems. The main problem is the plateau effect [14,33]. When a gradient descent learning algorithm encounters a local minimum in its parameter space, the gradient gets smaller and smaller with each succeeding step of the learning algorithm.

Amari developed an algorithm to avoid local minima by following the curvature of a manifold in the parameter space of the perceptron [33–35, 41, 54–56]. By using a recursive estimate of the inverse of the Fisher information matrix of the parameters, the algorithm was able to accelerate learning in the directions with low information while not overstepping in any directions with high information.

Training a multilayer perceptron is a nonlinear least-squares optimization problem [20, 22]. Instead of using the Hessian of the objective function, a natural gradient algorithm uses the Fisher information matrix derived from a probability distribution that describes the parameter space. A typical formulation for the score function (i.e., the gradient of the log-likelihood function) is a matrix with a row for each training example and a column for each parameter. The Fisher information matrix would then be a square matrix whose dimension is equal to the number of parameters squared. For large problems with many

inputs, this dimension becomes quite large. For example, training a two-layer perceptron with n inputs, m outputs and p hidden nodes would require $O((m+n)^2p^2)$ memory cells to store the Fisher information matrix.

4.2 Contributions

The new learning algorithm described in this dissertation was developed to reduce the memory requirements and computational overhead of Amari's Adaptive Natural Gradient Learning algorithm [34]. The central idea is that the parameters are matrices and not vectors. While it is true that matrices are isomophic to vectors [57], the action of these parameters is the transformation of input vectors into output vectors. Thus, the direct sum of these matrices should be used instead of stacking the rows or columns of the matrices and then stacking the matrices. The chief benefit of this new formulation is that the Fisher information matrix is now a block diagonal matrix. In the example from the previous paragraph, only $O(m^2+p^2)$ memory cells would be required in a natural gradient learning algorithm. Furthermore, inverting the matrix would only require $O(m^3+p^3)$ operations instead of $O((m+n)^3p^3)$. This new method uses less memory and less processing time than Amari's Adaptive Natural Gradient.

The experiments have shown that the performance of this simplified natural gradient learning algorithm performs comparably to Amari's Adaptive Natural Gradient Learning on a variety of problems. They also show that the improvement to performance of these natural gradient algorithms comes from adaptively using lower learning rates in the beginning and then increasing them only when all the information has been filtered out of the error vectors.

4.3 Future Work

There are many areas of research in Machine Learning and elsewhere to which this research could be applied. This section will discuss only two of them: Speech Recognition and Nonlinear Dimension Reduction.

Speech Recognition is a very difficult problem [58–60]. A speech recognition system is complex and a neural network usually serves the function of estimating the probability of a

phoneme given a few observations of the voice spectogram [61,62]. These networks are quite large having 26 inputs for the Mel Frequency Cepstral Coefficients and their derivatives, roughly 100 hidden units and nearly 50 outputs, one for each phoneme. Thus, a smaller Fisher information matrix would be very important given the large size of the network. Such a network would have 7,750 parameters. An Adaptive Natural Gradient Learning algorithm would require 60,062,500 memory cells to store the Fisher information matrix. Using single-precision floating-point format requiring four bytes for each variable would require 229 megabytes of memory. The Simplified Natural Gradient Learning algorithm would only require 12,500 memory cells. There is a large amount of training data, too. Hence, the reduction in learning cycles required to reach good performance levels will decrease overall training time.

Nonlinear Dimension Reduction was discussed in sec. 3.6. In those experiments, the neural network training algorithms viewed the Nonlinear Dimension Reduction method as a black box and tried to imitate its output. There is much more that can be done. There is some statistical data that can be gleaned from the original data and the reduced data. This data could be used in the maximum likelihood model from which the Fisher information matrix is derived. Doing so may improve the performance of the training algorithms significantly. The relationship between the fitting errors of the Nonlinear Dimension Reduction algorithm and the errors of the neural network should also be investigated to discover the confidence intervals of the neural network outputs.

4.4 Conclusion

A brief summary of this research and its potential for further research has been discussed. While not all potential research paths were enumerated, the two most promising were briefly discussed. This final chapter concludes this dissertation.

References

[1] F. Mittelbach and M. Goossens, *The LaTeX Companion*, 2nd ed., ser. Tools and Techniques for Computer Typesetting. Boston, MA: Addison-Wesley, 2004.

[2] T. H. Huxley, "On the hypothesis that animals are automata, and its history," *Fortnightly Review*, vol. XVL, p. 555, 1874.

[3] W. James, "Are we automata?" *Mind*, vol. 4, pp. 1–22, 1879.

[4] W. McColluch and W. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[5] D. O. Hebb, *The Organization of Behavior*. New York: John Wiley & Sons, 1949.

[6] A. Turing, "Intelligent machinery," in *The Essential Turing: The Ideas That Gave Birth to the Computer Age*, B. J. Copeland, Ed., pp. 395–432. New York: Oxford University Press, 2004.

[7] A. Turing, "Computing machinery and intelligence," *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950.

[8] A. Turing, "Can automatic calculating machines be said to think?" in *The Essential Turing: The Ideas That Gave Birth to the Computer Age*, B. J. Copeland, Ed., pp. 487–506. New York: Oxford University Press, 2004.

[9] S. Parker, *How the Body Works*. New York: Reader's Digest Association, 1994.

[10] K. Barnes and S. Weston, *The Human Body: How It Works*. New York: Barnes & Noble, 1997.

[11] M. A. Arbib, Ed., *The Handbook of Brain Theory and Neural Networks*, 2nd ed. Cambridge, MA: MIT Press, 2002.

[12] D. J. C. MacKay, *Information Theory, Inference, and Learning Algorithms*. New York: Cambridge University Press, 2003.

[13] M. Abramowitz and I. A. Stegun, Eds., *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Washington, DC: US Government Printing Office, 1972.

[14] C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford, UK: Oxford University Press, 1995.

[15] M. Biehl and H. Schwarze, "Learning by online gradient descent," *Journal of Physics*, vol. A, no. 28, pp. 643–656, 1995.

[16] S. Bös and S. Amari, "Annealed online learning in multilayer neural networks," in *On-line Learning in Neural Networks*, D. Saad, Ed., pp. 209–229. New York: Cambridge University Press, 1999.

[17] K. Fukumizu, "Effect of batch learning in multilayer neural networks," in *Proceedings of 5th International Conference on Neural Information Processing (ICONIP'98)*, pp. 67–70, 1998.

[18] T. Heskes and B. Kappen, "On-line learning processes in artificial neural networks," in *Mathematical Foundations of Neural Networks*, J. Taylor, Ed., pp. 199–233. Amsterdam, Netherlands: Elsevier, 1993.

[19] N. Murata, "A statistical study of on-line learning," in *On-line Learning in Neural Networks*, D. Saad, Ed., pp. 63–92. New York: Cambridge University Press, 1999.

[20] R. L. Burden and J. D. Faires, *Numerical Analysis*, 8th ed. Pacific Grove, CA: Brooks Cole, Dec. 2004.

[21] R. A. Horn and C. R. Johnson, *Matrix Analysis*. New York: Cambridge University Press, Feb. 1990.

[22] J. Nocedal and S. J. Wright, *Numerical Optimization*, ser. Springer Series in Operations Research. New York: Springer, 1999.

[23] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[24] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Washington DC: Spartan Books, 1962.

[25] M. L. Minsky and S. A. Papert, *Perceptrons*. Cambridge, MA: MIT Press, 1969.

[26] A. M. Chen, H. Lu, and W. Luo, "On the geometry of feed-forward neural network error surfaces," *Neural Computation*, vol. 5, no. 6, pp. 910–927, 1993.

[27] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing*. Cambridge, MA: MIT Press, 1986.

[28] R. Rojas, *Neural Networks*, ch. 7. New York: Springer-Verlag, 1996.

[29] M. D. Buhmann and M. J. Ablowitz, *Radial Basis Functions: Theory and Implementations*. New York: Cambridge University Press, 2003.

[30] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*, 1st ed. New York: Springer, 1991.

[31] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, no. 7, pp. 179–188, 1936.

[32] E. Anderson, "The irises of the gasp peninsula," *Bulletin of the American Iris Society*, no. 59, pp. 2–5, 1935.

[33] S. Amari, "Natural gradient works efficiently in learning," *Neural Computation*, vol. 10, no. 2, pp. 251–276, 1998.

[34] S. Amari, H. Park, and K. Fukumizu, "Adaptive method of realizing natural gradient learning for multilayer perceptrons," *Neural Computation*, vol. 12, no. 6, pp. 1399–1409, 2000.

[35] H. Park, S. Amari, and K. Fukumizu, "Adaptive natural gradient learning algorithms for various stochastic models," *Neural Networks*, vol. 13, no. 7, pp. 755–764, 2000.

[36] G. E. P. Box and N. R. Draper, *Empirical Model-Building and Response Surfaces*, p. 424. New York: John Wiley & Sons, 1987.

[37] R. A. Fisher, "On the mathematical foundations of theoretical statistics," *Philosophical Transactions of the Royal Society of London*, vol. 222, pp. 309–68, 1922.

[38] G. Casella and R. L. Berger, *Statistical Inference*, 2nd ed. Pacific Grove, CA: Duxbury Press, 2002.

[39] H. L. V. Trees, *Detection, Estimation, and Modulation Theory, Part I.* New York: John Wiley & Sons, 1968.

[40] T. K. Moon and W. C. Stirling, *Mathematical Methods and Algorithms for Signal Processing*. Upper Saddle River, NJ: Prentice Hall, Aug. 1999.

[41] S. Amari and H. Nagaoka, *Methods of Information Geometry*, ser. Translations of Mathematical Monographs, vol. 191. New York: Oxford University Press, 2000.

[42] M. K. Murray and J. W. Rice, *Differential Geometry and Statistics*, ser. Monographs on Statistics and Applied Probability, vol. 48. London: Chapman & Hall/CRC, 1993.

[43] J. Frank, P. Cheeseman, and J. Stutz, "When gravity fails: Local search topology," *Journal of Artificial Intelligence Research*, vol. 7, pp. 249–281, 1997.

[44] T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*. New York: Wiley-Interscience, June 2005.

[45] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

[46] J. B. Tenenbaum, V. de Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 290, pp. 2319–2323, Dec. 2000.

[47] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 290, pp. 2323–2326, Dec. 2000.

[48] M. Balasubramanian and E. L. Schwartz, "The isomap algorithm and topological stability," *Science*, vol. 295, Jan. 2002.

[49] D. K. Agrafiotis and H. Xu, "A self-organizing principle for learning nonlinear manifolds," in *Proceedings of the National Academy of Sciences of the United States of America*, vol. 99, no. 25, pp. 15 869–15 872, 2002.

[50] D. K. Agrafiotis, "Stochastic proximity embedding," *Journal of Computational Chemistry*, vol. 24, no. 10, pp. 1215–1221, 2003.

[51] C. M. Bachmann, T. L. Ainsworth, and R. A. Fusina, "Exploiting manifold geometry in hyperspectral imagery," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 43, no. 3, pp. 441–454, Mar. 2005.

[52] K. Q. Weinberger and L. K. Saul, "An introduction to nonlinear dimensionality reduction by maximum variance unfolding," in *AAAI*, 2006.

[53] I. Newton, Letter to Robert Hooke, Feb. 1676. Available: `http://en.wikiquote.org/wiki/Isaac_Newton#Sourced`

[54] S. Amari, "Information geometry of the EM and em algorithms for neural networks," *Neural Networks*, vol. 8, no. 9, pp. 1379–1408, 1995.

[55] S. Amari, "Neural learning in structured parameter spaces — natural riemannian gradient," in *Advances in Neural Information Processing Systems*, M. C. Mozer, M. I. Jordan, and T. Petsche, Eds., vol. 9, p. 127. Cambridge, MA: The MIT Press, 1997.

[56] S. Amari, H. Park, and T. Ozeki, *Geometrical singularities in the neuromanifold of multilayer perceptrons*, no. 14. Cambridge, MA: MIT Press, 2002.

[57] G. L. Bradley, *A Primer of Linear Algebra*. Upper Saddle River, NJ: Prentice Hall, 1975.

[58] L. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition*. Upper Saddle River, NJ: Prentice Hall, 1993.

[59] F. Jelinek, *Statistical Methods for Speech Recognition*. Upper Saddle River, NJ: Prentice Hall, 1998.

[60] X. Huang, A. Acero, and H.-W. Hon, *Spoken Language Processing: A Guide to Theory, Algorithm and System Development*. Upper Saddle River, NJ: Prentice Hall, 2001.

[61] J.-P. Hosom, R. Cole, and M. Fanty, "Speech recognition using neural networks," Center for Spoken Language Understanding (CSLU) Oregon Graduate Institute of Science and Technology, Technical Report, July 1999. Available: `http://cslu.cse.ogi.edu/tutordemos/nnet_recog/recog.html`

[62] J.-P. Hosom, J. de Villiers, R. Cole, M. Fanty, J. Schalkwyk, Y. Yan, and W. Wei, "Training hidden markov model/artificial neural network (hmm/ann) hybrids for automatic speech recognition (asr)," Center for Spoken Language Understanding (CSLU) OGI School of Science and Engineering (OGI) Oregon Health and Science University (OHSU), Technical Report, Feb. 2006. Available: `http://speech.bme.ogi.edu/tutordemos/nnet_training/tutorial.html`

Appendices

Appendix A
MATLAB Code for the Exclusive OR (XOR) Experiment

The MATLAB code for the Ordinary Gradient Learning, Simplified Natural Gradient Learning and Adaptive Natural Gradient Learning algorithms that learn the XOR function are included in this section.

A.1 Ordinary Gradient Learning

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This code trains a 2-2-1 network to learn the XOR function.
% It uses the Ordinary Gradient Learning Algorithm (OGL).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The number of learning epochs
L = 10000;

% The learning rate
eta = 0.25;

% Weight decay rate
alpha = 1e-2;

% Number of hidden nodes
q = 2;

% Target values
t = [ -1 ; 1 ; 1 ; -1 ];

% Input vectors
x = [-1 -1 ; -1 1 ; 1 -1 ; 1 1 ];

% Number of training examples
m = length(t);

% Number of inputs
n = size(x,2);

% Number of outputs
k = size(t,2);
```

```
% Initialize weights to random numbers between 1 and -1
w1 = 2 * rand(n+1,q) - 1;
w2 = 2 * rand(q+1,k) - 1;

% These arrays store the sum-squared error and effective learning rate
% respectively
oglsse = zeros(L,1);
oglelr = zeros(L,1);

for l = 1:L
    u1 = [ x ones(m, 1) ];  % Append ones to inputs for biases
    z1 = u1 * w1;           % Calculate hidden layer activations
    y1 = tanh(z1);          % Apply squashing function
    u2 = [ y1 ones(m, 1) ]; % Append ones to hidden layer outputs for
                            % bias
    z2 = u2 * w2;           % Calculate output layer activations
    y2 = tanh(z2);          % Apply squashing function
    e2 = t - y2;            % Calculate output error

    % Calculate output layer gradient
    g2 = -u2' * (e2 .* (1 - y2 .^2 )) + alpha * w2;

    % Calculate backpropagated error for hidden layer gradient
    b1 = (e2 * w2(1:q,1:k)') .* (1 - y1 .^ 2);

    g1 = -u1' * b1 + alpha * w1;    % Calculate hidden layer gradient
    w1 = w1 - min(eta, 1 / l) * g1; % Update hidden layer weights
    w2 = w2 - min(eta, 1 / l) * g2; % Update output layer weights
    oglsse(l) = trace(e2' * e2);    % Record sum-squared error
    oglelr(l) = min(eta, 1 / l);    % Record learning rate
end
```

A.2 Simplified Natural Gradient Learning

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This code trains a 2-2-1 network to learn the XOR function.
% It uses the Simplified Natural Gradient Learning Algorithm (SNGL).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The number of learning epochs
L = 10000;

% The learning rate
eta = 0.25;

% The weight decay rate
alpha = 1e-2;

% The number of hidden nodes
q = 2;

% The target values
t = [ -1 ; 1 ; 1 ; -1 ];

% The input vectors
x = [-1 -1 ; -1 1 ; 1 -1 ; 1 1 ];

% The number of training examples
m = length(t);

% The number of inputs
n = size(x,2);

% The number of outputs
k = size(t,2);

% Initialize weights to random numbers between 1 and -1
w1 = 0.2 * rand(n+1,q) - 0.1;
w2 = 0.2 * rand(q+1,k) - 0.1;

% These arrays store the sum-squared error and effective learning rate
% respectively
snglsse = zeros(L,1);
snglelr = zeros(L,1);

for l = 1:L
```

```
    u1 = [ x ones(m, 1) ];  % Append ones to the input vectors for the
                            % biases
    z1 = u1 * w1;           % Calculate hidden node activations
    y1 = tanh(z1);          % Apply the squashing function
    u2 = [ y1 ones(m, 1) ]; % Append ones to the hidden layer for the
                            % biases
    z2 = u2 * w2;           % Calculate output node activations
    y2 = tanh(z2);          % Apply the squashing function
    e2 = t - y2;            % Compute the error
    d2 = 1 - y2 .^2;        % Compute the Jacobian matrix for the
                            % output layer

    % Calculate the output layer's gradient
    g2 = -u2' * (e2 .* d2) + alpha * w2;

    % Calculate the output layer's Fisher information matrix
    G2 = ((e2 .* d2)' * u2 * u2' * (e2 .* d2)) / m + alpha * eye(k);

    d1 = 1 - y1 .^ 2;       % Compute the Jacobian matrix for the
                            % hidden layer

    % Calculate the backpropagated error of the hidden layer
    b1 = (e2 * w2(1:q,1:k)') .* d1;

    % Calculate the hidden layer's gradient
    g1 = -u1' * b1 + alpha * w1;

    % Calculate the hidden layer's Fisher information matrix
    G1 = (b1' * u1 * u1' * b1) / m + alpha * eye(q);

    % Update the weight matrices
    w1 = w1 - min(eta, 1 / l) * g1 * inv(G1);
    w2 = w2 - min(eta, 1 / l) * g2 * inv(G2);

    % Record the sum-squared error and effective learning rate for
    % this epoch
    snglsse(l) = trace(e2' * e2);
    snglelr(l) = min(eta, 1/l) / max([eig(G1); eig(G2)]);
end
```

A.3 Adaptive Natural Gradient Learning

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This code trains a 2-2-1 network to learn the XOR function.
% It uses the Amari's Adaptive Natural Gradient Learning algorithm
% (ANGL).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Number of learning epochs
L = 10000;

% Learning rate
eta = 0.25;

% Adaptation rate
delta = 0.1;

% Number of hidden nodes
q = 2;

% Target values
t = [ -1 ; 1 ; 1 ; -1 ];

% Input vectors
x = [-1 -1 ; -1 1 ; 1 -1 ; 1 1 ];

% Number of training examples
m = length(t);

% Number of inputs
n = size(x,2);

% Number of outputs
k = size(t,2);

% Initialize weights to random numbers between 1 and -1.
w1 = 2 * rand(n+1,q) - 1;
w2 = 2 * rand(q+1,k) - 1;

% Record the sum-squared error and effective learning rate for each
% epoch.
anglsse = zeros(L,1);
anglelr = zeros(L,1);
```

```
% Number of weights in hidden layer
n1 = (n + 1) * q;

% Number of weights in output layer
n2 = (q + 1) * k;

% Initialize Fisher information matrix inverse to identity matrix
G = eye(n1 + n2);

for l = 1:L
    u1 = [ x ones(m, 1) ];  % Append ones to input vectors for biases
    z1 = u1 * w1;           % Compute hidden node activations
    y1 = tanh(z1);          % Apply the squashing function
    u2 = [ y1 ones(m, 1) ]; % Append ones to hidden layer outputs
    z2 = u2 * w2;           % Compute output node activations
    y2 = tanh(z2);          % Apply the squashing function
    e2 = t - y2;            % Compute the error

    % Compute the output layer gradient
    g2 = -u2' * (e2 .* (1 - y2 .^2 ));

    % Compute the hidden layer backpropagated error
    b1 = (e2 * w2(1:q,1:k)') .* (1 - y1 .^ 2);

    % Compute the hidden layer gradient
    g1 = -u1' * b1;

    % Column order both matrices into one gradient vector
    r = [ g1(:); g2(:) ];

    % Multiply by the inverse Fisher information matrix
    p = G * r;

    % Reshape gradient vector back into two matrices
    p1 = reshape(p(1 : n1), n + 1, q);
    p2 = reshape(p(n1 + 1 : n1 + n2), q + 1, k);

    % Update weights
    w1 = w1 - min(eta,1/l) * p1;
    w2 = w2 - min(eta,1/l) * p2;

    % Update estimate of inverse Fisher information matrix
    G = (1+min(delta,1/l)) * G - min(delta,1/l) * (p*p') / (1+r'*p);

    % Record the sum-squared error and effective learning rate for
    % this epoch.
```

```
    anglsse(l) = trace(e2' * e2);
    anglelr(l) = min(eta,1/l) * min(eig(G));
end
```

Appendix B
MATLAB Code for the Low-Density Parity Check Code Experiment

The following function is used in all three algorithms.

```
function y = flipbits(x,p)
    r = rand(size(x));
    i = find(r < p);
    z = zeros(size(x));
    z(i) = 1;
    y = mod(x + z, 2);
end
```

B.1 Ordinary Gradient Learning

```
L = 10000;
alpha = 0.001;
beta = 0.01;

q = 10;

x = [0 0 0 0 0; 0 0 0 0 1; 0 0 0 1 0; 0 0 0 1 1;
     0 0 1 0 0; 0 0 1 0 1; 0 0 1 1 0; 0 0 1 1 1;
     0 1 0 0 0; 0 1 0 0 1; 0 1 0 1 0; 0 1 0 1 1;
     0 1 1 0 0; 0 1 1 0 1; 0 1 1 1 0; 0 1 1 1 1;
     1 0 0 0 0; 1 0 0 0 1; 1 0 0 1 0; 1 0 0 1 1;
     1 0 1 0 0; 1 0 1 0 1; 1 0 1 1 0; 1 0 1 1 1;
     1 1 0 0 0; 1 1 0 0 1; 1 1 0 1 0; 1 1 0 1 1;
     1 1 1 0 0; 1 1 1 0 1; 1 1 1 1 0; 1 1 1 1 1];

c = [ 0     0     0     0     0;     1     1     1     0     0;
      1     1     0     1     0;     0     0     1     1     0;
      1     0     1     0     1;     0     1     0     0     1;
      0     1     1     1     1;     1     0     0     1     1;
      0     0     1     1     1;     1     1     0     1     1;
      1     1     1     0     1;     0     0     0     0     1;
      1     0     0     1     0;     0     1     1     1     0;
      0     1     0     0     0;     1     0     1     0     0;
      0     1     0     1     1;     1     0     1     1     1;
      1     0     0     0     1;     0     1     1     0     1;
      1     1     1     1     0;     0     0     0     1     0;
      0     0     1     0     0;     1     1     0     0     0;
```

```
      0      1      1      0      0;      1      0      0      0      0;
      1      0      1      1      0;      0      1      0      1      0;
      1      1      0      0      1;      0      0      1      0      1;
      0      0      0      1      1;      1      1      1      1      1];

t = [c x];

[m n] = size(x);
[m k] = size(t);

eta = 0.2;

w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

oglsse = zeros(L,1);
oglelr = zeros(L,1);

for l = 1:L
    i = randperm(m);
    u1 = [x(i,:) ones(m,1)];
    y1 = sigmoid(u1 * w1);
    u2 = [y1 ones(m,1)];
    y2 = sigmoid(u2 * w2);

    e2 = flipbits(t(i,:), beta) - y2;
    d2 = y2 - y2 .^ 2;
    g2 = -u2' * (e2 .* d2) + alpha * w2;
    w2 = w2 - eta * g2;

    e1 = e2 * w2(1:q,:)';
    d1 = y1 - y1 .^ 2;
    g1 = -u1' * (e1 .* d1) + alpha * w1;
    w1 = w1 - eta * g1;

    oglsse(l) = trace((t(i,:) - y2)' * (t(i,:) - y2));
    oglelr(l) = eta;
end
```

B.2 Simplified Natural Gradient Learning

```
L = 10000;
alpha = 0.001;
beta = 0.01;

q = 10;

x = [0 0 0 0 0; 0 0 0 0 1; 0 0 0 1 0; 0 0 0 1 1;
     0 0 1 0 0; 0 0 1 0 1; 0 0 1 1 0; 0 0 1 1 1;
     0 1 0 0 0; 0 1 0 0 1; 0 1 0 1 0; 0 1 0 1 1;
     0 1 1 0 0; 0 1 1 0 1; 0 1 1 1 0; 0 1 1 1 1;
     1 0 0 0 0; 1 0 0 0 1; 1 0 0 1 0; 1 0 0 1 1;
     1 0 1 0 0; 1 0 1 0 1; 1 0 1 1 0; 1 0 1 1 1;
     1 1 0 0 0; 1 1 0 0 1; 1 1 0 1 0; 1 1 0 1 1;
     1 1 1 0 0; 1 1 1 0 1; 1 1 1 1 0; 1 1 1 1 1];

c = [ 0      0      0      0      0;      1      1      1      0      0;
      1      1      0      1      0;      0      0      1      1      0;
      1      0      1      0      1;      0      1      0      0      1;
      0      1      1      1      1;      1      0      0      1      1;
      0      0      1      1      1;      1      1      1      1      1;
      1      1      1      0      1;      0      0      0      0      1;
      1      0      0      1      0;      0      1      1      1      0;
      0      1      0      0      0;      1      0      1      0      0;
      0      1      0      1      1;      1      0      1      1      1;
      1      0      0      0      1;      0      1      1      0      1;
      1      1      1      1      0;      0      0      0      1      0;
      0      0      1      0      0;      1      1      0      0      0;
      0      1      1      0      0;      1      0      0      0      0;
      1      0      1      1      0;      0      1      0      1      0;
      1      1      0      0      1;      0      0      1      0      1;
      0      0      0      1      1;      1      1      1      1      1];

t = [c x];

[m n] = size(x);
[m k] = size(t);

eta = 0.005;

w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

snglsse = zeros(L,1);
snglelr = zeros(L,1);
```

```
for l = 1:L
    i = randperm(m);
    u1 = [x(i,:) ones(m,1)];
    y1 = sigmoid(u1 * w1);
    u2 = [y1 ones(m,1)];
    y2 = sigmoid(u2 * w2);

    e2 = flipbits(t(i,:), beta) - y2;
    d2 = y2 - y2 .^ 2;
    g2 = -u2' * (e2 .* d2) + alpha * w2;
    G2 = ((e2 .* d2)' * u2 * u2' * (e2 .* d2)) / m + alpha * eye(k);
    w2 = w2 - min(eta,1/l) * g2 * inv(G2);

    d1 = y1 - y1 .^ 2;
    b1 = (e2 * w2(1:q,1:k)') .* d1;
    g1 = -u1' * b1 + alpha * w1;
    G1 = (b1' * u1 * u1' * b1) / m + alpha * eye(q);
    w1 = w1 - min(eta,1/l) * g1 * inv(G1);

    snglsse(l) = trace((t(i,:) - y2)' * (t(i,:) - y2));
    snglelr(l) = min(eta,1/l) / max(max(eig(G2)), max(eig(G1)));
end
```

B.3 Adaptive Natural Gradient Learning

```
L = 10000;
q = 10;
beta = 0.01;

x = [0 0 0 0 0; 0 0 0 0 1; 0 0 0 1 0; 0 0 0 1 1;
     0 0 1 0 0; 0 0 1 0 1; 0 0 1 1 0; 0 0 1 1 1;
     0 1 0 0 0; 0 1 0 0 1; 0 1 0 1 0; 0 1 0 1 1;
     0 1 1 0 0; 0 1 1 0 1; 0 1 1 1 0; 0 1 1 1 1;
     1 0 0 0 0; 1 0 0 0 1; 1 0 0 1 0; 1 0 0 1 1;
     1 0 1 0 0; 1 0 1 0 1; 1 0 1 1 0; 1 0 1 1 1;
     1 1 0 0 0; 1 1 0 0 1; 1 1 0 1 0; 1 1 0 1 1;
     1 1 1 0 0; 1 1 1 0 1; 1 1 1 1 0; 1 1 1 1 1];

c = [ 0      0      0      0      0;      1      1      1      0      0;
      1      1      0      1      0;      0      0      1      1      0;
      1      0      1      0      1;      0      1      0      0      1;
      0      1      1      1      1;      1      0      0      1      1;
      0      0      1      1      1;      1      1      0      1      1;
      1      1      1      0      1;      0      0      0      0      1;
      1      0      0      1      0;      0      1      1      1      0;
      0      1      0      0      0;      1      0      1      0      0;
      0      1      0      1      1;      1      0      1      1      1;
      1      0      0      0      1;      0      1      1      0      1;
      1      1      1      1      0;      0      0      0      1      0;
      0      0      1      0      0;      1      1      0      0      0;
      0      1      1      0      0;      1      0      0      0      0;
      1      0      1      1      0;      0      1      0      1      0;
      1      1      0      0      1;      0      0      1      0      1;
      0      0      0      1      1;      1      1      1      1      1];

t = [c x];

[m n] = size(x);
[m k] = size(t);

eta = 0.01;

w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

anglsse = zeros(L,1);
anglelr = zeros(L,1);

n1 = (n + 1) * q;
```

```
n2 = (q + 1) * k;
G = eye(n1 + n2);
delta = 0.001;

for l = 1:L
    i = randperm(m);
    u1 = [x(i,:) ones(m,1)];
    y1 = sigmoid(u1 * w1);
    u2 = [y1 ones(m,1)];
    y2 = sigmoid(u2 * w2);

    e2 = flipbits(t(i,:), beta) - y2;
    d2 = y2 - y2 .^ 2;
    g2 = -u2' * (e2 .* d2);

    d1 = y1 - y1 .^ 2;
    b1 = (e2 * w2(1:q,1:k)') .* d1;
    g1 = -u1' * b1;

    r = [g1(:); g2(:)];
    p = G * r;
    p2 = reshape(p(n1 + 1 : n1 + n2), q + 1, k);
    p1 = reshape(p(1 : n1), n + 1, q);
    w2 = w2 - eta * p2;
    w1 = w1 - eta * p1;
    G = (1 + min(delta, 1/l)) * G - min(delta, 1/l) * (p*p') / (1 + r'*p);

    anglsse(l) = trace((t(i,:) - y2)' * (t(i,:) - y2));
    anglelr(l) = eta * min(diag(G));
end
```

Appendix C
MATLAB Code for the Fisher's Iris Data Experiment

The following data was used in all three algorithms.

```
iris_data = [
   50 33 14 02 1;
   64 28 56 22 3;
   65 28 46 15 2;
   67 31 56 24 3;
   63 28 51 15 3;
   46 34 14 03 1;
   69 31 51 23 3;
   62 22 45 15 2;
   59 32 48 18 2;
   46 36 10 02 1;
   61 30 46 14 2;
   60 27 51 16 2;
   65 30 52 20 3;
   56 25 39 11 2;
   65 30 55 18 3;
   58 27 51 19 3;
   68 32 59 23 3;
   51 33 17 05 1;
   57 28 45 13 2;
   62 34 54 23 3;
   77 38 67 22 3;
   63 33 47 16 2;
   67 33 57 25 3;
   76 30 66 21 3;
   49 25 45 17 3;
   55 35 13 02 1;
   67 30 52 23 3;
   70 32 47 14 2;
   64 32 45 15 2;
   61 28 40 13 2;
   48 31 16 02 1;
   59 30 51 18 3;
   55 24 38 11 2;
   63 25 50 19 3;
   64 32 53 23 3;
   52 34 14 02 1;
   49 36 14 01 1;
```

```
54 30 45 15 2;
79 38 64 20 3;
44 32 13 02 1;
67 33 57 21 3;
50 35 16 06 1;
58 26 40 12 2;
44 30 13 02 1;
77 28 67 20 3;
63 27 49 18 3;
47 32 16 02 1;
55 26 44 12 2;
50 23 33 10 2;
72 32 60 18 3;
48 30 14 03 1;
51 38 16 02 1;
61 30 49 18 3;
48 34 19 02 1;
50 30 16 02 1;
50 32 12 02 1;
61 26 56 14 3;
64 28 56 21 3;
43 30 11 01 1;
58 40 12 02 1;
51 38 19 04 1;
67 31 44 14 2;
62 28 48 18 3;
49 30 14 02 1;
51 35 14 02 1;
56 30 45 15 2;
58 27 41 10 2;
50 34 16 04 1;
46 32 14 02 1;
60 29 45 15 2;
57 26 35 10 2;
57 44 15 04 1;
50 36 14 02 1;
77 30 61 23 3;
63 34 56 24 3;
58 27 51 19 3;
57 29 42 13 2;
72 30 58 16 3;
54 34 15 04 1;
52 41 15 01 1;
71 30 59 21 3;
64 31 55 18 3;
60 30 48 18 3;
```

```
63 29 56 18 3;
49 24 33 10 2;
56 27 42 13 2;
57 30 42 12 2;
55 42 14 02 1;
49 31 15 02 1;
77 26 69 23 3;
60 22 50 15 3;
54 39 17 04 1;
66 29 46 13 2;
52 27 39 14 2;
60 34 45 16 2;
50 34 15 02 1;
44 29 14 02 1;
50 20 35 10 2;
55 24 37 10 2;
58 27 39 12 2;
47 32 13 02 1;
46 31 15 02 1;
69 32 57 23 3;
62 29 43 13 2;
74 28 61 19 3;
59 30 42 15 2;
51 34 15 02 1;
50 35 13 03 1;
56 28 49 20 3;
60 22 40 10 2;
73 29 63 18 3;
67 25 58 18 3;
49 31 15 01 1;
67 31 47 15 2;
63 23 44 13 2;
54 37 15 02 1;
56 30 41 13 2;
63 25 49 15 2;
61 28 47 12 2;
64 29 43 13 2;
51 25 30 11 2;
57 28 41 13 2;
65 30 58 22 3;
69 31 54 21 3;
54 39 13 04 1;
51 35 14 03 1;
72 36 61 25 3;
65 32 51 20 3;
61 29 47 14 2;
```

```
   56 29 36 13 2;
   69 31 49 15 2;
   64 27 53 19 3;
   68 30 55 21 3;
   55 25 40 13 2;
   48 34 16 02 1;
   48 30 14 01 1;
   45 23 13 03 1;
   57 25 50 20 3;
   57 38 17 03 1;
   51 38 15 03 1;
   55 23 40 13 2;
   66 30 44 14 2;
   68 28 48 14 2;
   54 34 17 02 1;
   51 37 15 04 1;
   52 35 15 02 1;
   58 28 51 24 3;
   67 30 50 17 2;
   63 33 60 25 3;
   53 37 15 02 1
];

sorted_data = sortrows(iris_data, 5);

train_index = [ 1:30 51:80  101:130];
test_index  = [31:50 81:100 131:150];

train_data  = sorted_data(train_index, 1:4);
test_data   = sorted_data(test_index,  1:4);

train_class = sorted_data(train_index, 5);
test_class  = sorted_data(test_index,  5);

I3 = eye(3);

train_target = I3(train_class, 1:3);
test_target  = I3(test_class,  1:3);

mu = mean(train_data(:,1:4));
C = cov(train_data(:,1:4));
R = chol(C);
S = inv(R);
x = (train_data - repmat(mu, size(train_data, 1), 1)) * S;
t = train_target;
z = (test_data - repmat(mu, size(test_data, 1), 1)) * S;
```

C.1 Ordinary Gradient Learning

```
eta = 0.01;
alpha = 0.1;
L = 10000;
q = 4;

[m n] = size(x);
[m k] = size(t);

w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

oglsse = zeros(L,1);
oglent = zeros(L,1);
oglcnt = zeros(L,1);
oglelr = zeros(L,1);

for l = 1:L
    i = randperm(m);
    u1 = [ x(i,:) ones(m,1) ];
    y1 = tanh(u1 * w1);
    u2 = [ y1 ones(m,1) ];
    z2 = exp(u2 * w2);
    y2 = diag(1 ./ sum(z2,2)) * z2;

    e2 = t(i,:) - y2;
    g2 = -u2' * e2 + alpha * w2;
    w2 = w2 - eta * g2;

    b1 = (e2 * w2(1:q,:)') .* (1 - y1 .^ 2);
    g1 = -u1' * b1 + alpha * w1;
    w1 = w1 - eta * g1;

    oglsse(l) = trace(e2' * e2);

    u1 = [ z ones(size(test_data, 1), 1) ];
    y1 = tanh(u1 * w1);
    u2 = [ y1 ones(size(test_data, 1), 1) ];
    v2 = exp(u2 * w2);
    y2 = diag(1 ./ sum(v2, 2)) * v2;

    p = prod(y2 .^ test_target, 2);
    oglent(l) = -sum(p .* log(p)) / m;
    oglcnt(l) = length(find(p < 0.5));
    oglelr(l) = eta;
```

```
end
```

C.2 Simplified Natural Gradient Learning

```
eta = 0.01;
tau = 0.5;
alpha = 0.1;
L = 10000;
q = 4;

[m n] = size(x);
[m k] = size(t);

w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

snglsse = zeros(L,1);
snglelr = zeros(L,1);
snglent = zeros(L,1);
snglcnt = zeros(L,1);

for l = 1:L
    i = randperm(m);
    u1 = [ x(i,:) ones(m,1) ];
    y1 = tanh(u1 * w1);
    u2 = [ y1 ones(m,1) ];
    z2 = exp(u2 * w2);
    y2 = diag(1 ./ sum(z2,2)) * z2;
    e2 = t(i,:) - y2;

    g2 = -u2' * e2 + alpha * w2;
    G2 = (u2' * e2)' * u2' * e2 / (m - 1) + alpha * eye(k);
    p2 = g2 * inv(G2);
    w2 = w2 - min(eta, tau/l) * p2;

    b1 = (e2 * w2(1:q,:)') .* (1 - y1 .^ 2);
    g1 = -u1' * b1 + alpha * w1;
    G1 = (u1' * b1)' * u1' * b1 / (m - 1) + alpha * eye(q);
    p1 = g1 * inv(G1);
    w1 = w1 - min(eta, tau/l) * p1;

    snglelr(l) = min(eta, tau/l) / max(max(eig(G1)), max(eig(G2)));
    snglsse(l) = trace(e2' * e2);

    u1 = [ z ones(size(test_data, 1), 1) ];
    y1 = tanh(u1 * w1);
    u2 = [ y1 ones(size(test_data, 1), 1) ];
    v2 = exp(u2 * w2);
```

```
    y2 = diag(1 ./ sum(v2, 2)) * v2;

    p = prod(y2 .^ test_target, 2);
    snglent(l) = -sum(p .* log(p)) / m;
    snglcnt(l) = length(find(p < 0.5));
end
```

C.3 Adaptive Natural Gradient Learning

```
eta = 0.01;
L = 10000;
q = 4;

[m n] = size(x);
[m k] = size(t);

n1 = (n+1) * q;
n2 = (q+1) * k;

delta = 0.001;

w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

G = eye(n1 + n2);

anglsse = zeros(L,1);
anglelr = zeros(L,1);
anglent = zeros(L,1);
anglcnt = zeros(L,1);

for l = 1:L
    i = randperm(m);
    u1 = [x(i,:) ones(m,1)];
    y1 = tanh(u1 * w1);
    u2 = [y1 ones(m,1)];
    z2 = exp(u2 * w2);
    y2 = diag(1 ./ sum(z2,2)) * z2;
    e2 = t(i, 1:k) - y2;
    g2 = -u2' * e2;
    b1 = (e2 * w2(1:q, 1:k)') .* (1 - y1 .^ 2);
    g1 = -u1' * b1;
    r = [g1(:); g2(:)];
    p = G * r;
    G = (1 + min(delta,1/l)) * G - min(delta,1/l) * (p*p') / (1 + r'*p);
    p1 = p(1:n1);
    p2 = p(n1+1:n1+n2);
    w1 = w1 - eta * reshape(p1, n+1, q);
    w2 = w2 - eta * reshape(p2, q+1, k);
    anglsse(l) = trace(e2' * e2);
    anglelr(l) = eta * min(eig(G));

    u1 = [ z ones(size(test_data, 1), 1) ];
```

```
    y1 = tanh(u1 * w1);
    u2 = [ y1 ones(size(test_data, 1), 1) ];
    v2 = exp(u2 * w2);
    y2 = diag(1 ./ sum(v2, 2)) * v2;

    pr = prod(y2 .^ test_target, 2);
    anglent(l) = -sum(pr .* log(pr)) / m;
    anglcnt(l) = length(find(pr < 0.5));
end
```

Appendix D
MATLAB Code for the Mackey-Glass Chaotic Time Series Experiment

The following function was used to generate the data.

```
N = 10000;
mgs = 0.9 * ones(N+1,1);
a = 0.2;
b = 0.1;
tau = 17;
k = 1;

x = zeros(501,4);
t = zeros(501,1);

xt = zeros(501,4);
tt = zeros(501,1);

for n = tau+1:N
    mgs(n+1) = (1-b) * mgs(n) + a * mgs(n - tau)/(1 + mgs(n - tau)^10);
end

for n = 200:700
    x(k,:) = [mgs(n) mgs(n-6) mgs(n-12) mgs(n-18)];
    t(k) = mgs(n+6);
    k = k + 1;
end

k = 1;

for n = 5000:5500
    xt(k,:) = [mgs(n) mgs(n-6) mgs(n-12) mgs(n-18)];
    tt(k) = mgs(n+6);
    k = k + 1;
end
```

D.1 Ordinary Gradient Learning

```
L = 100000;
tau = 1;
eta = 0.001;
alpha = 0.01;

q = 10;

[m n] = size(x);
[m k] = size(t);

w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

oglsse = zeros(L,1);
oglelr = zeros(L,1);

for l = 1:L
    i = randperm(m);
    u1 = [x(i,:) ones(m,1)];
    y1 = tanh(u1 * w1);
    u2 = [y1 ones(m,1)];
    y2 = u2 * w2;
    e2 = t(i) - y2;
    g2 = -u2' * e2 + alpha * w2;
    b1 = e2 * w2(1:q,:)';
    d1 = ones(size(y1)) - y1 .^ 2;
    e1 = b1 .* d1;
    g1 = -u1' * e1 + alpha * w1;
    w1 = w1 - min(eta, tau / l) * g1;
    w2 = w2 - min(eta, tau / l) * g2;
    oglsse(l) = e2' * e2;
    oglelr(l) = min(eta, tau / l);
end
```

D.2 Simplified Natural Gradient Learning

```
L = 100000;
tau = 0.1;
eta = 1e-3;
alpha = 0.001;

q = 10;

[m n] = size(x);
[m k] = size(t);

w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

snglsse = zeros(L,1);
snglelr = zeros(L,1);

for l = 1:L
    i = randperm(m);
    u1 = [x(i,:) ones(m,1)];
    y1 = tanh(u1 * w1);
    u2 = [y1 ones(m,1)];
    y2 = u2 * w2;
    e2 = t(i) - y2;
    g2 = -u2' * e2 + alpha * w2;
    G2 = (u2' * e2)' * u2' * e2 / (m - 1) + alpha * eye(k);
    b1 = e2 * w2(1:q,:)';
    d1 = ones(size(y1)) - y1 .^ 2;
    e1 = b1 .* d1;
    g1 = -u1' * e1 + alpha * w1;
    G1 = (u1' * e1)' * u1' * e1 / (m - 1) + alpha * eye(q);
    w1 = w1 - min(eta, tau / l) * g1 * inv(G1);
    w2 = w2 - min(eta, tau / l) * g2 * inv(G2);
    snglsse(l) = e2' * e2;
    snglelr(l) = min(eta, tau / l) / max(G2, max(eig(G1)));
end
```

D.3 Adaptive Natural Gradient Learning

```
L = 100000;
eta = 1e-3;
delta = 1e-5;

[m n] = size(x);
[m k] = size(t);
n1 = (n + 1) * q;
n2 = (q + 1) * k;

q = 10;

G = eye(n1 + n2);

w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

anglsse = zeros(L,1);
anglelr = zeros(L,1);

for l = 1:L
    i = randperm(m);
    u1 = [x(i,:) ones(m,1)];
    y1 = tanh(u1 * w1);
    u2 = [y1 ones(m,1)];
    y2 = u2 * w2;
    e2 = t(i) - y2;
    g2 = -u2' * e2;
    g1 = -u1' * ((e2 * w2(1:q,1:k)') .* (1 - y1 .^ 2));
    r = [g1(:); g2(:)];
    p = G * r;
    G = (1 + min(delta,1/l)) * G - min(delta,1/l) * (p*p') / (1 + r'*p);
    p1 = p(1:n1);
    p2 = p(n1 + 1 : n1 + n2);
    w1 = w1 - eta * reshape(p1, n + 1, q);
    w2 = w2 - eta * reshape(p2, q + 1, k);
    anglsse(l) = e2' * e2;
    anglelr(l) = eta * min(diag(G));
end
```

Appendix E
MATLAB Code for the Semicircle and S-Curve Unfolding Experiments

```
t = 0:0.01:1;
t = t(:);
m = length(t);
angle = 5*pi*t/3 + pi/6;
x = [cos(angle) sin(angle)] + 0.1 * randn(m,2);
```

E.1 Ordinary Gradient Learning

```
L = 100000;
eta = 1e-5;
alpha = 1e-2;

q = 5;

[m n] = size(x);
[m k] = size(t);

w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

oglsse = zeros(L,1);
oglelr = zeros(L,1);

for l = 1:L
    i = randperm(m);
    u1 = [x(i,:) ones(m,1)];
    y1 = tanh(u1 * w1);
    u2 = [y1 ones(m,1)];
    y2 = u2 * w2;
    e2 = t(i,:) - y2;
    g2 = -u2' * e2 + alpha * w2;
    b1 = e2 * w2(1:q,:)';
    d1 = ones(size(y1)) - y1 .^ 2;
    e1 = b1 .* d1;
    g1 = -u1' * e1 + alpha * w1;
    w1 = w1 - eta * g1;
    w2 = w2 - eta * g2;
    oglsse(l) = trace(e2' * e2);
    oglelr(l) = eta;
end
```

E.2 Simplified Natural Gradient Learning

```
L = 100000;
eta = 1e-3;
alpha = 1e-2;

q = 5;

[m n] = size(x);
[m k] = size(t);

w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

snglsse = zeros(L,1);
snglelr = zeros(L,1);

for l = 1:L
    i = randperm(m);
    u1 = [x(i,:) ones(m,1)];
    y1 = tanh(u1 * w1);
    u2 = [y1 ones(m,1)];
    y2 = u2 * w2;
    e2 = t(i,:) - y2;
    g2 = -u2' * e2 + alpha * w2;
    b1 = e2 * w2(1:q,:)';
    d1 = ones(size(y1)) - y1 .^ 2;
    e1 = b1 .* d1;
    g1 = -u1' * e1 + alpha * w1;
    G1 = (g1' * g1) / (m - 1) + alpha * eye(q);
    G2 = (g2' * g2) / (m - 1) + alpha * eye(k);
    w1 = w1 - min(eta, 1/l) * g1 * inv(G1);
    w2 = w2 - min(eta, 1/l) * g2 * inv(G2);
    snglsse(l) = trace(e2' * e2);
    snglelr(l) = min(eta, 1/l) / max([eig(G2); eig(G1)]);
end
```

E.3 Adaptive Natural Gradient Learning

```
L = 100000;
eta = 1e-5;
delta = 1e-4;

q = 5;

[m n] = size(x);
[m k] = size(t);
n1 = (n + 1) * q;
n2 = (q + 1) * k;

G = eye(n1 + n2);
w1 = 0.2 * rand(n + 1, q) - 0.1;
w2 = 0.2 * rand(q + 1, k) - 0.1;

anglsse = zeros(L,1);
anglelr = zeros(L,1);

for l = 1:L
    i = randperm(m);
    u1 = [x(i,:) ones(m,1)];
    y1 = tanh(u1 * w1);
    u2 = [y1 ones(m,1)];
    y2 = u2 * w2;
    e2 = t(i,:) - y2;
    g2 = -u2' * e2;
    b1 = e2 * w2(1:q,:)';
    d1 = ones(size(y1)) - y1 .^ 2;
    e1 = b1 .* d1;
    g1 = -u1' * e1;
    r = [g1(:); g2(:)];
    p = G * r;
    G = (1 + min(delta,1/l)) * G - min(delta,1/l) * (p*p') / (1 + r'*p);
    p1 = p(1:n1);
    p2 = p(n1 + 1 : n1 + n2);
    w1 = w1 - eta * reshape(p1, n + 1, q);
    w2 = w2 - eta * reshape(p2, q + 1, k);
    anglsse(l) = trace(e2' * e2);
    anglelr(l) = eta * min(diag(G));
end
```

Vita

Michael R. Bastian

Published Conference Papers

- Sobolev Gradients and Neural Networks, Michael R. Bastian, Jacob H. Gunther and Todd K. Moon, *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, 2008.
- An Improvement to the Natural Gradient Learning Algorithm for Multilayer Perceptrons, Michael R. Bastian, Jacob H. Gunther and Todd K. Moon, *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, 2005.
- Color quantization and dithering gamuts, B. Morse, T. Howard, S. Larson, M. Bastian and E. Mortensen, *Int. Conf. on Signal and Image Processing (SIP)*, 1999.
- A Bayesian exclusionary rule for Hough transforms, B. Morse, D. Ashton and M. Bastian, *Int. Conf. on Computer Vision, Pattern Recognition, and Image Processing (CVPRIP) in Joint Conference on Information Sciences (JCIS)*, 1999.