GENERALIZATION AND NEURAL NETWORKS

By

FOREST DAN FORESEE

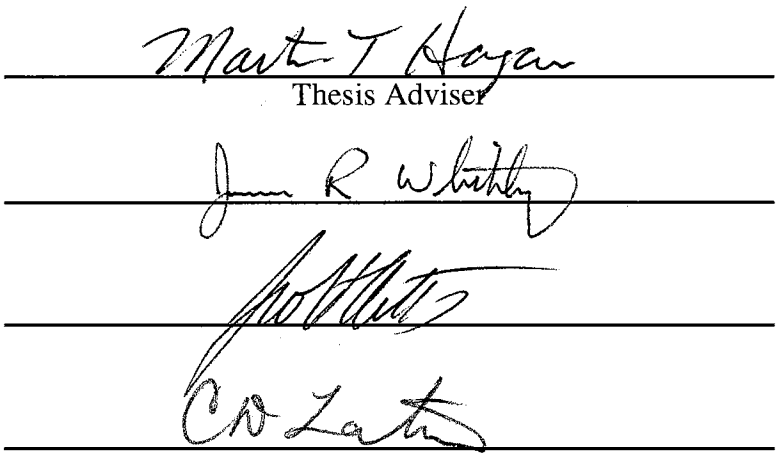
Bachelor of Science
Oklahoma State University
Stillwater, Oklahoma
1979

Master of Engineering
Oklahoma State University
Stillwater, Oklahoma
1981

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 1996

GENERALIZATION AND NEURAL NETWORKS

Thesis Approved:



Thesis Adviser

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to thank my major adviser, Dr. Martin Hagan, for his guidance, encouragement, patience and friendship. I would also like to thank my other committee members Dr. Carl Latino, Dr. Scott Acton and Dr. James Whiteley for their helpful suggestions and assistance.

My thanks also goes to the management of Lucent Technologies for their support in this endeavor. Specifically, thanks goes to Mike Stallings, Steven Gregus, Sid Hardy, Mike Carolina and in particular to Al Arms.

I would like to express my deep appreciation to my wife, Tina, for her years of loving support and encouragement. Thanks also to my children Dustin, Austin and Tara for their love and patience. Finally, I wish to thank my parents for all they have done to support all my endeavors, especially the encouragement for this one.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

With the recent dramatic increase in neural network usage, pressure is increasing to improve their performance. Neural networks are trained by using sample sets. However, the samples usually contain noise. When a network is trained, it learns the noise in the training set along with the underlying function. The resulting network will not respond properly to new input because of the noise. The neural network is said to not generalize very well. Our goal is to improve neural network generalization performance.

A common problem with surveying current literature in this area is the diversity of the field of researchers. The theories of neural networks are developed by people with a broad range of backgrounds. Different fields typically have their own language and nomenclature. This makes reading and applying their work difficult.

We will survey and categorize common techniques aimed at improving generalization. A few of the most promising techniques will be chosen for examination. We will put each into a common mathematical language for easy comparison. Then we will explore their strengths and weaknesses by implementing each of them in the Levenberg-Marquardt training algorithm and running experiments.

We will then draw on this knowledge and propose an improved algorithm based on the best existing techniques. We will test our new algorithm on real-world problems and discuss the results.

We will show that our GNBR (Gauss-Newton approximation to Bayesian Regularization) algorithm consistently produces optimal generalization performance results without dramatically increasing computational overhead. Indeed, any reasonably sized model trained with the GNBR algorithm, following our recipe for application, will often produce optimal results the first time. Our recipe will include some simple checks to give confidence in the training results.

Let us now outline the flow of this document. Chapter 2 will serve as a refresher of pertinent neural network background material. Its main thrust will be to put commonly used equations into the form we will be using and to introduce our nomenclature. This chapter will end with a list of parameters we will use throughout this document for easy reference.

In Chapter 3, we will discuss generalization and how it affects neural network performance. We will also give a short introduction to several existing generalization techniques. Finally, we will list the most promising methods, which we will present in more detail in future chapters.

The first method we will present is a model comparison criterion. It is an extension to the AIC commonly used in statistics for comparing regression models. This will be examined in Chapter 4.

Regularization is used quite often in statistics to restrict parameter values. In Chapter 5, we will examine its effect on neural network training. In Chapter 6, we will look at a method of automatically optimizing the amount of regularization to use on a neural network architecture, given the particular set of data used to train it. This will be accomplished through Bayesian analysis of network training.

A very popular training technique used today is "stopped training". In Chapter 7, we will explore this technique and compare it to regularization.

Then, in Chapter 8 we will outline a recipe for application of our new GNBR algorithm, which is first developed in Chapter 6. We will also apply it to four real-world problems to demonstrate the consistently optimal generalization performance it gives.

Chapter 9 will contain a summary of the main results and contributions of this work. This will be followed by recommendations for future work.

CHAPTER 2

BACKGROUND

Introduction

The intent of this chapter is to introduce relevant neural network background in a mathematical framework that will be used in the chapters to follow. It will remind the reader of applicable fundamentals while introducing the notation we are using. We will begin with a short analysis of a single neuron. This will be followed by a description of a 2-layer network architecture and a short discussion of its abilities.

Training is the key to improving generalization. We will briefly show a conceptual relationship between training and generalization, and will then present some details of the simplest form of training. Note that this material is slanted toward the implementation of the algorithm we will use.

"Toy" problems which will be used to illustrate the various concepts discussed in this dissertation are identified next. Lastly, a list of the most often used parameters is supplied for future reference.

Single Neuron

A neuron is the smallest processing element of an "artificial" neural network. A block diagram of a neuron is shown in Figure 1. It has an input p and an output a. The equation of operation is

$$a = f(x) = f(wp + b) \tag{1}$$

The input is scaled by the weight w and added to the bias b. The weight and bias of a neuron are selectable parameters and can take on any real value. The transfer function of the neuron is represented as f. It can be a simple linear function with unity gain, or it can be a more complex nonlinear function such as the hyperbolic tangent sigmoid (tansig). Whatever it is, it provides a key translation between the input p and the output a.

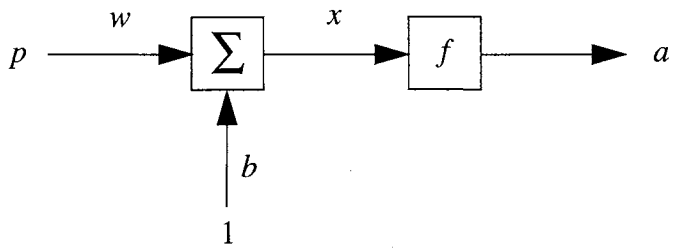


Figure 1 A Single Neuron

Let the transfer function f be an increasing function which is symmetric about zero. Then the bifurcation point of Eq. (1) is at $p = -\frac{b}{w}$. For both a linear function and a tansig function, if $p < -\frac{b}{w}$ then $a < 0$ whereas $p > -\frac{b}{w}$ maps to $a > 0$.

As an example calculation, consider $p = -\frac{b}{w} + \varepsilon$ for any $\varepsilon > 0$. From Eq. (1),

$$\begin{aligned} a &= f(wp + b) \\ &= f\left(w\left(-\frac{b}{w} + \varepsilon\right) + b\right) \\ &= f(-b + \varepsilon w + b) \\ &= f(\varepsilon w) \end{aligned} \tag{2}$$

If f is a linear transfer function such that $f(x) = x$, then the amount of the input p greater

than $-\frac{b}{w}$ is simply scaled by w. Note that, since w can be any real value, it can be selected to cause any specific output a for a particular input p. Now, if f is the tansig function, then

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3}$$

so f is bounded by $(-1, 1)$; i.e., $-1 < f(x) < 1 \ \forall x$. Note that the linear transfer function is good for providing real-valued output while the tansig function provides more of a binary type output, conducive to decisions and selections. An example of $a = f(0.25p + 0.75)$ for both the linear function and the tansig function is shown in Figure 2.

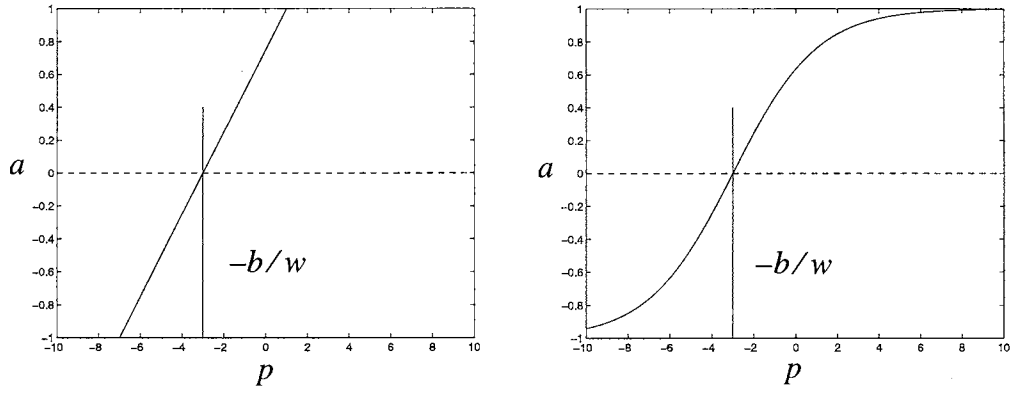


Figure 2 Linear and Tansig Functions

As one last note on a single neuron, we show it in Figure 1 as having only one input. It can also have many inputs. If $\mathbf{p}^T = \left[p_1\ p_2\ \ldots\ p_k\right]$, then $a = f(\mathbf{w}\mathbf{p} + b)$ where $\mathbf{w} = \left[w_1\ w_2\ \ldots\ w_k\right]$. The neuron now provides a weighted sum of the inputs combined with the bias before being translated by the function f. We will see how the single-input neurons can be combined with a multiple input neuron in the next section.

Neural Network Architecture

Consider the 2 layer neural network of Figure 3. It has two layers because the single input p is processed by S single neurons simultaneously before having their outputs $a_1^1, a_2^1, \ldots, a_S^1$ treated as multiple inputs to a single neuron whose result is the final output of the network a_1^2. The neurons receiving common input are considered to be in the same "layer". Thus we have two layers. There is one neuron in the output layer, so dubbed since its output is the output of the whole network.

Between the input p and the output layer is the "hidden" layer. The name stems from the fact that it does not have direct exposure to the output of the network. On one end of the network is the output layer; on the other end is the network input. The hidden layer of Figure 3 has S neurons. All S neurons may receive the same input p, but since the weight w_i^1 and the bias b_i^1 are typically different for each neuron, they will each respond with a different output a_i^1.

Notice that we show a network containing S hidden layer neurons with a single input as shown in Figure 1, and one output layer neuron with S inputs, from the hidden layer, as discussed at the end of the previous section. Imagine that the hidden layer neurons each have a bounded output as discussed previously, while the output layer neuron has a linear transfer function. We can now see that the size of the input p might determine which hidden layer neurons are presenting a value close to 1 as opposed to a -1 to the inputs of the output layer neuron. Recall that the linear output neuron will scale the inputs according to

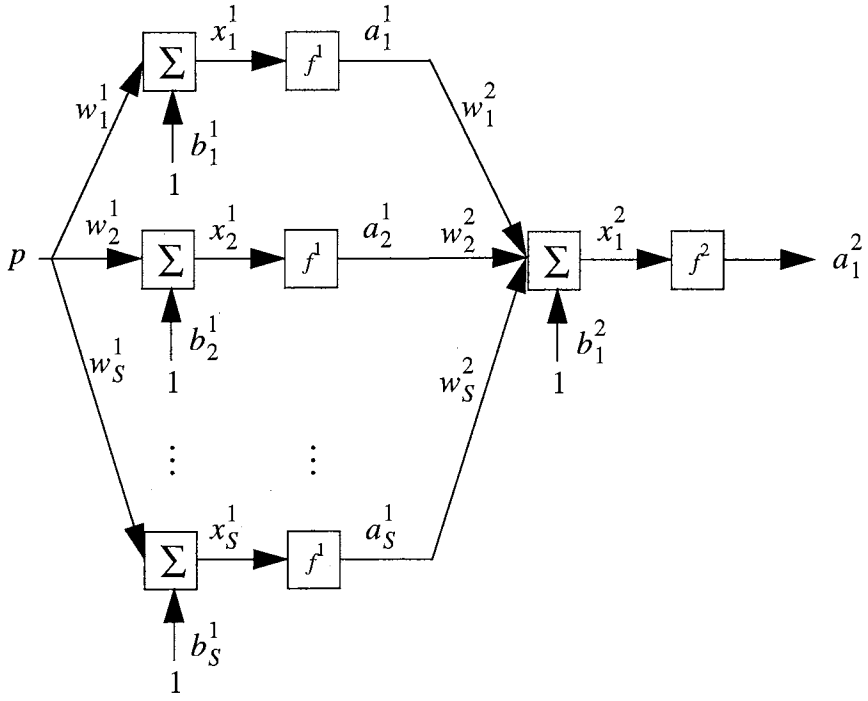


Figure 3 A 2-Layer Network with S Hidden Layer Neurons

the weights. So this neural network could provide a nonlinear scaled output based on the various combinations of outputs of the hidden layer neurons, which is in turn based on the value of the input p.

To see this better, we examine the equation of operation of the neural network of Figure 3 and follow with a simple example.

$$a_1^2 = f^2(\mathbf{w}^2 f^1(\mathbf{w}^1 p + \mathbf{b}^1) + \mathbf{b}^2) \tag{4}$$

where

$$\mathbf{w}^1 = \begin{bmatrix} w_1^1 \\ w_2^1 \\ \vdots \\ w_S^1 \end{bmatrix} \qquad \mathbf{b}^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \\ \vdots \\ b_S^1 \end{bmatrix} \qquad (\mathbf{w}^2)^T = \begin{bmatrix} w_1^2 \\ w_2^2 \\ \vdots \\ w_S^2 \end{bmatrix} \qquad \mathbf{b}^2 = \begin{bmatrix} b_1^2 \end{bmatrix} \tag{5}$$

The neural network we will use will be a single-input/single-output two layer network of the architecture shown in Figure 3. Further, we will assign the unity gain linear transfer function to the output layer neuron and the tansig function to each of the hidden layer neurons.

For an example, suppose there are only 2 hidden layer neurons. Thus our network will have only 3 neurons requiring 10 weights. (Note: quite often the weights and biases of a neural network are collectively referred to simply as the weights. This is in part because they have similar value to the network since they are all tunable parameters.) Our network will operate according to

$$\begin{aligned} a_1^2 &= f^2(\mathbf{w}^2 f^1(\mathbf{w}^1 p + \mathbf{b}^1) + \mathbf{b}^2) \\ &= f^2\left(\begin{bmatrix} w_1^2 & w_2^2 \end{bmatrix} f^1\left(\begin{bmatrix} w_1^1 \\ w_2^1 \end{bmatrix} p + \begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix}\right) + b_1^2\right) \\ &= \begin{bmatrix} w_1^2 & w_2^2 \end{bmatrix} f^1\left(\begin{bmatrix} w_1^1 \\ w_2^1 \end{bmatrix} p + \begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix}\right) + b_1^2 \end{aligned} \tag{6}$$

If we assign the weights of this network as

$$\mathbf{w}^1 = \begin{bmatrix} 4 \\ -1 \end{bmatrix} \qquad \mathbf{b}^1 = \begin{bmatrix} -2 \\ 0.5 \end{bmatrix} \qquad (\mathbf{w}^2)^T = \begin{bmatrix} -5 \\ -10 \end{bmatrix} \qquad \mathbf{b}^2 = \begin{bmatrix} 0.5 \end{bmatrix} \tag{7}$$

then the mapping function performed by this network would be as shown in Figure 4. More hidden layer neurons can produce more complex mappings. We will see many examples of this later.

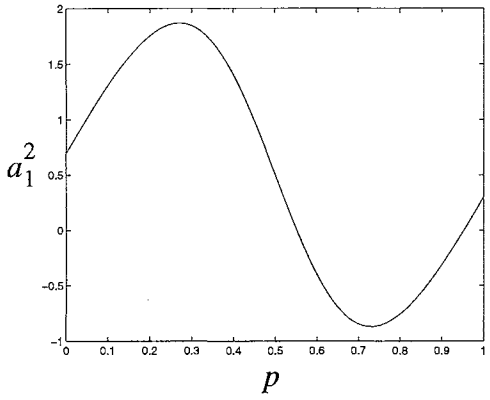


Figure 4 Typical 1-2-1 Network Mapping

Abilities

As indicated previously, the input p of a neural network can be mapped into more complex looking functions as more hidden layer neurons are added. In fact, Hornick and Stinchcombe show that any function (Borel integrable) can be approximated arbitrarily well by a 2-layer network given enough neurons in the hidden layer. [Whit92]

Thus far we have talked about neural networks that implement a certain mapping. Adjusting the weights to produce a desired mapping function is referred to as nonlinear regression, or function approximation.

Neural networks are certainly not limited to function approximation or regression. They can also perform discriminant analysis, or pattern recognition. For instance, if the transfer function in the output layer is bounded in the interval $(-1, 1)$ we can train a network with a single output neuron to produce a 1 when the input is in one class and a -1 when the input is in a second class. With multiple neurons in the output layer more complex classifications can be made.

In order to reduce the scope of work to a manageable size, this research will address only the function approximation (regression) aspects of neural network training. Many of the ideas discussed here, however, can be directly applied to pattern recognition (discriminant analysis) as well.

Training Concepts

Having defined the boundaries of our work, we will now discuss how to adjust the tunable parameters of a neural network, the weights. We will use supervised training to "teach" our network the function we wish it to learn. Supervised learning requires that we know the desired output (target) t for each input p, so errors can be calculated.

Of course, the training set (inputs and desired outputs) is finite in size. In many real-world cases the training set size might be quite small. The idea is that the training set represents a random sampling of the function we wish our network to learn to approximate. However, since we took measurements on the function, they surely contain errors. There are many sources of error; e.g., the measuring device, rounding, environmental inflections, or simple human error. If our training set were infinite in size, the average error would tend to zero. However, in practicality we are limited to a finite set of sample measurements.

Since these measurements contain noise, we must find a way to reduce the effects of the noise on training. Our desire is to train the network to respond as the true function does. The underlying function is sometimes severely obscured by the noise. We do not want our neural network to respond to novel inputs with an output typical of the noise it learned from the training set. How well a trained network mapping resembles the true function is a measure of generalization capability.

If a trained neural network is presented with an input value which it was not specifically trained for, we would like for the network to properly map it anyway. This is referred to as good generalization. Indeed, since our training set is finite in size, most of the future inputs will not have been in the training set. If the network learns the noise in the training set, then it will respond incorrectly to the novel input. This is referred to as poor generalization. Examples of good and poor generalization are shown in Figure 5. The training set samples are denoted by a "+" and the true function and the network response by lines. Notice the network mapping function on the right has learned the noise in the training set at the highest peak. In the figure on the left, the network response closely tracks the underlying saw function and has good generalization. Improving the generalization of a neural network is the aim of this work.

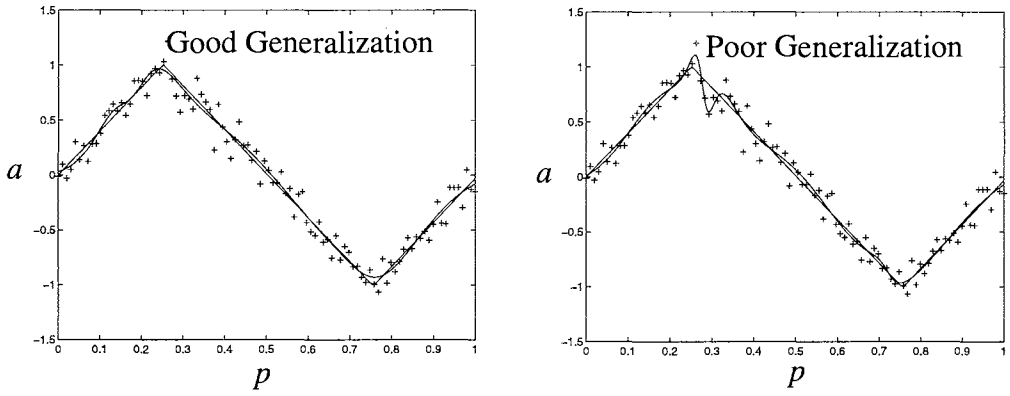


Figure 5 Example of Good and Poor Generalization

Training Methods

Tuning the parameters of the neural network according to the training set is central to our theme. To follow the ideas of training algorithms, we must now define several mathematical expressions. These ideas are common in the neural network literature, although

the notation is different for most every paper. The parameters used here will be reused throughout this paper.

With supervised learning we have a set of inputs and target outputs:

$$\{p_1, t_1\}, \{p_2, t_2\}, \ldots, \{p_n, t_n\},$$

where these pairs are generated from the following underlying process

$$t_i = g(p_i) + \varepsilon_i. \tag{8}$$

We define the training set error as

$$E_D = \sum_{i=1}^{n} (t_i - a_i)^2 = \sum_{i=1}^{n} e_i^2 = \mathbf{e}^T\mathbf{e} \tag{9}$$

where a_i is the output of the neural network when p_i is the input. (Note that for simplicity of presentation we are assuming a single-input/single-output system. The results can be easily generalized to the multiple-input/multiple-output case.) Also note that in the final relation of Eq. (9) we show the errors for the entire training set as the vector $\mathbf{e}$. Thus, $\mathbf{e}$ becomes an n length vector composed of the individual errors for the n size training set as

$$\mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix} \tag{10}$$

We can also define an "actual" error as

$$E_A = \sum_{i=1}^{n} (g(p_i) - a_i)^2. \tag{11}$$

We often adjust the weights and biases of the network to minimize the training set error E_D. However, we really want the network to minimize the actual error E_A. The problem is that E_A is not measurable.

There are also other errors we will wish to monitor later. For a set of independently chosen data, E_V performs the same function as E_D, except that E_V is not used by the training algorithm. This is the error associated with a validation set of data. We might observe this variable during training as a cross-check of proper learning.

Similarly, for yet another independently chosen set of data, we define E_T to be the testing error which could be evaluated after training as an indication of proper performance.

On a different note, we define

$$E_W = (\mathbf{w} - \mathbf{w}_0)^T(\mathbf{w} - \mathbf{w}_0) = \sum_{i=1}^{N} (w_i - w_{0i})^2 \tag{12}$$

This is the sum of the squares of the weights, ordered in $\mathbf{w}$. We will assume the nominal weight vector $\mathbf{w}_0 = \mathbf{0}$ unless stated otherwise. E_W has a form similar to E_D but is dependent solely on the values of the weights.

When training a neural network, we often look to minimize the errors of the training set E_D. But the actual function we minimize can take on many forms. For example, we may want to initialize some combination of E_D and E_W. We will refer to our general objective function as F, and we will be careful to indicate which form we are using for each

instance. For minimizing F, we will require the gradient and possibly the Hessian of the objective function F.

If our objective function is $F(\mathbf{w}) = E_D$, the gradient of F is

$$\begin{aligned} \nabla F(\mathbf{w}) &= \frac{\partial}{\partial \mathbf{w}} F(\mathbf{w}) \\ &= \frac{\partial}{\partial \mathbf{w}} E_D(\mathbf{w}) \\ &= \frac{\partial}{\partial \mathbf{w}} ((\mathbf{e}(\mathbf{w}))^T \mathbf{e}(\mathbf{w})) \\ &= 2\mathbf{J}^T(\mathbf{w})\mathbf{e}(\mathbf{w}) \end{aligned} \tag{13}$$

where

$$\mathbf{J}(\mathbf{w}) = \begin{bmatrix} \frac{\partial}{\partial w_1} e_1(\mathbf{w}) & \frac{\partial}{\partial w_2} e_1(\mathbf{w}) & \cdots & \frac{\partial}{\partial w_N} e_1(\mathbf{w}) \\ \frac{\partial}{\partial w_1} e_2(\mathbf{w}) & \ddots & & \vdots \\ \vdots & & & \\ \frac{\partial}{\partial w_1} e_n(\mathbf{w}) & \cdots & & \frac{\partial}{\partial w_N} e_n(\mathbf{w}) \end{bmatrix} \tag{14}$$

is the Jacobian matrix.

The matrix of second derivatives with respect to the weights is called the Hessian of F. It too, can be expressed using the Jacobian matrix.

$$
\begin{aligned}
\nabla^2 F(\mathbf{w}) &= \frac{\partial^2}{\partial \mathbf{w} \partial \mathbf{w}} F(\mathbf{w}) \\
&= \begin{bmatrix}
\frac{\partial^2}{\partial w_1 \partial w_1} F(\mathbf{w}) & \frac{\partial^2}{\partial w_1 \partial w_2} F(\mathbf{w}) & \cdots & \frac{\partial^2}{\partial w_1 \partial w_N} F(\mathbf{w}) \\
\frac{\partial^2}{\partial w_2 \partial w_1} F(\mathbf{w}) & \ddots & & \vdots \\
\vdots & & & \\
\frac{\partial^2}{\partial w_N \partial w_1} F(\mathbf{w}) & \cdots & & \frac{\partial^2}{\partial w_N \partial w_N} F(\mathbf{w})
\end{bmatrix} \\
&= 2\left[\mathbf{J}^T(\mathbf{w}) \frac{\partial}{\partial \mathbf{w}} \mathbf{e}(\mathbf{w}) + \sum_{i=1}^{n} e_i(\mathbf{w}) \nabla^2 e_i(\mathbf{w}) \right] \\
&= 2\mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) + 2\mathbf{S}(\mathbf{w})
\end{aligned}
\tag{15}
$$

We will assume $\mathbf{S}(\mathbf{w}) = \sum_{i=1}^{n} e_i(\mathbf{w}) \nabla^2 e_i(\mathbf{w})$ is close to zero. Thus,

$$
\nabla^2 F(\mathbf{w}) \approx 2\mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) \tag{16}
$$

The typical training method used today is backpropagation. This method propagates the resulting errors back through the network for weight modifications. The simplest form of backpropagation is steepest descent.

Steepest descent takes a step of size μ in the direction opposite of the gradient of the error function. It has the form shown in Eq. (17) below. This assures us that $F(\mathbf{w}^{(i+1)}) < F(\mathbf{w}^{(i)})$ so that we will approach a minimum point of our objective function $F(\mathbf{w})$.

$$
\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \mu \nabla F(\mathbf{w}^{(i)}) \tag{17}
$$

The valuation of $a = f(wp + b)$ is the forward propagation of a neuron for a single input stimulus p. For backpropagation of the resulting error at the output $(t - a)$, we must find the gradient $\nabla F(\mathbf{w})$ where $F(\mathbf{w}) = E_D = (\mathbf{e}(\mathbf{w}))^T \mathbf{e}(\mathbf{w})$. Note that the errors $\mathbf{e}$ are a function of the weights $\mathbf{w}$ since $e = t - a$ and $a = f(wp + b)$. So, we must use the chain rule of the form

$$\frac{\partial}{\partial w} f(x) = \frac{\partial}{\partial x} f(x) \cdot \frac{\partial}{\partial w} x(w)$$

where $x = wp + b$ is clearly a function of the weights. We will not dwell on the derivation of backpropagation but refer the reader to e.g. [HaDe96] for a detailed treatment. We will only show the final form of equations as they apply to our network architecture.

For each error e_i of our 2-layer neural network containing S hidden layer neurons, the forward equations are

$$\begin{aligned} \mathbf{a}^1 &= f^1(\mathbf{w}^1 p + \mathbf{b}^1) \\ a^2 &= f^2(\mathbf{w}^2 \mathbf{a}^1 + b^2) \end{aligned} \tag{18}$$

Note that the output a^2 and bias b^2 for our single neuron output network are scalars. Now, for each training set sample we define the sensitivities

$$c^2 = -2\left[\frac{\partial}{\partial x_1^2} f^2(x_1^2)\right] e = -2\left[\frac{\partial}{\partial x_1^2} f^2(x_1^2)\right](t-a)$$

$$\mathbf{c}^1 = \begin{bmatrix} \frac{\partial}{\partial x_1^1} f^1(x_1^1) & 0 & \cdots & 0 \\ 0 & \frac{\partial}{\partial x_2^1} f^1(x_2^1) & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\partial}{\partial x_S^1} f^1(x_S^1) \end{bmatrix} (\mathbf{w}^2)^T \left(\frac{\partial}{\partial x_1^2} f^2(x_1^2)\right) \tag{19}$$

The act of learning in a neural network is updating the weights. For each error we have

$$\begin{aligned} \mathbf{w}^{1(i+1)} &= \mathbf{w}^{1(i)} - \mu \mathbf{c}^1 p \\ \mathbf{w}^{2(i+1)} &= \mathbf{w}^{2(i)} - \mu c^2 (\mathbf{a}^1)^T \\ \mathbf{b}^{1(i+1)} &= \mathbf{b}^{1(i)} - \mu \mathbf{c}^1 \\ b^{2(i+1)} &= b^{2(i)} - \mu c^2 \end{aligned} \tag{20}$$

We see from Eq. (17) that applying the chain rule to obtain the gradient $\nabla F(\mathbf{w})$ resulted in the use of the sensitivities $\mathbf{c}^1$ and $\mathbf{c}^2$. Each type of weight or bias has its own update equation because of the different results obtained by the chain rule.

Now, for each epoch of training, we sum the results of all n errors, then take a step. Training typically ceases when a minimum level of error E_D is found. Recall that near a minimum point of F, the gradient approaches zero, and E_D has a minimum there.

There are many variations on the backpropagation algorithm. One method of depicting a large group of these variations is

$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \mu \mathbf{R} \nabla F(\mathbf{w}^{(i)}) \tag{21}$$

where the matrix $\mathbf{R}$ modifies the descent direction. In steepest descent, $\mathbf{R} = \mathbf{I}$ and we have simple gradient descent. More complex numerical optimization techniques modify the gradient using local second derivative information. This requires the estimated Hessian matrix introduced earlier.

The learning method of our choice for the duration of this work is the Levenberg-Marquardt algorithm. It is a Gauss-Newton form which tempers the second derivative Hessian estimate with tendencies toward steepest descent when inaccuracies due to only having local second derivative information result in an increase in the error function. This results in generally faster learning for small networks.

Toy Problems

For the purpose of debugging software implementations and exploiting characteristics of various generalization improvement techniques, we will use "toy" functions. The toy functions will serve as the true function we wish our trained neural network mapping to resemble. Knowing the true function, we can plot network responses and do visual comparisons as well as use non-noisy data for evaluating the actual errors E_A. Both these techniques will help us decide which generalization improvement technique can best approximate the true function.

We will obtain samples from each toy function. Then we will add noise to these true values to create noisy measurements. These will populate our training, validation and testing sets as they are needed. Unless otherwise noted, the same data sets will be used throughout this work for best comparisons. The initial weights are selected by the Nguyen-

Widrow method. [NgWi90] Again, for common architectures the same initial weights will be used unless otherwise noted.

The two toy problems chosen are the single cycle sine function and the single cycle saw function. Their forms are shown in Figure 6. The sine function is smooth whereas the saw function has sharp points.

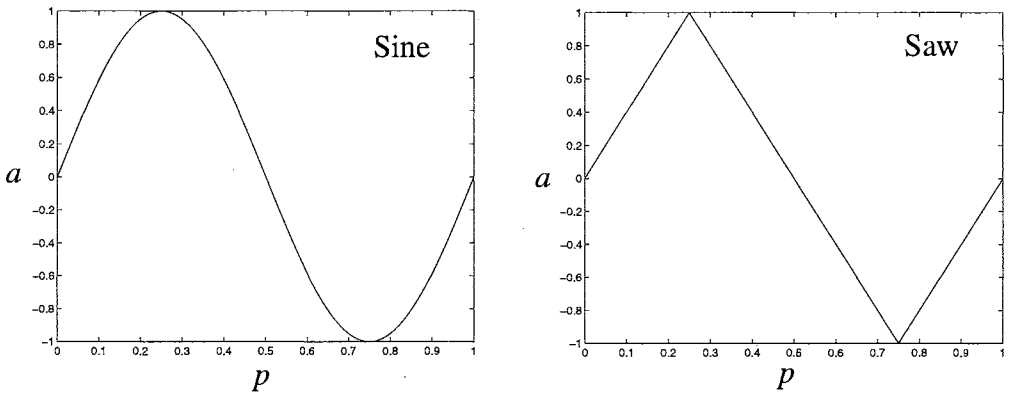


Figure 6 Toy Problems

Commonly Used Parameters

As a guide to notation, Table 1 shows the three types of typical variables used here. Table 2 shows a list of the most often used variables with a brief description. This list is not complete, but serves as a useful reminder of parameters carried from chapter to chapter.

Parameter Type	Style	Examples
scalar	non-bold Greek or italic letter	p, α, N
vector	bold small letter	**e**
matrix	bold capital letter	**F**

Table 1 Parameter Styles

Parameter	Type	Description
α	scalar	regularization parameter
β	scalar	error function parameter
$\gamma_{mo}, \gamma_{ma}, \gamma_{lj}$	scalar	Moody's/MacKay's/Ljung's effective number of parameters
λ	scalar	eigenvalue
λ^b, λ^h	scalar	eigenvalue of $\nabla^2\beta E_D$, $\nabla^2 F$
μ	scalar	training algorithm tunable parameter
σ_0	scalar	variance of noise in sample set
σ^2	scalar	variance
a, $\mathbf{a}$	scalar/vector	output of network
D	--	set of all training set data
e, $\mathbf{e}$	scalar/vector	single/all data set sample errors
e_i, $\mathbf{e}_i$	scalar/vector	single output value for single sample/part of $\mathbf{e}$
E_A	scalar	sum squared actual data errors
E_D	scalar	sum squared training set data errors
E_T	scalar	sum squared testing set data errors
E_V	scalar	sum squared validation set errors
E_W	scalar	sum squared weights
F	scalar	objective function
$\nabla F, \nabla^2 F$	vector/matrix	gradient/Hessian of F
$F(\mathbf{w}, D)$	scalar	objective function for D data set of size n
$F^{\alpha}(\mathbf{w})$	scalar	objective function for regularization factor of α
$\bar{F}(\mathbf{w})$	scalar	expectation of $F(\mathbf{w}, D)$ over training sets
$\bar{\bar{F}}$	scalar	expectation of $\bar{F}(\mathbf{w})$ over $\hat{\mathbf{w}}$
$\mathbf{G}$	matrix	matrix in *NIC*
GNBR	--	Gauss-Newton approximation to Bayesian Regularization
$\mathbf{H}$	matrix	Hessian of F
$\mathbf{I}$	matrix	identity matrix

Table 2 List of Parameters

Parameter	Type	Description
$\mathbf{j}_i$	vector	row part of $\mathbf{J}$ (produced by single sample)
$\mathbf{J}$	matrix	Jacobian of F
$\mathbf{K}$	matrix	expectation of $\mathbf{j}_i\mathbf{j}_i^T$
m	scalar	training step number
M	--	denotes a particular neural network model
$\mathbf{M}_\alpha$	matrix	weighted mean parameter based on α
$\mathbf{M}_m$	matrix	weighted mean parameter based on training steps m
n	scalar	number of training set samples
N	scalar	number of weights and biases in network
N^*	scalar	estimate of N
NIC	scalar	Network Information Criterion
p, $\mathbf{p}$	scalar/vector	input to network
$\mathbf{Q}$	matrix	matrix in NIC
$\mathbf{R}$	matrix	modifies gradient descent direction
S	scalar	number of hidden layer neurons
t, $\mathbf{t}$	scalar/vector	target(s) of sample input(s)
w_i, w	scalar	a neural network weight or bias
$\mathbf{w}$	vector	vector of weights and biases in the network
$\mathbf{w}^{\mathrm{ML}}$	vector	Most Likely weights based on min E_D
$\mathbf{w}^{\mathrm{MP}}$	vector	Most Probable weights based on Bayes' Optimization
w_o	scalar	constant
W	--	set of all weights and biases in network

Table 2 List of Parameters

Summary

In this chapter, we have presented relevant background for the purpose of introducing our notation. This notation will be our foundation for mathematical comparisons made in the following chapters.

We will now have a more extensive discussion of generalization and then present some existing procedures for improving generalization. We have bound this part of the work to feed-forward 2-layer neural networks with a single-input/single-output architecture for function approximation. The discussions and work in the following chapters will be limited to this situation.

In the next chapter, we will present applicable methods of improving generalization. Some of these we will explore further in later chapters.

CHAPTER 3

GENERALIZATION

Intent of Chapter

In this chapter, we will define generalization as it applies to our neural network architectures introduced in Chapter 2. Then we will discuss notable methods of improving generalization performance. Finally, we will note the methods we have decided to pursue in this dissertation.

Generalization

The intent of training a neural network is to represent in the network the function described by the training set. We would hope that the training set samples fully describe the form of the true function. For this reason, some of the requirements falling on the training set include the quantity and distribution of the samples. If the number and distribution of training set samples is not enough to fully describe the regularity or salient features of the true function, then the best a neural network can do is assign some random components to some of its parameters. We will not pursue training set requirements here. Rather, we will concentrate on how to ensure the most appropriate number of parameters are available in the neural network for a given training set.

The ability of a neural network to learn a function from a training set is driven by the relationship of two quantities: the number for parameters necessary to adequately de-

scribe the true function and the number of parameters available in the neural network. [BoLi96]

Too few or too many parameters in the neural network can have devastating effects on the ability of the neural network to properly learn the true function. It is not enough for the neural network to learn to appropriately map a particular training set sample input to the required output value. For any training set, given enough hidden layer neurons, a neural network can be trained to "memorize" the samples. However, overfitting can easily result.

Regression Comparison

The concept of overfitting in a neural network is much the same as in linear regression. For a finite set of data regressed onto an overparameterized model, the model can be made to fit the data, but testing points of the true function outside the sampled set may show that the model does not even grossly represent the true function.

Figure 7 shows a simple example of overfitting. The data is represented by circles. Polynomials of orders 6 and 8 were regressed onto the data. In the figure, the lines represent the actual polynomial mappings. The sixth order polynomial did not fit the data as well as the eighth order polynomial. However, the higher order polynomial does not appear to be the proper interpretation of the function the data represents. The two equations are

$$\begin{aligned} a &= -0.0037p^6 + 0.11p^5 - 1.28p^4 + 7.01p^3 - 19.2p^2 + 26.4p - 11.1 \\ a &= 0.002p^8 - 0.079p^7 + 1.33p^6 - 12.1p^5 + 64.5p^4 - 206p^3 + 382p^2 - 367p + 140 \end{aligned} \tag{22}$$

Definition: *Generalization* -- If a testing set is taken from the same probability function as the training set, then good generalization means the performance of the neural network will be about the same for both the training set and the testing set. [BoLi96]

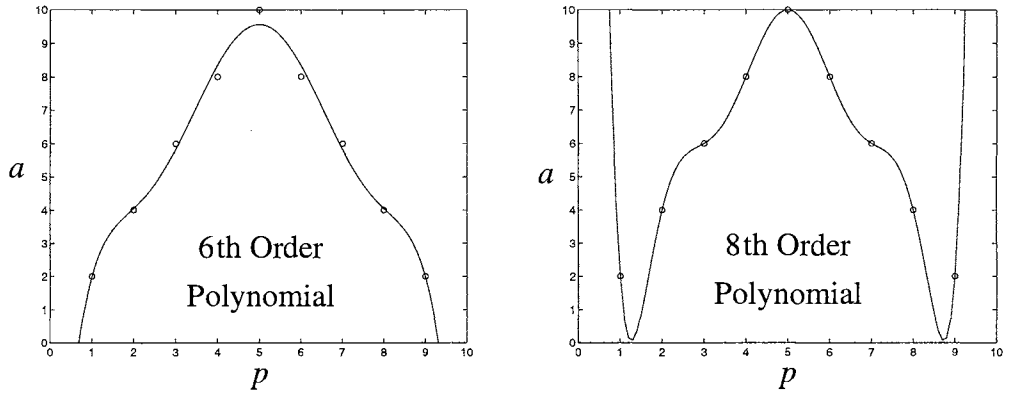


Figure 7 Fitting of Data for Polynomials of Order 6 and 8

With the above definition, note that good generalization is not equivalent to good approximation of the function being learned. Also, poor generalization of a neural network does not necessarily imply overfitting of the data. If the neural network does not contain enough parameters to learn the function, then underfitting results. In a general sense, this means that either the learned function is too smooth or that only a subset of the true function is represented.

For the toy problems where the true function is known, we will use E_A as a comparative empirical measure of generalization performance. When the true function is not known, we will sequester a portion of the available data samples that will not be used for training and use E_T for performance comparisons.

Improving Generalization

Neural networks have problems obtaining good generalization through training. These are much like the problems encountered in linear regression. Much time has been spent in the statistical world developing methods for comparing regression models and evaluating the performance of models. Neural network enthusiasts also have many ways

of comparing trained neural networks or empirically evaluating them and incorporating changes to improve generalization performance.

Here, we will briefly describe four distinct categories of generalization techniques used in neural networks. Pruning starts with an oversized neural network and removes "unnecessary" parts after training is finished. Growing starts with a small network and adds more neurons as needed. Statistical techniques attempt to evaluate the performance of trained neural networks for comparative purposes. Other techniques, mostly disparate in nature, exist but only a few have widespread acceptance.

Pruning Techniques

Pruning gets its name from gardening: if a limb or branch does not significantly contribute to the whole, excise it. There are many variations of pruning, though most stem from one of the two discussed here.

The simplest form of pruning examines the weights and biases of a trained neural network and removes or zeroes any which are close to zero. This static exercise assumes that a small weight contributes very little to the overall mapping function. However, given the nonlinearity of the neural networks, this need not be true. Static pruning may work to some degree, but results are not guaranteed. [SiDo91]

A more credible method of pruning is called Optimal Brain Damage. This method aims to remove unimportant weights by examining second derivative information of the objective function. Through perturbation analysis of the objective function, the second derivatives can be used to give an indication as to their importance in reducing the objective function. Those weights which have minimal effect on the objective function are removed

and the training resumes from there. Cases have been sited where an otherwise optimal fully-connected network was reduced by more than half the number of weights and yet gained significant improvements in generalization. [CuDe90]

Growing Techniques

Growing a neural network obviously involves adding to an existing architecture. On the simple side, one can choose a neural network which is inherently too small and train it and evaluate its performance. Then, add a single neuron to the hidden layer and continue training. In this process, neurons continue to be added as long as performance improves.

Growing methods can also be more deliberate. Two such techniques are stacked generalization and cascade correlation models. Both involve adding to existing architectures with the intent of improving generalization performance at a fundamental level.

Stacked Generalization

Stacked generalization (or ensemble methods) is a way of combining multiple independently trained neural networks to reduce generalization error. The neural networks may be of different architectures, or trained differently or simply using a different training set (e.g., the statistical leave-one-out implementation described on page 34). The outputs of all neural networks are combined in some fashion (e.g., weighted sum, average, winner-take-all, or nonlinear combination) to produce the overall "corrected" output. Certain constraints must be met in some variations (e.g., the sum of all the weights used to combine the networks must equal one, or the size of training set may need to be consistent for each network). [Wolp92]

This technique uses the idea that the error of each neural network is independent of the others so it can be "averaged" out. This also means that each neural network must be trained and stopped at a different local minimum. In [PeCo92], it is experimentally shown that there are usually few local minima, so the ensemble network may only require a few distinct neural networks. Also, each architecturally similar neural network is usually trained with a different cross validation subset of data. (See "Stopped Training" on page 36.)

Cascade-Correlation

The cascade-correlation network has a different structure than other neural networks noted here. It begins as a single layer of neurons, the output layer. All inputs available are connected to each of the output layer neurons. Typically, some form of gradient descent training method is used to minimize the training set error. When performance stops improving, the error level of the network is checked. If it is low enough, training stops. If not, then a single new neuron is added to the network.

The new neuron is chosen by first training several candidates with all available inputs to each. The neuron with the best correlation between its output and the established residual errors of the network is chosen to be inserted into the network. The input weights for the new neuron remain frozen, but the new connection from its output to the output layer neurons is modified along with the rest of the network as training resumes. This process is repeated by checking the final error level and adding another neuron, if necessary.

The key here is that each new neuron has all available inputs plus the outputs of each previously added neuron as its total complement of inputs. In this way, the new neu-

ron has the same inputs as the output layer neurons. With its output correlated with the errors of the original network, these errors are hopefully attenuated by the output layer neurons when connected as an input to them. If the first neural network learns only the most coarse characteristics of the desired function, then the subsequent cascades add increasingly higher degrees of detail. [FaLe90]

Statistical Techniques

This subsection gives a brief overview of some statistical techniques that have been adapted to apply to neural networks. Some may require a little background. We will provide that here.

Bootstrapping and jackknifing are two well known statistical methods employed for parameter estimation. These concepts are used in an attempt to overcome bias in the sample set used for network training. If all possible data were available in the training set then all measurements based on the exhaustive sample set must be accurate. However, the typical situation allows only a small subset of all possible data to be analyzed. This subset is usually the result of randomly sampling the true process.

One problem with random samples, however, is that they do not usually reflect the distribution of the true process. For example, let's say a neural network is trained on a particular training set. The resulting function will contain two types of error. First, if the learned function is on average different from the true function, then it contains bias. Second, if the learned function is very sensitive to the peculiarities of the training set, such that a different randomly chosen training set yields a different learned function, then it contains variance.

An example of this concept is shown in Figure 8. Both plots show the curved true function, the training set indicated as "+" points, and the learned function. In the plot on the left, we see that the average difference between the training set points and the learned function is large, but will remain about the same size for any randomly chosen training set. This is an example of high bias and low variance. In the plot on the right, the learned function fits the training set exactly, but will have large errors if compared to most any other training set. This learned function has no bias, but a very high variance.

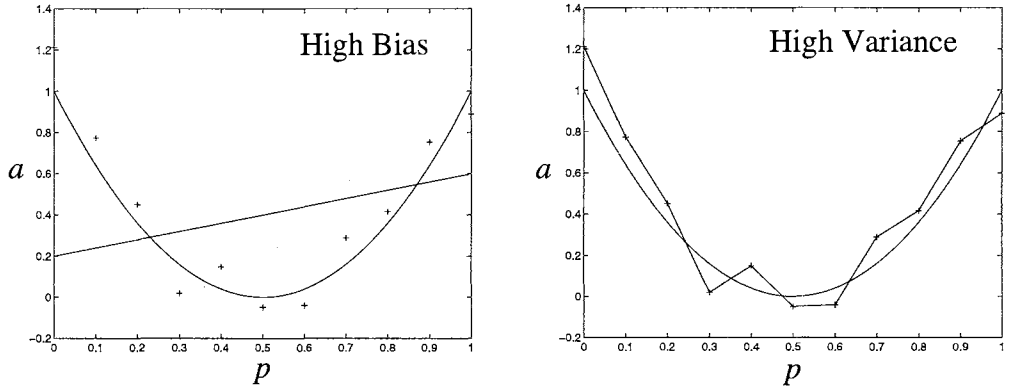


Figure 8 Examples of High Bias and High Variance

We can now see that if a neural network is trained to the level of the noise in the training set it will contain high variance error, whereas an untrained neural network might have high bias in its mapping function. (Unless the initial weights happen to reflect a minimum point of the error function.)

In this section, we will discuss four attempts to minimize these two error types. Bootstrapping and jackknifing characterize the bias and try to compensate for it. The *NIC* is a statistical measure of the training set fit of the learned function. Regularization attempts to minimize the bias and variance errors through training techniques.

Bootstrapping

Bootstrapping attempts to characterize the bias in sampling by analyzing many re-sampled sets. A resampled set is a set of samples taken from the original sample set. The samples are randomly taken from the original sample set one at a time. After each sample is taken, it is recorded in the new set and then placed back into the original set. In this manner, a sample has equal opportunity to be chosen every time. Thus, it is possible for the same sample to be recorded in the resampled set more than once. In fact, since the size of the resampled set is supposed to be equal to the size of the original set, the resampled set will surely have duplicates. This process is repeated for each resampled set required. When we sampled the true process, our sampling method did not preclude us from the possibility of drawing two identical samples. Resampling with replacement has the same idea.

Suppose we randomly sample our already chosen sample set, being careful to replace samples, giving them the possibility of being randomly chosen again. Now, calculate a statistic on the resampled set and then perform another independent resampling. After many resamplings, we average the statistic values found. The relationship between the average resampled statistic and the statistic for the complete sample set indicate the basis for a relationship between our sample set and the true process. We chose our samples for resampled sets from the original set in the same manner we chose our original set from the true function. If, for example, our average resampled statistic shows bias relative to the original set statistic we could assume the bias relationship can be extrapolated to the true function. Thus, the extrapolation of the two statistics will estimate the bias of the true function.

In bootstrapping we hope that the bias involved in the resampling relative to the training set is the same for the original training set relative to the true function being sampled (see [WeKu91]). Since we are extrapolating two statistics, we must use many resampled sets to ensure some degree of accuracy. This procedure, however, is usually computationally prohibitive since the rule of thumb is to perform at least 200 resamples and subsequent reestimations of the parameters.

Bootstrapping a neural network would involve training a neural network where the parameters being estimated are the weights and biases of the neurons. This procedure may work for neural networks whose trained state is "well defined" by the training set, such as generalized regression neural networks or probabilistic neural networks, but applying it to multilayered feed forward neural networks is problematic, at best. [Mast95]

First, the training takes time. Bootstrapping requires retraining the same neural network many times. Also, the randomness of which neuron takes on which feature will foul up averaging and comparing. Finally, falling into different local minima will require retraining from different initial conditions many times for each resampled set in an attempt to locate the same local minimum for all resampled training comparisons.

The work involved is too time consuming to get the best implementation possible for a single architecture. Also, there is no way to know how many different architectures must be trained before finding one with acceptable testing results. In addition, consider that bootstrapping is based more on experimental observations than on proven statistical techniques and thus is not guaranteed to produce more accurate parameter estimates.

Jackknifing

The jackknife method of parameter estimation assumes that the bias of the estimates is approximately inversely proportional to the sample size. Thus, extrapolation from two parameter estimates based on different sample sizes would give an estimate of the bias in the estimates. The first parameter estimate is, of course, based on the entire sample set, but we must use a subset of the entire sample set for the second parameter estimate.

The typical jackknifing method is to perform leave-one-out sampling. This involves estimating the parameter once for every new sample set created by leaving one data point out. If there are n samples in the original sample set, then there are n new sample sets for estimating the parameter. The average of the n new estimates is then compared to the original estimate that was based on the entire sample set. A linear extrapolation is then performed based on the two parameters. If we know the parameter estimates with set size n and $(n-1)$, then we can extrapolate from the parameter estimates at points $\frac{1}{n}$ and $\frac{1}{(n-1)}$ to zero, the point where n represents an infinite number of samples. [Mast95]

Jackknifing suffers from all the same problems as bootstrapping concerning computation time and local minima. Also, the basic assumption of the relation of bias to sample size cannot be affirmed until testing of the final estimated neural network parameters is performed.

NIC

The Network Information Criterion, NIC, is a statistical tool. The NIC is an extension of Akaike's Information Criterion, AIC, that is widely used to compare regression

models. The NIC is a statistic which consists of two parts. The first part is a measure of the accuracy of the network in fitting the training set. The second part is a measure of the complexity of the network. By minimizing the NIC we compromise between accuracy and complexity. Comparing multiple neural network architectures with the NIC will hopefully indicate the simplest network which sufficiently learns the true function. As in the rule of parsimony, the simpler network is thought to be least likely to overfit, thus providing better generalization. (See page 57.) The NIC will be discussed in detail in Chapter 4.

Regularization

Regularization is an attempt to restrict the size of the weights during training. The weights of the network are adjusted to minimize an objective function which is a combination of squared errors and squared weights. A regularization parameter multiplies the squared weights in the objective function. As this parameter is increased, more emphasis is placed on reducing the weights. Of course, too much emphasis on reducing the weights limits the neural network learning ability. This results in underfitting the true function with a learned function which is too smooth. Regularization will be discussed in detail in Chapter 5.

Bayesian Learning

What is missing with regularization is a way to automatically optimize the size of the regularization parameter. One answer to this problem is to put the training method in a probabilistic form, then solve for and optimize the regularization factor. This method uses Bayesian inference to find the most probable value of the regularization factor. This form of Bayesian learning will be discussed in detail in Chapter 6.

Other Techniques

Stopped Training

As with jackknifing, cross validation estimation also involves leave-one-out training. (See page 34.) However, for cross validation the one left out is used as a test case for performance evaluation. This is done for all samples in the training set, and the cross validation error estimate is the average error of the tested cases. There are many variations of this technique; e.g., in 10-fold cross validation, the training set is split into 10 sets, the neural network is trained 10 times, each time holding out a different 10% set, and subsequently testing on that set. [WeKu91]

In neural networks, stopped training is a variation of cross validation testing. A substantial portion of the training set, say 10% to 50%, is randomly chosen and held out and used for validation during training. The cross validation set error is observed during training and training ceases when the cross validation set error begins to increase, presumably at the onset of overfitting. This can only be done once, since retraining with different cross validation sets and choosing the neural network with the best validation test results invalidates the cross validation. Once the test set error is used for comparison, it has become a part of the training criteria. [Smit93]

An example case is shown in Figure 9 and Figure 10, where the underlying function is a saw wave. In Figure 9 is the sum of squared errors of the training set E_D and the sum of squared errors based on a separate validation set E_V progression during training of a 1-28-1 neural network trained on 24 points of training set data containing noise. Point A is the minimum value for E_V. Point B represents the final trained neural network. Notice the

upturn in the validation set error between these two points. It is at the onset of increasing E_V that overfitting may begin to happen. Eventually, the validation set error becomes much larger than the training set error.

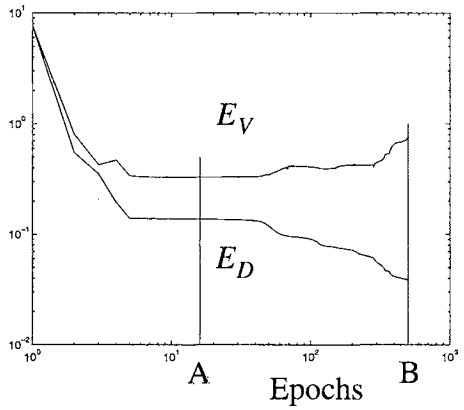


Figure 9 Training Set and Validation Set Errors

In Figure 10 we see the results of training to Point A and to Point B. The neural network function is plotted with the true saw function and the training set data, depicted as "+". For the plot at Point B, although the learned function passes very close to each training set data point, it obviously does not represent the true function very well. In contrast, the learned function as of Point A does not seem to show any signs of overfitting.

There is little statistical theory available to support stopped training, but there are many heuristics on how to use it. For example: use sufficient hidden units to avoid early local minima traps; use 10% - 50% of the training set as cross validation set; use a higher than typical percent for cross validation set if the total sample set is small; if the number of samples is greater than the number of neural network parameters but less than 30 times the number of neural network parameters, then use $\frac{100}{\sqrt{2N}}\%$ of samples for the cross valida-

tion set where N is the total number of weights and biases in the network; and, some training methods may be too fast for stopped training. (See, e.g., [Sarl95], [Prec94], [AmMu95], [Dodi94] and [Weig94].)

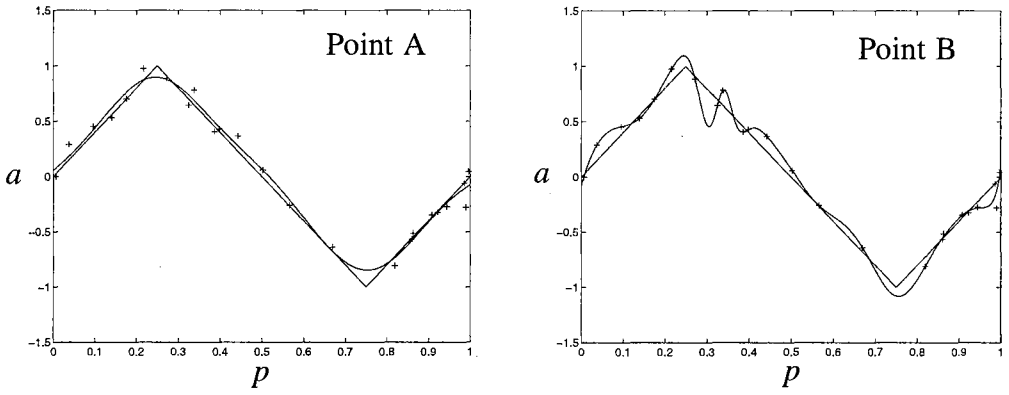


Figure 10 Learned Function at Point A and Point B

Stopped training is different from regularization techniques. Stopped training attempts to maximize generalization by stopping the training early, thus avoiding overfitting the data. Regularization avoids overfitting by limiting the complexity of the neural network so that it cannot learn minor features of the training set which are presumably not found in the true function. Also, regularization is surrounded by statistical theory that cannot be applied to early stopping techniques since a minima is never reached. In Chapter 7 we will describe the relationship between stopped training and regularization.

One problem with stopped training occurs when the number of training set samples is small. In this case, placing any samples in the cross validation set severely reduces the training set size. Another problem occurs when the cross validation set error never increases. This might occur when the neural network, because it is too small, is only able to learn coarse features of the true function that are common to both training and validation sets.

Also, some research shows that stopped training does not positively contribute to the neural network performance if the number of training set samples is greater than 30 times the number of neural network parameters and should not be used in this case (see [AmMu95]). Stopped training will be discussed in Chapter 7.

Discussion

Of all the techniques discussed, very few involve growing a neural network. Most adopt the idea of utilizing an oversized architecture and proceeding to constrain it somehow. Without constraints, the neural network will overfit the data. It is widely accepted that we can discourage overfitting in three ways.

First, we can limit the number of hidden neurons by pruning or statistically comparing different models. Techniques involving removing weights after training are straight forward and not computationally expensive; e.g., only one neural network is trained only one time. But, results are not guaranteed. Empirical comparison methods, however, have strong roots in statistics. [Smit93]

Second, we can constrain the weight values. This would include statistical regularization and weight decay methods. These methods bound the weight space of the learning problem, instead of restricting the actual number of dimensions.

Third, we can limit the amount of training. Here, of course, we are referring to stopped training methods involving validation data set criteria. Again, we are not reducing the dimensionality of the problem, but choosing to stop training before the high dimensional space is fully exploited by the gradient descent learning technique, which will lead to overfitting.

We could pursue many directions from here, but choose to investigate what appear to be the most promising ideas in each of the three categories of discouraging overfitting. In the next few chapters, we will introduce in greater detail the NIC, regularization and stopped training. In each case, we will implement the basic algorithms and show results of experimental trials. Where applicable, we will compare results.

Summary

In this chapter, we have described the concept of generalization as it applies to neural networks used in function approximation. We have also discussed the many methods of improving generalization. Most techniques are based on limiting the size of the network, limiting weight values or limiting training. We have chosen to pursue one technique in each of the three categories which we perceive as having the most potential.

In the next four chapters, we will introduce the NIC, regularization, Bayesian control of regularization and stopped training. Within each topic, we will describe the technique, implement the algorithm, run a few trials and discuss findings.

We will describe the NIC first in Chapter 4. Regularization will follow next in Chapter 5, which explores how regularization affects learning. Several simple examples are used to illustrate the application of regularization.

Chapter 6 will introduce Bayesian techniques for regularization. This will lead to better control of regularization through automatic tuning of the regularization factor.

Finally, stopped training will be discussed in Chapter 7. Validation set monitoring is the key feature of this chapter. As a very popular technique used today, we will compare the performance of the stopped training technique with the other generalization methods.

CHAPTER 4

NIC

Introduction

In this chapter, we will introduce the Network Information Criterion, NIC. It is a statistical measure one can use to compare performance of neural networks. We will show our adaptation of the key equations and how they can be approximated within the Levenberg-Marquardt algorithm. Finally, we will show our experimental results and note successes and problems encountered.

Background

The NIC is a statistically derived criterion for model selection. The objective is to select the neural network model which has the best generalization. If a model is too simple, containing too few parameters, it will underfit the data. Too complex a model may fit the training set data well, but may not reflect the true features of the underlying function because of overfitting. One way to minimize overfitting is to select the simplest model providing an appropriate fit to the training data. The NIC is designed to provide an empirical model comparison to do just that.

Once a network is fully trained, the NIC can be calculated and compared with the NIC value of another model. The lowest value represents the best model fit.

In [MuYo94], Amari outlines the development of the NIC. The NIC is based on Akaike's Information Criterion, AIC, which is used in regression model comparison. In fact, Amari shows it to be a generalized extension of the AIC to include comparison of underfitting models. Thus, an array of models for the same data set should produce a valley of NIC values. The lowest point represents the best fitting model, with underfitting on one side and overfitting on the other.

Since the NIC is based on the AIC, we would expect it to have two parts. One part would represent how well the model fits the training set data, and the other would be a measure of complexity, as in the following form.

$$NIC = g_1(\text{errors}) + g_2(\text{model complexity}) \tag{23}$$

The first part is based on the training set errors. As the size of the neural network increases, the errors should correspondingly diminish. However, the second part of the NIC is a size penalty function. So, although increasing the size of the network lowers the error term, it will simultaneously increase the complexity term.

Comparing the NIC values for different architecture sizes should allow us to find the neural network model producing the smallest possible NIC. This model should represent the best trade off between training set errors and model size.

We will now show our adaptation of the equations of the NIC. First, assume a standard objective function of

$$F = E_D \tag{24}$$

where E_D is the sum of the squares of the training set errors. We then express F in the

form of a log loss probability function. For this, we must recognize that the *NIC* is derived around a noise model. It assumes the training set data can be modeled as

$$a = f(p, \mathbf{w}) + \varepsilon \tag{25}$$

where p is the input to the true function $f(p, \mathbf{w})$ whose output is dependent upon parameters $\mathbf{w}$. The ε parameter represents noise added to the system to obtain the sample outputs a.

If the noise is independent of $\mathbf{w}$ and Gaussian, then the conditional distribution is

$$P(a|p, \mathbf{w}) = \psi(a - f(p, \mathbf{w})) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(a - f(p, \mathbf{w}))^2\right). \tag{26}$$

From Eq. (24) we note $e = a - t = a - f(p, \mathbf{w})$ are the errors we are minimizing during training. Thus, the log loss of $P(a|p, \mathbf{w})$ is

$$d(\mathbf{w}) = -\log P(a|p, \mathbf{w}) = \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}e^2. \tag{27}$$

Next, we need to find the gradient $\nabla d(\mathbf{w})$ and the Hessian $\nabla^2 d(\mathbf{w})$. From Eq. (9) and Eq. (13), the matrix form of the gradient $\nabla d(\mathbf{w})$ is

$$\begin{aligned} \nabla d(\mathbf{w}) &= \frac{\partial}{\partial \mathbf{w}}(-\log(P(a|p, \mathbf{w}))) \\ &= \frac{\partial}{\partial \mathbf{w}}\left(\frac{1}{2\sigma^2}(\mathbf{e}(\mathbf{w}))^T \mathbf{e}(\mathbf{w})\right) \\ &= \frac{1}{2\sigma^2} \cdot \frac{\partial}{\partial \mathbf{w}} F(\mathbf{w}) \\ &= \frac{1}{2\sigma^2} \cdot 2\mathbf{J}^T(\mathbf{w})\mathbf{e}(\mathbf{w}) \\ &= \frac{1}{\sigma^2}\mathbf{J}^T(\mathbf{w})\mathbf{e}(\mathbf{w}) \end{aligned} \tag{28}$$

For the Hessian $\nabla^2 d(\mathbf{w})$, using Eq. (16), we have

$$\begin{aligned}
\nabla^2 d(\mathbf{w}) &= \frac{\partial^2}{\partial \mathbf{w} \partial \mathbf{w}}(-\log(P(a|p, \mathbf{w}))) \\
&= \frac{\partial^2}{\partial \mathbf{w} \partial \mathbf{w}}\left(\frac{1}{2\sigma^2}(\mathbf{e}(\mathbf{w}))^T \mathbf{e}(\mathbf{w})\right) \\
&= \frac{1}{2\sigma^2} \cdot \frac{\partial^2}{\partial \mathbf{w} \partial \mathbf{w}} F(\mathbf{w}) \\
&= \frac{1}{2\sigma^2} \cdot 2\mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) \\
&= \frac{1}{\sigma^2}\mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w})
\end{aligned} \tag{29}$$

Finally, we denote the variance of the gradient $\nabla d(\mathbf{w})$ and the expectation of the Hessian $\nabla^2 d(\mathbf{w})$ by

$$\mathbf{G} \equiv \mathrm{Var}\{\nabla d(\mathbf{w})\} \tag{30}$$

$$\mathbf{Q} \equiv \mathrm{E}\{\nabla^2 d(\mathbf{w})\} \tag{31}$$

Now, with Eq. (30) and Eq. (31) we find the *NIC* of a model trained to minimize Eq. (24) as

$$NIC = \frac{1}{n}F + \frac{1}{n}\mathrm{tr}(\mathbf{G}\mathbf{Q}^{-1}) \tag{32}$$

where n is the number of training set samples and "tr" is the trace of the given matrix. Thus we see the *NIC* is the sum of the final trained objective function and another factor which can be shown to be a function of the number of parameters (weights) in the network. According to [MuYo94], if the model does not underfit the data for a single-input/single-output neural network trained with Eq. (24), then $\mathbf{G} = \sigma^2\mathbf{Q}$, where σ^2 is the sample variance

of the training set errors. This means $\frac{1}{\sigma^2}\mathrm{tr}(\mathbf{G}\mathbf{Q}^{-1})$ of Eq. (32) should be proportional to the number of parameters in the neural network, which we will call N^*. So, our *NIC* implementation reduces to the average squared error plus a ratio of the number of neural network parameters to the training set sample size.

Method of Application

Given the equations above, we need only train a number of a neural networks and evaluate them. The parameters $\mathbf{G}$ and $\mathbf{Q}$ can be estimated as shown in Eq. (33) and Eq. (34)

$$\mathbf{G} \approx \frac{1}{n}\sum_{i=1}^{n} (\nabla d_i(\mathbf{w}))(\nabla d_i(\mathbf{w}))^T \tag{33}$$

$$\mathbf{Q} \approx \frac{1}{n}\sum_{i=1}^{n} (\nabla^2 d_i(\mathbf{w})) \tag{34}$$

where ∇d_i is our log loss function expressed for a single training set sample i. We must now adapt these equations for use in our algorithm. [Mura93]

Following the development of the Levenberg-Marquardt algorithm in [HaDe96], in Eq. (12), we note that for $F(\mathbf{w}) = E_D = \mathbf{e}^T(\mathbf{w})\mathbf{e}(\mathbf{w})$ the gradient can be expressed as

$$\nabla F(\mathbf{w}) = 2\mathbf{J}^T(\mathbf{w})\mathbf{e}(\mathbf{w}) \tag{35}$$

and from Eq. (16), the Hessian can be approximated by

$$\nabla^2 F(\mathbf{w}) \approx 2\mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) \tag{36}$$

where the Jacobian matrix $\mathbf{J}(\mathbf{w})$ is defined in Eq. (14), and all the training set errors are contained in $\mathbf{e}(\mathbf{w})$ as defined in Eq. (10). These equations are in matrix form for application to all errors in the training set at once.

To calculate Eq. (33) and Eq. (34), we need to separate the Jacobian and errors into n components. Each component represents the effect of one training set sample. Recalling the format of Eq. (14), we define

$$\mathbf{J} \equiv \begin{bmatrix} \mathbf{j}_1^T \\ \mathbf{j}_2^T \\ \vdots \\ \mathbf{j}_n^T \end{bmatrix} \tag{37}$$

where $\mathbf{j}_1$ is the $N \times 1$ part of the Jacobian attributed to the first training set sample, just as e_1 in $\mathbf{e}(\mathbf{w})$ is the output error for that same sample. Examining the matrix forms in Eq. (13) and Eq. (16), we see that for a single training set sample i we have

$$\nabla F_i(\mathbf{w}) = \frac{\partial}{\partial \mathbf{w}} e_i^2(\mathbf{w}) = 2\mathbf{j}_i(\mathbf{w}) e_i(\mathbf{w}) \tag{38}$$

and

$$\nabla^2 F_i(\mathbf{w}) = \frac{\partial^2}{\partial \mathbf{w} \partial \mathbf{w}} e_i^2(\mathbf{w}) \approx 2\mathbf{j}_i(\mathbf{w})(\mathbf{j}_i(\mathbf{w}))^T \tag{39}$$

We can apply these to Eq. (28) and Eq. (29). Thus for individual training set samples, we can rewrite Eq. (33) and Eq. (34) as

$$\begin{aligned}\mathbf{G} &= \frac{1}{n}\sum_{i=1}^{n}(\nabla d_i(\mathbf{w}))(\nabla d_i(\mathbf{w}))^T \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{2\sigma^2}(2\mathbf{j}_i(\mathbf{w})e_i(\mathbf{w}))\right)\left(\frac{1}{2\sigma^2}(2\mathbf{j}_i(\mathbf{w})e_i(\mathbf{w}))\right)^T \\ &= \frac{1}{n\sigma^4}\sum_{i=1}^{n}(\mathbf{j}_i(\mathbf{w})e_i(\mathbf{w}))(\mathbf{j}_i(\mathbf{w})e_i(\mathbf{w}))^T\end{aligned} \tag{40}$$

and

$$\begin{aligned}\mathbf{Q} &= \frac{1}{n}\sum_{i=1}^{n}(\nabla^2 d_i(\mathbf{w})) \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{2\sigma^2}(2\mathbf{j}_i(\mathbf{w})(\mathbf{j}_i(\mathbf{w}))^T)\right) \\ &= \frac{1}{n\sigma^2}\sum_{i=1}^{n}(\mathbf{j}_i(\mathbf{w})(\mathbf{j}_i(\mathbf{w}))^T)\end{aligned} \tag{41}$$

The reason for putting $\mathbf{G}$ and $\mathbf{Q}$ into these terms is that the elements of these functions are readily available in our algorithm. As for σ^2, given our noise model of Eq. (25), we can estimate σ^2 over the entire training set as

$$\sigma^2 \approx \frac{1}{n}\sum_{i=1}^{n}(a_i - f(p_i, \mathbf{w}))^2 \tag{42}$$

Again, we note from Eq. (24) that $e = a - t = a - f(p, \mathbf{w})$. Thus, $\sigma^2 \approx \frac{1}{n}E_D$.

Now we have all the parts of the NIC available in the form used in the Levenberg-Marquardt algorithm. After training a neural network, we can use the last computed values of $\mathbf{j}_i(\mathbf{w})$, $e_i(\mathbf{w})$ and E_D to find the NIC for the trained network. For our application, the NIC of Eq. (32) becomes

$$NIC = \frac{1}{n}E_D + \frac{1}{n}\mathrm{tr}(\mathbf{G}\mathbf{Q}^{-1}) \tag{43}$$

where σ^2 contained in $\mathrm{tr}(\mathbf{G}\mathbf{Q}^{-1})$ is the sample variance estimated as

$$\sigma^2 \approx \frac{1}{n}E_D. \tag{44}$$

Trials

For our first experiment with the NIC, we trained a number of neural network architectures using noisy data from a four period sine wave. The equation for the noisy sine

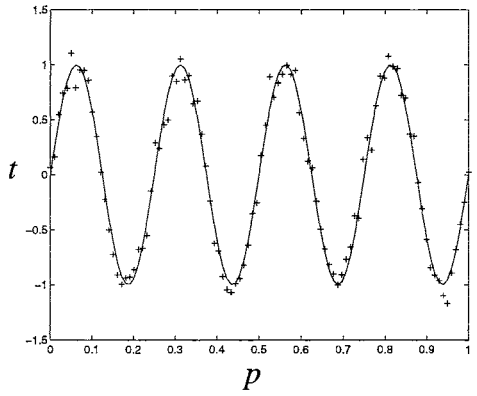


Figure 11 Four Period Sine Function

function is $t = \sin(4(2\pi p)) + 0.1(\varepsilon)$ where ε is random normally distributed noise of zero mean and variance one. Figure 11 shows the four period sine function and the noisy training set. Figure 12 shows the NIC values for nine different sizes of hidden layers rang-

ing from 2 to 30 hidden layer neurons. In both plots, the number of hidden layer neurons S counts along the x-axis. The 1-5-1 architecture was the simplest architecture to generally fit the data. This is reflected in Figure 12 as the lowest NIC value. As the architectural complexity increases, the NIC generally increases, until another fundamental level of fitting is able to occur. The 1-15-1 network, for example, has a sufficient number of parameters to capture some significant effects of the noise in the data, resulting in overfitting. Although the NIC drops at this point, due to a strong reduction in the E_D, it is still significantly higher then the 1-5-1 NIC.

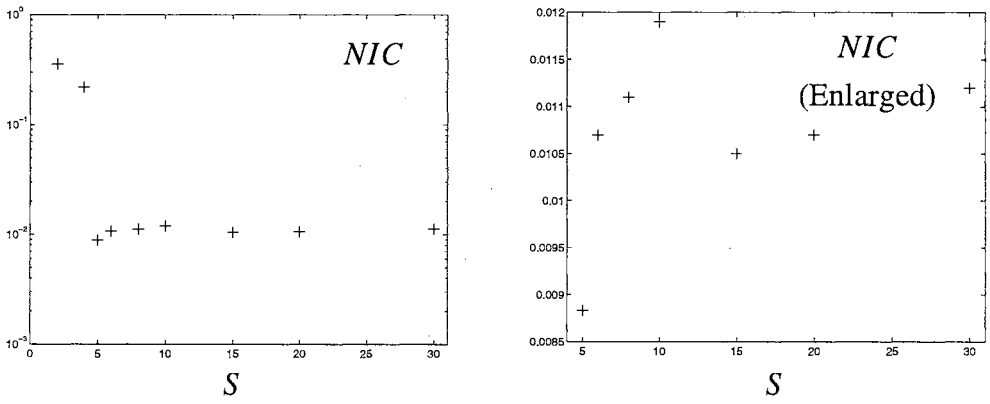


Figure 12 *NIC* for Four Period Sine Wave

In general, this was not a very enlightening example. However, applying the NIC to the saw function was very intriguing. For the true saw function, fitting the sharp points better requires increasing the number of parameters. However, at some point, the extra parameters will try to fit the sharp areas of the noisy data set as if they were sharp parts of the true function.

Figure 13 shows examples of underfitting and overfitting for the saw function. Just as for the sine wave, we added random normally distributed noise of mean zero and variance 0.01. We can see that the first fundamental fit of the data happens with the 1-2-1 network. But the learned function is obviously too smooth (only because we know the underlying function). The 1-6-1 network did the best job of fitting the saw function since it contained the highest level of complexity without overfitting.

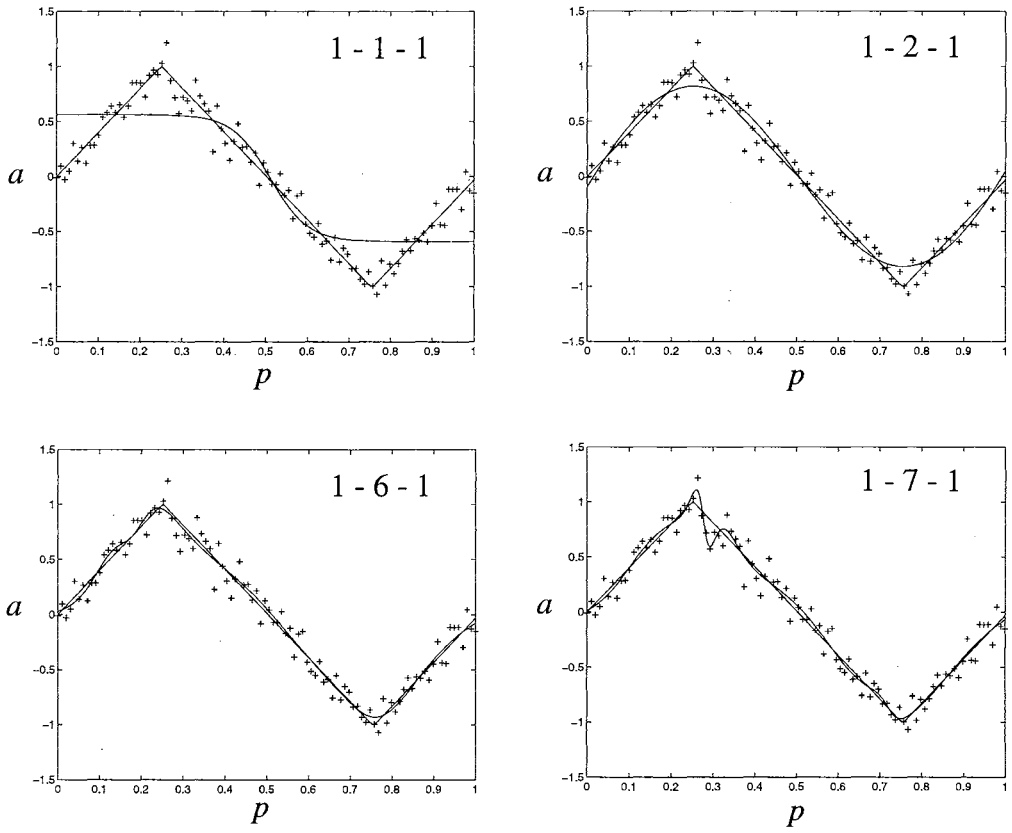


Figure 13 Actual, Training Set, and Learned Functions for Different Architectures

Having made this cursory comparison, we now examine the numbers. Figure 14 shows the results of training the neural network architectures for the saw problem with the number of hidden layer neurons plotted on the horizontal axis for each. In Figure 14, we

see that the *NIC* would have us believe that the 1-2-1 architecture is the best model. It has the lowest calculated *NIC* value. However, the plot of actual squared errors (the errors between the fitted model and the underlying saw function) in Figure 14 shows the 1-6-1 to be the clear winner. In fact, the small area of the saw function which is overfit by the 1-7-1 and 1-8-1 networks still results in lower total actual squared error than the choice indicated by the *NIC* of the 1-2-1 architecture. (Recall that the actual squared errors cannot be computed in practice, since we do not generally know the true underlying function.)

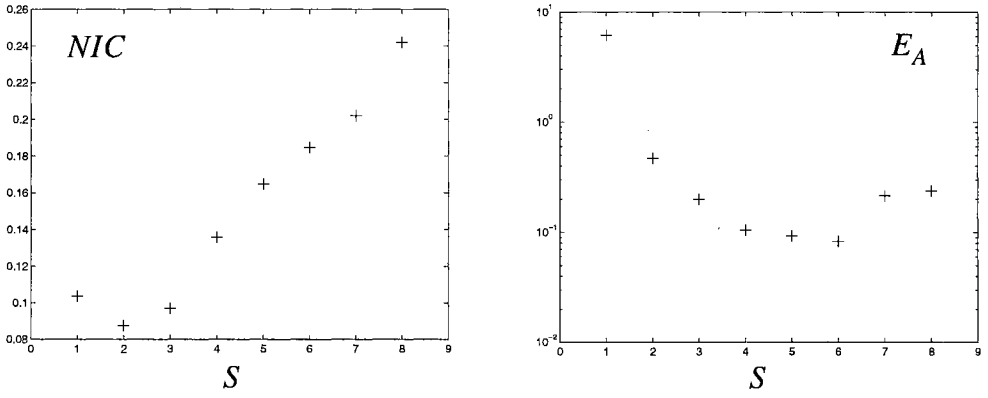


Figure 14 *NIC* and Actual Squared Error for Different Architectures

In trying to find a clue as to what is happening, we examine three key items. First, recall our application of the *NIC* from Eq. (43). The E_D will be large for all underfitting models and distinguishably smaller for any model which can fit the data. The other term is the trace of a square matrix of dimension equal to the total number of parameters in the neural network.

Figure 15 is a plot of the sample derived estimate of the number of parameters N^* and the final training set error E_D for several architectures. They are plotted with the num-

ber of hidden layer neurons along the x-axis. The estimated number of parameters are indicated as a "+" for each model while the true number of parameters are shown as a "o". Thus, our estimates are very good.

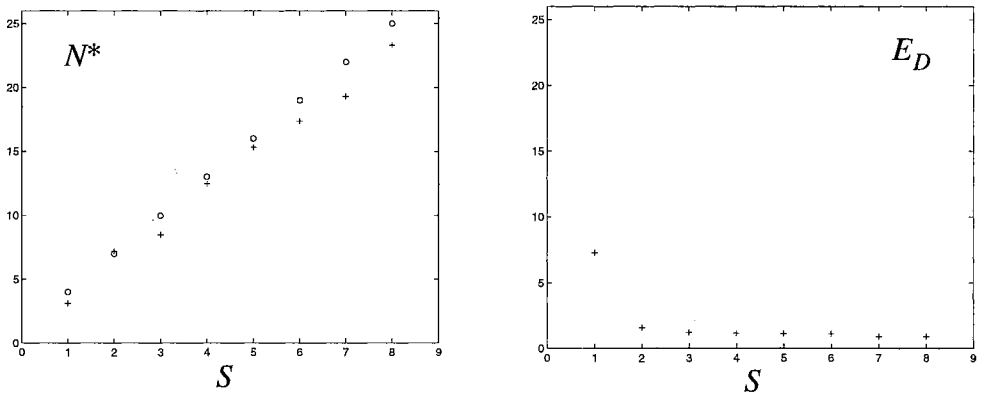


Figure 15 N^* and E_D for the Saw Function

For a second clue, note the size of N^* in Figure 15 in comparison to the size of E_D. The parameter N^* plays the dominate role in Eq. (43) for all but the simplest architectures, thus our trend of the NIC shown in Figure 14.

For a third and final indication, look at the normalized E_W in Figure 16. The E_W is a sum of squares of weight values. Similar to the E_D, it is a simplistic way of tracking weight size. For architecture comparison purposes, however, E_W must be normalized by dividing it by the number of parameters it represents since each neural network architecture has a different number of parameters.

In Figure 15, while the mid range networks could do little to distinguish themselves in reducing the E_D, the high end architectures have the complexity to do so. They reduce

the E_D slightly, yet distinguishably, but at the cost of the very large average squared weight shown in Figure 16. Thus, significant overfitting seems to be synonymous with a tremendous surge in weight size.

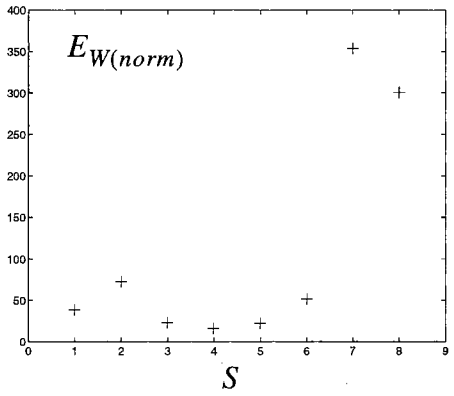


Figure 16 Normalized E_W for the Saw Function

Summary

In this chapter we have shown how we adapted the *NIC* to our algorithm making the assumption of Gaussian noise in the training set. Our results for preliminary experiments were shown using our toy problems where we know the optimal neural network size.

The *NIC* would seem to be a great way to compare different models and even different model types, based on the widely accepted standard for regression modeling -- the AIC. However, it requires calculating the inverse of the Hessian. Not only is this computationally expensive, but overparameterization may cause singularity problems. Also, several models must be trained before the *NIC* comparisons can be made. In addition, there is no guide for which model types or sizes to try.

Most importantly, the NIC is not fool proof. Our experiments indicate that the best model might be the most complex model which is not complex enough to seriously overfit. However, without prior knowledge of the true function we cannot monitor the possibilities of overfitting. But, we can watch the progression of E_W during training. We can even include the E_W term in our objective function as an attempt to discourage high weight sizes, thus reducing possibilities of overfitting.

These ideas are the subject of the next two chapters as we move on from empirical model comparisons to statistical regularization techniques for improving generalization.

CHAPTER 5

REGULARIZATION

Introduction

In this chapter, we will explore the addition of regularization to the objective function. Most of the background material comes from [Mood92]. After noting the key equations, we will incorporate regularization into the Levenberg-Marquardt algorithm. Some graphical results will show how regularization affects the resulting neural network mapping function. We will then introduce the concept of the effective number of parameters that we will build on in the next chapter. Finally, we will summarize our findings.

Background

Regularization attempts to restrict the size of the parameters in a model. This simply means that we anticipate that the function has some degree of smoothness. It is equivalent to assuming that small changes in the input will not result in a large change in the output. Given the noise inherent to measuring techniques, this assumption is sound and is one of the least restrictive assumptions that can be made on the target function (see [ReMa95]).

For the objective function

$$F = E_D + \alpha E_W, \tag{45}$$

the regularization term, E_W, creates a biasing of weight sizes. (Note, the bias referred to here is different from the discussion on page 30 in Chapter 3.) The regularization term causes an unnatural tendency of the weights to migrate, during training, toward a particular value.

There are many possible regularization functions. Two of the more popular functions are

$$E_W = \sum_i (w_i - w_o)^2 \tag{46}$$

and

$$E_W = \sum_i \frac{w_i^2}{(w_i^2 - w_o^2)} \tag{47}$$

where w_i is a weight in the neural network and w_o is a constant. Eq. (46) is a sum of squares of weights adjusted by a constant (see [SjLj92]). If $w_o = 0$, the regularization term simply attempts to reduce the magnitude of all the weights. In Eq. (47), the effect is to penalize weights with values much larger than the nonzero constant w_o (see [WeRu91]).

Thus, just as the training set error term sways the weights toward reducing the errors, the regularization term sways the values of the weights toward a predetermined constant. For example, in Eq. (46) all the weights will be swayed toward the value w_o. If a weight w_i is smaller than w_o, then the weight will tend to grow. If it is larger than w_o, then the weight will tend to diminish in value.

In Eq. (45), we are not forcing the weights to become a particular value, but simply causing a tendency to migrate in that direction. The regularization parameter α dictates how dominant a role the biasing plays in the overall objective function. If α is very small, the error term must be reduced to a comparable amount before the regularization term will have much effect on the weight sizes. If α is very large, the error will be allowed to grow in a situation where weight reduction is paramount.

Though any reasonable scheme will work, we chose to use

$$E_W = \sum_i w_i^2 \tag{48}$$

for the regularization term for all the work contained in this report. It penalizes large weights to gain a similar effect to parsimony, preferring lower order polynomials in statistical regression. Note that in statistics, a parsimonious model fits the available data adequately without using any unnecessary parameters. It is well known that parsimonious models produce better forecasts.

We note here that Eq. (9) and Eq. (48) are sums of squares and therefore nonnegative. Now, for $\alpha > 0$, Eq. (45) is equivalent to the *smoothing functional* introduced by Tikhonov. The regularizing effect of minimizing this equation reduces parameter values for solving overparameterized problems. [Moro93] [TiGo90] [Tikh63]

Recognizing that our data contains noise, we are not trying to find a model which could have produced our exact data, but rather we wish to find a model which approximates the true process. Thus we want a model that statistically represents our data, yet has as few parameters as possible. [Pank83]

One way to implement parsimony would be to reduce the actual number of weights in the neural network and measure the effect on the training set error. We saw this in Chapter 4 with the NIC. Here, we are considering an alternate method of implementing parsimony. If we bound the weight sizes, we are limiting their usefulness.

Eq. (48) attempts to drive the size of the weights down, thus reducing or eliminating their effect on the overall objective function, hopefully achieving a more parsimonious model. Obviously, if $\alpha E_W >> E_D$ in Eq. (45), then the next step in training will emphasize reducing the weight values, perhaps at the price of a modest increase in the training set error E_D. Thus, without regularization the neural network is allowed to use all of its parameters to the fullest extent during training. Of course, this may lead to overtraining.

In contrast, with too much emphasis on regularization, α too large, the weights are too restricted in size to contribute much to the training set error reduction. In this case, the mapping function learned by the neural network is too smooth. This corresponds to underfitting the data.

However, with an appropriate value of α in Eq. (45), an optimum usage of parameters can balance minimizing training set errors with the principle of parsimony. It is in this balance that we will find the neural network with the best generalization performance.

Method of Application

From Eq. (45) above, our new total objective function now includes both the sum of the squares of the training set errors, E_D, and the sum of the squares of the weights and biases, E_W, tempered by the regularization factor α. With $\mathbf{w}$ being a vector of the weights

and biases, $\mathbf{e}$ a vector of training set errors and $\mathbf{J}$ the Jacobian matrix, we can follow the flow of the Levenberg-Marquardt algorithm development found in [HaDe96] and find that

$$\nabla F(\mathbf{w}) = 2\mathbf{J}^T\mathbf{e} + 2\alpha\mathbf{w} \tag{49}$$

and

$$\nabla^2 F(\mathbf{w}) \approx 2\mathbf{J}^T\mathbf{J} + 2\alpha\mathbf{I} \tag{50}$$

The parameter update equation now becomes

$$\begin{aligned} \mathbf{w}_{k+1} &= \mathbf{w}_k - [\nabla^2 F(\mathbf{w})|_{\mathbf{w}=\mathbf{w}_k}]^{-1}[\nabla F(\mathbf{w})|_{\mathbf{w}=\mathbf{w}_k}] \\ &= \mathbf{w}_k - [\mathbf{J}^T\mathbf{J} + (\alpha+\mu)\mathbf{I}]^{-1}[\mathbf{J}^T\mathbf{e} + \alpha\mathbf{w}_k] \end{aligned} \tag{51}$$

where μ is the Levenberg-Marquardt algorithm tunable parameter.

Trials

For a demonstration of how regularization affects neural network generalization, consider the saw function of Figure 6 of Chapter 2. We chose to explore regularization on the saw function because of the sharp points it has. In linear regression, too few parameters would look too smooth rounding out the peaks. Too many parameters might overfit, but hopefully regularization will prevent that.

We will use a 1-6-1 neural network architecture to learn the saw function. The training set contains 100 random samples of the saw function with normally distributed noise of zero mean and 0.01 variance added. For the following examples, the Levenberg-Marquardt training algorithm will use Eq. (51) for weight updates. These examples will illustrate the effect of the regularization parameter α on the network performance. Note that the plots for E_D, E_A and E_W show their value during the course of training. The x-

axis for these plots is the training step (epoch) number. The plots of the learned functions also show the saw function of Figure 6 and the noisy training set, indicated as a "+" for each point.

Figure 17 shows an example of overfitting. Although α is not zero, which would totally eliminate the influence of the size of the weights on the objective function, it is small enough to show overfitting of the data. Note the extra "squiggle" in the learned neural network function. The neural network is overfitting the data and losing track of the actual saw function. There is also a significant increase in the actual error, calculated by using noiseless data samples. The point during training where the actual error increases corresponds to when E_D decreases and E_W increases. The weights increase enough to lower the training set error slightly, and in the process overfit the data.

On the other hand, Figure 18 shows the underfitting that happens for α too large. Here, the weights were so restricted that the resulting function resembles a smooth sinusoid instead of a sharp saw function. Even though the final E_D is approximately the same value as in Figure 17, the E_W is two orders of magnitude smaller. Looking only at E_D and E_W, we might think the neural network learned a function that fit the data very well. But, in this demonstration, we have the luxury of knowing what the real function looks like. And even though the actual error seems to be non-increasing throughout training, the plot of the neural network function with the saw function shows the truth of underfitting.

Finally, with an appropriate value of α, nice results can be obtained as shown in Figure 19. Comparing Figure 19 with the previous two figures, we see that the final E_D

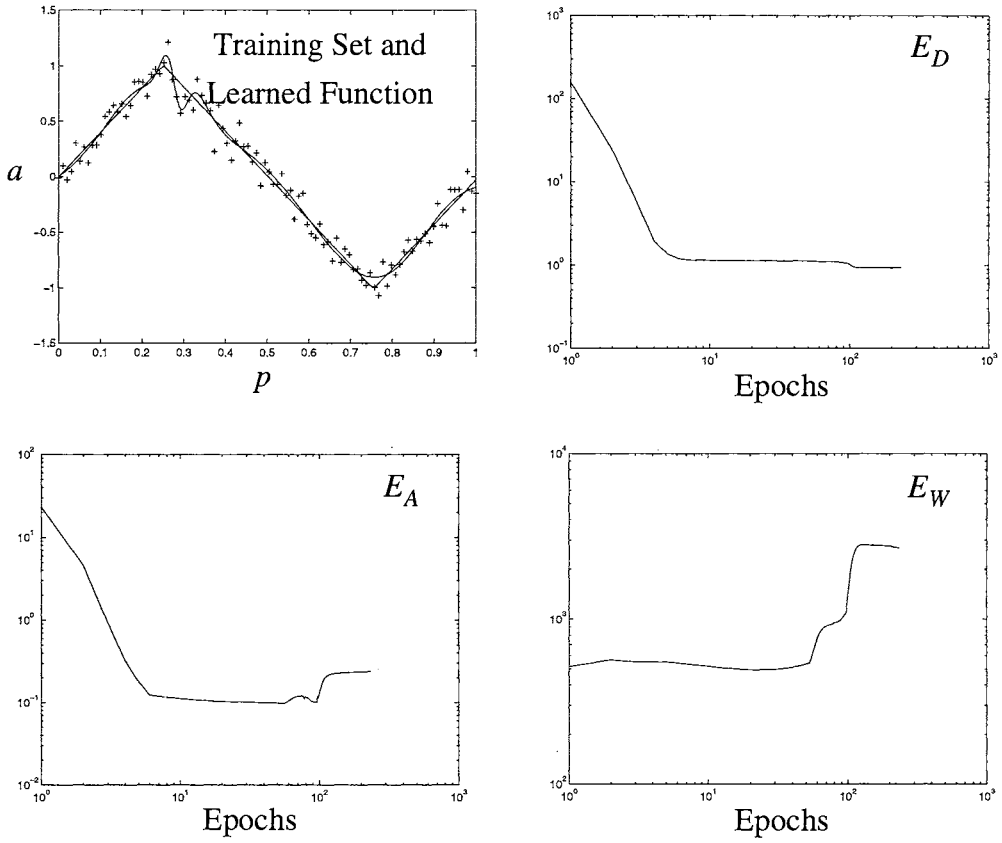


Figure 17 Overfitting Example for $\alpha = 0.001$

and E_W fall between those for overfitting and underfitting. However, the actual error is much improved over both other cases. This is quite noticeable in the saw versus neural network function plot.

A plot of the final values obtained from training trials for several values of α is given in Figure 20. Notice that for best results, α cannot be at either extreme of E_D or E_W. Sometimes a small change in α results in drastic changes in E_D or E_W. Also, note the valley in the actual error plot. The actual error increases for α too large, demonstrating underfitting. The actual error also increases for α too small, demonstrating overfitting.

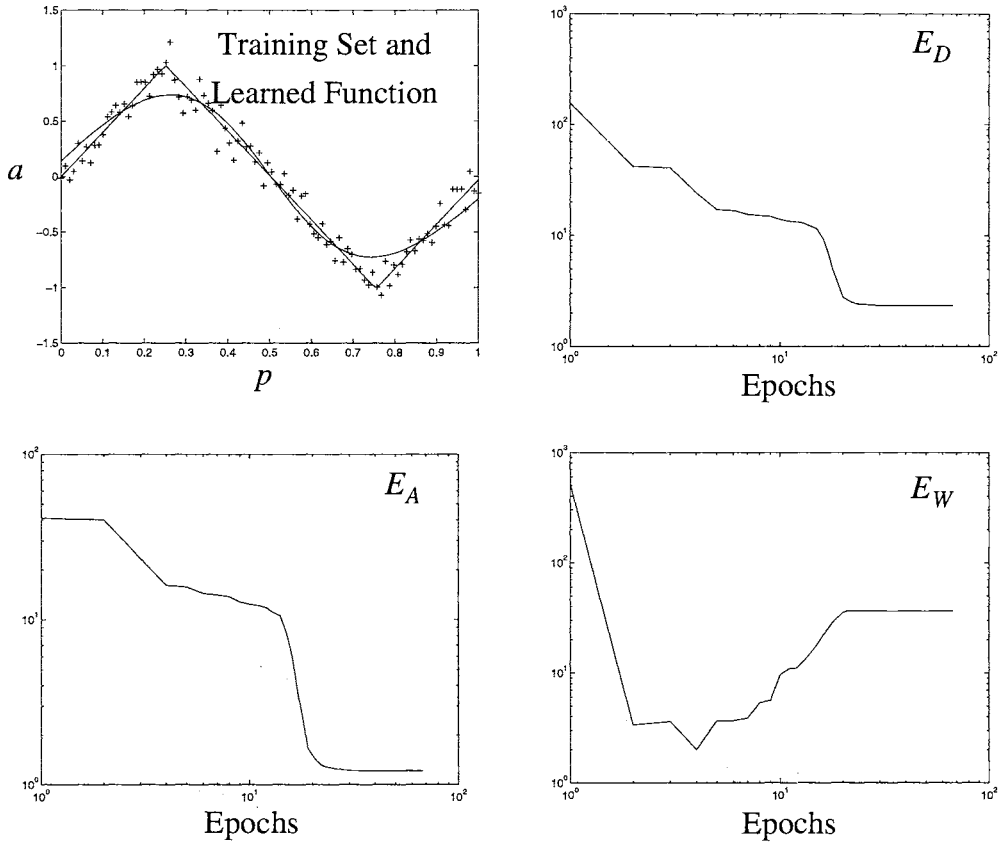


Figure 18 Underfitting Example for $\alpha = 10$

Effective Number of Parameters

A neural network has weights and biases which can be adjusted, much as coefficients of power terms are adjusted in a regression model. (See "Regression Comparison" on page 25.) Without pruning or growing, each neural network has a known total number of parameters available to it, as determined by the architecture.

Without the regularization term in Eq. (45), a neural network will use all of its parameters to reduce the training set error during learning. With the addition of the regularization term, however, weights found to be redundant will be constrained, thus reducing their usefulness. Here, the learning algorithm tries to find a balance between minimizing the training set error and lowering weight values.

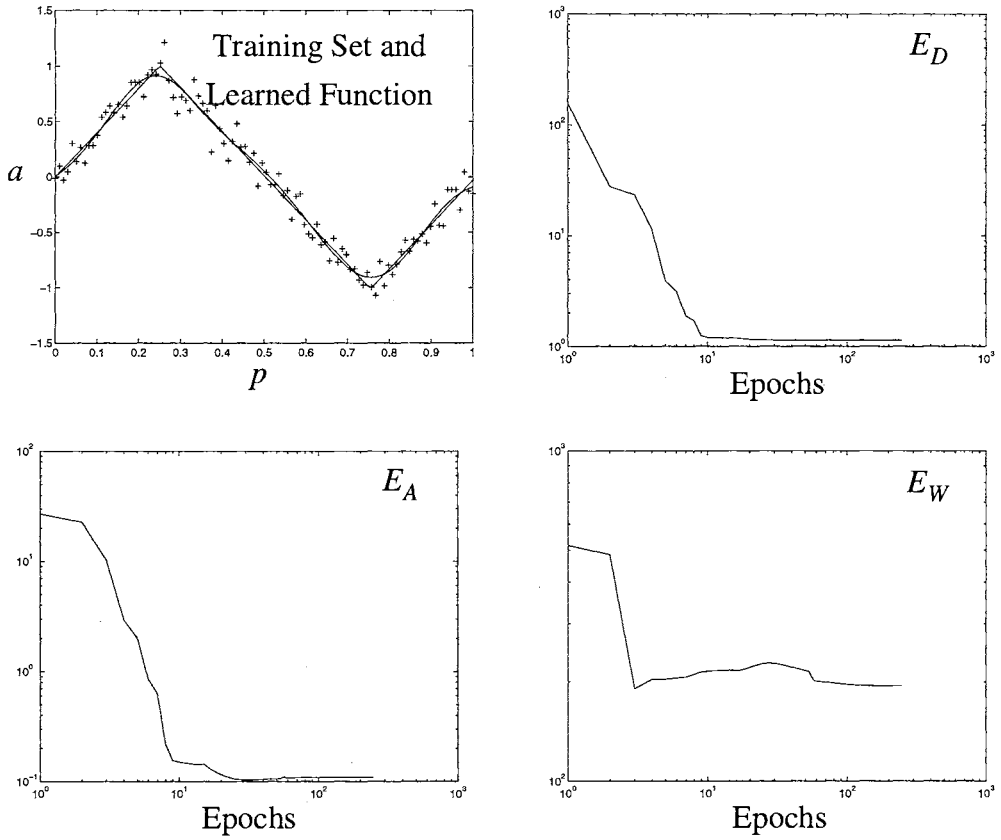


Figure 19 Data Fitting Example for $\alpha = 0.01$

The regularization term, in effect, places a constraint on the weights. Thus, each weight contributes to both the training set error E_D and the regularization term E_W. If a particular weight is comparatively very large, the contribution to the overall objective function is probably much stronger in the regularization term than in the training set error term; i.e., a large valued weight probably increases E_W to a value beyond the benefit it provides to lowering E_D. This weight is said to not be very effective at reducing the training set error, since its effect on E_D is excessively counteracted by the increase in E_W. However, if a weight is comparatively very small, then it may contribute much more to reducing the

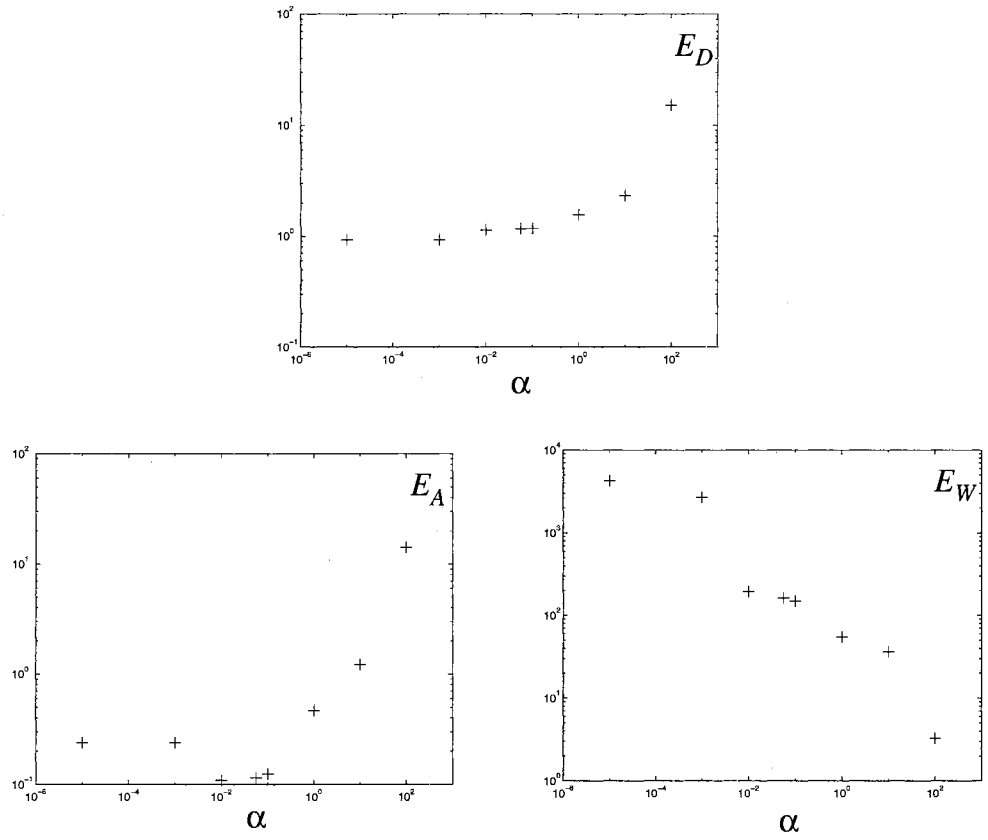


Figure 20 Final Results for Various Values of α

training set error than its value has caused the regularization term to increase. This weight is said to be utilized very effectively in learning the function.

From these ideas, Moody introduces the concept of the effective number of parameters. The effective number of parameters is a measure of the number of weights significantly contributing to the reduction of the training set error E_D. During training, the effective number of parameters will begin as some random value dictated by weight initialization. Then, it will change with every step. How it changes depends on the contribution to the total objective function F by the weights. However, the final value is strongly dependent on α. [Mood92]

Moody introduces his equation for calculating the effective number of parameters very abruptly in [Mood92]. He references a derivation to be found in a yet to be published follow-up paper. Although this paper is still unpublished we will introduce his equation anyway, however without derivation or explanation of the underlying ideas.

Since Moody uses $F = E_D + \alpha E_W$ where $E_D = \frac{1}{2}\sum_{i=1}^{n} e_i^2$ and $\mathrm{H} = \nabla^2 F$, then

$$\begin{aligned} \gamma_{\mathrm{mo}} &= \mathrm{tr}(\mathbf{J}\mathbf{H}^{-1}\mathbf{J}^T) \\ &= \mathrm{tr}((\mathbf{J}^T\mathbf{J})\mathbf{H}^{-1}) \\ &= \mathrm{tr}((\nabla^2 E_D)(\nabla^2 F)^{-1}) \end{aligned} \tag{52}$$

which is the trace of the product of the Hessian of the part of F responsible for the errors times the inverse of the Hessian for the whole objective function F.

Now, for our objective function of Eq. (45) $F = E_D + \alpha E_W$, where

$E_D = \sum_{i=1}^{n} e_i^2$, we have from Eq. (16) that $\nabla^2 E_D \approx 2\mathbf{J}^T\mathbf{J}$. Also, from Eq. (50) we have

$\mathbf{H} = \nabla^2 F \approx 2\mathbf{J}^T\mathbf{J} + 2\alpha\mathbf{I}$. Therefore, for our objective function Moody's effective number of parameters is

$$\begin{aligned} \gamma_{\mathrm{mo}} &= \mathrm{tr}((\nabla^2 E_D)(\nabla^2 F)^{-1}) \\ &\approx \mathrm{tr}((2\mathbf{J}^T\mathbf{J})(2\mathbf{J}^T\mathbf{J} + 2\alpha\mathbf{I})^{-1}) \\ &\approx \mathrm{tr}(\mathbf{J}(\mathbf{J}^T\mathbf{J} + \alpha\mathbf{I})^{-1}\mathbf{J}^T) \end{aligned} \tag{53}$$

This is how we calculate the effective number of parameters in our training algorithm.

Moody claims that $\gamma_{mo} \rightarrow N$, the total number of weights in the neural network, as $\alpha \rightarrow 0$. This reinforces the idea of no regularization will allow a neural network to overfit using all of the parameters, and a very large regularization parameter will cause the network to underfit the data by driving all weights to zero regardless of the training set error. Looking at Figure 21, note that large α does indeed limit the effectiveness of the neural network weights in minimizing the objective function, while very small α allows all 19 of the weights to be fully utilized in our 1-6-1 neural network. Compare this plot with those of Figure 20 to confirm the effect α has on the weights.

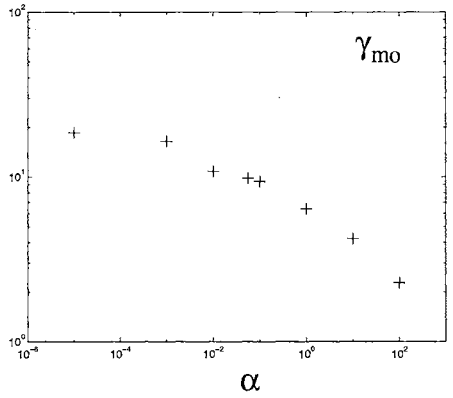


Figure 21 Effective Number of Parameters for the Saw Function

Summary

In this chapter, we adapted our training algorithm to include regularization. The results of our preliminary experiments showed that adding a regularization term to the objective function could temper the number of significant parameters utilized by the neural network and, therefore, greatly affect the final number of effective parameters, allowing better generalization. We also experimented with a way to calculate the effective number

of parameters used by a neural network in performing the mapping function described by the training set.

Problems entailed in this implementation include the required calculation of the inverse of the Hessian. Not only is this computationally expensive, but for overparameterized problems the Hessian may be singular or near singular.

Also, recall that the fundamental assumption of regularization is that the actual desired mapping function has a certain degree of smoothness. Although this is likely, given the typical presence of noise in measured data, still there is the question of how much smoothness. Without foresight, the neural network must be retrained many times changing the regularization factor α in an attempt to find the best compromise of training set error and smoothness of function.

In the next chapter, we will explore a method of automatically choosing the optimal regularization factor α. We will use Bayesian ideas to optimize regularization dynamically during training.

CHAPTER 6

BAYESIAN LEARNING AND THE GNBR ALGORITHM

Introduction

In Chapter 5, we saw the dramatic effect regularization has on neural network training results. The regularization term is added to the squared error term of the objective function in an attempt to reduce final weight sizes. Small weights tend to be conducive to smoother functions which generalize better.

The main problem of regularization is knowing how much emphasis to place on regularization, as opposed to error reduction; i.e., the size of the regularization parameter. Clearly there is a trade-off. An oversized regularization parameter leads to underfitting the data, resulting in inordinately high errors in the training set. In contrast, an undersized parameter results in overfitting the data, where the training set errors may be very small but generalization is poor.

In the previous chapter, we have viewed $\mathbf{w}$ as a deterministic parameter. We chose the value of $\mathbf{w}$ that minimized the error function E_D. Once given the training set data D, we used a gradient descent type method to minimize E_D. The training performed an iterative search for the single most likely parameter value $\mathbf{w}^{ML}$ that could have produced the data D. This approach is commonly referred to as the maximum likelihood method.

Now, we will explore a different approach. We will assume our parameter $\mathbf{w}$ is a random variable. A random variable is represented by a probability density function. We can assign such a probability density function to our random variable $\mathbf{w}$. This probability density function will describe our preconceived ideas about the values of $\mathbf{w}$ before having ever seen the training set D. This is called the prior probability density. We can also express our likelihood function based on E_D as a conditional probability density function for $\mathbf{w}$, given the training set D. Combining these density functions properly, using Bayes' theorem, will produce the posterior distribution. While the prior distribution is typically very broad, indicating our uncertainty of $\mathbf{w}$, the posterior distribution is very narrow. The peak value of the posterior density function is the most probable value $\mathbf{w}^{MP}$. This approach is called Bayesian inference.

In this chapter, we will express our neural network training in Bayesian form. This will lead to an interesting comparison to the regularization techniques of Chapter 5. Using a second level of Bayesian inference, we will try to find the most probable regularization factor thus eliminating the guesswork of the previous chapter. We will then adapt our training algorithm to this approach and run simulations for comparison to previous work. This will be followed by an in depth analysis of the effective number of parameters as seen from two different viewpoints. Finally, a relationship between the regularization weight parameters and the unregularized ones will be derived for analysis.

The Bayesian application to neural networks found in this chapter is based on material found in [MacK95], [MacK92a], [MacK92b], [MacK92c] and [Bish95]. They are

also responsible for the qualitative comparison of $\mathbf{w}^{ML}$ and $\mathbf{w}^{MP}$. The second point of view of the theoretical derivation of the effective number of parameters and the mathematical comparison of $\mathbf{w}^{ML}$ and $\mathbf{w}^{MP}$ is taken from [SjLj92], [LjSj92] and [SjLj94].

We will express the pertinent equations in our mathematical framework for comparative purposes. Then they will be adapted for use in our algorithm. Our demonstrations will show how updating the regularization parameter between training steps will result in the best amount of regularization found experimentally in Chapter 5.

We will also explain how the effective number of parameters calculation works and in particular why the value of any given weight is never actually driven to zero by regularization. We will also explain the relationship between $\mathbf{w}^{ML}$ and $\mathbf{w}^{MP}$ and therefore what actually happens when applying regularization. Then we will adapt into our framework a third theory that is aimed explicitly at deriving γ and mathematically compare it to the others.

Background

In a Bayesian framework, neural network learning is interpreted in a probabilistic fashion. Consider the following application of Bayes' theorem:

$$P(\mathbf{w}|D) = \frac{P(D|\mathbf{w})P(\mathbf{w})}{P(D)}. \tag{54}$$

Without the training set D, the probability distribution over the weights $\mathbf{w}$ is $P(\mathbf{w})$, called the prior. With the observed training set data D, a set of weights $\mathbf{w}$ can be evaluated by an error function, say E_D, to give the probability that the neural network with weights $\mathbf{w}$

produced the training set data D. This is the likelihood function which we maximize to train the neural network (minimizing the squared error). On the left side of Eq. (54) is $P(\mathbf{w}|D)$. This is called the posterior probability and describes the probability that a particular set of weights $\mathbf{w}$ could have produced the training set data D given our prior knowledge of the weights. The denominator of Eq. (54) $P(D)$ is a normalizing factor that ensures the posterior probabilities sum to one.

Using common names, Eq. (54) looks like

$$\text{Posterior} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Normalization Factor}}. \tag{55}$$

The neural network interpretation of Eq. (54) is that we want to maximize the probability of $\mathbf{w}$ given D, $P(\mathbf{w}|D)$, by maximizing the likelihood function $P(D|\mathbf{w})$ during training while using prior knowledge about the weights $P(\mathbf{w})$. But, given a particular neural network model, we would generally have no known preferences for a particular set of weights $\mathbf{w}$ for that model. Thus, with no reasonable prior knowledge, we simply set all prior probabilities equal. Since $P(D)$ is just a normalization factor, we conclude that to find the best choice of weights in the presence of data $P(\mathbf{w}|D)$ we must maximize the likelihood $P(D|\mathbf{w})$. This is unregularized training.

Figure 22 is a simplified depiction of learning. The prior $P(\mathbf{w})$ is flat indicating preference given to a broad set of weights $\mathbf{w}$. Since it is a probability distribution over a broad spectrum, the uniform height is small. Once the training set data has been used for training by finding the likelihood $P(D|\mathbf{w})$, the prior can be converted into a posterior dis-

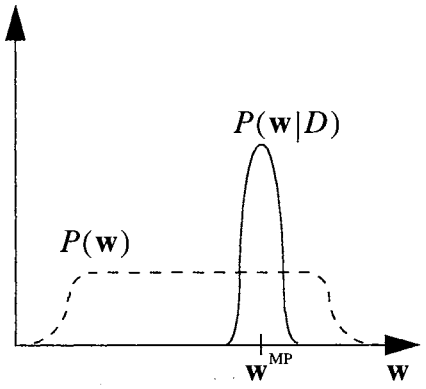


Figure 22 Prior and Posterior of Bayesian Learning

tribution $P(\mathbf{w}|D)$ using Eq. (54). Note that the posterior distribution shows preference to a strongly narrowed spectrum of weights. And, since the area under it must sum to 1, as with any probability distribution, it must have a much higher peak than the prior. The higher posterior peak with narrower base are indications of having learned something about the correlation between the training set data and the weight spectrum. The highest point represents the most probable weights $\mathbf{w}^{MP}$.

Before we describe the actual function used in the evaluation of Eq. (54), recall our objective function that includes regularization is

$$F = \beta E_D + \alpha E_W. \tag{56}$$

Note that we have added a factor β to the error term E_D. In the previous chapter, we realized that regularization could be very helpful in achieving good generalization if we could only optimize the regularization factor α. In this application of Bayes' theorem, it is convenient to simultaneously optimize an error factor β.

Now, rewriting Eq. (54) with Eq. (56) in mind, the new form of our neural network learning function is shown in Eq. (57) where M is a particular neural network model.

$$P(\mathbf{w}|D, \alpha, \beta, M) = \frac{P(D|\mathbf{w}, \beta, M)P(\mathbf{w}|\alpha, M)}{P(D|\alpha, \beta, M)} \tag{57}$$

Note that the prior density describes our knowledge of what $\mathbf{w}$ should be without having seen the training set D. So, the prior density function $P(\mathbf{w}|\alpha, M)$ is not dependent on β. Also, our likelihood function is based on minimizing the errors in E_D. It has nothing to do with E_W, so the likelihood $P(D|\mathbf{w}, \beta, M)$ is not dependent on α. The posterior density function, however, is dependent on α and β by the defined relationship of Eq. (57) as they apply to our neural network learning problem.

First, consider the prior distribution. From Eq. (56), the size of the weights depends on the regularization factor α and the number of weights which defines a model M. If we consider the weight distribution to have a Gaussian form, it would look like

$$P(\mathbf{w}|\alpha, M) = \frac{1}{Z_W(\alpha)}\exp(-\alpha E_W) = \frac{1}{Z_W(\alpha)}\exp\left(-\frac{1}{1/\alpha}\sum_{i=1}^{N} w_i^2\right). \tag{58}$$

where $Z_W(\alpha) = \int \exp(-\alpha E_W) d\mathbf{w}$ which normalizes the distribution. A multivariate Gaussian distribution is written as

$$P(w_1, \ldots w_N) = \frac{1}{(2\pi\sigma_W^2)^{N/2}}\exp\left(-\frac{1}{2\sigma_W^2}\sum_{i=1}^{N} w_i^2\right) \tag{59}$$

where w_i are zero mean independent random variables with variance σ^2.

Now, we can see that $Z_W(\alpha)$ simplifies to $\left(\frac{\pi}{\alpha}\right)^{\frac{N}{2}}$ since $\sigma_W^2 = \frac{1}{2\alpha}$. Examining the operation of Eq. (58), we see that if the weights $\mathbf{w}$ are large, then $E_W = \sum_{i=1}^{N} w_i^2$ is large, and our prior probability $P(\mathbf{w}|\alpha, M)$ is small. Thus, the highest probability for $P(\mathbf{w}|\alpha, M)$ is the $\mathbf{w}$ which has the smallest weight values. So, Eq. (58) prefers smaller weights, just as we wanted for good generalization, since we assume the true function to have some degree of smoothness.

Next, consider the likelihood $P(D|\mathbf{w}, \beta, M)$. We have always assumed that the training set measurements contained noise. Recall Eq. (25) from Chapter 4, $a = f(p, \mathbf{w}) + \varepsilon$. If we assume the noise ε to be Gaussian with zero mean, then we can write the likelihood probability as

$$P(D|\mathbf{w}, \beta, M) = \frac{1}{Z_D(\beta)}\exp(-\beta E_D) = \frac{1}{Z_D(\beta)}\exp\left(-\frac{1}{(1/\beta)}\sum_{i=1}^{n} e_i^2\right) \tag{60}$$

where $Z_D(\beta) = \int \exp(-\beta E_D) d\mathbf{e}$ is the normalizing constant over the n-dimensional training set data space. Similar to Eq. (59), if we assume all training set data points to be measured independently, then with zero mean independent Gaussian noise we have for the training set errors of E_D

$$P(e_1, \ldots e_N) = \frac{1}{(2\pi\sigma_D^2)^{n/2}}\exp\left(-\frac{1}{2\sigma_D^2}\sum_{i=1}^{n} e_i^2\right). \tag{61}$$

Thus with a noise level of $\sigma_D^2 = \frac{1}{2\beta}$ we see that $Z_D(\beta)$ simplifies to $\left(\frac{\pi}{\beta}\right)^{\frac{n}{2}}$. Just as in the regularization term, the likelihood probability distribution prefers smaller errors which is what we minimize during training.

Before going further, let's examine what we have so far. Since $P(D|\alpha, \beta, M)$ is just a normalizing constant, we can rewrite Eq. (57) as

$$P(\mathbf{w}|D, \alpha, \beta, M) \propto P(D|\mathbf{w}, \beta, M)P(\mathbf{w}|\alpha, M). \tag{62}$$

Evaluating the right hand side using Eq. (58) and Eq. (60) we have

$$\begin{aligned} P(D|\mathbf{w}, \beta, M)P(\mathbf{w}|\alpha, M) &= \left[\frac{1}{Z_W(\alpha)}\exp(-\alpha E_W)\right]\left[\frac{1}{Z_D(\beta)}\exp(-\beta E_D)\right] \\ &= \frac{1}{Z_W(\alpha)Z_D(\beta)}\exp(-\beta E_D - \alpha E_W) \\ &= \frac{1}{Z_W(\alpha)Z_D(\beta)}\exp(-F) \end{aligned} \tag{63}$$

Recalling that $Z_W(\alpha)$ and $Z_D(\beta)$ are constants, we can now see that maximizing the posterior density function $P(\mathbf{w}|D, \alpha, \beta, M)$ is accomplishing the same thing as minimizing our regularized objective function F of Eq. (56)!

We should note that we assumed prior knowledge of $\mathbf{w}$, since Eq. (58) gives preference to smaller weight values. If we assumed no prior knowledge of the weights, the prior density function $P(\mathbf{w}|\alpha, M)$ would simply be a constant over the entire weight space. This is interpreted as σ_W^2 being very large and from $\sigma_W^2 = \frac{1}{2\alpha}$ we must have a very small

regularization factor α. Thus, as we would expect, no prior knowledge of $\mathbf{w}$ represents unregularized training.

Now we have suitable representations for the probability density functions of Eq. (57). From Eq. (63), maximizing Eq. (57) accomplishes the same function as minimizing our objective function F. Thus, for a chosen α, we know the prior density function $P(\mathbf{w}|\alpha, M)$. Also, for a particular β we can use our training algorithm to find the likelihood value $\mathbf{w}^{ML}$ that minimizes E_D thus maximizing the likelihood $P(D|\mathbf{w}, \beta, M)$. Since α and β are fixed constants, we can negate the normalizing constant $P(D|\alpha, \beta, M)$ of Eq. (57) and use Eq. (62) to compare the effects of different regularization factors. This is what we did in Chapter 5.

Now we look at optimizing α and β. Applying Bayes' theorem to the task, we get

$$P(\alpha, \beta|D, M) = \frac{P(D|\alpha, \beta, M)P(\alpha, \beta|M)}{P(D|M)}. \tag{64}$$

Our aim is to maximize the posterior $P(\alpha, \beta|D, M)$ producing the most probable values α^{MP} and β^{MP}. Using the same ideas as before, we note that $P(D|M)$ is simply a normalizing constant and that without prior knowledge of α and β we assign all possibilities of $P(\alpha, \beta|M)$ to equal values. Thus, to optimize α and β we must maximize the likelihood $P(D|\alpha, \beta, M)$. Take note that this is the normalizing factor of Eq. (57)! This factor is called the evidence for α and β.

Here is the key point. If we could estimate the posterior of Eq. (57), we could use Eq. (58) and Eq. (60) to estimate the evidence $P(D|\alpha, \beta, M)$ giving us α^{MP} and β^{MP}. Indeed, we can estimate $P(\mathbf{w}|D, \alpha, \beta, M)$ with some accuracy if we are near a minimum of our objective function F. If we are sufficiently close to a minimum of F, we can approximate the surface of the objective function as a quadratic. With this in mind, we perform a Taylor series expansion of $P(\mathbf{w}|D, \alpha, \beta, M)$.

First, we will need the Taylor series expansion of $F(\mathbf{w})$ around $\mathbf{w}^{MP}$

$$F(\mathbf{w}) \approx F(\mathbf{w}^{MP}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{MP})^T \mathbf{H}^{MP} (\mathbf{w} - \mathbf{w}^{MP}) \tag{65}$$

where $\mathbf{H}^{MP} = \nabla^2 F(\mathbf{w})\big|_{\mathbf{w} = \mathbf{w}^{MP}}$ (MP stands for most probable) and $\nabla F(\mathbf{w}) = 0$ for $\mathbf{w} = \mathbf{w}^{MP}$ since $\mathbf{w}^{MP}$ represents the set of weights at a local minimum of F.

Since all parts of Eq. (57) are Gaussian, we expand $P(\mathbf{w}|D, \alpha, \beta, M)$ as a Gaussian distribution.

$$\begin{aligned} P(\mathbf{w}|D, \alpha, \beta, M) &= \frac{1}{Z_F} \exp(-F(\mathbf{w})) \\ &\approx \frac{1}{Z_F} \exp\left(-F(\mathbf{w}^{MP}) - \frac{1}{2}(\mathbf{w} - \mathbf{w}^{MP})^T \mathbf{H}^{MP} (\mathbf{w} - \mathbf{w}^{MP})\right) \\ &\approx \left(\frac{1}{Z_F} \exp(-F(\mathbf{w}^{MP}))\right) \exp\left(-\frac{1}{2}(\mathbf{w} - \mathbf{w}^{MP})^T \mathbf{H}^{MP} (\mathbf{w} - \mathbf{w}^{MP})\right) \end{aligned} \tag{66}$$

With the multivariate Gaussian form of

$$P(\mathbf{w}) = \frac{1}{\sqrt{(2\pi)^N \left|(\mathbf{H}^{MP})^{-1}\right|}} \exp\left(-\frac{1}{2}(\mathbf{w} - \mathbf{w}^{MP})^T \mathbf{H}^{MP} (\mathbf{w} - \mathbf{w}^{MP})\right) \tag{67}$$

we have that $Z_F = (2\pi)^{N/2}(\det((\mathbf{H}^{\mathrm{MP}})^{-1}))^{1/2}\exp(-F(\mathbf{w}^{\mathrm{MP}}))$.

Now, plugging Eq. (58), Eq. (60) and Eq. (66) into Eq. (57), we have

$$
\begin{aligned}
P(\mathbf{w}|D, \alpha, \beta, M) &= \frac{P(D|\mathbf{w}, \beta, M)P(\mathbf{w}|\alpha, M)}{P(D|\alpha, \beta, M)} \\
P(D|\alpha, \beta, M) &= \frac{\left[\frac{1}{Z_D(\beta)}\exp(-\beta E_D)\right]\left[\frac{1}{Z_W(\alpha)}\exp(-\alpha E_W)\right]}{\frac{1}{Z_F}\exp(-F(\mathbf{w}))} \\
&= \frac{Z_F}{Z_D(\beta)Z_W(\alpha)} \cdot \frac{\exp(-\beta E_D - \alpha E_W)}{\exp(-F(\mathbf{w}))} \\
&= \frac{Z_F}{Z_D(\beta)Z_W(\alpha)}
\end{aligned}
\tag{68}
$$

If we take the log of Eq. (68) and collect terms, we have

$$
\begin{aligned}
\log P(D|\alpha, \beta, M) &= \log(Z_F) - \log(Z_D(\beta)) - \log(Z_W(\alpha)) \\
&= \frac{N}{2}\log(2\pi) - \frac{1}{2}\log\det(\mathbf{H}^{\mathrm{MP}}) - F(\mathbf{w}^{\mathrm{MP}}) - \frac{n}{2}\log\left(\frac{\pi}{\beta}\right) - \frac{N}{2}\log\left(\frac{\pi}{\alpha}\right) \\
&= -F(\mathbf{w}^{\mathrm{MP}}) - \frac{1}{2}\log\det(\mathbf{H}^{\mathrm{MP}}) + \frac{n}{2}\log(\beta) + \frac{N}{2}\log(\alpha) + \frac{N}{2}\log 2 - \frac{n}{2}\log\pi
\end{aligned}
\tag{69}
$$

Recall from Eq. (64), to maximize $P(\alpha, \beta|D, M)$ we must find the maximum of $P(D|\alpha, \beta, M)$. We need only to take the derivatives of Eq. (69) with respect to α and β, set the derivatives to zero, and solve for α^{MP} and β^{MP}.

First consider what happens in both cases to the second term of Eq. (69). Since $\mathbf{H}$ is the Hessian of F in Eq. (56), we can separate it as

$\mathbf{H} = \nabla^2 F = \nabla^2(\beta E_D) + \nabla^2(\alpha E_W) = \beta\mathbf{B} + 2\alpha\mathbf{I}$ where $\mathbf{B} = \nabla^2 E_D$ and $\mathbf{I}$ is the N dimensional identity matrix. If we let λ^h be an eigenvalue of $\mathbf{H}$ and λ^b be an eigenvalue of

$\beta\mathbf{B}$, then $\lambda^h = \lambda^b + 2\alpha$ for all corresponding eigenvalues. Now we take the derivative of Eq. (69) with respect to α. Since the determinant of a matrix can be expressed as the product of its eigenvalues, we can reduce it as shown in Eq. (70) where $\mathrm{tr}(\mathbf{H}^{-1})$ is the trace of the inverse of the Hessian $\mathbf{H}$.

$$\begin{aligned}
\frac{\partial}{\partial\alpha}\frac{1}{2}\log\det\mathbf{H} &= \frac{1}{2\det\mathbf{H}}\frac{\partial}{\partial\alpha}\left(\prod_{k=1}^{N}\lambda^h\right) \\
&= \frac{1}{2\det\mathbf{H}}\frac{\partial}{\partial\alpha}\left(\prod_{i=1}^{N}(\lambda_i^b + 2\alpha)\right) \\
&= \frac{1}{2\det\mathbf{H}}\left[\sum_{i=1}^{N}\left(\prod_{j\neq i}(\lambda_j^b + 2\alpha)\right)\frac{\partial}{\partial\alpha}(\lambda_i^b + 2\alpha)\right] \\
&= \frac{\sum_{i=1}^{N}\left(\prod_{j\neq i}(\lambda_j^b + 2\alpha)\right)}{\prod_{i=1}^{N}(\lambda_i^b + 2\alpha)} \\
&= \sum_{i=1}^{N}\frac{1}{\lambda_i^b + 2\alpha} = \mathrm{tr}(\mathbf{H}^{-1})
\end{aligned} \tag{70}$$

Now, we define the parameter γ as in Eq. (71) and expand it for use in our next step. The parameter γ is referred to as the effective number of parameters. We will explore it in depth in a later section.

$$\gamma \equiv N - 2\alpha \mathrm{tr}(\mathbf{H}^{-1})$$
$$= N - 2\alpha \sum_{i=1}^{N} \frac{1}{\lambda_i^b + 2\alpha} = \sum_{i=1}^{N} \left(1 - \frac{2\alpha}{\lambda_i^b + 2\alpha}\right) = \sum_{i=1}^{N} \left(\frac{\lambda_i^b}{\lambda_i^b + 2\alpha}\right) = \sum_{i=1}^{N} \frac{\lambda_i^b}{\lambda_i^h} \tag{71}$$

Now for the derivative with respect to β.

$$\begin{aligned}
\frac{\partial}{\partial \beta} \frac{1}{2} \log \det \mathbf{H} &= \frac{1}{2 \det \mathbf{H}} \frac{\partial}{\partial \beta} \left(\prod_{k=1}^{N} \lambda_k^h \right) \\
&= \frac{1}{2 \det \mathbf{H}} \frac{\partial}{\partial \beta} \left(\prod_{i=1}^{N} (\lambda_i^b + 2\alpha) \right) \\
&= \frac{1}{2 \det \mathbf{H}} \left[\sum_{i=1}^{N} \left(\prod_{j \neq i} (\lambda_j^b + 2\alpha) \right) \frac{\partial}{\partial \beta} (\lambda_i^b + 2\alpha) \right] \\
&= \frac{\frac{1}{2} \sum_{i=1}^{N} \left(\prod_{j \neq i} (\lambda_j^b + 2\alpha) \left(\frac{\lambda_i^b}{\beta} \right) \right)}{\prod_{i=1}^{N} (\lambda_i^b + 2\alpha)} \\
&= \frac{1}{2\beta} \sum_{i=1}^{N} \frac{\lambda_i^b}{\lambda_i^b + 2\alpha} = \frac{\gamma}{2\beta}
\end{aligned} \tag{72}$$

where the fourth step is derived from the fact that λ_i^b is an eigenvalue of $\beta\mathbf{B}$ and therefore the derivative of λ_i^b with respect to β is just the eigenvalue of $\mathbf{B}$ which is λ_i^b / β.

Now we are finally ready to take the derivatives of Eq. (69) and set them equal to zero. The derivative of Eq. (69) with respect to α, using Eq. (70), will be

$$\begin{aligned}\frac{\partial}{\partial\alpha}\log P(D|\alpha,\beta,M) &= -\frac{\partial}{\partial\alpha}F(\mathbf{w}^{\mathrm{MP}}) - \frac{\partial}{\partial\alpha}\frac{1}{2}\log\det(\mathbf{H}^{\mathrm{MP}}) + \frac{\partial}{\partial\alpha}\frac{N}{2}\log\alpha \\ &= -\frac{\partial}{\partial\alpha}(\alpha E_W(\mathbf{w}^{\mathrm{MP}})) - \mathrm{tr}(\mathbf{H}^{\mathrm{MP}})^{-1} + \frac{N}{2\alpha^{\mathrm{MP}}} \\ &= -E_W(\mathbf{w}^{\mathrm{MP}}) - \mathrm{tr}(\mathbf{H}^{\mathrm{MP}})^{-1} + \frac{N}{2\alpha^{\mathrm{MP}}} = 0\end{aligned} \tag{73}$$

Rearranging terms, and with Eq. (71) we have

$$\begin{aligned}E_W(\mathbf{w}^{\mathrm{MP}}) &= \frac{N}{2\alpha^{\mathrm{MP}}} - \mathrm{tr}(\mathbf{H}^{\mathrm{MP}})^{-1} \\ 2\alpha^{\mathrm{MP}}E_W(\mathbf{w}^{\mathrm{MP}}) &= N - 2\alpha^{\mathrm{MP}}\mathrm{tr}(\mathbf{H}^{\mathrm{MP}})^{-1} = \gamma \\ \alpha^{\mathrm{MP}} &= \frac{\gamma}{2E_W(\mathbf{w}^{\mathrm{MP}})}\end{aligned} \tag{74}$$

Now for β and using Eq. (72),

$$\begin{aligned}\frac{\partial}{\partial\beta}\log P(D|\alpha,\beta,M) &= -\frac{\partial}{\partial\beta}F(\mathbf{w}^{\mathrm{MP}}) - \frac{\partial}{\partial\beta}\frac{1}{2}\log\det(\mathbf{H}^{\mathrm{MP}}) + \frac{\partial}{\partial\beta}\frac{n}{2}\log\beta \\ &= -\frac{\partial}{\partial\beta}(\beta E_D(\mathbf{w}^{\mathrm{MP}})) - \frac{\gamma}{2\beta^{\mathrm{MP}}} + \frac{n}{2\beta^{\mathrm{MP}}} \\ &= -E_D(\mathbf{w}^{\mathrm{MP}}) - \frac{\gamma}{2\beta^{\mathrm{MP}}} + \frac{n}{2\beta^{\mathrm{MP}}} = 0\end{aligned} \tag{75}$$

Rearranging terms,

$$\begin{aligned}E_D(\mathbf{w}^{\mathrm{MP}}) &= \frac{n}{2\beta^{\mathrm{MP}}} - \frac{\gamma}{2\beta^{\mathrm{MP}}} \\ \beta^{\mathrm{MP}} &= \frac{n-\gamma}{2E_D(\mathbf{w}^{\mathrm{MP}})}\end{aligned} \tag{76}$$

For a review of what we have done, we redefined our performance index in terms of Bayes' theorem, Eq. (57), assuming Gaussian models. Estimating the posterior of Eq. (57) near a minimum of our objective function enabled us to estimate the normalization fac-

tor $P(D|\alpha, \beta, M)$ which is the evidence for α and β shown in Eq. (64). Maximizing the evidence has given us a way to calculate optimal values for our factors α^{MP} and β^{MP}. Though we are not near a minimum at the onset of training, there is no reason not to use Eq. (74) and Eq. (76) to estimate our objective function's factors, since the estimations will become more accurate during the course of training.

Method of Application

Applying the Bayesian method of optimization to our objective function factors is straight forward. First, we modify our algorithm to minimize our new objective function shown in Eq. (56). For the initial step calculations, we must choose reasonable initial conditions for α and β. Since the initial weight settings are random, we have two choices. First, we may choose to maximize initial emphasis on E_D by initializing the parameters to $\alpha = 0$ and $\beta = 1$. Second, we may choose to calculate α and β in the normal way described below just as if a step had been taken. Though both methods of initialization work, we chose the second method for our trials. Now we can set the algorithm in motion and take a first step.

Next, we use the resulting $\mathbf{H} = \nabla^2 F(\mathbf{w}) \approx 2\beta \mathbf{J}^T \mathbf{J} + 2\alpha \mathbf{I}_N$ to calculate the current γ from Eq. (71). This allows us to adjust α and β for the next step using Eq. (74) and Eq. (76). The new α and β should be closer to optimal since our last step of the algorithm moved our weight vector $\mathbf{w}$ closer to a minimum of F. Note that we are using a Gauss-Newton approximation to the Hessian that is available to us in the Levenberg-Marquardt

algorithm. The resulting procedure will be called the Gauss-Newton approximation to Bayesian Regularization (GNBR).

Now we can take another training step being careful to realize we have changed our objective function by changing α and β. We continue the cycle until we are sufficiently close to a minimum of F to stop training. For our implementation, we chose to stop training only when the Levenberg-Marquardt learning parameter μ was increased to the maximum value allowed. At this point, machine accuracy limits the ability of the algorithm to lower the objective function F.

GNBR Trials

For the saw function of Chapter 5, we now test our GNBR algorithm. We will use the same training set of Chapter 5 and apply it to a 1-6-1 neural network. The resulting learned function is shown in Figure 23 along with the actual squared errors; i.e., the sum of the squares of errors using nonnoisy data. A comparison of these results with Figure 17, Figure 18 and Figure 19 of Chapter 5 show that the Bayesian training seems to automatically hone in on the best choice for regularization factor. The form of the function learned is most similar to that of Figure 19. It does not show the overfitting or underfitting aspects of the others, but instead seems to show a very good approximation of the actual saw function. The E_A plot in Figure 23 also shows a final value very close to that of Figure 19. Notice that it does not contain the upward movement on the tail end of training indicative of overfitting as shown in Figure 17.

To make further comparisons, we now reveal that the trials concerning the saw function of Chapter 5 were actually done using Eq. (56) instead of Eq. (45). The parameters

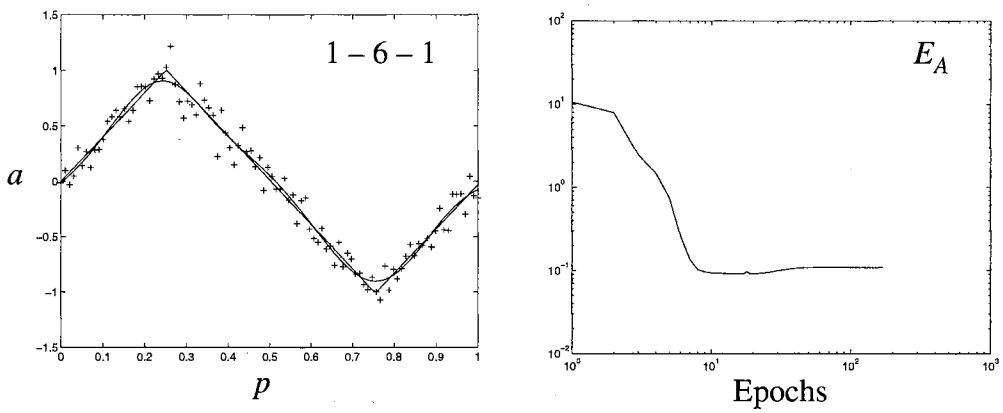


Figure 23 Final Function and Actual Squared Error Progression

were fixed during the trials with $\beta = 78.8354$. We chose this value since it is the final value found for β through Bayesian optimization. Note that this constant factor for E_D in no way compromises the trial results. If $\beta = 1$, then all the results of Chapter 5 would still be true but for correspondingly scaled values for α. Compare Eq. (56) with Eq. (77) below.

$$\frac{F}{\beta} = E_D + \frac{\alpha}{\beta} E_W \tag{77}$$

Changing β in our objective function changes the valuation of the function to $\frac{F}{\beta}$. The minimum of $\frac{F}{\beta}$ occurs at the same weight values as the minimum of F. Thus, the only effect changing β has on the function is that it scales the regularization factor to be $\frac{\alpha}{\beta}$. So if we fix β to be a constant other than one, the results of Chapter 5 will be the same but for correspondingly scaled regularization factors. With this in mind, we continue our comparisons.

The final actual squared error was very close to the lowest found experimentally in Figure 20. The variables E_D and E_W are plotted in Figure 24. Their final values were both in the mid range of those shown in Figure 20, hopefully indicating neither overfitting nor underfitting.

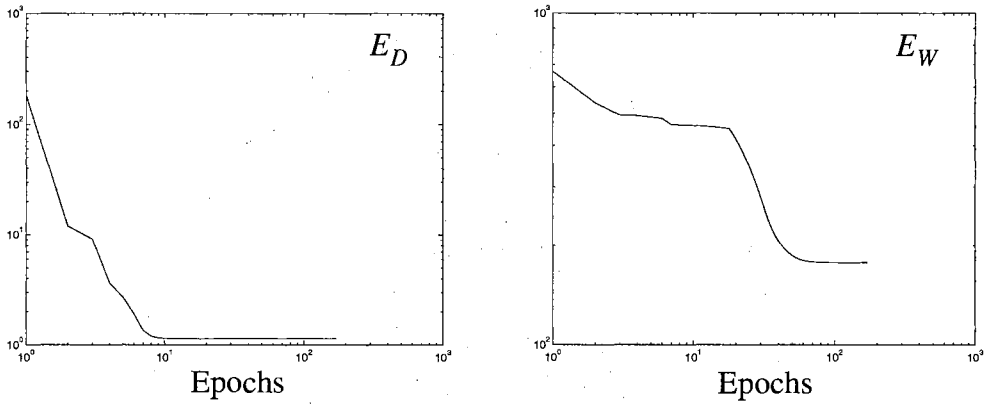


Figure 24 E_D and E_W for the Saw Function of the 1-6-1 Network

In Figure 25, we plot α and β during training. Note that the initial values of $\alpha = 0$ and $\beta = 1$ seemed to be immediately recovered from after the first step.

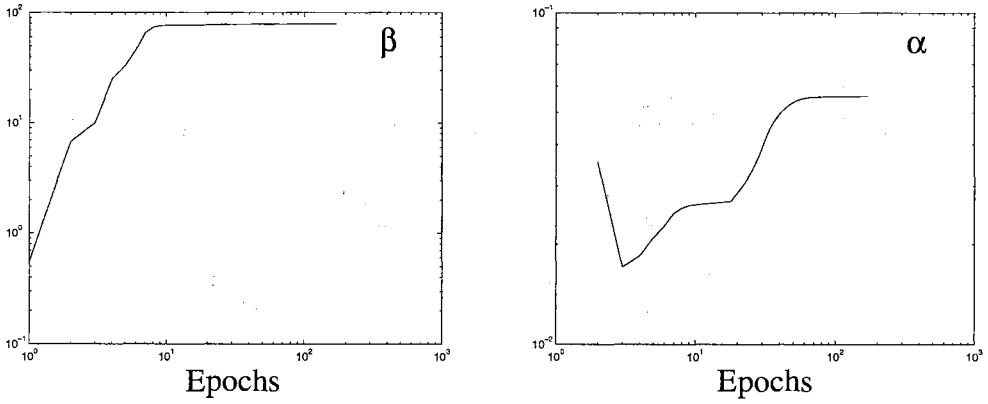


Figure 25 α and β for Saw Function of the 1-6-1 Network

The only problem encountered with Bayesian training using the Levenberg-Marquardt algorithm was how the training was stopped. While attempting to take the last step, μ is increased beyond a maximum value. From [HaDe96], μ is normally increased in the Levenberg-Marquardt algorithm to a point where the objective function is lowered, depicting a compromise between Gauss-Newton and steepest descent methods. With large μ, the algorithm takes a steepest descent step which in theory must yield a lower objective function.

However, α and β are updated with each epoch such that the typical minimum specified objective function may never be reached as a stopping criteria. In fact, plugging Eq. (74) and Eq. (76) into our objective function yields

$$\begin{aligned} F &= \beta E_D + \alpha E_W \\ &= \frac{1}{2}(n-\gamma) + \frac{1}{2}\gamma \\ &= \frac{1}{2}n \end{aligned} \tag{78}$$

Thus every adjustment of α and β resets F to half the number of training samples.

Now the natural stopping criteria becomes the point when increasing μ does not reduce the objective function. This is the case when the algorithm has been compromised by the numerical limitation of the machine; i.e., the require step size is smaller than the accuracy of the machine. Therefore, this is also the proper exit criteria for our regularization algorithm.

Effective Number of Parameters

In this section, we will concentrate on the meaning of Eq. (71). The eigenvalues of $\nabla^2 F$ and $\nabla^2 \beta E_D$ are key to these discussions. Once the concepts are explained, we will return to our experiments to see examples of the relationship between these eigenvalues and the effective number of parameters.

Looking again at Figure 25, which shows the progression of α and β during training, we note that although the final values of α and β were 0.0559 and 78.835, respectively, they were not constant during training. Also, note how β increased first, giving more preference to reducing E_D. And, comparing with Figure 24 we see that E_D drops, then becomes somewhat stable. Once E_D is stable, α begins to increase thus driving E_W down. Note that E_D is still stable. This is when the algorithm is attempting to reduce the effective number of parameters without affecting error minimization.

Recall that in Chapter 5 we introduced the concept of the effective number of parameters. Moody calculates the effective number of parameters using Eq. (52), whereas MacKay defines it as Eq. (71) restated here as Eq. (79). Recall that α is our regularization factor, N is the total number of weights, and $\mathbf{H}$ is the Hessian for $\nabla^2 F$ of Eq. (56).

$$\gamma_{\text{ma}} \equiv N - 2\alpha \ \text{tr}(\mathbf{H}^{-1}). \tag{79}$$

Moody showed γ_{ma} to be a linearization of his γ_{mo}, where discrepancies exist between them for very small values of α. However, we found no measurable discrepancy throughout our experiments. Perhaps for our examples, α is much too large to show a dif-

ference. For α small enough to show a measurable difference, the regularization would allow undesirable overfitting. In our trials, γ_{mo} and γ_{ma} were numerically identical in value. But, although both are always tracked in all of our experiments, we endorse using Eq. (79) for actual calculations since it minimizes the additional computations. The Levenberg-Marquardt algorithm is a second order method, thus the Hessian is already available to us for use.

Figure 26 shows γ_{mo} for the saw example. Note that the initial calculation for γ_{mo} is an incorrect estimate. This is the result of the start-up situation where Eq. (56) begins in a random situation given the initialization of the weights. Also, α has been forced to zero for the first step.

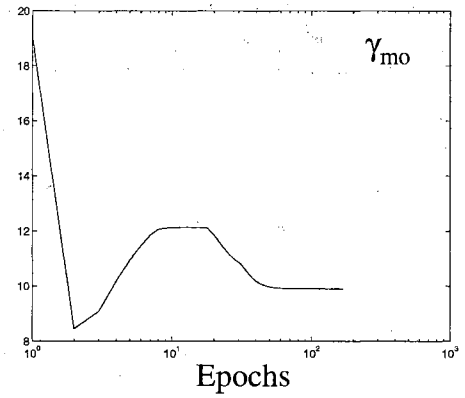


Figure 26 γ for the Saw Function for the 1-6-1 Network

Comparing Figure 26 with Figure 24 and Figure 25, we can see the possible overfitting occurring as β rises, allowing E_D reduction and therefore, an increase in γ_{mo}. However, once α rises sharply, E_W drops with a corresponding drop in γ_{mo}. This is hopefully reducing the possibility of overfitting.

If we began training with no eigenvalues of E_D significantly contributing to F, we would expect the initial value of γ_{mo} to be virtually zero and increase as training progresses. Some researchers believe this is important for proper neural network training. We will develop this concept further in Chapter 7. For our experiment, we took the same training set data and applied it to a 1-6-1 network. We used the same initial weights, however dividing them by 1000 to make them too small to initially contribute much to F.

The results are shown in Figure 27 and Figure 28. Comparing them with the results for our previous experiment in this chapter, shown in Figure 23 through Figure 26, we found the final results to be the same. The final function looked the same and the final values for E_A, E_D, E_W, α, β and γ_{mo} were all the identical. The E_W began very small and was allowed to grow to a final value under the control of α, which seemed to almost asymptotically drop on the log scale during training.

The effective number of parameters for this new experiment is shown in Figure 28. Aside from initialization, notice how γ_{mo} started very small and grew to the final value. This corresponds to the almost continual drop in E_D. This is an indication of weights growing to learn the function. If we look back at Eq. (71), we see the last relationship is the ordered sum of ratios of eigenvalues. Putting this into perspective, look at Eq. (56) again. We are comparing eigenvalue sizes of the Hessian of βE_D to those of the Hessian of the total objective function F. Each ratio is a measure of the contribution of the eigenvalue due to the error portion of the overall objective function.

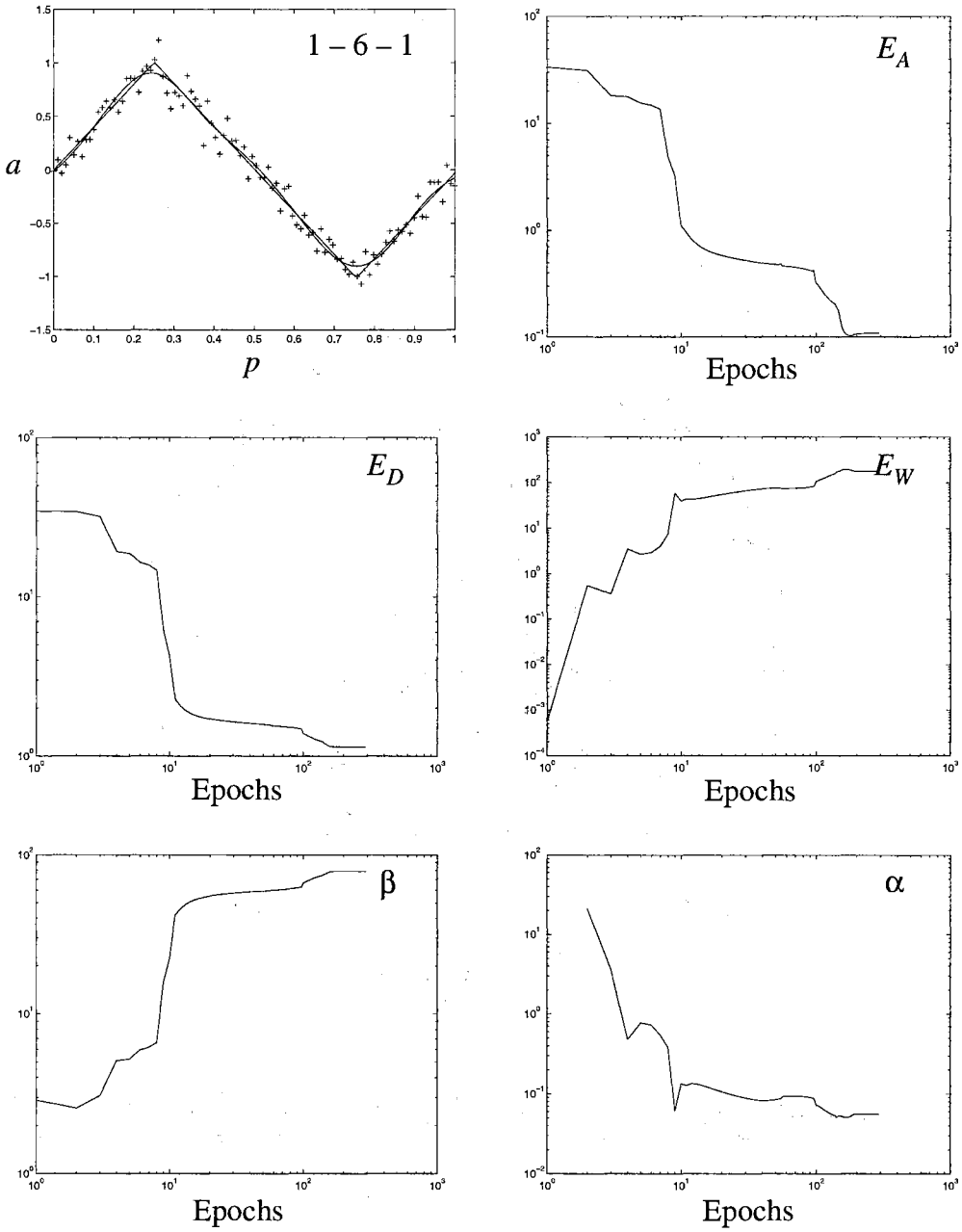


Figure 27 Results of Small Initial Weights for the 1-6-1 Network

Now let's look at the regularization process from a slightly different point of view, which will demonstrate how the effective number of parameters is reduced as regularization is added. First, recall Eq. (71):

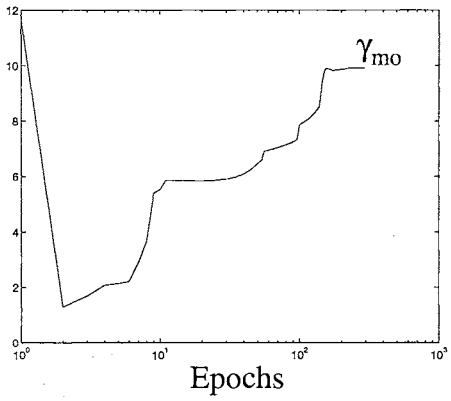


Figure 28 γ for Small Initial Weights for 1-6-1 Network

$$\gamma = \sum_{i=1}^{N} \left(\frac{\lambda_i^b}{\lambda_i^b + 2\alpha} \right) = \sum_{i=1}^{N} \gamma_i \tag{80}$$

We can see that $0 \le \gamma_i \le 1$. If $\alpha = 0$ (no regularization) then $\gamma_i = 1$ and $\gamma = N$, which means that all parameters are being fully used. If α is very large (overspecification of regularization) then $\gamma_i \approx 0$ and $\gamma \approx 0$, which means that effectively no parameters are being used.

Consider the two extreme cases. First, without data, our objective function is reduced to $F = \alpha E_W$ and minimizing F drives w_i toward zero. Since $\gamma = 2\alpha E_W$, then $\gamma \to 0$.

Recall that the prior density function for the weights is based on E_W which is a sum of squared weights. Since all of the terms have a coefficient of 1, and there are no cross-terms, the eigenvalues of $\nabla^2 E_W$ are positive and equal. A contour plot of the E_W surface would show circles indicating equal curvature in all directions centered at the origin. This is depicted as the circle in the simplified view of a 2 dimensional problem shown in Figure

29. The circle is centered at the origin since the minimum of E_W is the vector $\mathbf{0}$ where $w_i = 0$ for all i.

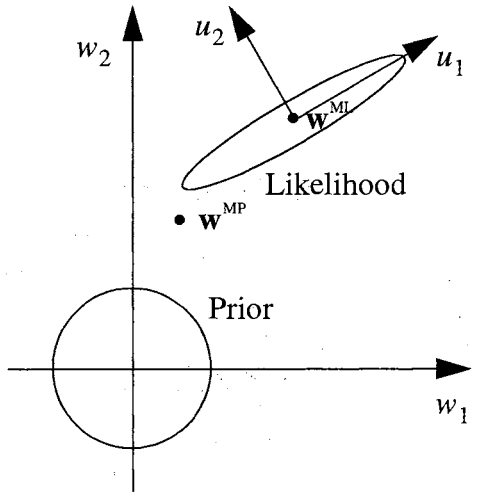


Figure 29 Simplified View of Eigenvector Relationships

Now, consider the other extreme. Without regularization, $F = \beta E_D$. At a minimum $\mathbf{w}^{ML}$ of E_D, all eigenvalues λ_i^b of $\nabla^2 \beta E_D$ are greater than or equal to zero. All weights are used to minimize E_D. Thus, $\gamma_i = 1$ for all weights and $\gamma = N$. Since not all λ_i^b are of the same size, we show an ellipse for the contour centered at $\mathbf{w}^{ML}$ in Figure 29.

Returning to the general regularized objective function, from the above discussion we note that eigenvalues λ_i^b of $\nabla^2 \beta E_D$ can be large or small while eigenvalues of $\nabla^2 \alpha E_W$ are equal in size. Consider again the relationship $\gamma_i = \dfrac{\lambda_i^b}{\lambda_i^b + 2\alpha}$. A large eigenvalue λ_i^b of $\nabla^2 \beta E_D$ has an associated eigenvector pointing in a direction of high curvature. Slight

movement of $\mathbf{w}^{ML}$ in this direction corresponds to a significant change in the value of E_D. Thus, a large eigenvalue plays an important role in reducing the error. Since the eigenvalues of $\nabla^2\alpha E_W$ are fixed in size, the large eigenvalue λ_i^b of $\nabla^2\beta E_D$ dominates them. This corresponds to $\lambda_i^b >> 2\alpha$ which implies $\gamma_i \rightarrow 1$ and the eigenvalue for the associated weight (or combination of weights) is shown to be an effective parameter. In Figure 29, eigenvalue λ_2 with associated eigenvector u_2 is a large eigenvalue and therefore contributes to the effective number of parameters with a value near unity.

In contrast, a small sized eigenvalue λ_i^b of $\nabla^2\beta E_D$ has an eigenvector pointing in a direction of only slight or no curvature. In this case, $\mathbf{w}^{ML}$ can be varied in this direction with minimal effect on minimizing the errors in E_D. Thus, the parameter (or combination of parameters) associated with this small eigenvalue is not a key player in reducing the errors. Now, $\lambda_i^b << 2\alpha$ and $\gamma_i \rightarrow 0$ and the responsible parameter is driven to smaller values through regularization. In Figure 29, the small eigenvalue λ_1 with associated eigenvector u_1 corresponds to such an ineffective parameter.

Our error term E_D is the squared error of a nonlinear function whose Hessian can be approximated as a quadratic near a minimum $\mathbf{w}^{ML}$. At such a minimum, the eigenvalues of $\nabla^2 E_D$ are positive but not necessarily equal, since a small change in one weight component w_i may not increase E_D as significantly as a corresponding change in another

weight component. In our 2-dimensional example shown in Figure 29, the ellipse centered at $\mathbf{w}^{ML}$ represents a single contour of E_D. Now the minimum of E_D is more sensitive to changes along u_2 than in changes along u_1. Thus, the eigenvalue of $\nabla^2 E_D$ with associated eigenvector pointing in the u_2 direction is much larger than the eigenvalue whose eigenvector is pointing in the u_1 direction since curvature of the contour of E_D is much stronger in the direction of u_2 than u_1.

Here is the real meaning of the effective number of parameters. If all the weights are important to realizing the true function, then they all have correspondingly large eigenvalues and their exact size is important, so α must be small enough to allow them to grow as needed. On the other hand, redundant parameters have eigenvalues smaller than α and are driven to insignificant sizes, since they do not contribute significantly to reducing the error term E_D in the objective function. Therefore, γ counts only the number of w_i whose values are strongly controlled by the data, rather than the prior probability.

So, weights (or combinations of weights) with a small eigenvalue show little dependence on data thus these weights tend to be driven to smaller values. In contrast, a weight (or combinations of weights) with a large eigenvalue contributes strongly to the reduction in E_D and its size is predominantly unaffected by the E_W term. Thus, in Figure 29, regularization producing the most probable weight vector $\mathbf{w}^{MP}$ is moved away from the error minimum $\mathbf{w}^{ML}$ and along u_1, due to its insensitivity in that direction indicated by the small

eigenvalue. The new minimum $\mathbf{w}^{MP}$ moves only slightly along the u_2 direction, since this direction corresponds to a large eigenvalue of $\nabla^2\beta E_D$, and therefore E_D must be very sensitive to movement in this direction.

In review, our Bayesian approach to learning has shown us how to optimize our objective function parameters, resulting in a most probable set of weights $\mathbf{w}^{MP}$ given the presence of the training set data. The relationship between $\mathbf{w}^{MP}$, which is the minimum of F, and $\mathbf{w}^{ML}$, which is the minimum of E_D alone, is graphically shown in a simplified view in Figure 29, where components of $\mathbf{w}^{ML}$ that only slightly contribute to the minimum of E_D are driven to smaller values in $\mathbf{w}^{MP}$.

Now we return to our original example in this chapter. Table 3 contains an example of eigenvalues from the saw function problem. The weights are not ordered by size, but the eigenvalues are. Note that for reasons discussed above, a particular eigenvalue may not correspond to a particular weight, but rather some combination of weights. In this example, $\gamma_{mo} = 9.9084$, which is the sum of the rightmost column values. Clearly, the eigenvalues much larger than $\alpha = 0.055915$ are the only effective parameters.

Looking at the weight values, it is not obvious which parameters are not effective. The first group of six are the final values of the weights between the input neuron and the six hidden layer neurons. The next 6 are hidden neuron biases. The last group of 6 are hidden layer output weights, and the final single value is the bias for the output neuron in our 1-6-1 architecture. With this in mind, note that the magnitudes of the first and sixth weight

Weight Value	λ_i^b	$\lambda_i^b + 2\alpha$	$\lambda_i^b/(\lambda_i^b + 2\alpha) = \gamma_i$
-0.661620	178108.302988	178108.414818	0.999999
7.112937	79025.772300	79025.884130	0.999999
-3.199016	8670.878694	8670.990524	0.999987
5.669189	5674.480460	5674.592290	0.999980
-4.204124	872.648132	872.759962	0.999872
0.661621	310.434534	310.546364	0.999640
0.156640	168.292386	168.404216	0.999336
-2.170455	5.565156	5.676986	0.980301
2.317223	1.272448	1.384278	0.919214
-4.850998	0.519392	0.631222	0.822836
1.064855	0.021364	0.133194	0.160398
-0.156640	0.002458	0.114288	0.021507
0.880564	0.000396	0.112226	0.003529
-1.874159	0.000206	0.112036	0.001839
3.045329	0.000000	0.111830	0.000000
2.388736	0.000000	0.111830	0.000000
-3.291639	0.000000	0.111830	0.000000
-0.880566	0.000000	0.111830	0.000000
-0.117509	0.000000	0.111830	0.000000

Table 3 Eigenvalues from the Saw Example

of each group of six are equal. These are the redundant, or ineffective, parameters. With all three pairs having opposite sign, they do not negate each others effect in the neural network. Rather, their effect is additive as if all pairs had the same sign. This means that the Bayesian training is somehow balancing the load instead of actually driving the redundant weights all the way to zero. This is a very intriguing observation that we will return to in a moment.

Reconsider for a moment the idea of Bayesian training. If the size of the neural network we are using is more complex than necessary, the Bayesian optimization of α and β

should drive into insignificance all extraneous parameters. Thus, it would seem that the resulting network should perform similarly for different numbers of hidden layer neurons.

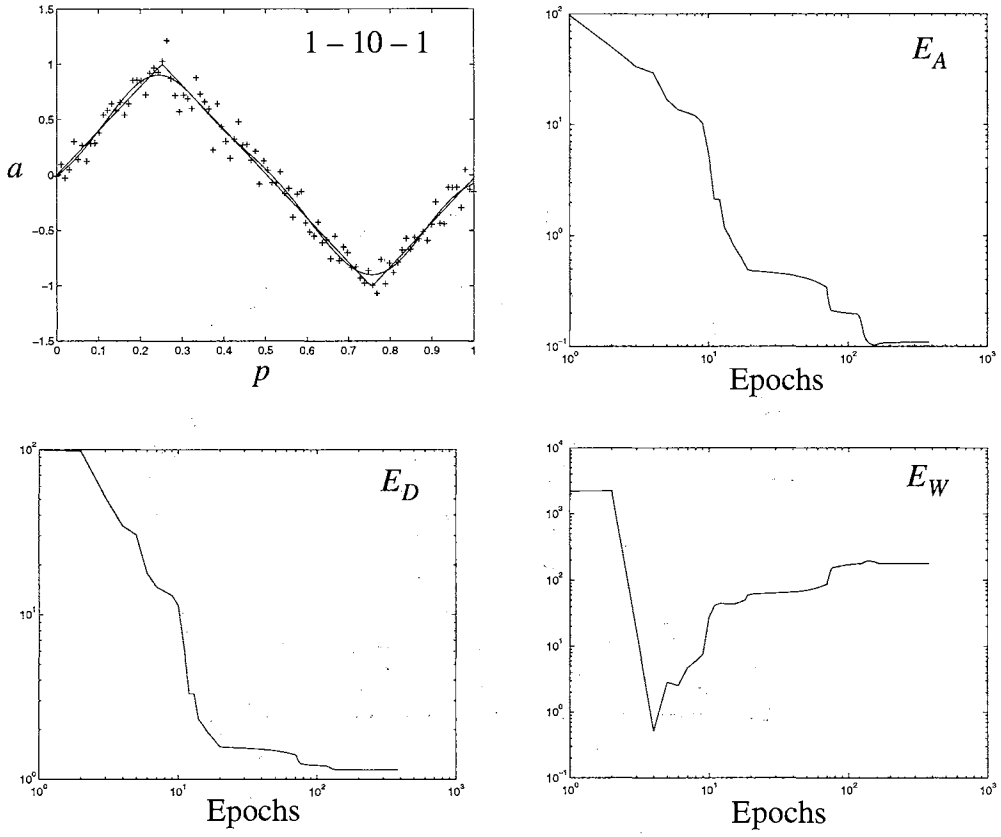


Figure 30 Results for the 1-10-1 Network

With this in mind, we tried training a 1-10-1 network using the same saw function training set. Figure 30 shows the resulting function approximation along with the E_D, E_W and E_A during training. Amazingly, the 1-10-1 network function approximation was nearly identical to the 1-6-1 result and the 1-10-1 network had almost exactly the same final values for E_D, E_W and E_A. In addition, the final value for effective number of parameters as shown in Figure 31 was virtually the same.

Table 4 contains a summary of results for other network sizes we tried. Note that for each size of network, the E_D, E_W, E_A and γ_{mo} all plateau once a sufficiently complex network is reached. Recall that unregularized training found the 1-6-1 network as optimal in actual squared error comparisons of Figure 14 in Chapter 4, although the 1-4-1 and 1-5-1 network architectures did almost as well. Here, we see that using optimal Bayesian regularization training allows us to use virtually any size neural network that has at least 4 hidden layer neurons. The training will automatically reduce the redundant parameters so as to not overfit the training set data.

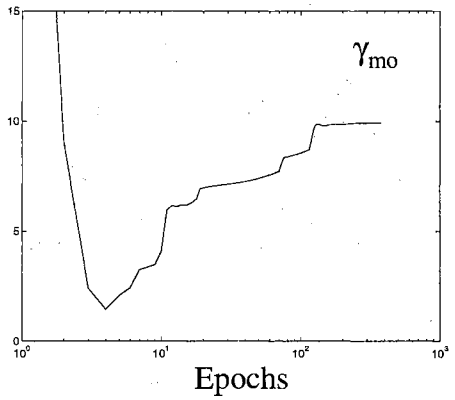


Figure 31 γ for the 1-10-1 Network

Now, let's return to our observation in the 1-6-1 network example, where two hidden layer neurons played identical additive roles. If we compare final weight values for different sizes of neural networks, we can make some very interesting observations.

Looking again at Table 3, we see that there are four completely dissimilar hidden layer neurons; i.e., they each have distinguishably different weights associated with them. With each hidden layer neuron having one input weight, one output weight and one bias, there are 12 distinctly unique weights in the 1-6-1 network. Comparing weight values with

S	E_D	E_W	E_A	N	γ_{mo}
2	1.612	203.0	.5031	7	5.659
3	1.214	187.8	.1954	10	8.468
4	1.144	177.0	.1080	13	9.843
5	1.143	177.2	.1085	16	9.906
6	1.143	177.2	.1088	19	9.908
7	1.143	177.2	.1090	22	9.910
8	1.143	177.1	.1091	25	9.911
9	1.143	177.1	.1092	28	9.912
10	1.142	177.1	.1093	31	9.913
14	1.142	177.0	.1095	43	9.915
20	1.142	177.0	.1097	61	9.916
30	1.142	177.0	.1098	91	9.918
40	1.142	176.9	.1099	121	9.919

Table 4 Comparison of Different Hidden Layer Sizes

other sizes of networks, we find these same 12 weight values to be common to them all! In fact, these 12 weights, plus an output neuron bias, make up the trained 1-4-1 network. Starting with the 1-4-1 network, the actual value of each of these weights slowly increase or decrease as another hidden layer neuron is added. The change in value is very small, always less that one percent.

Now that we see how similar the networks of different sizes are, we must wonder what the rest of the neurons are doing. We found that, just as in the 1-6-1 example, all hidden layer neurons outside the common four are always identical in magnitude. In Table 3, the two neurons beyond the common 4 had identical values for input weight, output weight and bias. The 1-10-1 network has six neurons beyond the four that are common. Each of the six neurons had identical values for input weights, output weights and biases.

Table 5 shows a summary of how many hidden layer neurons had identical values for the different architectures. It also shows what the values are. Notice how the values

decrease as the number of additive neurons increase. It is also interesting that the point of symmetry for the transfer function of the neurons is preserved for all network sizes. Recall that the point of symmetry for $f(wp + b)$ is at $-b/w$. (See e.g., [HaDe96] Chapter 2.)

Number Hidden Neurons	Number Additive Neurons	$W1$	$B1$	$-\frac{B1}{W1}$	$W2$	$B2$
5	1	.77267	-.16720	.216	-1.11036	-.18510
6	2	-.66162	.15664	.237	.88057	-.11751
7	3	.59155	-.14127	.239	-.75124	-.07142
8	4	.54104	-.12842	.237	-.66544	-.03722
9	5	.50212	-.11808	.235	-.60326	-.01072
10	6	-.47083	.10969	.233	.55557	.01044

Table 5 Weight Values of Additive Neurons

Noise Considerations

Recall that the results in the previous section were for normally distributed noise of zero mean and 0.01 variance added to the saw function. We will now compare those results to equivalent amounts of Laplacian and uniformly distributed noise. The three noisy training data sets and the noise alone are shown in Figure 32.

Table 6 shows the training results for the Laplacian noise training set. With this type of noise, it was a little more difficult to locate the minimum of E_A. Occasionally, a model had to be retrained with another set of initial weights. As more parameters were available, the number of local minimums increased. The other local minima, however, had only slightly higher E_A values.

A characteristic of these other minima is that the redundant neurons are not all identical. For example, the 1-10-1 model trained with the results shown in Table 6 has four

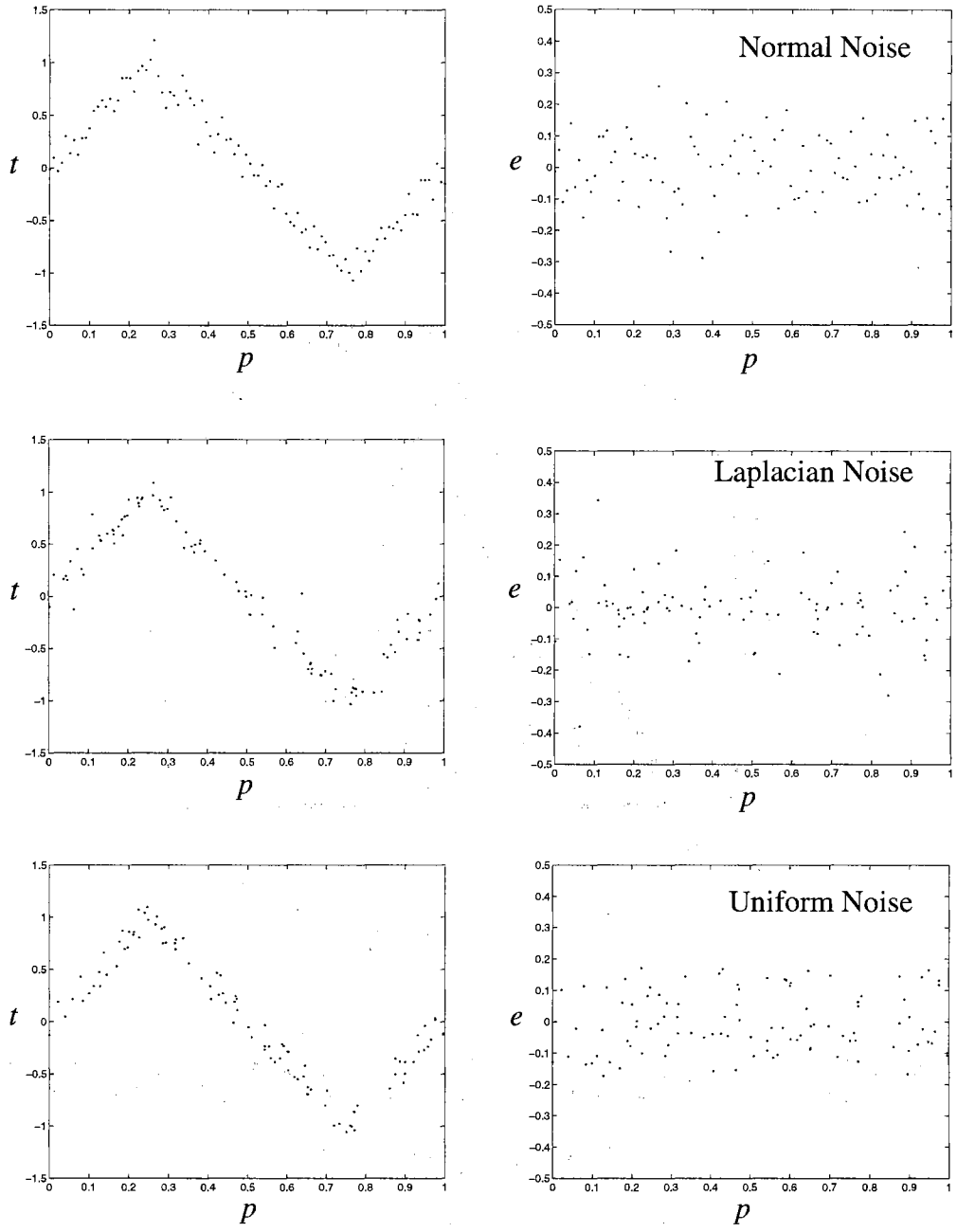


Figure 32 Noise and Noisy Training Sets

identical neurons. But, it is fairly easy for training to result in a model realization containing two different pairs of identical redundant neurons. Increasing the number of redundant neurons increases the number of alternate local minimums.

S	E_D	E_W	E_A	N	γ_{mo}
1	7.5436	116.2	6.0066	4	3.793
2	1.6250	214.8	.5770	7	5.676
3	1.0738	239.3	.1861	10	8.559
4	.9576	223.7	.0941	13	10.226
5	.9576	223.7	.0941	16	10.226
6	.9019	366.3	.0754	19	11.923
7	.9019	366.2	.0755	22	11.924
8	.9020	366.1	.0755	25	11.924
9	.9020	366.1	.0755	28	11.924
10	.9020	366.0	.0755	31	11.924

Table 6 Comparison of Models Trained with Laplacian Noise

It is also interesting to note that the 1-5-1 model had one redundant neuron with all other parameters common to the 1-4-1 model. The 1-6-1 model, and all more complex models with identical redundant neurons, produced the same optimal results.

Table 7 summarizes model results for training with uniformly distributed noise. This data set presented the most problems. The 1-5-1 model contains parameter values common to the larger models. Models trained with uniform noise exhibited some of the same tendencies as the Laplacian noise trained models concerning multiple sets of identical redundant neurons. In addition, there are now multiple minima with all identical redundant neurons. The performance of the different minima is significantly different.

Also, models containing at least 10 hidden layer neurons would occasionally locate a minimum point which had lower E_A and higher γ, though only slightly improved over the 1-5-1 model. When a minimum is located that is similar to another model, the actual values of common parameters are close but not nearly as close as for Laplacian or normally distributed noise models.

S	E_D	E_W	E_A	N	γ_{mo}
1	7.5436	116.2	6.0066	4	3.793
2	1.6250	214.8	.5770	7	5.676
3	1.0738	239.3	.1861	10	8.559
4	.9576	223.7	.0941	13	10.226
5	.9239	282.9	.0856	16	11.129
6	.9237	283.1	.0857	19	11.157
7	.9236	283.2	.0857	22	11.166
8	.9236	283.2	.0858	25	11.170
9	.9236	283.2	.0858	28	11.172
10	.9236	283.2	.0858	31	11.173
14	.9236	283.1	.0859	43	11.174

Table 7 Comparison of Models Trained with Uniform Noise

Although the models trained with uniformly distributed noise caused the most problems, we can always identify the redundant neurons. This will imply what the size is of the smallest sufficiently complex model that is needed for the training set. Since models with fewer redundant neurons train more consistently to the optimal minimum, the problems caused by uniform noise are controllable.

An Alternate Viewpoint

Recall from page 79 that MacKay's formulation for the effective number of parameters seemed to fall out of the α^{MP} and β^{MP} equations. Also, recall that Moody did not show a derivation for γ_{mo}. It would be nice if we had concrete mathematical foundation for the effective number of parameters. Lennart Ljung, with his significant body of work in system identification, gives us such a foundation. (See, e.g. [Ljun87].)

In this section, we will explore the works of Ljung and how they relate to our present work while adapting them to our framework. We will apply the standard measures of quality for a model that Ljung uses in his system identification textbook [Ljun87].

We will first analyze a neural network trained with the unregularized objective function $F = E_D$. Then we will look at the regularized model $F = E_D + \alpha E_W$. The results will be a new definition of the effective number of parameters. Also, a direct relationship between a regularized minimum and an unregularized minimum can be shown. This is not only important here, but will be used for comparison of the stopped training method described in Chapter 7 where we will resume exploring Ljung's work.

We begin by recalling how our training set data is modeled. As in Eq. (25), let the training set data be modeled as

$$a = f(p, \mathbf{w}) + \varepsilon \tag{81}$$

where p is the input, $f(p, \mathbf{w})$ is the true function, and ε is white noise with variance of $E(\varepsilon)^2 = \sigma_0$. Recall that these equations assume a single-input/single-output neural network for ease of presentation. They can be generalized for multiple inputs and outputs. Then with the unregularized objective function

$$F(\mathbf{w}, D) = E_D = \sum_{i=1}^{n} (a_i - f(p_i, \mathbf{w}))^2 = \sum_{i=1}^{n} (e_i(\mathbf{w}))^2 \tag{82}$$

we can train a neural network to minimize F such that

$$\mathbf{w}^{ML} = \arg\min_{\mathbf{w}} F(\mathbf{w}, D). \tag{83}$$

We know that as n grows larger, $\mathbf{w}^{ML}$ approaches $\mathbf{w}_0$, a minimum of the mean square error over the total population (as opposed to the error over a finite training set). Thus a measure of a model's ability to learn the true function $f(p, \mathbf{w}_0)$ could be

$$\bar{F}(\mathbf{w}) = EF(\mathbf{w}, D) = \mathrm{E}\left(\sum_{i=1}^{n} (a_i - f(p_i, \mathbf{w}))^2 \right) = \mathrm{E}\left(\sum_{i=1}^{n} (e_i(\mathbf{w}))^2 \right) \tag{84}$$

which is an expectation over all possible training sets, while the average trained model gives

$$\bar{F} = \mathrm{E}\bar{F}(\mathbf{w}^{\mathrm{ML}}) \tag{85}$$

performance, an expectation over the random variable $\mathbf{w}^{\mathrm{ML}}$.

Now we need to evaluate a few things. First, we need to approximate the gradient of $F(\mathbf{w}^{\mathrm{ML}}, D)$ around the minimum $\mathbf{w}_0$ by the Mean Value Theorem.

$$0 = \nabla F(\mathbf{w}^{\mathrm{ML}}, D) \approx \nabla F(\mathbf{w}_0, D) + \nabla^2 F(\mathbf{w}_0, D)(\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0) \tag{86}$$

Rearranging terms, we have

$$(\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0) \approx -[\nabla^2 F(\mathbf{w}_0, D)]^{-1} \nabla F(\mathbf{w}_0, D). \tag{87}$$

Second, from Eq. (82) and for $\mathbf{j}(\mathbf{w}_0) = \dfrac{\partial}{\partial \mathbf{w}} e(\mathbf{w}) \Big|_{\mathbf{w} = \mathbf{w}_0}$, we have for the gradient of Eq. (82)

$$\nabla F(\mathbf{w}_0, D) = 2 \sum_{i=1}^{n} \mathbf{j}_i(\mathbf{w}_0) e_i(\mathbf{w}_0). \tag{88}$$

Third, we will need the covariance of the gradient.

$$
\begin{aligned}
\operatorname{cov}(\nabla F(\mathbf{w}_0, D)) &= \mathrm{E}[\nabla F(\mathbf{w}_0, D)][\nabla F(\mathbf{w}_0, D)]^T \\
&= 4\mathrm{E}\left\{ \sum_{i=1}^{n} \mathbf{j}_i(\mathbf{w}_0) e_i(\mathbf{w}_0) \sum_{i=1}^{n} e_i^T(\mathbf{w}_0) \mathbf{j}_i^T(\mathbf{w}_0) \right\} \\
&= 4\mathrm{E}\left\{ \sum_{i=1}^{n} \mathbf{j}_i(\mathbf{w}_0) (e_i(\mathbf{w}_0))^2 \mathbf{j}_i^T(\mathbf{w}_0) \right\} \\
&= 4\sigma_0 \sum_{i=1}^{n} \mathrm{E}\{\mathbf{j}_i(\mathbf{w}_0) \mathbf{j}_i^T(\mathbf{w}_0)\} \\
&= 4n\sigma_0 \mathbf{K}
\end{aligned}
\tag{89}
$$

Here we note that the cross terms are uncorrelated, and we define $\mathbf{K} \equiv \mathrm{E}\{\mathbf{j}(\mathbf{w}_0)\mathbf{j}^T(\mathbf{w}_0)\}$.

We also see that since $\mathbf{w}_0$ is a minimum of $F(\mathbf{w}, D)$

$$
\mathrm{E}\{e_i(\mathbf{w}_0) e_i^T(\mathbf{w}_0)\} = \mathrm{E}\{(e_i(\mathbf{w}_0))^2\} = \sigma_0. \tag{90}
$$

Fourth, from Eq. (88), the expectation of the Hessian of Eq. (82) is

$$
\begin{aligned}
E\{\nabla^2 F(\mathbf{w}_0, D)\} &= E\left\{ 2 \sum_{i=1}^{n} \mathbf{j}_i(\mathbf{w}_0) \mathbf{j}_i^T(\mathbf{w}_0) + \left(\frac{\partial}{\partial \mathbf{w}} \mathbf{j}_i(\mathbf{w}_0) \right) e_i(\mathbf{w}_0) \right\} \\
&\approx 2 \sum_{i=1}^{n} E\{\mathbf{j}_i(\mathbf{w}_0) \mathbf{j}_i^T(\mathbf{w}_0)\} \\
&\approx 2n\mathbf{K}
\end{aligned}
\tag{91}
$$

where $E\left\{ \left(\frac{\partial}{\partial \mathbf{w}} \mathbf{j}_i(\mathbf{w}_0) \right) e_i(\mathbf{w}_0) \right\}$ is zero since $E\{e_i(\mathbf{w}_0)\} = 0$ and we assume the error is independent of the derivative.

Lastly, we need the covariance of $(\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0)$. This evaluation requires Eq. (87) and Eq. (91).

$$\begin{aligned}
\mathrm{cov}(\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0) &= \mathrm{E}\left\{(\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0)(\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0)^T\right\} \\
&\approx \mathrm{E}\left\{\left[-[\nabla^2 F(\mathbf{w}_0, D)]^{-1}\nabla F(\mathbf{w}_0, D)\right]\left[-[\nabla^2 F(\mathbf{w}_0, D)]^{-1}\nabla F(\mathbf{w}_0, D)\right]^T\right\} \\
&\approx 4n\sigma_0 \mathrm{E}\left\{[\nabla^2 F(\mathbf{w}_0, D)]^{-1}\mathbf{K}[[\nabla^2 F(\mathbf{w}_0, D)]^{-1}]^T\right\} \\
&\approx 2\sigma_0[\nabla^2 \bar{F}(\mathbf{w}_0)]^{-1}
\end{aligned} \tag{92}$$

Finally we are ready to evaluate Eq. (85). We start with a Taylor series approximation around $\mathbf{w}_0$ in Eq. (93). Note that $\nabla \bar{F}(\mathbf{w}_0) = 0$ since $\mathbf{w}_0$ is a minimum. Since we assume F is quadratic in the region around a minimum $\mathbf{w}_0$, then $\nabla^2 \bar{F}(\mathbf{w}_0)$ is positive definite. Thus it can be diagonalized where the quadratic can be written as a sum of products through the trace of the diagonal matrix. Then, substituting Eq. (92) we arrive at our result.

$$
\begin{aligned}
\bar{F} &= \mathrm{E}\bar{F}(\mathbf{w}^{\mathrm{ML}}) \\
&\approx \mathrm{E}\left\{ \bar{F}(\mathbf{w}_0) + \frac{1}{2}(\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0)^T \nabla^2 \bar{F}(\mathbf{w}_0)(\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0) \right\} \\
&\approx \mathrm{E}\bar{F}(\mathbf{w}_0) + \frac{1}{2}\mathrm{E}\,\mathrm{tr}\left\{ (\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0)^T \nabla^2 \bar{F}(\mathbf{w}_0)(\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0) \right\} \\
&\approx \mathrm{E}\left\{ \sum_{i=1}^{n} (e_i(\mathbf{w}_0))^2 \right\} + \frac{1}{2}\mathrm{E}\,\mathrm{tr}\left\{ \nabla^2 \bar{F}(\mathbf{w}_0)(\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0)(\mathbf{w}^{\mathrm{ML}} - \mathbf{w}_0)^T \right\} \qquad (93) \\
&\approx \mathrm{E}\left\{ \sum_{i=1}^{n} (\varepsilon)^2 \right\} + \frac{1}{2}\,\mathrm{tr}\left\{ \nabla^2 \bar{F}(\mathbf{w}_0)(2\sigma_0)[\nabla^2 \bar{F}(\mathbf{w}_0)]^{-1} \right\} \\
&\approx n\sigma_0 + \sigma_0\,\mathrm{tr}\mathbf{I}_N \\
&\approx \sigma_0(n+N)
\end{aligned}
$$

This says that every parameter contributes σ_0 to the model error $\bar{F}$, regardless of its importance to learning the true function. So, parameters which do not improve E_D degrade generalization. Obviously we want to minimize this effect. Thus we want N to be as small as possible. This is the process of model selection similar to the intent of the NIC of Chapter 4.

Now, suppose we add regularization and minimize

$$F^{\alpha}(\mathbf{w}) = E_D + \alpha E_W \qquad (94)$$

where $E_W = (\mathbf{w} - \mathbf{w}_0)^T(\mathbf{w} - \mathbf{w}_0)$, the square of the Euclidean distance of the neural network weight vector to the weight vector $\mathbf{w}_0$, which is a minimum of the true function. In reality, we typically will not know what $\mathbf{w}_0$ is, but we will deal with this later. Note that

to distinguish our regularization function from the unregularized F of Eq. (82), we add the superscript α in Eq. (94) and the next equation, Eq. (95), to remind us of which objective function we are discussing.

Now we will proceed just as before. Let

$$\mathbf{w}^{\mathrm{MP}} = \arg\min_{\mathbf{w}} F^{\alpha}(\mathbf{w}) \tag{95}$$

such that $\nabla F^{\alpha}(\mathbf{w}^{\mathrm{MP}}, D) = 0$. Assuming n is large enough that $\mathbf{w}^{\mathrm{MP}}$ is close to $\mathbf{w}_0$, we can expand the gradient by the Mean Value Theorem

$$0 = \nabla F^{\alpha}(\mathbf{w}^{\mathrm{MP}}, D) \approx \nabla F^{\alpha}(\mathbf{w}_0, D) + \nabla^2 F^{\alpha}(\mathbf{w}_0, D)(\mathbf{w}^{\mathrm{MP}} - \mathbf{w}_0). \tag{96}$$

Thus

$$(\mathbf{w}^{\mathrm{MP}} - \mathbf{w}_0) \approx -[\nabla^2 F^{\alpha}(\mathbf{w}_0, D)]^{-1} \nabla F^{\alpha}(\mathbf{w}_0, D). \tag{97}$$

Now, with $\mathbf{j}(\mathbf{w}_0) = \dfrac{\partial}{\partial \mathbf{w}} e(\mathbf{w})\Big|_{\mathbf{w} = \mathbf{w}_0}$ and defining $\mathbf{K} \equiv \mathrm{E}\{\mathbf{j}(\mathbf{w}_0)\mathbf{j}^T(\mathbf{w}_0)\}$ as before, the gradient of Eq. (94) is

$$\nabla F^{\alpha}(\mathbf{w}_0, D) = 2 \sum_{i=1}^{n} \mathbf{j}_i(\mathbf{w}_0) e_i(\mathbf{w}_0) \tag{98}$$

where the regulatory term E_W is zero at $\mathbf{w}_0$. Note that this is exactly the same as Eq. (88) for the unregularized objective function since $E_W\big|_{\mathbf{w} = \mathbf{w}_0} = 0$. Thus, the covariance of the gradient at $\mathbf{w}_0$ is the same as in Eq. (89).

$$\mathrm{E}[\nabla F^{\alpha}(\mathbf{w}_0, D)][\nabla F^{\alpha}(\mathbf{w}_0, D)]^T = 4n\sigma_0 \mathbf{K} \tag{99}$$

The expectation of the Hessian of Eq. (94) is found to be the sum of the Hessian for the unregularized objective function shown in Eq. (91) and a new term.

$$\begin{aligned} E\{\nabla^2 F^{\alpha}(\mathbf{w}_0, D)\} &= E\{\nabla^2 E_D + \nabla^2(\alpha E_W)\} \\ &\approx E\{\nabla^2 F(\mathbf{w}_0, D)\} + \alpha E\left\{\frac{\partial^2}{\partial \mathbf{w}\partial \mathbf{w}}(\mathbf{w}-\mathbf{w}_0)^T(\mathbf{w}-\mathbf{w}_0)\right\} \\ &\approx 2n\mathbf{K} + 2\alpha E\left\{\frac{\partial}{\partial \mathbf{w}}(\mathbf{w}-\mathbf{w}_0)\right\} \\ &\approx 2n\mathbf{K} + 2\alpha\mathbf{I}_N \end{aligned} \tag{100}$$

Now, using Eq. (97) and Eq. (99) the covariance of $(\mathbf{w}^{\mathrm{MP}} - \mathbf{w}_0)$ is

$$\begin{aligned} \mathrm{cov}(\mathbf{w}^{\mathrm{MP}} - \mathbf{w}_0) &= \mathrm{E}\{(\mathbf{w}^{\mathrm{MP}} - \mathbf{w}_0)(\mathbf{w}^{\mathrm{MP}} - \mathbf{w}_0)^T\} \\ &\approx \mathrm{E}\left\{\left[-[\nabla^2 F^{\alpha}(\mathbf{w}_0, D)]^{-1}\nabla F^{\alpha}(\mathbf{w}_0, D)\right]\left[-[\nabla^2 F^{\alpha}(\mathbf{w}_0, D)]^{-1}\nabla F^{\alpha}(\mathbf{w}_0, D)\right]^T\right\} \\ &\approx 4n\sigma_0\left\{[2n\mathbf{K} + 2\alpha\mathbf{I}_N]^{-1}\mathbf{K}[[2n\mathbf{K} + 2\alpha\mathbf{I}_N]^{-1}]^T\right\} \\ &\approx \sigma_0\left\{[n\mathbf{K} + \alpha\mathbf{I}_N]^{-1}(n\mathbf{K})[n\mathbf{K} + \alpha\mathbf{I}_N]^{-1}\right\} \end{aligned} \tag{101}$$

Finally, we are again ready to evaluate Eq. (85) for our regularized objective function. We show this in Eq. (102). As before, we use a Taylor series approximation around a minimum $\mathbf{w}_0$ at which point the gradient $\nabla\bar{F}(\mathbf{w}_0) = 0$. We also rearrange the terms by assuming that is approximately quadratic and $\bar{F}(\mathbf{w})$ is positive definite at a minimum $\mathbf{w}_0$. Then, we substitute from Eq. (101). Finally we use the fact that $\nabla^2\bar{F}(\mathbf{w}_0) \approx \mathrm{E}\nabla^2 F(\mathbf{w}_0, D) \approx 2n\mathbf{K}$ from Eq. (84) and Eq. (91).

$$
\begin{aligned}
\bar{F} &= \mathrm{E}\bar{F}(\mathbf{w}^{\mathrm{MP}}) \\
&\approx \mathrm{E}\left\{\bar{F}(\mathbf{w}_0) + \frac{1}{2}(\mathbf{w}^{\mathrm{MP}} - \mathbf{w}_0)^T \nabla^2 \bar{F}(\mathbf{w}_0)(\mathbf{w}^{\mathrm{MP}} - \mathbf{w}_0)\right\} \\
&\approx \mathrm{E}\left\{\sum_{i=1}^{n} (e_i(\mathbf{w}_0))^2\right\} + \frac{1}{2}\mathrm{E}\,\mathrm{tr}\left\{\nabla^2 \bar{F}(\mathbf{w}_0)(\mathbf{w}^{\mathrm{MP}} - \mathbf{w}_0)(\mathbf{w}^{\mathrm{MP}} - \mathbf{w}_0)^T\right\} \\
&\approx \mathrm{E}\left\{\sum_{i=1}^{n} (\varepsilon)^2\right\} + \frac{1}{2}\sigma_0\,\mathrm{tr}\left\{(2n\mathbf{K})[n\mathbf{K} + \alpha\mathbf{I}_N]^{-1}(n\mathbf{K})[n\mathbf{K} + \alpha\mathbf{I}_N]^{-1}\right\} \\
&\approx n\sigma_0 + \sigma_0\,\mathrm{tr}\left\{(n\mathbf{K})[n\mathbf{K} + \alpha\mathbf{I}_N]^{-1}(n\mathbf{K})[n\mathbf{K} + \alpha\mathbf{I}_N]^{-1}\right\}
\end{aligned}
\tag{102}
$$

Now, since all nondiagonal matrices are the same, namely $(n\mathbf{K})$, we can diagonalize all matrices simultaneously.

$$
\bar{F} \approx n\sigma_0 + \gamma_{\mathrm{lj}}\sigma_0 = \sigma_0(n + \gamma_{\mathrm{lj}}) \tag{103}
$$

where

$$
\gamma_{\mathrm{lj}} \equiv \sum_{i=1}^{N} \frac{(n\lambda_i)^2}{(n\lambda_i + \alpha)^2} = \sum_{i=1}^{N} \frac{\lambda_i^2}{\left(\lambda_i + \frac{\alpha}{n}\right)^2} \tag{104}
$$

is the effective number of parameters (as defined by Ljung) and λ_i are the eigenvalues of $\mathbf{K}$. Recalling Eq. (91) and that λ_i^b is an eigenvalue of $\nabla^2(\beta E_D)$, we note that $\frac{\lambda_i^b}{\beta} = 2n\lambda_i$. We can see that there is a linear relationship between the eigenvalues. Also, comparing Eq. (104) to Eq. (71) we note the form of λ_{lj} is very similar to that of λ_{ma}. Recall that each

component γ_i of the summation for both is bounded by $0 \le \gamma_i \le 1$. Since each γ_i tends to be close to either 0 or 1, we see that the squaring difference between γ_{lj} and γ_{ma} is of little consequence. However, Ljung's equation for γ_{lj} was purposefully derived where MacKay's was a by-product of other derivations.

Comparing to Moody's parameter, recall from Eq. (52) that

$$\gamma_{mo} = \mathrm{tr}((\nabla^2 E_D)(\nabla^2 F)^{-1}) \tag{105}$$

where $F = E_D + \alpha E_W$. If we let $\Lambda = \mathbf{B}^{-1}(\nabla^2 E_D)\mathbf{B}$ be a diagonal matrix, with λ_i as the diagonal elements, for symmetric $\mathbf{B}$, then

$$\begin{aligned}
\gamma_{mo} &= \mathrm{tr}((\mathbf{B}\Lambda\mathbf{B}^{-1})(\mathbf{B}\Lambda\mathbf{B}^{-1} + 2\alpha\mathbf{I})^{-1}) \\
&= \mathrm{tr}((\mathbf{B}\Lambda\mathbf{B}^{-1})(\mathbf{B}(\Lambda + 2\alpha\mathbf{I})\mathbf{B}^{-1})^{-1}) \\
&= \mathrm{tr}((\mathbf{B}\Lambda\mathbf{B}^{-1})(\mathbf{B}(\Lambda + 2\alpha\mathbf{I})^{-1}\mathbf{B}^{-1})) \\
&= \mathrm{tr}(\mathbf{B}\Lambda(\Lambda + 2\alpha\mathbf{I})^{-1}\mathbf{B}^{-1}) \\
&= \mathrm{tr}(\Lambda(\Lambda + 2\alpha\mathbf{I})^{-1}) \\
&= \begin{bmatrix} \frac{\lambda_i}{\lambda_i + 2\alpha} & 0 & \cdots & 0 \\ 0 & \frac{\lambda_i}{\lambda_i + 2\alpha} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \frac{\lambda_i}{\lambda_i + 2\alpha} \end{bmatrix} \\
&= \sum_{i=1}^{N} \frac{\lambda_i}{\lambda_i + 2\alpha} = \gamma_{ma}
\end{aligned} \tag{106}$$

This is identical to MacKay's γ_{ma} parameter!

Now, let's return to the analysis of Eq. (103). Notice how Eq. (93) compares to Eq. (103). For $\alpha = 0$ (no regularization), $\gamma_{lj} = N$ and Eq. (103) becomes Eq. (93). With regularization, however, $\gamma_{lj} < N$. In fact, only large eigenvalues λ_i of $\nabla^2 E_D$ significantly contribute to γ_{lj}. Thus, small eigenvalues which do not improve E_D do not degrade the performance of the trained neural network. Note that "large" and "small" are relative to α, so the regularization factor determines the neural network performance.

During training, some eigenvalues of $\nabla^2 E_D$ can get very small. These small eigenvalues are very sensitive to disturbances. Recall that λ_1 associated with eigenvector u_1 in Figure 29, is a small eigenvalue. When the parameter estimate $\mathbf{w}^{\mathrm{ML}}$ is moved along u_1 it only slightly effects the squared error, E_D. We see from Eq. (100) that with regularization, the eigenvalues of $\nabla^2 F^{\alpha}(\mathbf{w})$ cannot be smaller than 2α. This limits the minimum size of the eigenvalues of redundant parameters of $\mathbf{w}$, thus making them less sensitive to the noise peculiarities of a particular training set.

For an examination of how the regularized $\mathbf{w}^{\mathrm{MP}}$ differs from the unregularized $\mathbf{w}^{\mathrm{ML}}$, from Eq. (94),

$$\begin{aligned} 0 = \nabla F^{\alpha}(\mathbf{w}^{\mathrm{MP}}, D) &= \nabla E_D + \nabla(\alpha E_W) \\ &= \nabla F(\mathbf{w}^{\mathrm{MP}}, D) + 2\alpha(\mathbf{w}^{\mathrm{MP}} - \mathbf{w}_0) \end{aligned} \tag{107}$$

where $\nabla F(\mathbf{w}^{\mathrm{MP}}, D)$ is the unregularized objective function. Expanding around a near minimum point $\mathbf{w}^{\mathrm{ML}}$ of $F(\mathbf{w}, D)$ using the Mean Value Theorem and Eq. (91),

$$\begin{aligned}\nabla F(\mathbf{w}^{MP}, D) &= \nabla F(\mathbf{w}^{ML}, D) + \nabla^2 F(\mathbf{w}^{ML}, D)(\mathbf{w}^{MP} - \mathbf{w}^{ML}) \\ &\approx 2n\mathbf{K}(\mathbf{w}^{MP} - \mathbf{w}^{ML})\end{aligned} \tag{108}$$

where $\nabla F(\mathbf{w}^{ML}, D) \approx 0$ since $\mathbf{w}^{ML}$ is the minimum of $F(\mathbf{w}, D)$. Substituting Eq. (108) into Eq. (107) we have

$$\begin{aligned}\nabla F(\mathbf{w}^{MP}, D) + 2\alpha(\mathbf{w}^{MP} - \mathbf{w}_0) &= 0 \\ 2n\mathbf{K}(\mathbf{w}^{MP} - \mathbf{w}^{ML}) &= -2\alpha(\mathbf{w}^{MP} - \mathbf{w}_0) \\ n\mathbf{K}(\mathbf{w}^{MP} - \mathbf{w}^{ML}) &= -\alpha(\mathbf{w}^{MP} - \mathbf{w}^{ML} + \mathbf{w}^{ML} - \mathbf{w}_0) \\ n\mathbf{K}(\mathbf{w}^{MP} - \mathbf{w}^{ML}) &= -\alpha(\mathbf{w}^{MP} - \mathbf{w}^{ML}) - \alpha(\mathbf{w}^{ML} - \mathbf{w}_0) \\ (n\mathbf{K} + \alpha\mathbf{I})(\mathbf{w}^{MP} - \mathbf{w}^{ML}) &= \alpha(\mathbf{w}_0 - \mathbf{w}^{ML}) \\ (\mathbf{w}^{MP} - \mathbf{w}^{ML}) &= \frac{\alpha}{n}\left[\mathbf{K} + \frac{\alpha}{n}\mathbf{I}\right]^{-1}(\mathbf{w}_0 - \mathbf{w}^{ML})\end{aligned} \tag{109}$$

Now, rearranging,

$$\begin{aligned}\mathbf{w}^{MP} &= \mathbf{w}^{ML}\mathbf{I} + \frac{\alpha}{n}\left(\mathbf{K} + \frac{\alpha}{n}\mathbf{I}\right)^{-1}\mathbf{w}_0 - \frac{\alpha}{n}\left(\mathbf{K} + \frac{\alpha}{n}\mathbf{I}\right)^{-1}\mathbf{w}^{ML} \\ &= \left(\mathbf{I} - \frac{\alpha}{n}\left(\mathbf{K} + \frac{\alpha}{n}\mathbf{I}\right)^{-1}\right)\mathbf{w}^{ML} + \frac{\alpha}{n}\left(\mathbf{K} + \frac{\alpha}{n}\mathbf{I}\right)^{-1}\mathbf{w}_0 \\ &= (\mathbf{I} - \mathbf{M}_\alpha)\mathbf{w}^{ML} + \mathbf{M}_\alpha\mathbf{w}_0\end{aligned} \tag{110}$$

where $\mathbf{M}_\alpha = \frac{\alpha}{n}\left(\mathbf{K} + \frac{\alpha}{n}\mathbf{I}\right)^{-1}$. So, the regularization estimate $\mathbf{w}^{MP}$ is a weighted average of the unregularized estimate $\mathbf{w}^{ML}$ and the nominal value $\mathbf{w}_0$.

In practice, we cannot use the minimum of the mean square error over the entire population, $\mathbf{w}_0$, since we do not know it. However, in the next chapter, we will revisit these equations with interesting results for the special case of assuming the condition $\mathbf{w}_0 = 0$.

As one last note, we have found that regularization plays a key role in preventing overfitting. We have thus far seen two methods for improving generalization: model selection with the NIC, and adding a regularization term to the objective function. Ljung prefers to use regularization with an oversized network, rather than choosing the smallest network which does not overfit. In their experiments, the largest network that does not overfit produces worse overall performance than a larger model trained with regularization. They conclude that removing superfluous hidden layer neurons will remove some of the important parameters, too. We see that this is certainly possible given our discussion on page 93.

Summary

We have shown how we can use MacKay's Bayesian analysis of neural network learning to recalculate the optimum values of α and β after each training epoch. Demonstrations of our implementation in the GNBR algorithm show how this yields a dynamically changing emphasis on reducing the training set error versus reducing the effective number of parameters. Since these adjustments automatically optimize regularization factors, the resulting network is the best representation of the underlying function. Our experiments with the GNBR algorithm have demonstrated common results for all neural network architectures of a minimum size.

We have placed the formulas for the effective number of parameters adapted from three separate works into a common mathematical framework for comparison. Moody's γ_{mo} was found to be identical to MacKay's γ_{ma}, however the derivation of γ_{mo} is unknown while γ_{ma} is a by-product of other derivations. The third version comes from Ljung pro-

viding a firm mathematical foundation for his parameter γ_{lj}. Interestingly, the terms summed in γ_{lj} are almost the square of their counterparts in γ_{ma}. However, since all terms are bounded by $(0, 1)$ and tend to be driven to their extremes, all three versions of γ act similarly.

Although there are several ways to calculate γ, they all have drawbacks. They either require inverting the Hessian or calculating eigenvalues, both are computationally expensive. However, since the Hessian is already available to us through our training algorithm, the additional overhead is minimal!

Using MacKay's Bayesian learning technique alone requires alternately updating α and β versus γ, where the computation involved for calculating γ is expensive. However, our GNBR implementation takes advantage of information available in the training algorithm. The new GNBR algorithm provides optimal neural network performance with only a minor increase in computational overhead. We will summarize this major development with a recipe for application in Chapter 8. But first, we have one more method to examine.

There is a unique training method that is used very often today. It is called stopped training. Ljung's work has exposed a relationship between regularization and the popular stopped training methods. We will pursue this in depth in the next chapter.

CHAPTER 7

STOPPED TRAINING

Introduction

Ljung's work in neural networks seems to have gone almost unnoticed. Yet he has brought very sound mathematics to bear on the ad hoc procedure of stopped training.

Stopped training is often employed in training neural networks because of its simplicity. However, until recently it has lacked a mathematical foundation. Ljung has brought what appears to be the first rigorous explanation of the method. Interestingly, it is a straight forward extension of the concepts presented in his well known System Identification textbook. His analysis seems to go a long way in explaining why stopped training works, and also sheds light on why stopped training sometimes unexpectedly fails.[Ljun87]

What is most intriguing is how he is able to relate stopped training directly to regularization. This ground breaking comparison is the focus of this chapter. It is a continuation of section 'An Alternate Viewpoint' on page 103. The key references for this work are [Ljun87], [LjSj92], [SjLj92] and [SjLj94].

In this chapter, we will use Ljung's techniques to put stopped training into our framework for a direct analytical comparison to regularization. This comparison will explain some important characteristics of stopped training. We will also describe an experi-

mental comparison between stopped training and the Bayesian regularization optimization technique of the previous chapter.

Background

The stopped training method stops neural network training when a point is reached where further training would only lead to overfitting. This is done by monitoring the errors in an independently chosen validation sample set.

There are many variations on backpropagation training. These range from simple fixed step size gradient descent to Newton methods that require second derivative information. Different applications may require different training techniques. These are some of the reasons why many embrace stopped training. It can be applied to most any algorithm with only simple modifications, and the modifications do not change anything about the algorithm except the stopping criterion.

Let's say that we employ our favorite unregularized training algorithm for reducing the errors of a training set for a neural network. We know from the gradient descent technique that the error rate must decline with each training step. It will not stop until the limit of accuracy of the implementing machine is reached. This is why most algorithms use a particular minimum error size or minimum gradient magnitude as the stopping criteria.

Now, consider what is happening during training. Anyone who has watched a simple single-input/single-output neural network being trained and seen intermediate results during training has noticed that a network tends to take on the grossest features of the training set first. The finest features are always last to be learned. Often many of the finer fea-

tures have more to do with noise of the training set than of characteristics of the true function.

The longer we allow the unregularized neural network to be trained, the finer the details the network learns. This could be viewed as a developmental process that passes through stages resembling fully trained networks of increasing levels of complexity. The question is how do we know at which level of complexity to stop training in order to avoid overfitting. [Smit93]

If, during the training of a neural network, we were to monitor the errors of a noiseless sample set we might see results similar to Figure 33. In this figure, E_D represents the squared errors in the training set and E_A represents the squared errors on the noiseless set (actual error).

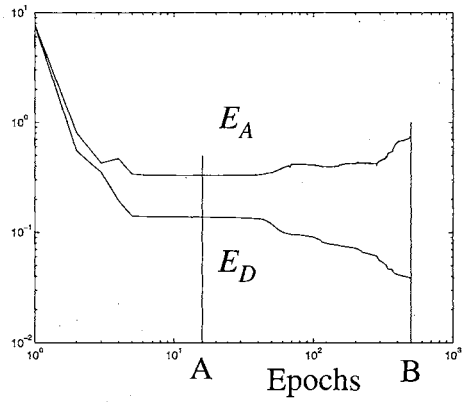


Figure 33 Example of Training to the Point of Overfitting

The errors in both sets are reduced initially. Then, at some point during training the network begins to learn the characteristics of noise. In Figure 33, this point is labeled A. Now, although the error continues to fall for the training set, E_D, the comparative error for

the noiseless data set, E_A, begins to increase. The network represented by stopping training at point B will overfit the training set. Thus, stopping training at point A should result in a network which has learned the most it can about the true function without significant overfitting. This network should have the best overall generalization, since it has not yet been trained to the level of the noise in the training set. Indeed, the learned functions shown in Figure 34 show that the overfitting found at point B has not yet happened at point A.

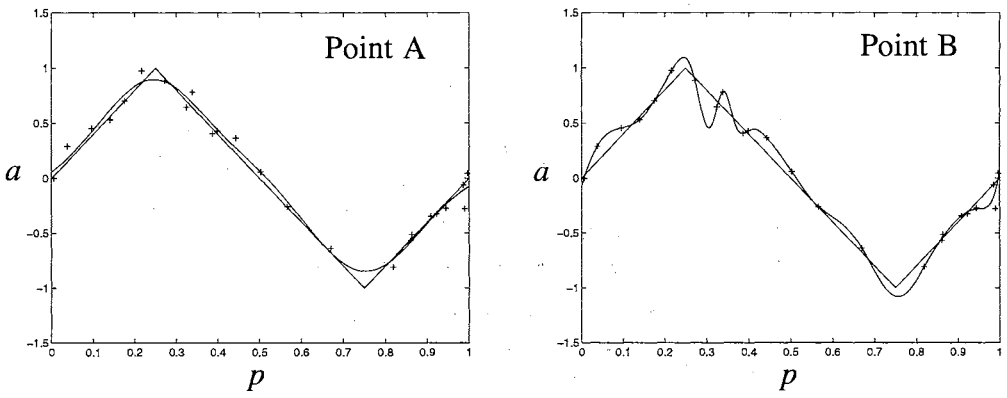


Figure 34 Function Plot at Point A and Point B

This procedure is promising, but there are several difficulties to address. First, in real-world problems noiseless data is not available. So we must substitute a second set of independently chosen samples. Usually this means dividing up the available samples into two sets: the training set and the validation set. This can be a problem when data is sparse because we need to use all the data we can to train the neural network.

Second, it is difficult to distinguish between training which learns the true function and training which learns noise. That is, there is not a decisive point in training where learning the true function stops and learning noise begins. The two stages overlap, as symbolically shown in Figure 35. Just as in the bias and variance discussion on page 30, from

the first to the last step of training the bias is being reduced, and at the same time the variance is being increased. No matter when training is stopped, some overfitting will have happened.

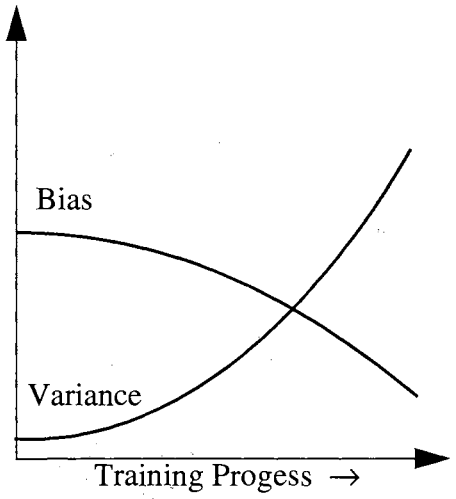


Figure 35 Bias and Variance of a Network During Training

Third, because noise is contained in both the training set and the validation set, the validation set error may not be a strictly decreasing function. In fact, the validation set error may have multiple valleys, some at a lower error level than others. Many ways to handle this problem have been suggested. They range from monitoring every fifth training step to calculating a percentage increase of a moving average of the error over a minimum point. (Recall the list on page 37.) In any case, there is no guarantee that a particular criteria that worked in one situation will work for all cases.

Fourth, the neural network may not be complex enough for stopped training to work. In real world problems, we have few clues as to how complex a network must be to learn the true function. Too complex a network will cause the variance to increase much more rapidly as training progresses than a network barely capable of learning the true func-

tion. In the latter case, the variance curve in Figure 35 would be virtually flat. If a network has the minimum complexity to fit a given function, then stopped training is relatively insensitive to the precision of the stopping criteria. In fact, the validation set error of Figure 33 may never distinctly increase.

However, for a complex network the bias and variance curves of Figure 35 may have very strong slopes. It may be a challenge to locate the optimal stopping point, especially with quick training techniques like quasi-Newton methods. Also, the level of the error at this sensitive stopping point may be higher than for the less complex network.

Stopped training clearly has some problems. Experience shows that results are not guaranteed. With a mathematical understanding of how the process works, however, one might be able to avert some of these problems.

An Analysis

We will start be analyzing the general weight update formula

$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \mu \mathbf{R} \nabla F(\mathbf{w}^{(i)}) \tag{111}$$

Note that in this formula $\mathbf{R}$ modifies the gradient direction. For simple backpropagation, $\mathbf{R} = \mathbf{I}$. For Gauss-Newton method, $\mathbf{R}$ is an approximation of the Hessian $\mathbf{H}$ of our objective function $F = E_D$. Thus, Eq. (111) represents a wide range of updating methods.

We now perform a Taylor series expansion of $\nabla F(\mathbf{w}^{(i)})$ around $\mathbf{w}^{\mathrm{ML}}$.

$$\nabla F(\mathbf{w}^{(i)}) \approx \nabla F(\mathbf{w}^{\mathrm{ML}}) + \frac{1}{2} \nabla^2 F(\mathbf{w}^{\mathrm{ML}})(\mathbf{w}^{(i)} - \mathbf{w}^{\mathrm{ML}}) \tag{112}$$

Since from Eq. (83), $\mathbf{w}^{\mathrm{ML}} = \arg \min_{\mathbf{w}} F(\mathbf{w}, D)$ then $\nabla F(\mathbf{w}^{\mathrm{ML}}) = 0$. Also, from Eq. (91)

$\nabla^2 F(\mathbf{w}^{\mathrm{ML}}, D) \approx 2n\mathbf{K}$ since $\mathbf{w}^{\mathrm{ML}}$ is assumed to be close to the minimum $\mathbf{w}_0$ of the true function. Substituting into Eq. (112), we have

$$\nabla F(\mathbf{w}^{(i)}) \approx n\mathbf{K}(\mathbf{w}^{(i)} - \mathbf{w}^{\mathrm{ML}}) \tag{113}$$

Using Eq. (111) and Eq. (113), we can evaluate $\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$ using $\mathbf{M}_m = (\mathbf{I} - \mu n\mathbf{R}\mathbf{K})^m$.

$$\begin{aligned} \mathbf{w}^{(1)} &= \mathbf{w}^{(0)} - \mu\mathbf{R}\nabla F(\mathbf{w}^{(0)}) \\ &= \mathbf{w}^{(0)} - \mu n\mathbf{R}\mathbf{K}(\mathbf{w}^{(0)} - \mathbf{w}^{\mathrm{ML}}) \\ &= (\mu n\mathbf{R}\mathbf{K})\mathbf{w}^{\mathrm{ML}} + (\mathbf{I} - \mu n\mathbf{R}\mathbf{K})\mathbf{w}^{(0)} \\ &= (\mathbf{I} - \mathbf{M}_1)\mathbf{w}^{\mathrm{ML}} + \mathbf{M}_1\mathbf{w}^{(0)} \end{aligned} \tag{114}$$

and

$$\begin{aligned} \mathbf{w}^{(2)} &= \mathbf{w}^{(1)} - \mu\mathbf{R}\nabla F(\mathbf{w}^{(1)}) \\ &= (\mathbf{I} - \mathbf{M}_1)\mathbf{w}^{\mathrm{ML}} + \mathbf{M}_1\mathbf{w}^{(0)} - \mu n\mathbf{R}\mathbf{K}(\mathbf{w}^{(1)} - \mathbf{w}^{\mathrm{ML}}) \\ &= (\mathbf{I} - \mathbf{M}_1)\mathbf{w}^{\mathrm{ML}} + \mathbf{M}_1\mathbf{w}^{(0)} - (\mathbf{I} - \mathbf{M}_1)(\mathbf{w}^{(1)} - \mathbf{w}^{\mathrm{ML}}) \\ &= 2(\mathbf{I} - \mathbf{M}_1)\mathbf{w}^{\mathrm{ML}} + \mathbf{M}_1\mathbf{w}^{(0)} - (\mathbf{I} - \mathbf{M}_1)[(\mathbf{I} - \mathbf{M}_1)\mathbf{w}^{\mathrm{ML}} + \mathbf{M}_1\mathbf{w}^{(0)}] \\ &= [2(\mathbf{I} - \mathbf{M}_1) - (\mathbf{I} - \mathbf{M}_1)^2]\mathbf{w}^{\mathrm{ML}} + [\mathbf{M}_1 - (\mathbf{I} - \mathbf{M}_1)\mathbf{M}_1]\mathbf{w}^{(0)} \\ &= (2\mu n\mathbf{R}\mathbf{K} - (\mu n\mathbf{R}\mathbf{K})^2)\mathbf{w}^{\mathrm{ML}} + \mathbf{M}_1^2\mathbf{w}^{(0)} \\ &= (\mathbf{I} - (\mathbf{I} - \mu n\mathbf{R}\mathbf{K})^2)\mathbf{w}^{\mathrm{ML}} + \mathbf{M}_2\mathbf{w}^{(0)} \\ &= (\mathbf{I} - \mathbf{M}_2)\mathbf{w}^{\mathrm{ML}} + \mathbf{M}_2\mathbf{w}^{(0)} \end{aligned} \tag{115}$$

Comparing the final form of Eq. (114) and Eq. (115), we see that

$$\mathbf{w}^{(m)} = (\mathbf{I} - \mathbf{M}_m)\mathbf{w}^{\mathrm{ML}} + \mathbf{M}_m\mathbf{w}^{(0)} \tag{116}$$

where $\mathbf{w}^{(0)}$ is our initial weight vector.

Now, compare Eq. (116) with Eq. (110) of Chapter 6. If our initial weight vector $\mathbf{w}^{(0)}$ is the same as our regularization constant $\mathbf{w}_0$, then the two equations describe a similar relationship between our initial set of weights $\mathbf{w}_0$ and the unregularized estimate $\mathbf{w}^{\mathrm{ML}}$!

In order to compare $\mathbf{M}_\alpha$ and $\mathbf{M}_m$, we will assume a gradient descent training method where $\mathbf{R} = \mathbf{I}$. Also, we assume $\mathbf{K}$ is diagonalized, since it can be made so through a change of basis.

Recall that λ^b is an eigenvalue of $\nabla^2 E_D = \nabla^2 F(\mathbf{w}, D) = 2n\mathbf{K}$. A diagonal element of $\mathbf{M}_m$ can therefore be written

$$(M_m)_{jj} = \left(1 - \frac{\mu n}{2n}\lambda^b\right)^m = \left(1 - \frac{1}{2}(\mu\lambda^b)\right)^m \tag{117}$$

Likewise, a diagonal element of $\mathbf{M}_\alpha$ can be written

$$(M_\alpha)_{jj} = \frac{\alpha}{n}\left(\frac{1}{2n}\lambda^b + \frac{\alpha}{n}\right)^{-1} = \left(\frac{1}{2\alpha}\lambda^b + 1\right)^{-1} \tag{118}$$

Let's compare these two elements as λ^b is varied. A plot for each of these elements over a range of $\frac{1}{\lambda^b}$ for $m = 100$, $\mu = 0.01$ and $\alpha = 1$ is shown in Figure 36. We know that for Eq. (111) to be stable for the gradient descent algorithm we must have $\mu < \frac{2}{(\lambda^b)_{\max}}$. (See, e.g. [HaDe96].) Thus we are only interested in properties of $\mathbf{M}_m$ and $\mathbf{M}_\alpha$ for $\frac{1}{\lambda^b} > \frac{\mu}{2}$. Indeed, we are really interested in comparing properties concerning small λ^b, since we

know that regularization drives redundant parameters and their associated eigenvalues to small sizes. We would like to draw some similar conclusions based on early stopping. From Figure 36, $\mathbf{M}_m$ and $\mathbf{M}_\alpha$ seem to be identical for small λ^b.

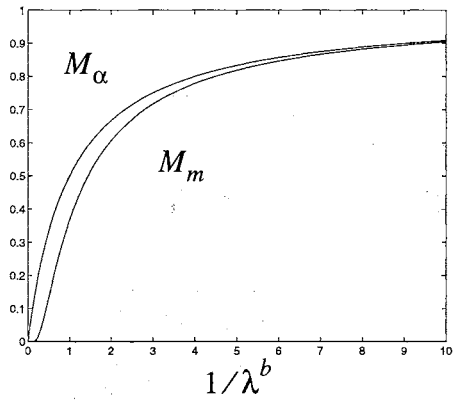


Figure 36 Plot of Diagonal Elements of $\mathbf{M}_m$ and $\mathbf{M}_\alpha$ Parameters

Let's compare the diagonal elements of $\mathbf{M}_m$ and $\mathbf{M}_\alpha$ for very small values of λ^b. Setting the equations for the diagonal elements equal and taking the derivative with respect to λ^b, we find for asymptotically small values of λ_b that

$$
\begin{aligned}
\left(\frac{1}{2\alpha}\lambda^b + 1\right)^{-1} &= \left(1 - \frac{1}{2}(\mu\lambda^b)\right)^m \\
-\log\left(\frac{1}{2\alpha}\lambda^b + 1\right) &= m\log\left(1 - \frac{1}{2}(\mu\lambda^b)\right) \\
-\frac{1}{\left(\frac{1}{2\alpha}\lambda^b + 1\right)} \cdot \frac{1}{2\alpha} &= m \cdot \frac{1}{\left(1 - \frac{1}{2}(\mu\lambda^b)\right)} \cdot \left(-\frac{1}{2}\mu\right) \\
\frac{1}{\left(\frac{1}{2\alpha}\lambda^b + 1\right)} &= \frac{m\mu\alpha}{\left(1 - \frac{1}{2}(\mu\lambda^b)\right)} \\
m &= \frac{1}{\mu\alpha} \cdot \frac{\left(1 - \frac{1}{2}(\mu\lambda^b)\right)}{\left(1 + \frac{1}{2\alpha}\lambda^b\right)} \\
m &\approx \frac{1}{\mu\alpha}
\end{aligned}
\tag{119}
$$

Now we can see that the number of training algorithm iterations m is directly linked to the regularization parameter α! From Eq. (104) in Chapter 6 we know that the number of effective parameters depends on the regularization factor α. A larger α yields a smaller effective number of parameters. In Eq. (119), m is inversely proportional to α. Substituting into Eq. (104), we see that the effective number of parameters grows with each training step taken. So, stopping training early from an initial weight setting of $\mathbf{w}_0$ has the same effect as applying regularization and using $E_W = (\mathbf{w} - \mathbf{w}_0)^T(\mathbf{w} - \mathbf{w}_0)$.

Now we take a look at the special case of allowing $\mathbf{w}_0$ to be the origin. Starting with Eq. (111) and expanding it with substitutions from Eq. (82) and Eq. (88) and allowing $\mathbf{R} = \mathbf{I}$, we have

$$
\begin{aligned}
\mathbf{w}^{(i+1)} &= \mathbf{w}^{(i)} - \mu \mathbf{R} \nabla F(\mathbf{w}^{(i)}) \\
&= \mathbf{w}^{(i)} - 2\mu \sum_{j=1}^{n} \mathbf{j}_j(\mathbf{w}^{(i)}) e_j(\mathbf{w}^{(i)}) \\
&= \mathbf{w}^{(i)} - 2\mu \sum_{j=1}^{n} \left[\frac{\partial}{\partial \mathbf{w}} e_j(\mathbf{w}^{(i)}) \right] e_j(\mathbf{w}^{(i)}) \\
&= \mathbf{w}^{(i)} - 2\mu \sum_{j=1}^{n} \left[\frac{\partial}{\partial \mathbf{w}} (a_j - f(p_j, \mathbf{w}^{(i)})) \right] (a_j - f(p_j, \mathbf{w}^{(i)})) \\
&= \mathbf{w}^{(i)} - 2\mu \sum_{j=1}^{n} \left[\frac{\partial}{\partial \mathbf{w}} (-f(p_j, \mathbf{w}^{(i)})) \right] (a_j - f(p_j, \mathbf{w}^{(i)}))
\end{aligned}
\tag{120}
$$

For the hidden layer of the neural network, the log-sigmoid transfer function $f^1(x) = \dfrac{1}{1 + \exp(-x)}$ is often used. The derivative of this function is $f^1(x) \cdot (1 - f^1(x))$. For large net input $x>>1$ the derivative is almost zero. This will cause the elements of $\dfrac{\partial}{\partial \mathbf{w}} f(p, \mathbf{w})$ associated with weights in the hidden layer to be very small. Thus, if any data and weight combination are mapped by the sigmoid function into the flat region, where the function value is near zero on one end or near one on the other, the weight updates of Eq. (120) will be near zero and the weight will be "frozen".

Now, we add regularization to the objective function. With

$$
\nabla F^{\alpha}(\mathbf{w}^{(i)}) = 2 \sum_{j=1}^{n} \mathbf{j}_j(\mathbf{w}^{(i)}) e_j(\mathbf{w}^{(i)}) + 2\alpha \mathbf{w}^{(i)}
$$

we can easily see from Eq. (120) that even though a weight may start in a "frozen" situation, it will at least be moved according to

$$\begin{aligned} \mathbf{w}^{(i+1)} &= \mathbf{w}^{(i)} - \mu(2\alpha\mathbf{w}^{(i)}) \\ &= (1 - 2\alpha\mu)\mathbf{w}^{(i)} \end{aligned} \tag{121}$$

For $0 < \alpha < \frac{1}{\mu}$, which stabilizes the update equation, the "frozen" weight will be driven toward zero, enforcing our regularization. Here the usefulness of the weight is regained since it is now in the active area of the log-sigmoid function. Of course, along the way $\frac{\partial}{\partial \mathbf{w}} f(p, \mathbf{w})$ will no longer be near zero and learning will resume with a now useful weight.

Thus, the special situation of allowing $\mathbf{w}_0 = \mathbf{0}$ plays the important role of ensuring any weight needed to minimize the objective function is available to play its part.

In review, Ljung has shown us that if we set our initial weights $\mathbf{w}_0$ to the vector $\mathbf{0}$, or very near it, and perform early stopping, we will be accomplishing virtually the same thing as minimizing a regularized objective function, since there exists a direct relationship between the number of training steps taken and the regularization factor α. Further, by using $\mathbf{w}_0$ near the origin, we have the added benefit of avoiding "frozen" weights.

Extending this idea, if we trained to minimize E_D, with resulting weights $\mathbf{w}^{ML}$, our neural network weights should have passed through (or at least near) the regularized minimum $\mathbf{w}^{MP}$. This is shown in a simplified graphical view in Figure 37. Thus for these conditions, stopped training could well result in locating $\mathbf{w}^{MP}$ or a point near it.

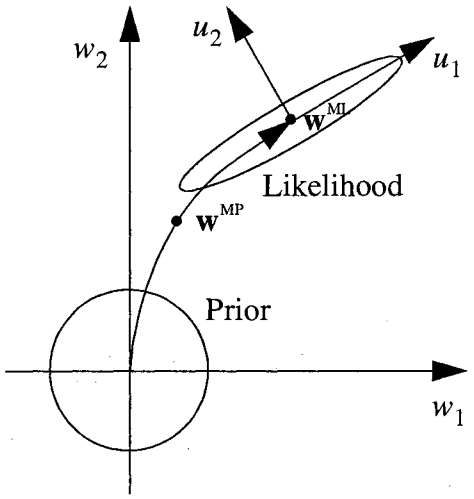


Figure 37 Simplified View of Regularization and Stopped Training Relationship

Method of Application

There are many choices to be made when implementing stopped training. These choices fall into two categories: the size of the validation set, and the stopping criteria.

For the size of the validation set, there do not seem to be any hard rules that have a rigorous mathematical foundation that can be applied in all cases. Rather, most knowledge is based on experience. However, we did find one recent publication worth noting.

In [AmMu95], Amari found that when the number of training set samples n is larger than thirty times the number of neural network parameters N, generalization is made worse by using a validation set to stop training. They show this through asymptotic statistical theory. They further show that for $N < n < 30N$ only $\frac{n}{\sqrt{2N}}$ samples should be taken from the training set and used as a validation set. The mathematical treatment of these rules make them worth noting. Of course, even these rules are based on assumptions about the network and the objective function used for training.

As for the stopping criteria, several types of circumstances, as noted on page 120, must be accounted for. The simplest strategy would be to stop the training on the first iteration for which the validation error function E_V goes up. Experience, however, shows that stopping after a single uptick fails to produce consistent results. The most encompassing set of stopping rules we have seen is found in [Prec94].

We now turn our attention to the problem at hand. We have discussed the possible validation set size and stopping criteria we would investigate using if we were to train a neural network using stopped training methodology. However, the spirit of stopped training is simply to observe the validation set error E_V and react to it. That is what we will do in our experiments in the next section.

Trials

For comparison of stopped training with the Bayesian optimization of the regularization method described in Chapter 6, we chose to try two experiments. First, we wish to show a case for which an under specified regularization parameter leads to overfitting for some fixed α and β of our regularized objective function. Second, we want to show a case where no overfitting occurs. We will observe the validation set error E_V for Bayesian optimized parameter values of α and β.

In our first experiment, we fixed $\alpha = 0.0001$ and $\beta = 78.8354$. While β is set to the final Bayesian optimal value, α is made very small to encourage the use of all the network parameters. The results are shown in Figure 38. We used the same 1-6-1 network architecture trained with the saw function as before. We even used the same initial weights

and training set D, but we added an additional validation set equal in size to D for easy comparisons. Note that the learned function plots in Figure 38, and Figure 39 which follows, show the training set as "+" and the validation set as ".".

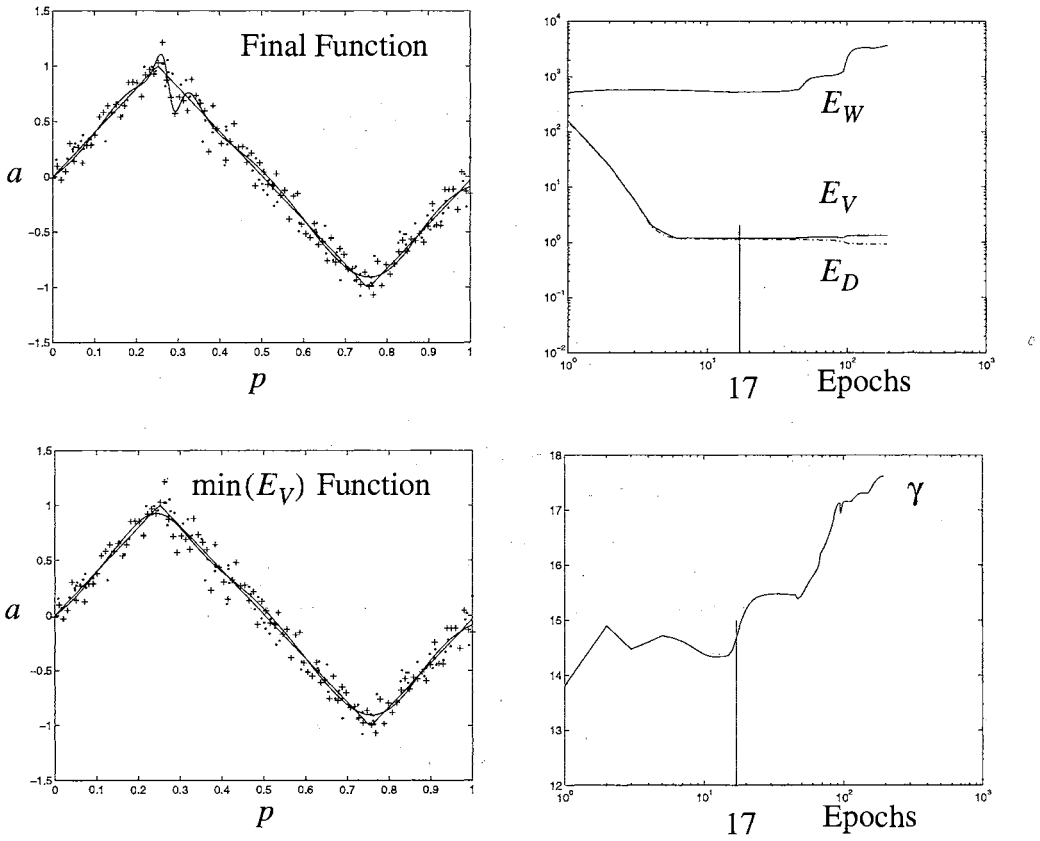


Figure 38 Results for Fixed α

From Figure 38 we can see that E_V tracked E_D at the beginning of training. In this area of training the network is learning features common to the two sets of data. Since they are common, it is hoped they reflect features in the true function. However, late in training the weight sizes grow tremendously. While this lowers E_D, it causes an increase in E_V. Overfitting is surely the result, since features being learned in this phase are predominately

from the peculiarities of the noise in the training set which are not in the validation set and therefore probably not in the true function.

The final learned function does indeed show overfitting. Parts of the saw function are not represented very well. However, this is the final trained network. If we applied stopped training to this case, the results would be different. For the simplest stopped training implementation, the network parameters responsible for the smallest validation set error E_V were saved. The smallest value for E_V was found at the 17th training step. A plot of the learned function at this intermediate point is shown in Figure 38. It looks very much like our optimal network function found through Bayesian regularization! Indeed, the developmental process described on page 119 seems to be true even for our fast Gauss-Newton algorithm.

If we examine the effective number of parameters during training, we see the final value is a very high 17.6. It increased rapidly as the weight values increased. At the 17th training step it is only about 15. This intermediate value is still much higher than in our Bayesian experiment, which resulted in a value of 9.9. This discrepancy occurs because all of the derivations of the effective number of parameters assumed we were at or near a minimum. Only the final value of γ has meaning, and all prior calculations are only gross indications, which grow more accurate as a minimum is reached. For this reason, the estimate of the effective number of parameters is not accurate under the stopped training method, since no minimum is ever reached.

Now let's look at our second experiment. We use the same conditions as before, except this time we optimize α and β. The results are plotted in Figure 39. They should

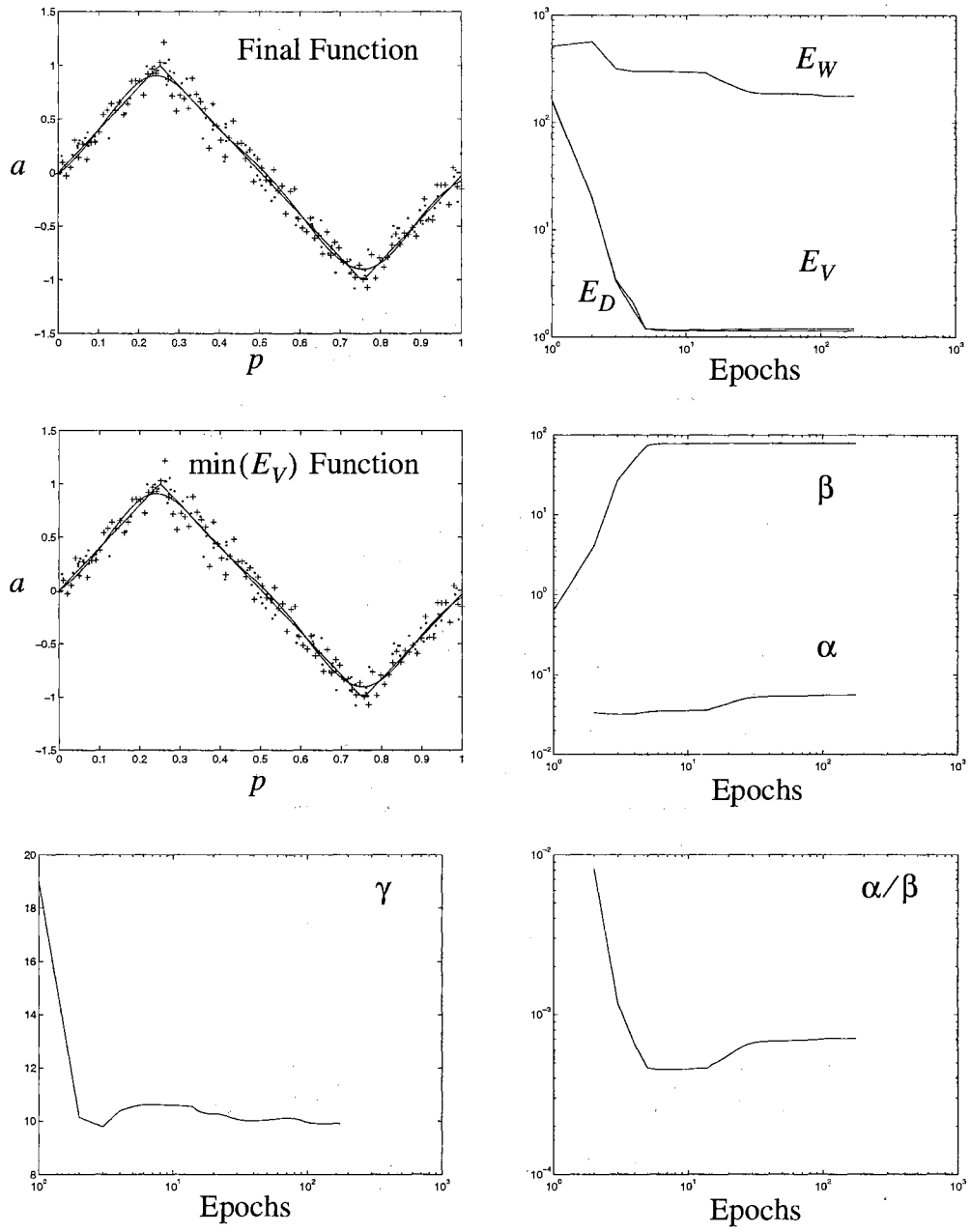


Figure 39 Results of Bayesian Learning Example

be familiar to us from Chapter 6. Notice how the validation set error E_V never substantially increased. In fact, it increased less than 2.5 percent from the minimum point at step 12 to the end of training at step 173. If we compare the function learned at step 12 to the final function, we see very little difference. The sum of squared errors for the nonnoisy data,

E_A, is 0.1022 for step 12 and 0.1088 at the end of training. Thus, even in the eyes of stopped training, our Bayesian regularization algorithm operates well.

Summary

We have explained the stopped training method for improving generalization. We have also put stopped training into our common framework for a direct comparison to regularization. Our experiments show that stopped training is effective when used properly. However, there are several problem situations that can arise. The best defense against them is to examine the validation set error over the course of training.

In our examples, we have shown that the performance of stopped training is similar to Bayesian optimization of regularization. However, in typical situations the amount of data available is limited. Splitting the data into a separate training set and validation set means our neural network will be trained with less data. Since the training set size is now significantly smaller, we would expect the resulting network to have less accuracy in describing the true function.

Therefore, in typical situations the Bayesian optimization method would be preferred to stopped training. Since stopped training requires a significant amount of precious data to be used for an independent validation set, the Bayesian optimization technique will generally yield more reliable results.

Stopped training is the last generalization improvement method we have chosen to pursue. In the next chapter, we will outline a recipe for application of the GNBR algorithm and show results for real world problems.

CHAPTER 8

REAL-WORLD PROBLEMS

Introduction

In this chapter we will apply our algorithm to real-world problems. We have explored several methods for improving generalization performance. From these, we have developed the Gauss-Newton approximation to Bayesian Regularization (GNBR) algorithm. Now we will apply the GNBR to four real-world problems.

The first test problem is a problem relating age to the ratio of weight to height for preschool boys. The second problem relates two photosensor inputs to the position of a ball in the field of view. The third problem is the prediction of annual sunspot activity. The final problem is the prediction of a sample response of the Mackey-Glass chaotic equation.

These four problems were selected to represent a diverse cross-section of applications: a single-input/single-output regression; a two-input/single-output regression; a real-world time series; and a simulated chaotic system.

First we will review a step-by-step recipe for application, and then we will show the results of the real-world problems. Overall results will be summarized at the end of this chapter.

Recipe for Application

Here is the general recipe we will follow for application:

1) *Normalize Data* -- Bound each input and target output by $[-1, 1]$. For a time series, bound the entire set before formatting for training usage.

2) *Testing Set* -- For our purposes, we will segregate a 10% sample of the data for a test set. This will serve as our cross-validation comparison check. In practice, all the data would be used for training unless over-fitting is suspected.

3) *Training* -- Choose a reasonably sized model and train with formatted data. Use the method described in section 'Method of Application' on page 82. For the Levenberg-Marquardt training algorithm, stop training by allowing μ to be driven to a maximum value.

4) *Check* γ -- If the final effective number of parameters γ is within S of the total number of parameters N, be suspicious that the model is not sufficiently complex. Increase S and retrain. Continue to increase the number of hidden neurons S until γ stops increasing.

5) *Other Checks* -- If the "noise" in the data is narrowly dispersed, then E_D, E_W and γ will remain constant for all sufficiently large models. For a wide dispersion of noise, retraining with new initial weights may be necessary to ensure consistent results (although our experiments showed surrounding minima to be very close to optimal results). As a final check for a sufficiently complex model, compare actual parameter values (weights and biases): for two models, look for parameter values in common; for the larger model, look for duplicate parameter values.

Both are good indications of redundant neurons and therefore sufficient model complexity.

We used these ideas as our primary guide for training neural networks with the GNBR algorithm. In the next few sections we show the results for the four real-world problems.

Age versus Weight/Height of Preschool Boys

This data set was taken from [SeWi89] but originated from [EpFo72]. There are two variables in the data set. One is the age of the preschool boys in the study, and the other is their weight to height ratio.

Figure 40 shows the normalized data "+" along with a trained neural network response. There are 72 points. Our process of normalizing the data is simply to scale it to be bounded by ± 1. For example if D is the set of ages, we could transform it according to

$$D_{norm} = \left[\frac{D - D_{min}}{D_{max} - D_{min}}\right] \times 2 - 1 \tag{122}$$

such that the elements of $D_{norm} \in [-1, 1]$.

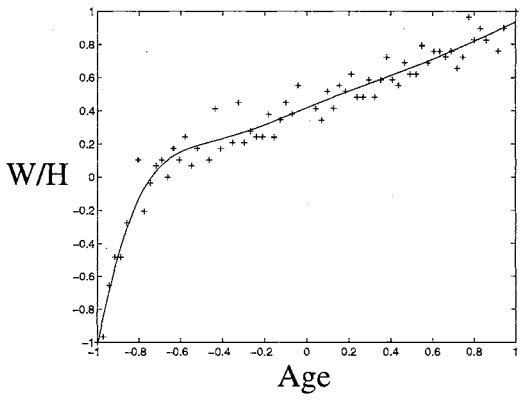


Figure 40 Preschool Boys Data Set

With 72 data points, we randomly chose 7 for extraction into our testing set. The rest formed our training set, with age as the input and weight/height as the target set. Table 8 summarizes the results of training a broad range of models. (Recall that S is the number of hidden layer neurons, E_D is the squared training set error, E_W is the squared weights, E_T is the squared test set error, N is the number of actual parameters, and γ is the effective number of parameters.) Notice that for all models with $S \geq 4$ the errors and γ are the same. A quick look at the final parameter values revealed why: all parameters associated with redundant neurons are very near zero! Thus, any model with $S > 4$ responds essentially the same as the 1-4-1 model. The GNBR algorithm has worked unquestionably well in producing the same optimal results for increasingly complex neural networks.

S	E_D	E_W	E_T	N	γ
1	1.2621	15.42	.04749	4	3.055
2	.4427	45.65	.00954	7	5.984
3	.4441	43.37	.00866	10	6.059
4	.4413	41.63	.01244	13	7.244
5	.4413	41.63	.01244	16	7.244
6	.4413	41.63	.01244	19	7.244
7	.4413	41.63	.01244	22	7.244
8	.4413	41.63	.01244	25	7.244
10	.4413	41.63	.01244	31	7.244
20	.4413	41.63	.01244	61	7.244

Table 8 Model Comparison for Boys Data Set

Figure 40 shows the resulting function learned while Figure 41 shows the error tracking for training a 1-6-1 model. The mapping learned by the neural network seems balanced in the data and very smooth. The plots of E_D and E_W are relatively flat for about

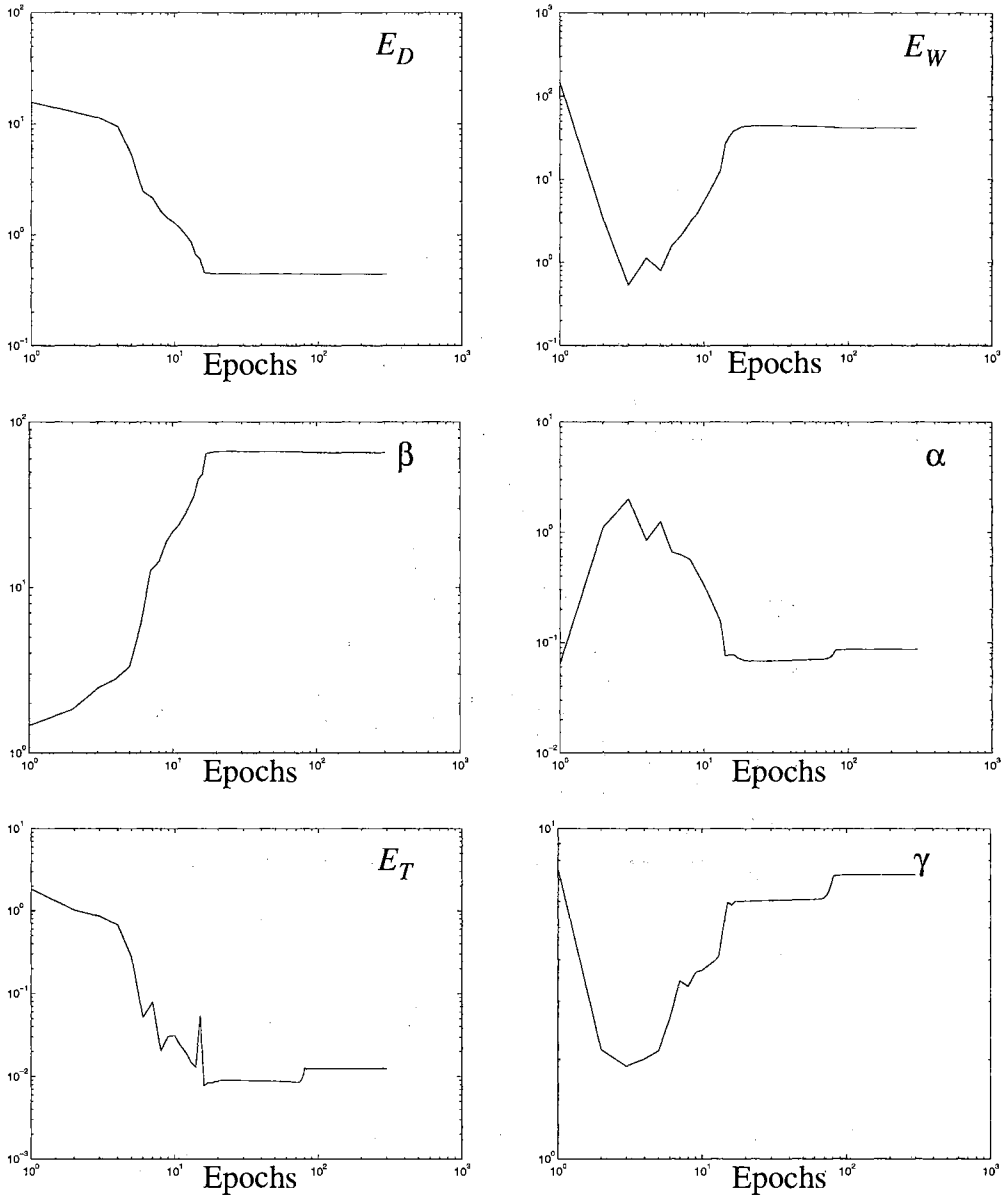


Figure 41 Training for Boys Data Set

the last 280 epochs. However, at about 80 epochs into training γ increases along with E_T. The normal interpretation would be that the increase in γ results from a better fit in the data, and a corresponding increase in E_T indicates fitting training set characteristics not common

to the testing set. This describes overfitting. But E_D does not show a drop to indicate better fitting and E_W does not increase as it does when overfitting occurs.

Compare the interim learned function plots in Figure 42. They show training results for 14, 16, 70 and 300 epochs. The testing data is indicated with "o" while the training set is shown as "+". From 14 to 16 epochs, the function clearly improves near $p = 1$, but there is little difference between the 16 and 70 epochs results. Both of these differences follow observations in E_T.

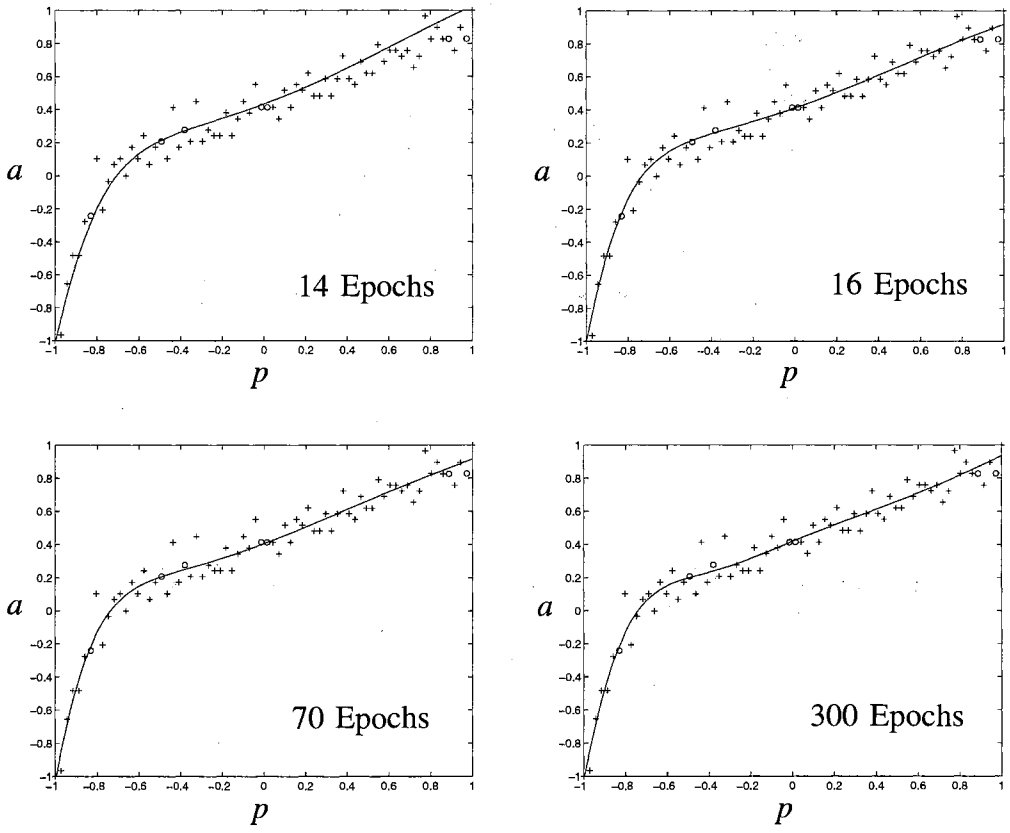


Figure 42 Interim Function Plots for 1-6-1 Training

The increase in E_T near 80 epochs is mostly attributed to the change in the learned function near $p = -0.4$. Comparing 70 to 300 epochs, the function in this area moved closer to the group of lower valued training set target points which increased the distance between the function and two test set points. This caused the increase in E_T.

The results of E_T for this particular training session are clearly misleading. Prior to the final increase in E_T, the neural network was slightly overfitting areas impacting E_T. In the final stages of training, the movement to a slightly better fit of the data resulted in moving away from the testing set data points. Using E_T as a stopping training criterion would have resulted in a less than optimal network. Indeed, retraining with different initial weights always resulted in the same final value for E_T but only occasionally showed the valley in E_T we chose to show here for demonstration. This indicates that the method of early stopping, which was presented in Chapter 7, will not always provide the optimum network.

Also, note in Table 8 that the $S = 2$ and 3 models yielded smaller E_T than the models with larger S; however, E_D and E_W are larger, γ is not consistent, and there are no common parameters between these models. These models are not complex enough for the mapping task, but those places where the models are flawed are not exposed by the test set. They were not complex enough to put the extra "bend" in the resulting function near $p = -0.4$. The smoother function for the smaller networks simply gives false indications for testing set results. A different testing set may not reflect the same results.

Sensors

Two optical sensors provide information about the position of an object which passes through their visual field. The two sensors are located in different positions, thus providing independent measurements. The data consists of the two sensor outputs and the object position. Figure 43 shows two different views of the data "o" and the final 2-8-1 network mapping "+".

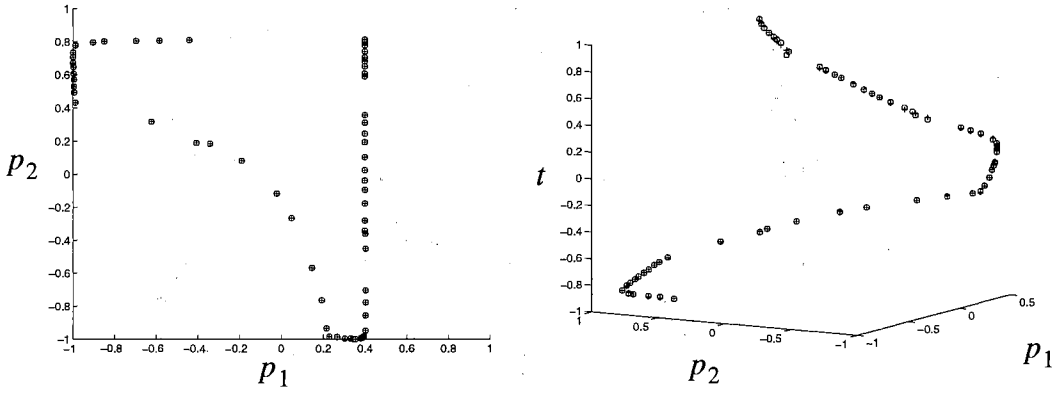


Figure 43 Sensor Data Set

The data was normalized and used to train a two-input/single-output network. Seven of the 67 data points occupied the randomly chosen testing set. The results are shown in Table 9. The 2-8-1 neural network is the smallest model providing optimal fit of the data. Notice how well the GNBR algorithm works. Even when 61 parameters are available, only about 21 are effectively utilized. Each increase in model complexity corresponds to an increase in γ, but only until a sufficiently complex model is found where the optimal γ is 20.978. Also, as complexity increases E_D and E_T tend to drop until sufficient complexity is available for them to reach the optimal errors of 0.003596 and 0.000973, respectively.

S	E_D	E_W	E_T	N	γ
2	.024190	126.59	.002924	9	8.035
3	.008660	87.60	.002114	13	12.118
4	.004857	36.94	.001012	17	15.567
5	.004507	36.70	.000987	21	16.727
6	.003903	41.73	.001010	25	19.027
7	.003917	40.98	.000979	29	19.269
8	.003596	46.51	.000973	33	20.978
9	.003596	46.51	.000973	37	20.978
10	.003596	46.51	.000973	41	20.978
15	.003596	46.51	.000973	61	20.978

Table 9 Model Comparison for Sensors Data Set

Just as in the previous problem, redundant neurons had parameter values near zero; however, sometimes retraining was necessary to locate the proper minimum. The 2-8-1 model has one redundant neuron, but we couldn't seem to train to the optimal minimum point with a 2-7-1 model. It would seem that the optimal minimum is "surrounded" by other local minimums causing our training algorithm to get stuck in one of them.

Figure 44 shows how training progressed for the 2-8-1 model. As is typical, the parameter β stabilizes before α makes a final adjustment. Notice how well the regularization works. The point where E_W drops significantly, E_D and E_T do not increase. Instead, they drop slightly while γ increases. The GNBR algorithm is reducing the redundant weight values and increasing the effectiveness of the necessary weights to produce optimal results.

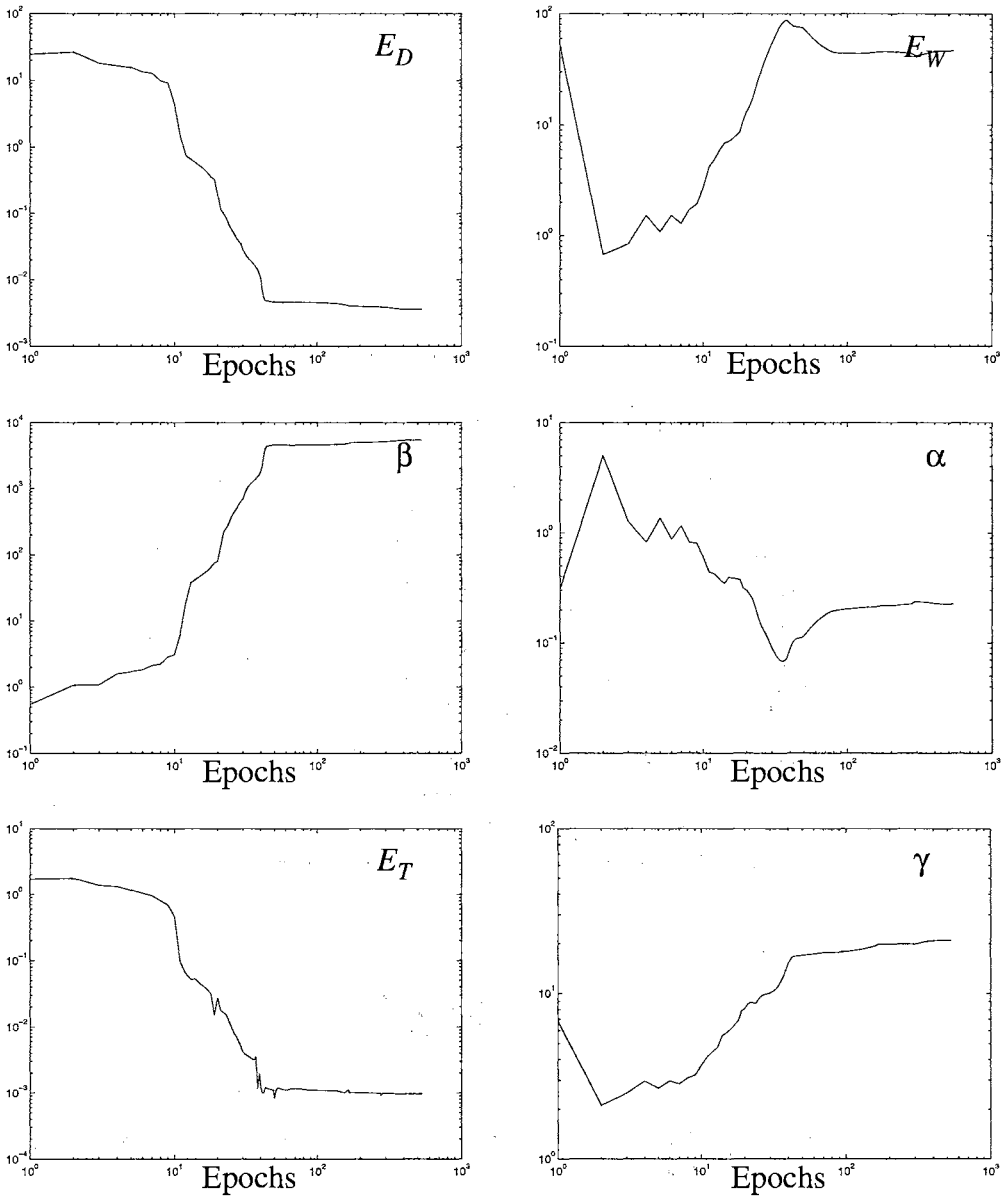


Figure 44 Training for a 2-8-1 Model with Sensor Data

Sunspots

The sunspot data set was taken from [Tong90]. It is the annual sunspot observations from 1700 to 1988. This is a time series problem so we used delayed values to "predict" the current target value for training. Box and Jenkins in [BoJe76] suggest a second order autoregressive model for this problem. Thus, after normalizing, the data set was formatted

using the two previous values as inputs to the neural network and the current value as the target. The last 29 points (10 percent of the data) were assigned to a testing set, the rest for training. Figure 45 shows the sunspot data set used for training where the year represents the relative position in the training set.

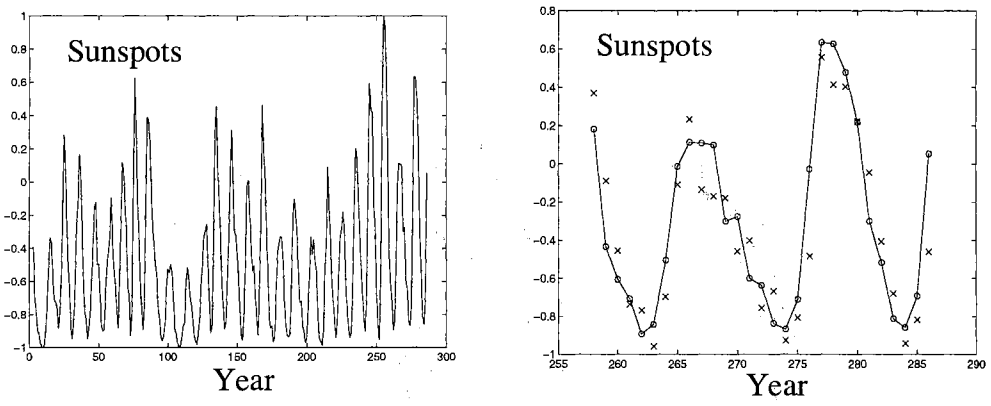


Figure 45 Sunspot Training Data Set and Test Set

Table 10 shows the training results for several models. The optimal network size was found to be a 2-4-1 model. This is the smallest neural network where increasing the number of actual parameters N does not cause an increase in the number of effective parameters γ. The optimal γ utilizes 12.52 out of a possible 17 parameters. In fact, for a 2-50-1 network having 46 redundant neurons with 184 associated parameters, the GNBR effectively inhibits their superfluous contributions. Again, the redundant neurons had associated parameters near zero, resulting in the same effective mapping function for all models with four or more hidden layer neurons.

Note that E_T is smallest for the 2-2-1 network. This is clearly not the optimal model, since increasing the number of hidden neurons S by one lowers E_D and increases γ.

S	E_D	E_W	E_T	N	γ
1	6.999	3.993	1.474	5	4.507
2	5.426	13.88	1.123	9	8.030
3	5.297	13.61	1.459	13	10.55
4	5.105	13.79	1.187	17	12.52
5	5.105	13.79	1.187	21	12.52
6	5.105	13.79	1.187	25	12.52
8	5.105	13.79	1.187	33	12.52
10	5.105	13.79	1.187	41	12.52
15	5.105	13.79	1.187	61	12.52
20	5.105	13.79	1.187	81	12.52
30	5.105	13.79	1.187	121	12.52
50	5.105	13.79	1.187	201	12.52

Table 10 Model Comparison for Sunspot Data Set

Following our recipe for application of the GNBR, the 2-4-1 network clearly represents the minimally complex model. Of course, as indicated above, any neural network with $S \geq 4$ will provide the same optimal generalization performance.

Figure 45 shows the testing set data points "o" with the corresponding 2-6-1 network one-step-ahead predictions "x". Our network produces residuals very comparable to those found in [BoJe76] for their AR(2) model. Figure 46 shows the training progress for the 2-6-1 model. As usual, β stabilizes before α, but in this case E_W does not reflect the final significant change in α that is shown in γ. The result, however, is a definitive drop in E_T providing the best testing set error for the lowest training set error E_D.

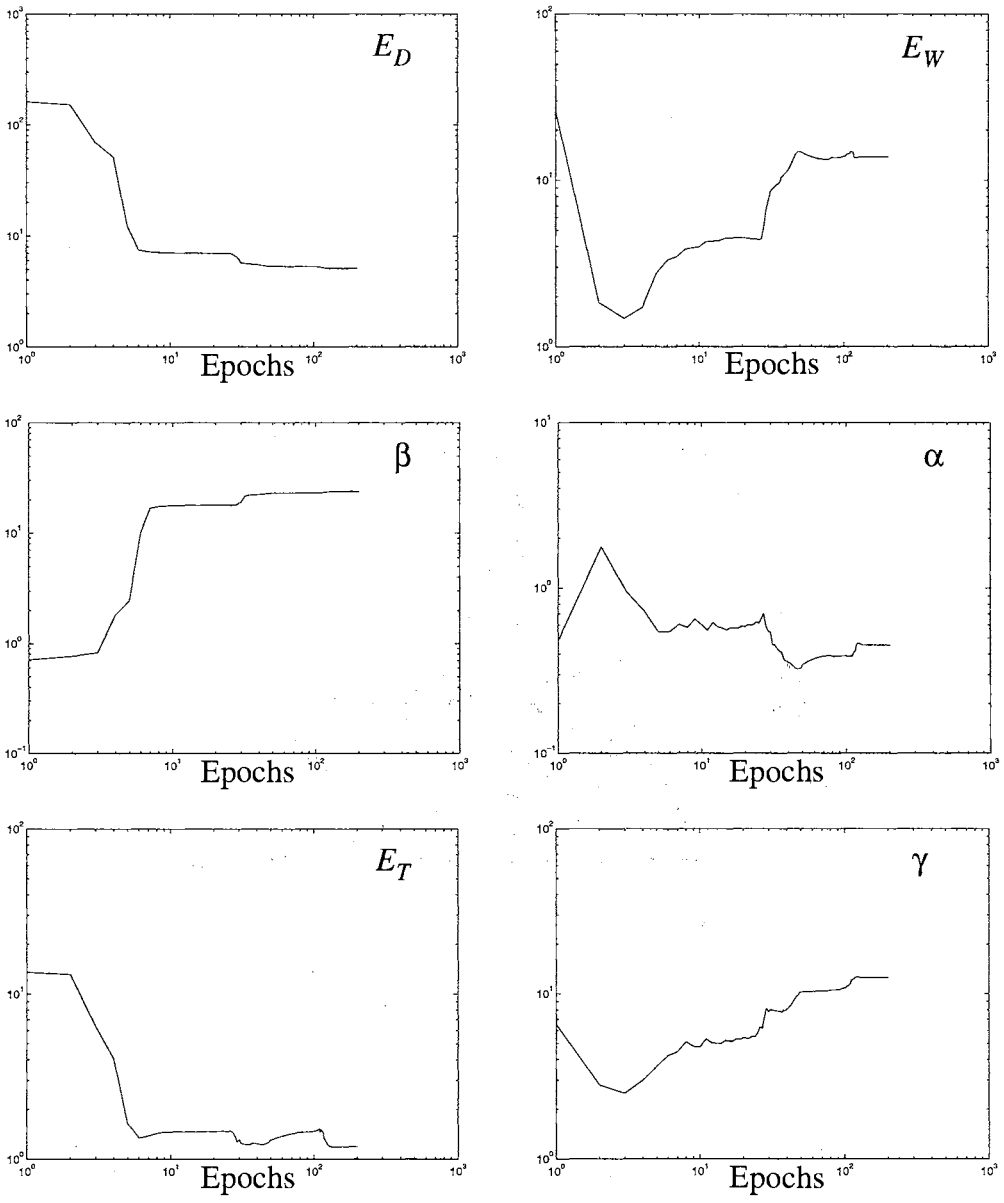


Figure 46 Training for a 2-6-1 Model with Sunspot Data

Mackey-Glass Equation

The Mackey-Glass chaotic equation, taken from [PlCo96], is

$$\dot{x}(t) = \frac{ax(t-\tau)}{1+x(t-\tau)^{10}} - bx(t) \tag{123}$$

We set the characteristic parameters to $a = 0.2$, $b = 0.1$ and $\tau = 17$. Letting $\Delta t = 1$,

we iterated the equation to produce a time series. Skipping the first 1000 iterates (transient period), we captured the second 1000 data points. The data set is shown in Figure 47.

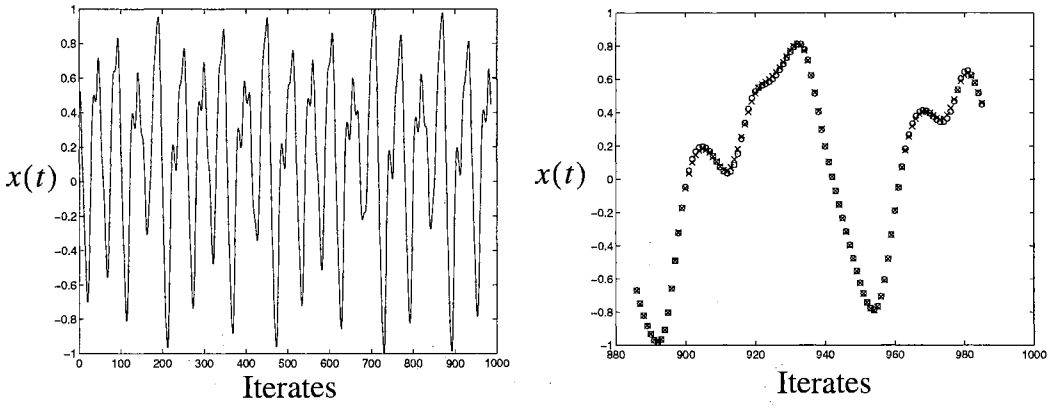


Figure 47 Mackey-Glass Training Data Set and Test Set

We then normalized the data set and took the last 100 points (10 percent) as a testing set. Using two time-delayed inputs, the first and eighteenth, we formatted the data and trained 2-S-1 models. Table 11 shows the results.

S	E_D	E_W	E_T	N	γ
1	2.532	68.56	0.4012	5	4.077
2	0.09857	107.2	0.01471	9	8.178
3	0.08527	98.81	0.01241	13	11.82
4	0.07438	14.70	0.01078	17	16.71
5	0.07384	15.07	0.01080	21	19.47
6	0.07376	14.88	0.01078	25	20.91
7	0.07375	14.41	0.01082	29	22.07
8	0.07375	14.41	0.01082	33	22.08
9	0.07374	14.41	0.01082	37	22.09
10	0.07374	14.42	0.01082	41	22.09
12	0.07369	15.27	0.01079	49	21.61

Table 11 Model Comparison for Mackey-Glass Data Set

The effective number of parameters γ reached a maximum of about 22 with the 2-7-1 network. Though N is increased for larger models, γ stayed at 22 indicating the 2-7-1 model is the smallest model with sufficient complexity to fit the data. Beyond this size, E_D and E_T remain stable at 0.0737 and 0.0108 respectively. Error values for larger networks show a slight deviation from the values for the 2-7-1 model. This is because the larger models have redundant parameters that are nonzero. Thus the common parameters are not exactly alike and the resulting networks do not respond identically, but they are very close. Indeed, the GNBR algorithm produces consistent optimal results, similar to those obtained for the saw function of Chapter 6. For each model the redundant neurons have identical values. Although in this case the parameters are nonzero, a simple examination still indicates that we have a minimum sized network with sufficient complexity to obtain optimal results.

Looking at Figure 47 again, the testing set is shown as "o" while the predicted results from the 2-7-1 neural network are shown as "x". This is a time series problem similar to the sunspot problem. However, the Mackey-Glass data set is much larger. Note how well the network output fits the test set data.

As a final note, Figure 48 shows the training progress for a 2-7-1 model. Notice how the significant weight changes allowed near 100 epochs of training resulted in improved performance for E_T. E_D improved only slightly, but γ shows a significant improvement. Again, this is where the GNBR is reducing the value of redundant weights and increasing the effectiveness of those necessary to fit the data.

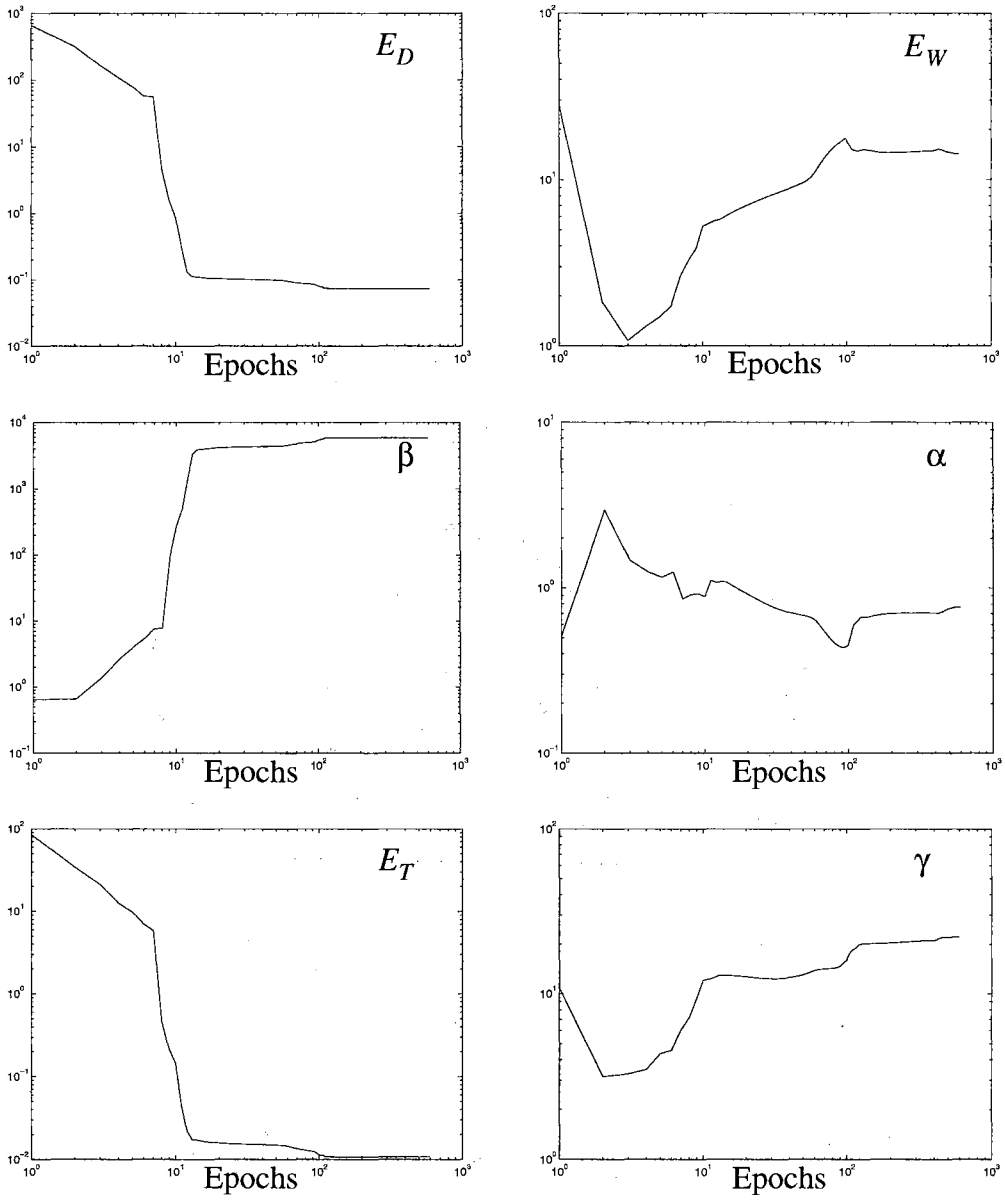


Figure 48 Training for a 2-7-1 Model with Mackey-Glass Data Set

Summary

In this chapter, we have outlined our recipe for application of our GNBR algorithm. It is simple to follow and easy to implement in conjunction with the Levenberg-Marquardt training algorithm.

We applied the GNBR to a variety of real-world problems to demonstrate the performance of our algorithm. In each case it was clear that for any reasonable model of at least a certain minimal complexity, the GNBR algorithm resulted in an optimal neural network realization.

We further showed how analyzing the resulting implementation was crucial to detecting false indications of optimal results. Specifically, a testing set that is required for the stopped training method of improving generalization was shown to sometimes result in a less than optimal realization. Examples of false indications during training and during model comparison were shown. In addition to removing precious data from the training set, stopped training does not produce reliable and consistent results.

In contrast, our GNBR algorithm always produced optimal results for a sufficiently complex neural network. Indeed, a comparison of parameter values for two oversized networks showed obvious and consistent results. Given the simplicity of implementing the GNBR in the Levenberg-Marquardt training algorithm, with minimal increase in computation, our GNBR algorithm has clear benefits over other techniques for improving generalization performance.

CHAPTER 9

CONCLUSIONS

In this chapter we present a brief summary of results. This is followed by recommendations for future work.

Summary of Results

We have discussed the key methods for improving generalization performance in feed-forward 2-layer neural networks trained for function approximation. These techniques fall into the following categories: limiting the size of the network (e.g., pruning, growing and model selection), limiting weight values through regularization, and limiting training with stopped training techniques. We performed an in-depth examination of the Network Information Criterion (NIC) for model selection, regularization, and stopped training.

One contribution of this work is a theoretical comparison of several important generalization techniques. We modified the development of each technique to place them into a common mathematical framework. This allowed a direct comparison between what would otherwise appear to be divergent techniques. Further, the framework we used is consistent with the Levenberg-Marquardt algorithm. This made each technique easy to implement for experimental comparisons.

We analyzed the strengths and weaknesses of each method, both theoretically and experimentally. This experience opened the way for the development of a new algorithm.

We found that the best overall strategy is to incorporate Bayesian optimization of regularization parameters into the training algorithm. This has several advantages over other techniques. Stopped training requires segregation of important data for a validation set. The NIC requires training numerous models to select the best implementation. Regularization also requires numerous training cycles to experimentally locate the optimal regularization parameter for a model.

The main contribution of this work is the development of the GNBR (Gauss-Newton approximation to Bayesian Regularization) algorithm -- an implementation of Bayesian regularization that uses a Gauss-Newton approximation of the Hessian matrix of the objective function. This approximation makes Bayesian regularization feasible because it drastically reduces the amount of computation required. We have shown in Chapters 6 and 8 that the GNBR algorithm consistently produces networks with excellent generalization capability.

A reasonably sized model trained with the GNBR algorithm with properly normalized data often produces optimal results the first time. Chapter 8 outlines our recipe for applying the GNBR algorithm and includes some simple checks to give confidence in the results.

We chose to implement the Bayesian optimization procedure in combination with the fast Levenberg-Marquardt training algorithm. We found the modification to the training algorithm to be straight forward and the computational overhead to be minimal, since

most of the information required for the GNBR algorithm was directly available from the Levenberg-Marquardt training algorithm.

We successfully applied the GNBR to four real-world problems. They included single variable regression, two variable regression and time series, for demonstration of diverse applications. The analysis of these problems provides both demonstration and insight into our recipe.

Recommendations for Future Work

In some of the real-world problems, the redundant parameters in a large network were actually driven to zero by the GNBR algorithm. Given our discussion on page 94 we would not typically expect individual redundant parameters to be driven to zero. We believe that the normalization process applied to the data sets may create this effect. Mathematical exploration of this area is needed to understand the transformation. A guarantee that a data transformation will result in negligible redundant parameter values has obvious advantages.

Another area for future work concerns the application to time series or other multivariate input problems. For a single-input/single-output model, it is easy to spot underfitting due to insufficient model complexity; the relationship between the effective number of parameters γ and the actual number of parameters N gives sufficient indications. However, for multiple-input models, adding a single hidden layer neuron may not cause a corresponding increase in γ, whereas a second added neuron might. This may have to do with the effectiveness of "neuron sharing" for what may otherwise be mutually exclusive contributions to the output neuron by different inputs. Some experimentation with multiple-

input models without cross connections to the hidden layer neurons might yield insight into this situation. This would explain why multiple plateaus for γ exist for a set of ordered models of minimally increasing complexity.

REFERENCES

[AmMu93] S. Amari and N. Murata, "Statistical Theory of Learning Curves under Entropic Loss Criterion," *Neural Computation*, vol. 5, pp. 140-153, 1993.

[AmMu95] S. Amari, N. Murata, K.-R. Muller, M. Finke and H. Yang, "Asymptotic Statistical Theory of Overtraining and Cross-Validation," METR 95-06, Department of Mathematical Engineering and Information, Physics, University of Tokyo, August, 1995. *(ftp://archive.cis.ohio-state.edu/pub/neuroprose/amari.overtraining .ps.Z)*

[Bish95] C. M. Bishop, *Neural Networks for Pattern Recognition*, New York: Oxford University Press, Inc., 1995.

[BoLi96] N. K. Bose and P. Liang, *Neural Network Fundamentals with Graphs, Algorithms, and Applications*, New York: McGraw-Hill, Inc., 1996.

[BoJe76] G. E. P. Box and G. M. Jenkins, *Time Series Analysis Forecasting and Control,* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1976.

[CuDe90] Y. L. Cun, J. S. Denker and S. A. Solla, "Optimal Brain Damage," in *Advances in Neural Information Processing Systems 2*, D. Touretzky, ed., pp. 598-605, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.

[Dodi94] R. Dodier, "Increase of Apparent Complexity Is Due to Decrease of Training Set Error," in *Proceedings of the 1993 Connectionist Models Summer School*, M. C. Mozer, P. Smolensky, D. S. Touretzky, J. L. Elman and A. S. Weigend, eds., pp. 343-350, Hillsdale, N.J.: Lawrence Erlbaum Associates, Inc., 1994.

[EpFo72] E. S. Eppright, H. M. Fox, B. A. Fryer, G. H. Lamkin, V. M. Vivian and E. S. Fuller, "Nutrition of Infants and Preschool Children in the North Central Region of the United States of America," *World Rev. Nutrition and Dietetics*, vol. 14, pp. 269-332, 1972.

[FaLe90] A. E. Fahlman and C. Lebiere, "The Cascade-Correlation Learning Architecture," in *Advances in Neural Information Processing Systems 2*, D. Touretzky, ed., pp. 524-532, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.

[HaDe96] M. T. Hagan, H. B. Demuth and M. Beale, *Neural Network Design*, Boston: PWS Publishing Co., 1996.

[Ljun87] L. Ljung, *System Identification: Theory for the User*, Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1987.

[LjSj92] L. Ljung and J. Sjöberg, "A System Identification Perspective on Neural Nets," Technical Report LiTH-I-ISY-1373, Department of Electrical Engineering, Linköping University, Sweden, 1992. *(ftp://ftp.control.isy.liu.se/pub/Reports/1992/1373.ps.Z)*

[MacK95] D. J. C. MacKay, "Bayesian Methods for Supervised Neural Networks," in *The Handbook of Brain Theory and Neural Networks*, ed. M. A. Arbib, pp. 144-149, Cambridge, MA: MIT Press, 1995.

[MacK92a] D. J. C. MacKay, "Bayesian Model Comparison and Backprop Nets," in *Advances in Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson and R. P. Lippmann, eds., pp. 839-846, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1992.

[MacK92b] D. J. C. MacKay, "A Practical Framework for Backpropagation Networks," *Neural Computation*, vol. 4, pp. 448-472, 1992.

[MacK92c] D. J. C. MacKay, "Bayesian Interpolation," *Neural Computation*, vol. 4, pp. 415-447, 1992.

[Mast95] T. Masters, *Advanced Algorithms for Neural Networks: A C++ Sourcebook*, New York: John Wiley & Sons, Inc., 1995.

[Mend87] J. M. Mendel, *Lessons in Digital Estimation Theory*, Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1987.

[Mood92] J. E. Moody, "The Effective Number of Parameters: An Analysis of Generalization and Regularization in Nonlinear Learning Systems," in *Advances in Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson and R. P. Lippmann, eds., pp. 847-854, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1992.

[Moro93] V. A. Morozov, *Regularization Methods for Ill-Posed Problems,* Boca Raton, Florida: CRC Press, Inc., 1993.

[Mura93] N. Murata, "Learning Curves, Model Selection and Complexity of Neural Networks," in *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan and C. L. Giles, eds., pp. 607-614, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1993.

[MuYo94] N. Murata, S. Yoshizawa and S. Amari, "Network Information Criterion -- Determining the Number of Hidden Units for an Artificial Neural Network Model," *IEEE Transactions on Neural Networks*, vol. 5, pp. 865-872, 1994.

[NgWi90] D. Nguyen and B. Widrow, "Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values of the Adaptive Weights," *Proceedings of the IJCNN*, vol. 3, pp.21-26, 1990.

[Pank83] A. Pankratz, *Forecasting with Univariate Box-Jenkins Models*, New York: John Wiley & Sons, 1983.

[PeCo92] M. P. Perrone and L. N. Cooper, "When Networks Disagree: Ensemble Methods for Hybrid Neural Networks," to appear in *Neural Networks for Speech and Image Processing*, R. J. Mammoni, ed., Chapman-Hall, 1993.
(ftp://archive.cis.ohio-state.edu/pub/neuroprose/perrone.MSE-averaging.ps.Z)

[PlCo96] M. Plutowski, G. Cottrell and H. White, "Experience with Selecting Exemplars from Clean Data," *Neural Networks*, vol. 9, pp. 273-294, 1996.

[Prec94] L. Prechelt, "Proben1 -- A Set of Neural Network Benchmark Problems and Benchmarking Rules," Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, Germany, 30 September 1994.
(ftp://ftp.ira.uka.de/pub/papers/techreports/1994/1994-21.ps.Z)

[ReMa95] R. Reed and R. J. Marks II, "Neurosmithing: Improving Neural Network Learning," in *The Handbook of Brain Theory and Neural Networks*, ed. M. A. Arbib, pp. 639-644, Cambridge, MA: MIT Press, 1995.

[Sarl95] W. S. Sarle, "Stopped Training and Other Remedies for Overfitting," to appear in *Proceedings of the 27th Symposium of the Interface*, 1995.
(ftp://ftp.sas.com//pub/neural/interface95.ps)

[SeWi89] G. A. F. Seber and C. J. Wild, *Nonlinear Regression,* New York: John Wiley & Sons, 1989.

[SiDo91] J. Sietsma and R. J. F. Dow, "Creating Artificial Neural Networks That Generalize," *Neural Networks*, vol. 4, pp. 67-79, 1991.

[SjLj92] J. Sjöberg and L. Ljung, "Overtraining, Regularization, and Searching for Minimum in Neural Networks," Technical Report LiTH-I-ISY-1297, Department of Electrical Engineering, Linköping University, Sweden, 1992.
(ftp://ftp.control.isy.liu.se/pub/Reports/1991/1297.ps.Z)

[SjLj94] J. Sjöberg and L. Ljung, "Overtraining, Regularization, and Searching for Minimum with Application to Neural Networks," Technical Report LiTH-ISY-R-1567, Department of Electrical Engineering, Linköping University, Sweden, 1994.
(ftp://ftp.control.isy.liu.se/pub/Reports/1994/1567.ps.Z)

[Smit93] M. Smith, *Neural Networks for Statistical Modeling*, New York: Van Nostrand Reinhold, 1993.

[TiGo90] A. N. Tikhonov, A. V. Goncharsky, V. V. Stepanov and A. G. Yagola, *Numerical methods for the Solution of Ill-Posed Problems,* Dordrecht, The Netherlands: Kluwer Academic Publishers, 1990.

[Tikh63] A. N. Tikhonov, "The solution of ill-posed problems and the regularization method," in *Dokl. Acad. Nauk USSR,* vol. 151:3, pp. 501-504, 1963.

[Tong90] H. Tong, *Non-linear Time Series: A Dynamical System Approach,* New York: Oxford University Press, 1990.

[Weig94] A. S. Weigend, "On Overfitting and the Effective Number of Hidden Units," in *Proceedings of the 1993 Connectionist Models Summer School*, M. C. Mozer, P. Smolensky, D. S. Touretzky, J. L. Elman and A. S. Weigend, eds., pp. 343-350, Hillsdale, N.J.: Lawrence Erlbaum Associates, Inc., 1994.

[WeKu91] S. M. Weiss and C. A. Kulikowski, *Computer Systems That Learn*, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1991.

[WeRu91] A. S. Weigend, D. E. Rumelhart and B. A. Huberman, "Generalization by weight-elimination applied to currency exchange rate prediction," in *Proceedings of the International Joint Conference on Neural Networks*, vol. 1, pp. 837-841, Piscataway, N.J.: IEEE, 1991.

[Whit92] H. White, *Artificial Neural Networks: Approximation and Learning Theory*, Cambridge, MA: Blackwell Publishers, 1992.

[Wolp92] D. H. Wolpert, "Stacked Generalization," *Neural Networks*, vol. 5, pp. 241-259, 1992.

VITA

Forest Dan Foresee

Candidate for the Degree of

Doctor of Philosophy

Thesis: GENERALIZATION AND NEURAL NETWORKS

Major Field: Electrical and Computer Engineering

Biographical:

Personal Data: Born in Cushing, Oklahoma, on July 8, 1959, the son of F. Don and Ruth E. Foresee.

Education: Graduated from Cushing High School, Cushing, Oklahoma, in May 1976; received Bachelor of Science degree in Electrical Engineering and Master of Electrical Engineering degree from Oklahoma State University, Stillwater, Oklahoma, in December 1979 and May 1981, respectively. Completed the requirements for the Doctor of Philosophy degree in Electrical Engineering at Oklahoma State University in December 1996.

Experience: Employed by Amoco Production Co. as a Research Engineer in 1980; employed by Oklahoma State University as a Teaching Assistant from 1980 to 1981; employed by Magnetic Peripherals Inc. as an Associate Electrical Engineer from 1981 to 1982; employed by Lucent Technologies, formerly of AT&T, as a Member of the Technical Staff from 1982 to present.

Professional Status and Memberships: Licensed as a Professional Engineer in the state of Oklahoma since July 1989; Member of National Society of Professional Engineers, Institute of Electrical and Electronic Engineers, and International Neural Network Society.