Training of artificial neural networks via interior point algorithms

by

Douglas Edward Welsh

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Department: Electrical Engineering and Computer Engineering
Major: Electrical Engineering

'niversity
Ames, Iowa

1993

This thesis is dedicated to my parents,
in recognition of the sacrifices they made
to ensure that my sisters and I received
quality educations.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

1.1 Artificial Neural Networks

The field of artificial neural networks (ANNs) has experienced a resurgence of interest in the last ten years. This is due in part to the use of neural networks for a variety of new applications, including forecasting . Neural networks are ideally suited to this application when the forecast is to be made based on the complex interactions of several variables, where the exact functional relationships between the variables is not known [1].

The most common form of neural network training–the determining of weights for interconnections between neurons– is known as the back-propagation algorithm. This algorithm is suited to networks having smooth, differentiable transfer functions. Unfortunately, back-propagation converges to only a local minimum in many instances or fails to converge at all. The development of new training algorithms for ANNs is an active area of research today.

1.2 Interior Point Linear Programming Algorithms

At about the same time that ANNs became popular again, the field of linear programming experienced a major advance with the introduction of Karmarkar's interior point algorithm [8]. This algorithm guaranteed polynomial-time solutions to large linear programs, where traditional methods such as the simplex technique could not. The interior point algorithms operate by shooting through the interior of the feasible region rather than moving from vertex to vertex of this region as in the simplex approach. Subsequent refinements to the Karmarkar's algorithm have reduced the computational complexity while at the same time speeding convergence to the solution. One of these improved algorithms is the affine scaling algorithm [17]. The affine scaling algorithm is guaranteed to converge to the optimal solution for a linear program where all the variables are constrained to be nonnegative. The original version of this algorithm operates by re-scaling the variables so they are equidistant from the nonnegativity constraint boundaries, and then the solution moves in the direction of a projected gradient.

1.3 Power System Short-Term Load Forecasting

Modern electric utilities attempt to operate in the most economic fashion possible while offering highly reliable service to customers. The short-term load forecast (STLF) is a useful tool for accomplishing this task. This forecast estimates the power system load in the period from one hour to several weeks into the future [1]. With this information, the utility may schedule their equipment and personnel to meet the load demand in a secure fashion, while keeping costs to a minimum. Artificial neural networks are becoming popular tools for performing load forecasts, because this technique is simple to implement while still being able to handle the complex relationships between the many variables that affect the load.

1.4 Contents of the Thesis

Chapter 2 of this thesis furnishes background information about artificial neural networks, beginning with their origins and progressing through their design and applications. That chapter includes a description of an artificial neuron, as well as the Hopfield and feedforward networks and their operation. It concludes by presenting information about the application of ANNs to the STLF problem.

Chapter 3 contains a powerful extension to the affine scaling algorithm. The modification uses a version of the algorithm that allows it to work with negative variables and variables with nonzero lower bounds, which allows the algorithm to be applied to a more general class of problems [18]. The extension to the affine scaling algorithm presented here incorporates the restricted basis entry technique into the algorithm by altering the upper and lower bounds on variables. This modification is useful in the linear programs where a nonlinear curve is represented by a piecewise-linear (PWL) curve.

Also in Chapter 3 is the description of an artificial neural network that has a piecewise-linear transfer function in the hidden layer neurons. This type of transfer function allows the training problem to be formulated as a linear program. The training program is explained in detail in that chapter. (In this thesis, the term "transfer function" is used to describe the functional relationship between the input and output of a neuron. This interpretation is common in Electrical Engineering and should not be confused with the more precise definition used in the field of mathematics.)

Chapter 4 discusses the implementation of the ANN and training algorithm applied to the STLF problem. A forecast is made for a 12-hour period using the new approach. Results from this forecast are compared to those obtained from a network trained with the back-propagation algorithm.

Conclusions about the ANN, training algorithm and the forecast are presented in Chapter 5. That chapter concludes with suggestions for future work in this area.

CHAPTER 2. BACKGROUND

2.1 Introduction to Artificial Neural Networks

The human brain contains on the order of 100 billion neurons, simple processing elements that give off brief electrical pulses when they are sufficiently excited . Each neuron is electro-chemically connected to about 10,000 other neurons by way of branches called axons and chemical-filled gaps known as synapses. The pulses from a neuron travel through these connections to either excite or inhibit other neurons. It is this massively interconnected network that allows us as humans to accomplish a myriad of tasks including controlling our speech and movement.

The brain is also the site of human memory. When a child recognizes his mother's voice, or a senior citizen remembers her high school graduation, the mind is actually retrieving patterns that are stored away in the brain. But how does the brain store and retrieve memories? It is not done in the way that one of today's computers does so, whereby a discrete memory location is allocated for a bit of data, that location is assigned a value of high or low corresponding to the piece of information, and when asked to recall the data the value at the location is read and used wherever the program desires. Instead, the brain stores information in a distributed fashion--in its structure. It is the network of neurons, synapses and the specific strengths of the electrical interconnections than contain human memory.

An extremely crude abstraction of human memory would be a matrix whose elements represent the strength of the electrical interconnections between different neurons. Just looking at this matrix we, of course, could not decipher any of the information held within it, but we could know which neurons were better connected electrically to other neurons and vice-versa. We could perform mathematical operations on parts of the matrix, though, and possibly we could interpret these results. If the matrix were for a living being, its elements would not be static, but instead would change as new memories are added and other memories are lost (though, hopefully, not at the same rate). Looking at this process on a microscale, D.O. Hebb explained it well in his classic text *The Organization of Behavior a Neuropsychological Theory* "when an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased [4, p. 62]." This is how changes in memory take place. As a person is exposed to events, his sensory nerves cause neurons in the brain to be excited and pulse. If this pulsing is frequent enough, the interconnection strength between neurons changes, and new information is added to the memory.

It was this theory that prompted researchers to begin studying artificial neural networks –abstractions of human neural structure that can be "trained" to recognize patterns, classify items and make forecasts, among other applications. These networks rarely contain more than 1,000 neurons and, thus, are not able to generalize on the scale that the human brain works. The neural network is usually represented mathematically as a set of weights representing the interconnection strengths, along with some model or models of how the neurons behave when excited. Neural network research has included varying the interconnection schemes between neurons, altering neuron transfer functions, experimenting with mathematical techniques to determine the interconnection strengths which optimize the system, and determining where the networks are best suited for application.

2.1.1 Artificial neurons

A detailed model for an artificial neuron used in ANN's is shown in Figure 2.1. The inputs are applied to the neuron and are scaled and summed. A bias term is added to the sum which is then passed through the transfer function of the neuron to yield the output. The standard symbol for a neuron is simply a circle with the interconnections shown going into and out of the neuron.

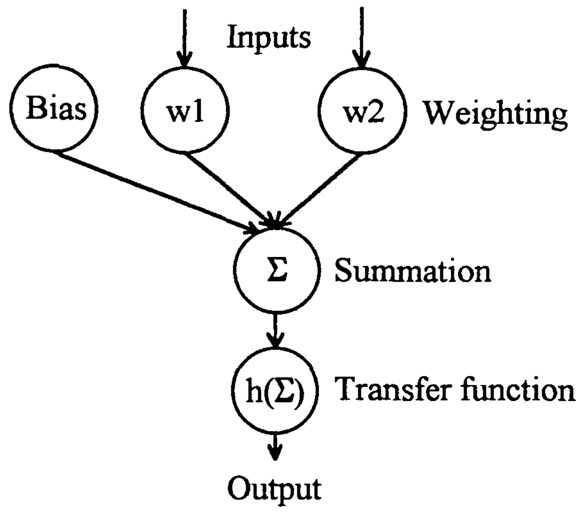

Figure 2.1. A detailed model of an artificial neuron

2.1.2 Neural network applications

Artificial neural networks have a wide range of application. One of the first applications of the networks was the pattern recognition problem. In this type of problem, items are taken from a set and classified into distinct groups based upon their characteristics. In some instances, ANN's can outperform the classical techniques for this application.

Neural networks can also serve as a model for human memory [7]. The Hopfield neural network–discussed in a later section– is one such model that is capable of recalling specific binary or bipolar patterns with which it has been trained. This network has also shown promise as a tool for optimization because of a special energy function that may be associated with the network's state.

One of the most common applications for ANNs is forecasting future trends. The best types of problems to apply ANNs to are ones where the variables that affect the output are known or can be determined, while the exact functional relationships between these variables cannot be determined. One such example of this condition is the power system short-term load forecasting problem. The inability to determine these functional relationships make the use of traditional techniques such as regression analysis impractical for these forecasts. ANNs are particularly valuable for problems that are extremely nonlinear. Neural networks are presently being used to forecast such things as stock market prices, and the risk associated with different loan applications.

2.1.3 Artificial neural network training

A given artificial neural network is not valuable for the applications listed above until certain network parameters are determined which cause the network to behave in a meaningful manner. This process is known as "training" the ANN. Specifically, neural network training involves the determination of weights for interconnections between neurons. The goal of the training is usually to minimize the sum of a function of output errors for a given set of inputs, or to ensure that the stable points of a network correspond to a set of training patterns. For example, training data may consist of a set of input vectors and a set of output vectors, where each input pattern is associated with a single output pattern. The goal of the training is to have the actual output of the network be as close, in some sense, to the associated output pattern when a given input pattern is applied.

2.1.4 Hopfield neural networks

The field of artificial neural networks was popular for several years in the 1940's as researchers came up with several new neuron models. It was during this time also that Hebb published *The Organization of Behavior*. The field experienced intermittent periods of popularity and disfavor until 1982 when J.J Hopfield published his paper *Neural networks and physical systems with emergent collective computational abilities* [7]. This paper began the resurgence of interest in artificial neural networks. The Hopfield network has a single layer of neurons, each of which is fed back to the inputs of each of the other neurons. The neurons are characterized by a bipolar or binary output and a threshold transfer function. The output of each neuron is fed back to the input of each other neuron via weighted interconnections. Conditions are often imposed on these feedback connections, including symmetric weights and no self-feedback. Absolute stability is guaranteed for networks having these particular restrictions on the weights [7]. The asynchronous network operates as follows. First, an initial output vector $\mathbf{x} \in R^n$ is selected and network output is held at this state. Then according to some probability distribution, a neuron is selected for update. If neuron i is selected during iteration k, its output is updated according to the following rule

$$x^{k+1} = \operatorname{sgn}\left(\sum_{j=1}^{n} w_{ij} x_j - t_i\right) \tag{2.1}$$

where x is the neuron's state, w_{ij} is the weight for the interconnection to neuron i from neuron j, and t_i is the threshold for neuron i. The output of the remaining neurons does not change at this stage. The network is then checked for stability, that is, to see if any neuron from the network would have its output change if it were selected for update. If the network is stable, stop. Otherwise, select another neuron for update and repeat the process.

The Hopfield network is trained with a set of binary or bipolar patterns. The usual goal of training a Hopfield network is to increase the likelihood that the network will stabilize to one of the training patterns during operation. Hopfield networks may also stabilize at spurious solutions, enter a limit cycle or wander chaotically during operation, depending on the restrictions placed on the weights. The most common method for training Hopfield networks is known as the sum of outer products algorithm. Another method uses linear programming to maximize the regions of convergence about the training patterns.

2.1.5 Feedforward networks

Feedforward networks are the most common type of ANN. The inputs are applied to the first layer of neurons, which usually have a linear transfer function. This input layer is connected to the hidden layer via weighted interconnections. Each hidden layer neuron has an associated transfer function, and the output of the hidden layer neurons serves as input to the next layer of neurons. Usually, only one hidden layer is present so this layer is directly connected to the output layer by weighted interconnections. Feedforward ANNs may have more than one hidden layer, but this topology is rarely used. The output layer neurons may have either a linear or nonlinear transfer function, depending on the design of the network. There are no limits to the number of neurons in a given layer. A simple feedforward network is shown in Figure 2.2. This network has four inputs, two hidden layer neurons and one output. The network is fully connected, meaning that each neuron is connected to all the neurons in the successive layer, and a bias is connected to the hidden and output layer neurons.

The most common nonlinear transfer function for a neuron is the sigmoid. Functionally this is expressed as

$$F(x) = \frac{1}{1 + e^{-cx}} \tag{2.2}$$

where c is a constant. Note that as c tends to infinity the sigmoid approximates a threshold function. Figure 2.3 shows a graph of the sigmoid function for $c=1$. Other common neuron transfer functions include linear threshold, hyperbolic tangent, gaussian and step functions.

The training set for a feedforward network is a set of input-output vectors. When a given training input pattern is applied to the input layer of the ANN, the error is some measure of the difference between the actual output of the network and the desired output vector. The interconnection weights are determined in the training process so that the output error for the training cases is minimized. The most common training algorithm for feedforward networks is a gradient technique known as back-propagation. This iterative method is a blame-assigning algorithm that changes a weight based on that interconnection's contribution to the error over the individual patterns. Each training pattern is presented several–possibly several thousand–times in the algorithm. Detailed discussions of the back-propagation algorithm may be found in any text on neural networks.

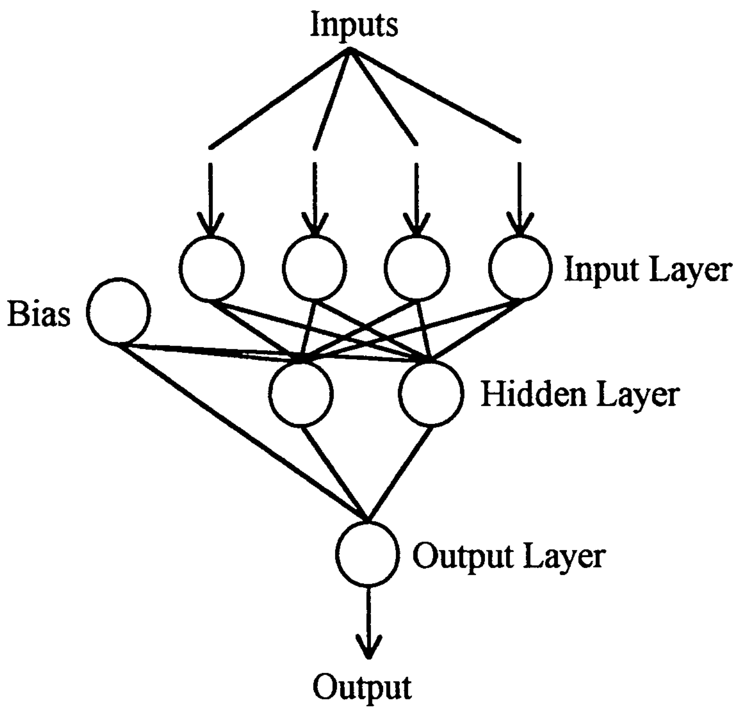

Figure 2.2. A feedforward artificial neural network

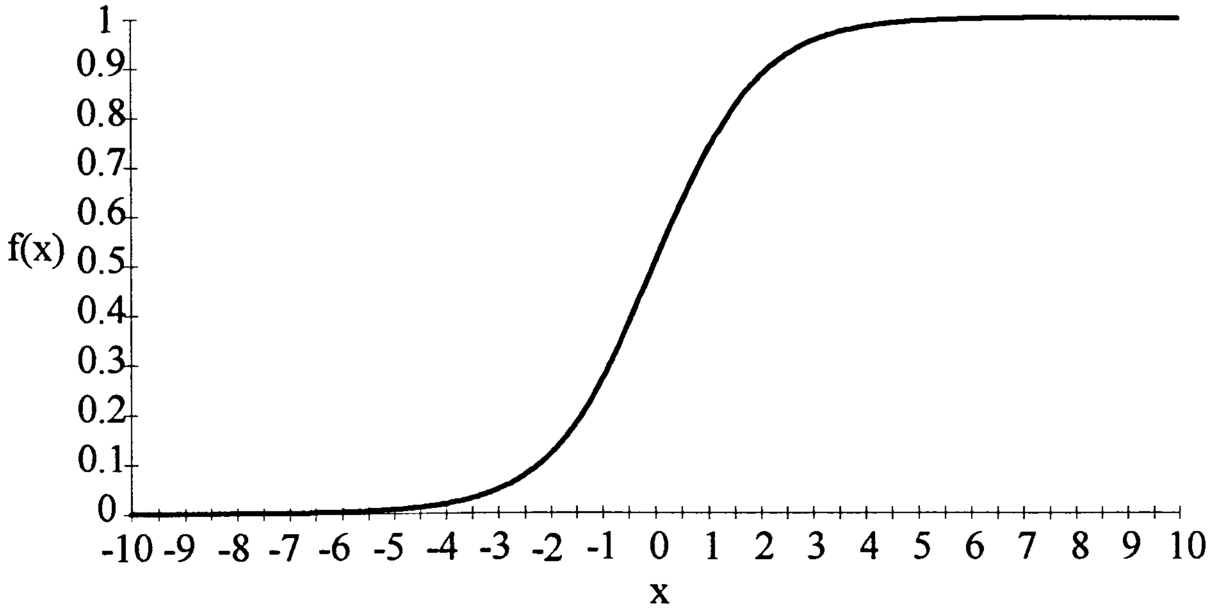

Figure 2.3. The sigmoidal curve

2.2 Introduction to Short-Term Load Forecasting

Short term power system load forecasting (STLF) involves the estimation of power system load from a few minutes to several weeks in advance. The load forecast is used for both economic and system security purposes. Economic applications of forecast information include economic dispatch, reserve margin allocation, unit commitment, interchange scheduling, fuel allocation, and minor maintenance scheduling [20]. The forecast may also be used in powerflow and contingency analysis to determine secure operating schemes [11]. Power system load is affected by a highly nonlinear combination of variables that can be categorized by dependency on weather, periodic or seasonal factors. Temperature is the most important weather-related factor. Humidity and cloud cover are other weather variables that affect power system load. Periodic variables that affect load include daily activities such as work, school and entertainment. Abnormal power levels resulting from special events such as the Superbowl and holidays are difficult to forecast accurately since these are uncommon and do not display a common load profile [6]. Seasonal considerations that affect load result from load growth, seasonal related loads, weather changes, and the number of daylight hours.

Short term load forecasting techniques are categorized according to how they deal with the different variables, particularly weather [13]. The time-series approach is a non-weather sensitive approach that uses historical load data for extrapolation to future load conditions. This method views abnormal data as bad data [1] and is also time consuming, computationally intensive and numerically unstable [14]. Regression models study consumer habits and weather behavior to derive linear models for the system load, however the linearization of the weather terms is unjustified [14]. The knowledge based system (KBS) approach is a rules-based method for forecasting load that attempts to convert the logic of the power system operator into mathematical equations for forecasting. The problem with KBS techniques is that they assume the presence of an expert to derive the rules from on-the-job training, and conversion of the operator's logic to equations sometimes also may be impractical [15]. Other forecast techniques include spectral analysis, exponential smoothing, state-space techniques, and Kalman filtering. All of these approaches require unwieldy historical databases of three to ten years of data to model the seasonal and annual load variations.

The load profiles tend to change with the time of the year. This is due to factors such as seasonal loads (i.e., heating and cooling) and growth. ANN's deal with these seasonal variables in two ways. First, a given model may be retrained regularly, sometimes daily, to reflect the load trend. The training data in this case is selected from a time-limited set, assuring that the training data is always recent enough to be unaffected by seasonal variations. This is known as the "moving window" technique for training data selection. The other method for dealing with seasonal variables is the use of separate models for different times of the year. Each model includes variables that are pertinent only to that particular season of the year.

The ANN method models the multivariate forecast problem without making complex dependency assumptions about the input variables, but instead relies on the selection of appropriate training and input data. The ANN performs what is essentially a pattern recognition function, based upon the historical data that is used to train the network [15]. With its ability to be retrained using recent historical data, the ANN is inherently updatable and eliminates the need for huge databases, thus the amount of data needed to train the network is at a minimum. The short term ANN methods reviewed do not have the ability to capture the seasonal and annual load growth trends that other techniques can capture. A proper comparison of techniques would require the ANNs to include all models incorporated within the classical techniques.

The use of ANN's for STLF may be reduced to several tasks. The process begins by determining the length and type of the forecast to be made. Forecasts are usually daily or weekly, and may estimate the hour-by-hour load or the peak load for the period, among other possibilities. Next, the appropriate type and quantity of training data is selected. The specific inputs required for a forecast vary by geography, demographics, and time of year. In general, a neural network that is used to forecast load for one region is not an optimal design for another region [11]. The amount of training data required to train the network also varies. With the moving window approach, it is useful to select data from several weeks prior to the forecast period. This width of this window is normally on the order of four weeks to prevent negative effects from seasonal load changes. Historical data from the same period in previous years may also be used, but caution must be exercised with regard to annual load growth which could tend to hurt results. Next, the structure of the ANN is determined. The number of input neurons is determined by the number of input variables, and the number of output neurons is determined in a similar fashion. No method exists for determining the required number of hidden layer neurons without experimenting. One popular method consists of adding neurons until no additional benefit is seen during the training period. Another approach "prunes" neurons that have low values on incoming and outgoing interconnections, because these neurons increase training time but do not play a large role in the forecast. Once all this has been completed, the neural network is trained with the algorithm of choice, a forecast is made, and the process begins again.

CHAPTER 3. THEORETICAL DEVELOPMENT

This chapter begins with a general discussion about linear programming. Following this information is a look the affine scaling algorithm, a powerful technique that is used to solve linear programs. Next, certain new extensions to the algorithm are explained.

A new artificial neural network whose training problem may be formulated as a linear program is described following the information about the affine scaling routine. The chapter concludes by describing the formulation of the training problem.

3.1 Linear Programming

Linear programming is an optimization technique that deals with formulating and solving problems of the following type:

$$\begin{aligned}&\underset{\mathbf{x}}{\text{minimize}}\ \ \mathbf{c}^t\mathbf{x}\\ &\text{subject to the constraints:}\\ &\mathbf{Ax} = \mathbf{b} \qquad\qquad (3.1)\\ &\text{and}\\ &\mathbf{x}_i \geq 0,\ \forall\ i = 1, 2, \ldots , n\end{aligned}$$

where **c** is an n-dimensional column vector representing the cost per unit element of column vector **x**, **A** is an m x n matrix, and **b** is an m-dimensional column vector. This type of problem formulation with linear costs and linear constraints occurs frequently in resource allocation and transportation problems where some minimum cost is desired while meeting certain constraints.

In most cases, the initial problem formulation will include some inequality constraints of the form

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \qquad (3.2)$$

that result because of the physical limitations of the problem. To deal with this, a dummy variable known as a slack variable is introduced into the problem. This requires the addition of a column to the **A** matrix plus the addition of one row to the **x** vector. This modification creates the equality constraint:

$$\sum_{j=1}^{n} a_{ij}x_j + s_i = b_i \tag{3.3}$$

where s_i is a slack variable. This form is consistent with the above standard form . A constraint that initially yields an equation where the left side is greater than or equal to the right hand side can be put into the standard form by multiplying the constraint by minus unity, and adding a corresponding slack variable as above. Two of the more common inequality constraints for LP problems require elements of **x** to remain below some maximum value

$$x_i \leq u_i \ \forall \ i = 1, 2, \ldots , n \tag{3.4}$$

or above some minimum value

$$x_i \geq l_i \ \forall \ i = 1, 2, \ldots , n \tag{3.5}$$

A variable without upper or lower bounds is known as a *free variable*. Several methods are presently available to implement free variables in linear programs, most of which require additional variables and hence additional memory and computation time. The most simple and common method involves splitting the free variable into two variables, one of which represents the positive portion of the variable and the other the negative portion. If x_i is a free variable, set

$$x_i = u_i - v_i \tag{3.6}$$

so that both u_i and v_i are nonnegative. This method allows linear programs to be more general but at the cost of an additional variable. Another method for introducing free variables into a linear program uses an additional vector to keep track of the signs for the different variables.

Any vector $\mathbf{x}_f$ that satisfies all the constraints imposed by the problem formulation including the upper bound and nonnegativity constraints is known as a *feasible solution*. A vector $\mathbf{x}_n$, which fails to satisfy at least one of the constraints is called an *infeasible solution*.

The region formed by the set of all feasible solutions is known as the *feasible region*. The feasible solution $\mathbf{x}^*$ which minimizes the *objective function* $\mathbf{c}^t\mathbf{x}$ is the *optimal solution*.

For any feasible solution $\mathbf{x}$, we may set

$$\mathbf{c}^t\mathbf{x} - z = 0 \tag{3.7}$$

where z is a fixed constant. The set of points satisfying this equation is a hyperplane in n-dimensional space (for a two variable problem this is a line). By changing z in a two variable problem, a family of parallel lines is established.

The feasible region for the two-variable problem

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}} \;\; -x_1 - 2x_2 \\ &\text{subject to:} \\ &x_1 + x_2 \le 7 \\ &0 \le x_1 \le 4,\; 0 \le x_2 \le 6 \end{aligned} \tag{3.8}$$

is shown in Figure 3.1. The feasible region for the linear program is represented by the shaded area in the figure and the cost function is shown for various values of *z*. The feasible region is guaranteed to be convex by the problem formulation. Notice that as z changes, a line parallel to the previous cost function results.

By forming a matrix **B** from the first m linearly independent columns of **A**, we may uniquely solve the equation

$$\mathbf{B}\mathbf{x}_b = \mathbf{b} \tag{3.9}$$

where $\mathbf{x}_b$ is the first m variables of $\mathbf{x}$. These variables are called *basic variables*, and are said to be in the *basis*. The remaining variables, all equal to zero, are said to be *nonbasic variables*. Letting the remaining elements of $\mathbf{x}$ be zero, we now have a feasible solution that lies on a vertex of the feasible region. This type of solution is known as a *basic feasible solution*. It can be shown that the optimal solution of a nondegenerate (one where no edge of

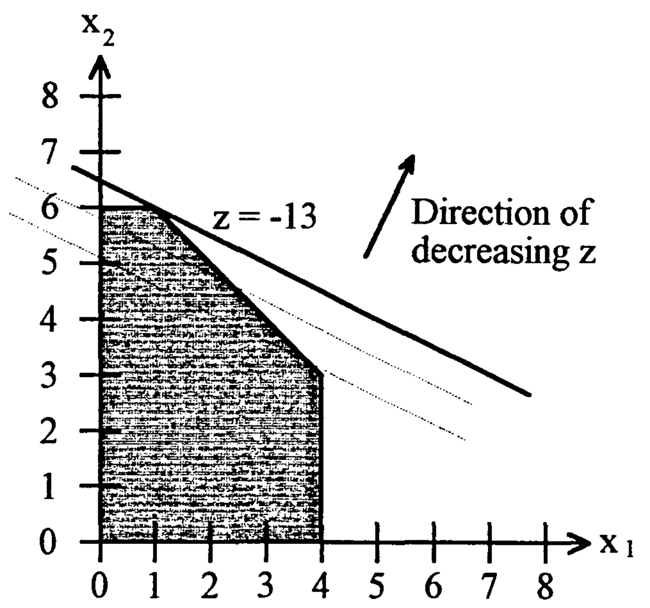

Figure 3.1. The feasible region and a family of cost vectors for the program in equation 3.8

the feasible region is parallel to the cost function) linear programming problem occurs at a basic feasible solution. An optimal solution for a problem of this type occurs where the cost function is at its minimum while the intersection of the cost function vector and the feasible region is nonempty. For nondegenerate problems, this is a single point. The point (1,6) is the optimal solution for the problem described in equation 3.8. It is not difficult to see in this two dimensional example that as *z* becomes more negative and translates through the feasible region, it will eventually intersect only at this vertex of the feasible region.

3.1.1 Piecewise-linear functions and restricted basis entry

Some functions we may wish to use in a linear program are nonlinear. Occasionally, it is possible to include these equations in the program by using a piecewise-linear (PWL) approximation to the function. A piecewise-linear approximation to a curve is shown in Figure 3.2.

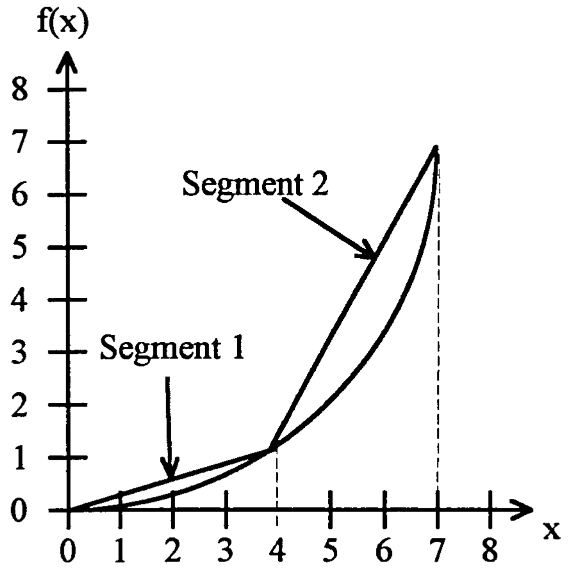

Figure 3.2. A piecewise-linear approximation to a curve

This PWL approximation contains two line segments, the first of which approximates the curve over the range [0,4], and the second of which is valid over [4,7]. In a linear program, the PWL function would be represented by two variables, x_1 and x_2. The variables that represent PWL segments are known as *segment variables*. Then we have the constraint that

$$x = x_1 + x_2$$

while the function value is determined by

$$f(x) = S_1 x_1 + S_2 x_2$$

where S_i is the slope of segment i. Also note that x_2 must be zero until x_1 is at its maximum, otherwise the function is meaningless. The different LP solution algorithms deal with this restriction in a variety of ways. The simplex algorithm, one of the most common, incorporates extra logic into the routine to implement this *restricted basis entry* restriction. A means for implementing restricted basis entry for another algorithm, the affine scaling algorithm, will be introduced for the first time later in this thesis.

The technique shown here for transforming a nonlinear function into a PWL approximation is taken from the fifth edition of *Introduction to Operations Research* by F.S. Hillier and G.J. Lieberman [5]. Interested readers should also see *Methods and Applications of Linear Programming* by L. Cooper and D. Steinberg [2] for a more advanced approach to this conversion.

3.1.2 Linear programming algorithms

Several algorithms have previously been developed to step from vertex to vertex along the boundary of the feasible region. The simplex algorithm, and later the revised simplex algorithm, begin at an initial basic feasible solution obtained as described above, and then proceed from basic feasible solution to basic feasible solution until the optimal solution is found. This mature methodology is capable of incorporating upper and lower bounds for variables and many other special cases. The simplex technique is guaranteed to find the optimal solution to the problem, but for extremely large problems it can be slow to find this result. Theoretically, the technique could have to iterate through every basic feasible solution to find the optimum. This number would be

$$\#\text{ Basic feasible solutions } = \frac{n!}{n!(n-m)!} \tag{3.10}$$

In reality, the simplex algorithm rarely requires this many iterations. A newer algorithm for linear programming travels through the interior of the feasible region to find the solution. This interior point LP algorithm is able to solve large problems faster than the simplex algorithm.

The interior point linear programming algorithm was introduced by Narendra Karmarkar at Bell Labs in 1984 [8]. The algorithm guarantees polynomial time solutions to linear programming problems. Karmarkar's algorithm differs from the simplex from the beginning in that it begins with an interior feasible solution rather than a basic feasible solution. It is called an interior point algorithm because it does not operate on the boundary of the feasible region. Karmarkar's algorithm rescales the variables to place the solution in the center of a scaled feasible solution, allowing it to take a large step toward the optimal solution. An example of the path an interior point algorithm might take to find a solution is shown in Figure 3.3. Notice that the algorithm may take several large steps across the feasible region to approach the solution, which will then be followed by smaller steps as the optimal solution is neared.

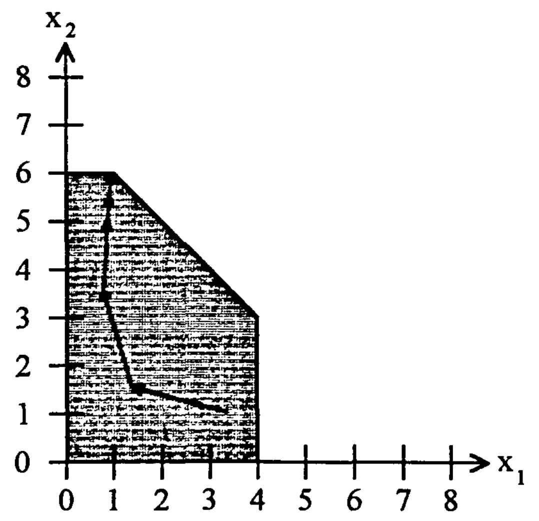

Figure 3.3. The path of an interior point algorithm for linear programming

The affine scaling algorithm is a modified version of Karmarkar's algorithm that was published in 1986, and this presentation follows the one presented in the original paper [17], except for the discussion about nonzero lower bounding and free variables that follows from [18]. The affine scaling algorithm is also an interior point algorithm but it differs from Karmarkar's algorithm in that it has a much more simple centering scheme and uses a gradient technique to decide the direction to move the solution. The affine scaling algorithm begins by finding an initial interior feasible solution. This is normally done by adding an artificial variable to the A matrix by setting

$$\mathbf{A}' = [\mathbf{A} : \mathbf{b} \ - \mathbf{A1}] \tag{3.11}$$

where **1** represents the vector of all ones. The initial feasible interior solution, $\mathbf{x}_0$, is then the (n+1)-dimensional vector of all ones. Assigning a very high cost (known as the big M method) to the artificial variable guarantees that it will go to zero in the optimal solution. If we assume the initial problem has a feasible solution, the optimal solutions for the initial problem and the big M formulation will be the same.

Gradient descent techniques will find an optimal solution to a problem in a reasonable number of iterations only if allowed to take large jumps through the feasible region. If a solution is pinned near a boundary then it is possible that only small steps may be taken towards the optimal solution. The affine scaling algorithm moves the solution away from the boundaries by normalizing all variables to unity. This makes sense because the feasible region is bounded by $\mathbf{x} \geq \mathbf{0}$ in the original formulation, so scaling all the variables to unity places them equidistant from at least one boundary. The scaling is done using the equation

$$\tilde{\mathbf{x}}_0 = \mathbf{D}_0^{-1}\mathbf{x}_0 \tag{3.12}$$

where $\mathbf{D}_0$ is a matrix with the elements of the initial solution, $\mathbf{x}_0$, on the diagonal. Clearly, $\tilde{\mathbf{x}}_0$ is the vector of all ones. The constraint matrix and cost vector must also be scaled to reflect the variable changes

$$\tilde{\mathbf{A}} = \mathbf{AD}_0 \tag{3.13}$$

and

$$\tilde{\mathbf{c}} = \mathbf{D}_0\mathbf{c} \tag{3.14}$$

The maximum rate of decrease of the transformed cost function would be obtained by moving in the negative direction of the cost vector $\tilde{\mathbf{c}}$. This, though, would lead to an infeasible solution by invalidating some equality constraints. The algorithm moves instead in the direction of the gradient projected onto the null space of $\tilde{\mathbf{A}}\,\mathbf{D}_0$. This projected gradient is

$$\mathbf{c}_P = \mathbf{P}\tilde{\mathbf{c}} \tag{3.15}$$

where

$$\mathbf{P} = \mathbf{I} - \mathbf{D}_0\tilde{\mathbf{A}}^T(\tilde{\mathbf{A}}\mathbf{D}_0{}^2\tilde{\mathbf{A}}^T)^{-1}\tilde{\mathbf{A}}\mathbf{D}_0 \tag{3.16}$$

is the projection matrix.

Next, the solution moves in the direction of $-\mathbf{c}_P$. The size of this move is chosen so that no element of $\tilde{\mathbf{x}}_1$ will become less than zero. The largest element of $\mathbf{c}_P$, γ, is used in the equation

$$\tilde{\mathbf{x}}_1 = \mathbf{1} - \frac{\alpha\,\mathbf{c}_P}{\gamma} \tag{3.17}$$

to guarantee feasibility for $\tilde{\mathbf{x}}_1$. Here $0 \le \alpha \le 1$ (typically, α is about 0.9) so $\tilde{\mathbf{x}}_1$ does not lie on a boundary, which would occur if $\alpha = 1$. Lastly, the algorithm maps back to get a new interior feasible solution

$$\mathbf{x}_1 = \mathbf{D}_0\tilde{\mathbf{x}}_0 \tag{3.18}$$

The algorithm is then repeated until some stopping condition is met. In abbreviated form the algorithm generates the sequence of points

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \frac{\alpha}{\gamma}\mathbf{DPDc} \tag{3.19}$$

The algorithm can also accommodate variables with finite upper bounds. Both the search direction and the step size must be changed with these additional restrictions, so

$$\mathbf{D} = \mathrm{diag}(\min(x_i,\, u_i - x_i)) \tag{3.20}$$

where $\mathbf{u}$ is the vector of upper bounds. The step size must be chosen to maintain the nonnegativity constraints and to keep the variables under their respective maximums. For this

$$\gamma = \max_i(\max(\frac{e_i \cdot \mathbf{Dc}_P}{x_i},\, \frac{e_i \cdot \mathbf{Dc}_P}{u_i - x_i})) \tag{3.21}$$

where e_i is the ith unit vector. The rest of the algorithm remains the same as before. In many problems, one or several variables have a nonzero lower bound. Two simple changes in the algorithm change it to allow such a restriction. The lower bound changes $\mathbf{D}$ to

$$\mathbf{D} = \mathrm{diag}(\min(x_i - l_i,\, u_i - x_i)) \tag{3.22}$$

where **L** is the vector of lower bounds. We similarly alter the equation for γ so

$$\gamma = \max_i(\max(\frac{e_i \cdot \mathbf{D}\mathbf{c}_P}{x_i - l_i}, \ -\frac{e_i \cdot \mathbf{D}\mathbf{c}_P}{u_i - x_i})) \qquad (3.23)$$

As in the upper bound version, the change seen in equation 3.42 alters the search direction for the algorithm by including the lower bound constraints while equation 3.43 guarantees that the step size will be small enough that it does not violate any of these constraints.

The freedom to have negative lower bounds in the affine scaling algorithm implies that we may now have free variables in programs without the cost of increasing the size of the problem or adding additional logic. This is done by setting the limits on free variables to extreme values. In practice, these bounds should not be set more than a few orders of magnitude larger than the bounds on other variables, due to numerical stability problems that may result from working with numbers over too large a range. With the ability to have negative variables, free variables may now be incorporated into a problem by setting the upper and lower bounds for these variables to extreme values. This, in turn, reduces the number of variables in linear programs because in the standard approach variables that are not constrained to be nonnegative are represented as two variables. Because the time required to solve a linear program is in part a function of the number of variables involved, the solution time for a given problem is reduced with this alteration. The nonzero lower bound procedure, along with upper bounding, is used here to introduce a method for implementing the restricted basis entry algorithm.

3.2 Restricted Basis Entry in the Affine Scaling Algorithm

The discussion in this section is presented in terms of implementing the restricted basis entry algorithm for a PWL curve. The procedure, though, is a general one and may be implemented wherever it is called for.

There are several ways to incorporate the restricted basis entry procedure into the affine scaling algorithm. One method sets the slopes of the segments not allowed to change equal to zero in the constraint matrix. Because the slope is zero, the gradient for that variable is very near zero, and upon update the variable does not change. While this method keeps the variable from changing significantly, it causes violation of the equality constraints for the pertinent output variable. Because this method does not result in feasible solutions it is not a reasonable solution to the problem.

The technique developed in this thesis implements the restricted basis entry by adjusting the upper and lower bounds of the variables according to some logic. A flow chart for the discussion in this section is shown in Figure 3.4. For the first case, where a variable x_1 must be near its maximum, *Max1*, before variable x_2 may depart from its lower bound, *min2*, the upper bound on x_2, *Max2*, is set to just above *min2*. This effectively sandwiches x_2 and keeps it near its lower bound. The conditions for x_2 to increase are that x_1 be very near *Max1*, and that the gradient of x_1 in the previous iteration is negative (thus indicating that it would increase if allowed to). Upon these conditions being met, the upper bound on the "entering" variable x_2 is set to its maximum, and x_1 must then be restricted to prevent it from decreasing significantly. This is done by changing *min1* to just below the present value of the variable and altering the affine scaling algorithm to allow nonzero lower bounds. If at some point x_2 decreases below some minimum value, x_1 must be allowed to decrease and x_2 must once again be held near its lower bound. The conditions that indicate this situation include a positive gradient for x_1 (thus indicating that it would decrease if allowed to) and a value less than some minimum for x_2. If these conditions are met, *Max2* is again set to a low value, and the lower bound for x_1 is set to its actual lower bound.

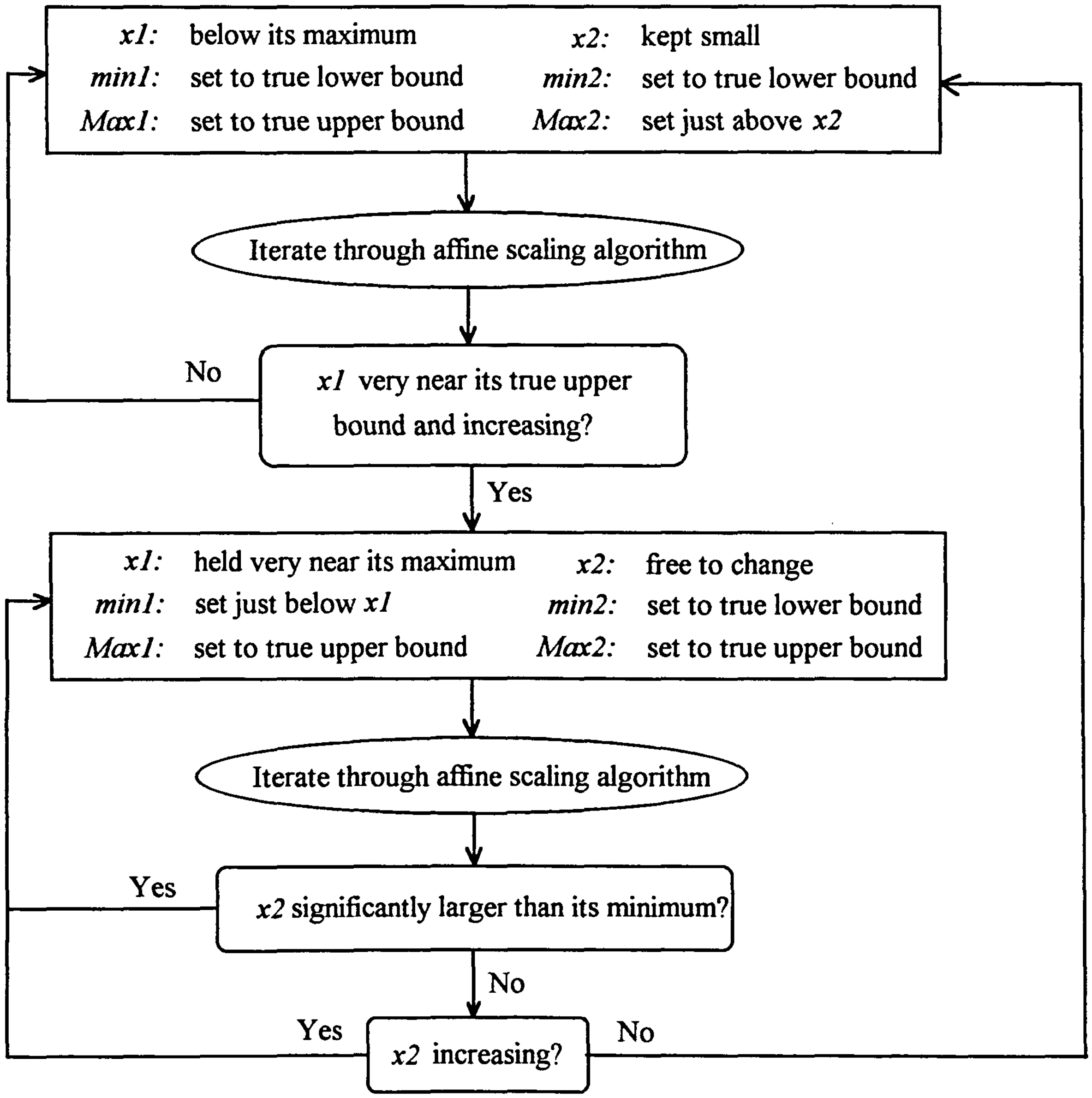
x1: below its maximum
x2: kept small
min1: set to true lower bound
min2: set to true lower bound
Max1: set to true upper bound
Max2: set just above x2
Iterate through affine scaling algorithm
No
x1 very near its true upper bound and increasing?
Yes
x1: held very near its maximum
x2: free to change
min1: set just below x1
min2: set to true lower bound
Max1: set to true upper bound
Max2: set to true upper bound
Iterate through affine scaling algorithm
Yes
x2 significantly larger than its minimum?
No
Yes
x2 increasing?
No

Figure 3.4. Flowchart for implementing restricted basis entry in the affine scaling algorithm

3.3 Behavior of the Affine Scaling Algorithm

The affine scaling routine behaves in an interesting manner that is valuable for the training algorithm to be developed in Section 3.4.3. Consider the six-variable linear program for which the optimal solution is known [12]

$$\begin{aligned}
&\underset{\mathbf{x}}{\text{minimize}}\ \ 2x_1 + x_2 + 3x_3 - 2x_4 + 10x_5 + 10^6 x_6 \\
&\text{subject to:} \\
&\qquad x_1 + x_3 - x_4 + 2x_5 = 5 \\
&\qquad x_2 + 2x_3 + 2x_4 + x_5 = 9 \\
&0 \le x_1 \le 7,\ 0 \le x_2 \le 10,\ 0 \le x_3 \le 1,\ 0 \le x_4 \le 5,\ 0 \le x_5 \le 3
\end{aligned} \tag{3.24}$$

that was solved using the affine scaling algorithm. Figure 3.5 shows a plot of the variables through several iterations of the algorithm. In the first iteration, the artificial variable is forced to near zero because of its high cost, while the other variables change only slightly. Not much happens during iterations two through four, and then suddenly on the fifth iteration the solution changes dramatically. The sixth iteration sees the variables change only slightly and the seventh through tenth iterations accomplish little. Depending on the precision required for the solution, these last two iterations may have been a waste. The behavior of the variables in this problem is not unusual for the algorithm.

Figure 3.6 shows a plot of the absolute error from the optimal solution over all the variables. The error decreases slightly in the first few iterations. Upon the fifth iteration the error drops off significantly with only modest improvements taking place with subsequent iterations as the solution is neared. For larger problems, the error may begin to drop significantly with the first iteration, and the rate of decrease may be less dramatic than in this small example, but the basic performance is similar. The big improvements in the objective function take place in the early iterations, and as the optimal solution is approached, the rate of improvement in the objective function decreases along with step size as a point of diminished returns for additional calculations is reached.

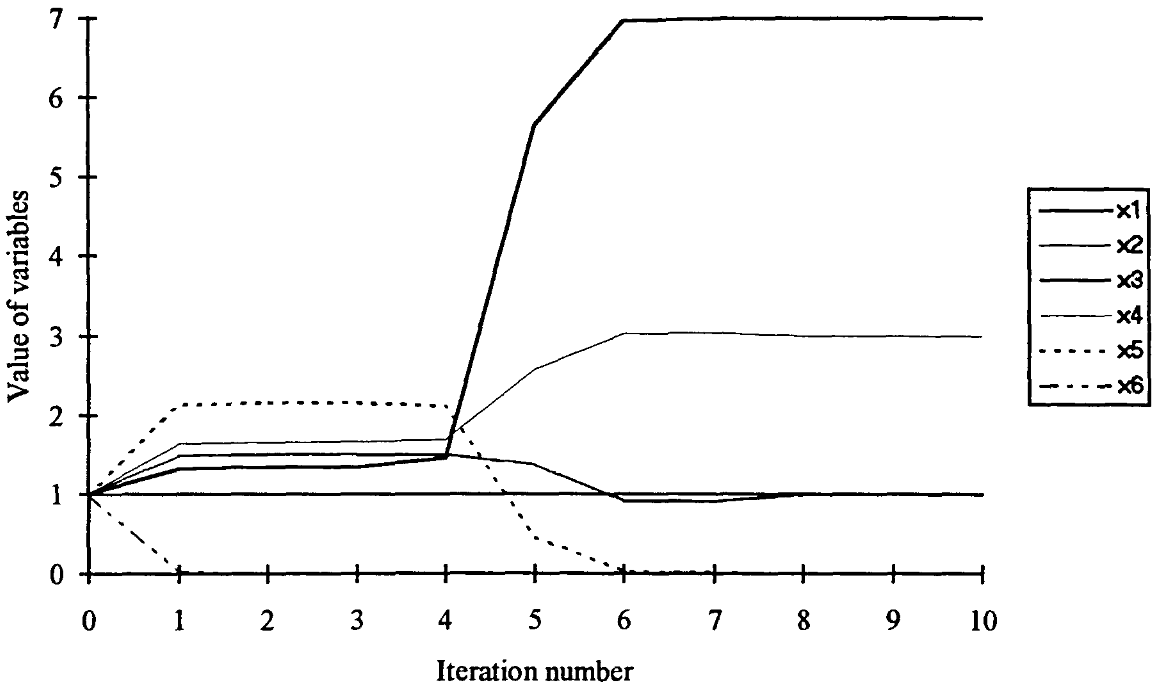

Figure 3.5. Behavior of the variables in the affine scaling algorithm.

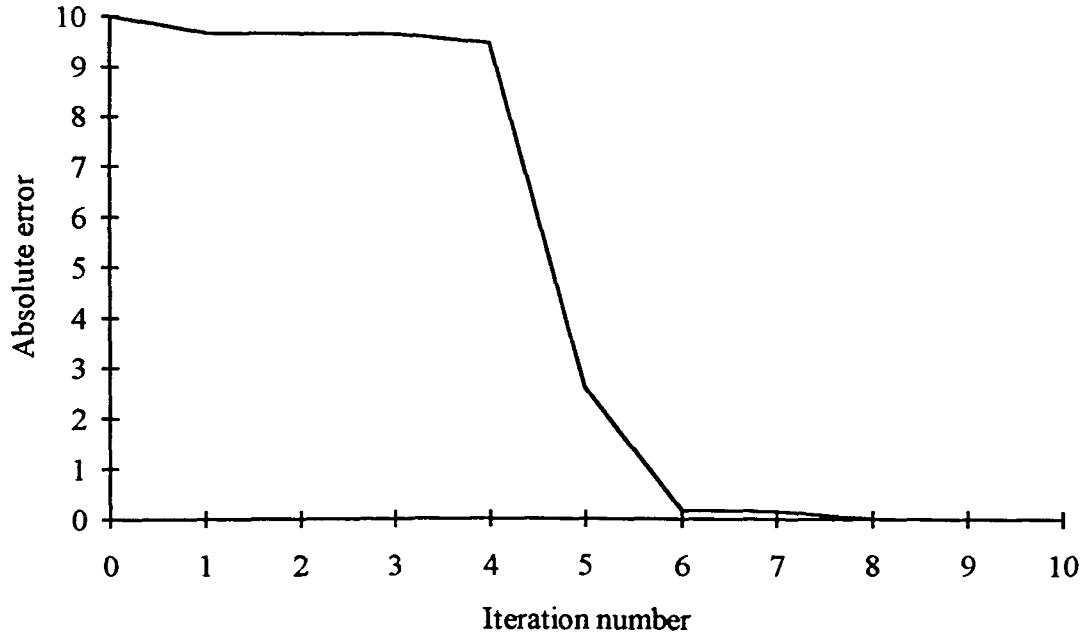

Figure 3.6. Behavior of the error in the affine scaling algorithm

3.4 Neural Network Training

Now that the affine scaling has been introduced and extended to solve a wider variety of problems, it is appropriate to take a look at a specific application of the extended algorithm. The algorithm is applied in this thesis to the problem of training an artificial neural network. In the remainder of this chapter is a more detailed explanation of the ANN model, development of the LP formulation for training the network, a method for finding an initial interior feasible solution for the problem, and a means for determining when to stop the algorithm.

3.4.1 Neural network structure

The artificial neural network described in this thesis is a feedforward model with the bias connected to each of the hidden and output layer neurons. The network is assumed to be fully connected. The input and output layer neurons have a linear transfer function, that is, the output of the neuron is the same as its input, while the hidden layer neurons have a piecewise-linear transfer function

As mentioned earlier in this work, hidden layer neurons in feedforward networks usually have smooth, continuously differentiable transfer functions to accommodate the gradient techniques that are used to train the network. That type of transfer function is not useful for a linear programming formulation, because it cannot be represented explicitly in the program. The transfer function for the hidden-layer neurons in the network studied here is shown in Figure 3.7. This transfer function is a piecewise-linear function, to accommodate the formulation, and it has three segments. In equation form the transfer function is

$$S(\sigma) = \begin{cases} 10\sigma, & \text{for } 0 \leq \sigma \leq 0.03 \\ 0.3 + 5(\sigma - 0.03), & \text{for } 0.03 \leq \sigma \leq 0.05 \\ 0.4 + 2(\sigma - 0.05), & \text{for } 0.05 \leq \sigma \leq 0.35 \end{cases} \tag{3.25}$$

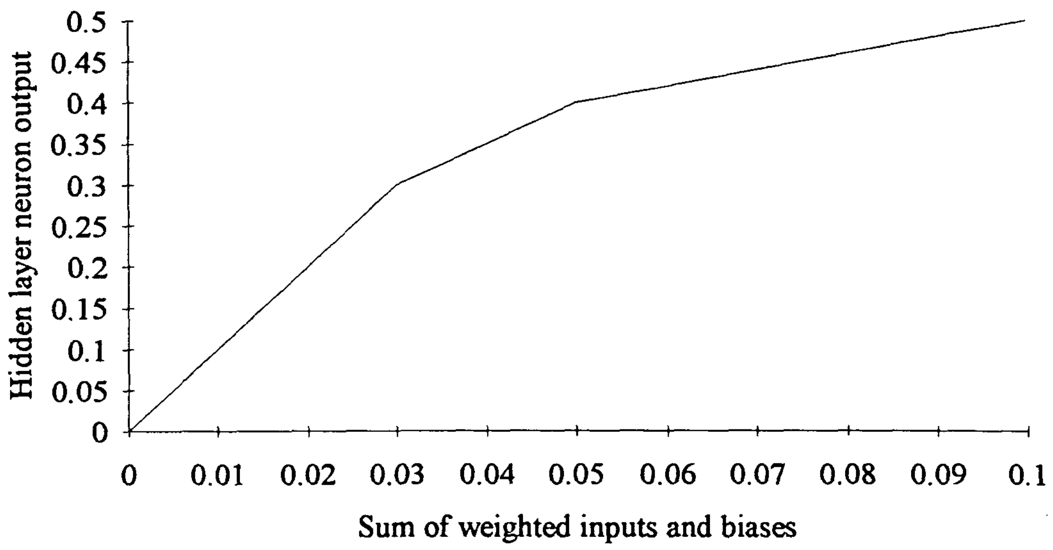

Figure 3.7. The transfer function used in the hidden layer neurons

where σ is the sum of the weighted inputs and bias. This transfer function was selected because it is similar to the sigmoid in the first quadrant. However, there is no reason to believe that this PWL function is superior to others.

3.4.2 Program formulation

Each segment of the transfer function represents a different variable in the linear programming formulation. The variable x_1 is assigned to the segment from $0 \leq \sigma \leq 0.03$, x_2 is for $0.03 \leq \sigma \leq 0.05$, and x_3 is for $0.05 \leq \sigma \leq 0.35$. Let *nsegs* be the number of segments in the PWL curve. For an input pattern $\mathbf{V}_h$ of length *nin*, and hidden neuron j, the segment variables must satisfy the relationship

$$\sum_{g=1}^{nsegs} x_{jg} - t_j - \sum_{i=1}^{nin} w_{ij} v_{ih} = 0 \tag{3.26}$$

where t_j is the bias term for neuron j, w_{ij} is the weight from input neuron i to the hidden neuron, and v_{ih} is element i of the input pattern vector. Every hidden layer neuron in the linear program must satisfy equation 3.26 for every pattern. This is the first of two kinds of functional constraints that are contained in the **A** matrix.

The output S_j of hidden-layer neuron j is determined by the segment variables and the slope m_g of each segment

$$S_j = \sum_{g=1}^{3} x_{jg} m_g \tag{3.27}$$

The output of neuron k in the output layer is the same as the input to these neurons since they are linear

$$\sigma_k = t_k + \sum_{j=1}^{d} w_{jk} S_j \tag{3.28}$$

and

$$S_k = \sigma_k \tag{3.29}$$

3.4.3 Derivation of the training algorithm

The back-propagation algorithm for neural network training seeks to minimize the mean-squared error (MSE) over the set of training patterns

$$\text{minimize: MSE} = \frac{1}{2} \sum_n (T_n - O_n)^2 \tag{3.30}$$

where T_n is the target output specified by the training data and O_n is the actual output. This particular error minimization is not practical for linear programming because of the squaring term. Instead, the linear program minimizes the sum of the absolute errors over the training patterns

$$\text{minimize:} \quad \sum_n |T_n - O_n| \tag{3.31}$$

Because linear programming does not accommodate absolute values, each error term is replaced by two variables, e^+ and e^-, both of which are constrained to be positive. The resulting equality is

$$T - O = e^+ - e^- \tag{3.32}$$

so equation 3.31 becomes

$$\text{minimize:} \quad \sum_{n} e_{n}^{+} + e_{n}^{-} \tag{3.33}$$

where for output neuron *k* and pattern *h*, we have the following equality

$$T_{kh} = e_{kh}^{+} - e_{kh}^{-} + t_{k} + \sum_{j=1}^{d} w_{jkh} S_{jh} \tag{3.34}$$

where d is the number of hidden-layer neurons. S_{jh} is not included explicitly as a variable in the linear program to reduce the number of variables. Instead, from we have

$$T_{kh} = e_{kh}^{+} - e_{kh}^{-} + t_{k} + \sum_{j=1}^{d} \sum_{g=1}^{3} w_{jkh} x_{jg} m_{g} \tag{3.35}$$

This is the second of the two types of equations included in the constraints.

A closer look at equation 3.35 reveals that it violates the linear programming assumption of additivity that does not allow two variables in the program to multiply each other. This violation occurs when the weights from the hidden layer to the output layer multiply the segment variables. It is not possible in a single linear program to adjust both the weights from the input layer to the hidden layer while at the same time adjusting the weights from the hidden layer to the output layer. The solution to this problem is a decomposition of the program from a single linear program into two linear programs.

3.4.4 Operation of the training algorithm

This discussion follows the diagram shown in Figure 3.8. In the first program–referred to as the input-hidden LP (IHLP)–the weights for the paths from the bias and the input layer neurons to the hidden layer neurons, as well as the bias connections to the output layer neurons are allowed to vary while the weights from the hidden layer to the output layer are held constant. The weights from the hidden to the output layers are not explicit in the IHLP but are implied by scaling the slopes of the segment variables, since for any constant β,

$$\beta S_{j} = \sum_{g=1}^{3} \beta x_{jg} m_{g} \tag{3.36}$$

This program iterates a set number of times, decreasing the absolute error between the target output and the actual output. If any constraints are violated (i.e., upper or lower bounds, etc.), the program ends and uses the most recent feasible solution to operate the network. the restricted basis entry procedure is also performed here to select the proper transfer function segment to operate on. Next the outputs from the hidden layer neurons are extracted using equation 3.27. This information, plus the output neuron bias terms and the error portion of the **x** vector for the IHLP is used to establish the second linear program, known as the hidden-output LP (HOLP). This program also iterates a set number of times, or until infeasibility occurs. The variables representing error and output bias terms are then replaced into the **x** vector in the IHLP and the segment slopes are scaled in that programs constraint matrix. The process continues until the fixed number of overall iterations, L, is reached, or until infeasiblity occurs.The input-hidden linear program is

$$\begin{aligned}
&\text{minimize: } \sum_{h=1}^{npats}\sum_{k=1}^{nout} e^{+}_{kh} + e^{-}_{kh} \\
&\text{subject to the constraints} \\
&T_{kh} = e^{+}_{kh} - e^{-}_{kh} + t_k + \sum_{j=1}^{nhid}\sum_{g=1}^{nsegs} w_{jkh} S_j,\ \forall\, h = 1,\dots,npats, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \forall\, k = 1,\dots,nout \\
&\sum_{g=1}^{nsegs} x_{jgh} - t_j - \sum_{i=1}^{nin} w_{ij} v_i = 0,\ \forall\, j = 1,\dots,nhid, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \forall\, h = 1,\dots,npats
\end{aligned} \tag{3.37}$$

where w_{jkh} from the hidden layer to the output layer is held constant through the iteration.

The hidden-output linear program is similar to the first except the w_{ij} are held constant. The initial feasible solution for each iteration of this program comes directly from the first program every time the algorithm switches from the first program to the second. From the

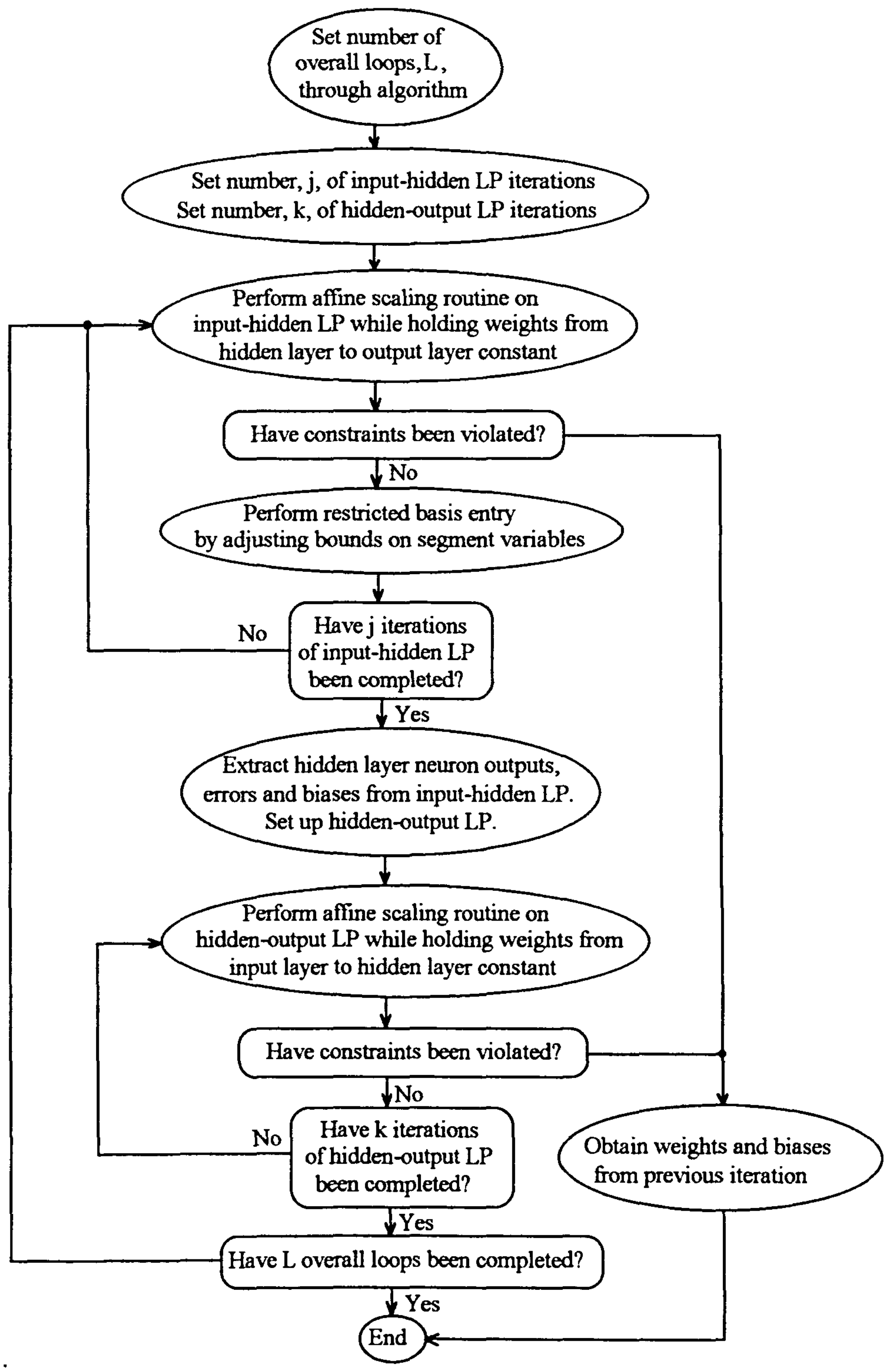
Set number of overall loops, L, through algorithm
Set number, j, of input-hidden LP iterations
Set number, k, of hidden-output LP iterations
Perform affine scaling routine on input-hidden LP while holding weights from hidden layer to output layer constant
Have constraints been violated?
No
Perform restricted basis entry by adjusting bounds on segment variables
No
Have j iterations of input-hidden LP been completed?
Yes
Extract hidden layer neuron outputs, errors and biases from input-hidden LP. Set up hidden-output LP.
Perform affine scaling routine on hidden-output LP while holding weights from input layer to hidden layer constant
Have constraints been violated?
No
No
Have k iterations of hidden-output LP been completed?
Obtain weights and biases from previous iteration
Yes
Have L overall loops been completed?
Yes
End

Figure 3.8. Flow chart for the neural network training algorithm

first program we get the second program

$$\begin{aligned}&\text{minimize: } \sum_{h=1}^{npats}\sum_{k=1}^{nout} e_{kh}^{+} + e_{kh}^{-}\\ &\text{subject to the constraints}\\ &T_{kh} = e_{kh}^{+} - e_{kh}^{-} + t_k + \sum_{j=1}^{nhid}\sum_{g=1}^{nsegs} w_{jkh} S_j, \ \forall\, h = 1,\ldots,\text{npats},\\ &\qquad\qquad\qquad\qquad\forall\, k = 1,\ldots,\text{nout}\end{aligned} \tag{3.38}$$

The size of the matrices and vectors for the first linear program is a function of the number of training patterns, the number of input, hidden and output neurons, and the number of segments in each piecewise-linear transfer function. For each of the *npats* training patterns, we must include *nhid* constraints (from equation 3.23) on the hidden layer neurons and *nout* constraints (from equation 3.32) on the output neurons. Thus the number of rows in $\mathbf{A}_{IHLP}$ is

$$\text{Rows in } \mathbf{A}_{IHLP} = npats\,(nout + nhid) \tag{3.39}$$

$\mathbf{A}_{IHLP}$ has two error variables per output neuron per training pattern and each segment of every hidden layer neuron must be represented for all patterns. Additionally, the weights and biases–free variables–are each represented by a variable. The number of columns in $\mathbf{A}_{IHLP}$ is

$$\text{Col's in } \mathbf{A}_{IHLP} = 2 * nout * npats + nhid(1 + nin + nsegs * npats) + nout(1 + nhid) \tag{3.40}$$

The size of the hidden-output LP is considerably smaller than the input-hidden LP. The constraints that apply to the segment variables in the input-hidden LP do not apply, so the number of rows is determined by

$$\text{Rows in } \mathbf{A}_{HOLP} = npats \times nout \tag{3.41}$$

while the number of columns is

$$\text{Columns in } \mathbf{A}_{HOLP} = nout(1 + 2npats + nhid) \tag{3.42}$$

As mentioned above, the initial feasible solution for the HOLP comes from the IHLP, and vice versa, once the algorithm is under way. Finding the first initial feasible solution to the IHLP, however, requires some additional work.

3.4.5 Finding an initial feasible solution to the IHLP

The method described earlier for finding an initial interior feasible solution to the linear program sets the initial solution to a vector of all ones for the affine scaling algorithm . Because the segment variables in the PWL transfer function have upper bounds less than one, the previously described method is not viable for this application. In this work, the weights and biases are chosen randomly in such a manner that the initial solution is interior feasible. The sum of weights and biases is less than the maximum allowable value for the first segment.

The total number of inputs to a hidden layer node is *nin*+1, which includes the input variables and a bias term. Keeping in mind that all the inputs have been scaled between zero and one, if we generate a column vector **r** of length (*nin*+1)**nhid* that is composed of uniform random numbers in (0,1) and scale

$$\mathbf{r}_{\text{scaled}} = \frac{\max(x_1)\ \mathbf{r}}{nin+1} \tag{3.43}$$

where $\max(x_1)$ is the maximum value for the first PWL segment. Then the elements of this vector may be used as the weights and biases for the connections into the hidden layer neurons. The first *nhid* elements of this vector are used for the bias terms and the rest are for the inputs. For every hidden layer neuron we have an inequality of the form

$$\sigma < \frac{(nin+1)\max(x_1)}{(nin+1)} = \max(x_1) \tag{3.44}$$

Setting all the other segment variables equal to a small value, ε, x_1 is

$$x_1 = \sigma - \varepsilon\ (nsegs-1) \tag{3.45}$$

The only variables yet to be determined are the error variables. These are extracted from the equation

$$error = T - bias - \sum_{j=1}^{nhid} \mathbf{x}_{seg}^{'} \mathbf{s} \tag{3.46}$$

where the output neuron bias terms are taken from a uniform distribution in (0,1). The variable $\mathbf{x}_{seg}$ is a column vector of length *nsegs* that contains the segment variables for a single hidden layer neuron, and $\mathbf{s}$ is a column vector of the slope segments. If *error* is greater than or equal to zero

$$\begin{aligned} e^{+} &= error + \varepsilon \\ e^{-} &= \varepsilon \end{aligned} \tag{3.47}$$

and vice versa if *error* is less than zero. This procedure is done for every output neuron for every training pattern.

Once the algorithm has cycled through the HOLP, the new feasible solution for the IHLP comes from the second program

3.4.6 Stopping the training algorithm

A proof is performed in [17] showing that the affine scaling algorithm converges in the limit to one point and that this point is the optimal solution to the linear program. This proof involves what is known as *dual linear programs*. Simply put, the dual linear program is an LP that can be derived from the original problem in equation 3.1 (known as the *primal linear program*). The dual LP to that problem is

$$\begin{aligned} &\text{maximize:} \quad \boldsymbol{\lambda}^{T}\mathbf{b} \\ &\text{subject to:} \quad \boldsymbol{\lambda}^{T}\mathbf{A} \le \mathbf{c}^{T} \end{aligned} \tag{3.48}$$

where the dual vector λ is a free variable [12]. The initial solution to a dual problem is an infeasible solution to the primal. The optimal solution of the dual problem is primal feasible and is the same as the optimal solution to the primal. If the dual problem is worked on at the same time as the primal, the distance between their respective variable vectors is a measure of how far away from the optimal solution the programs are. At the optimum, this distance is zero. The stopping rule specified in the paper by Vanderbei et al. [17] calculates this distance but requires a significant amount of computation. The time required to operate on the dual problem may be better spent performing an additional iteration of the LP.

The decision to stop the algorithm involves a choice between improving the solution and running the risk that due to numerical instability the solution will become infeasible with the next iteration. It is important to remember here the behavior of the affine scaling routine as discussed in Section 3.3, where it was shown that after several iterations, a point of diminishing returns is reached whereby additional iterations improve the objective function only slightly. It is also valuable to recall that the training algorithm is an iterative process with two separate LPs. Based on these facts, a heuristic stopping rule is used in this thesis where the overall training algorithm iterates a manually controlled, fixed number of times. This number is set high enough that the solution becomes infeasible (thus indicating that optimality had been neared in the previous iteration) at some point in the training session. This near-optimal solution is then used for operation of the network. The number of iterations of both the IHLP and the HOLP are kept low enough that unnecessary effort is not expended taking small steps toward optimality. The number of iterations for each of these LPs in each overall iteration is also set manually.

CHAPTER 4. IMPLEMENTATION AND RESULTS

In this chapter the new artificial neural network and training algorithm and a more traditional ANN trained by back-propagation are applied to the short-term load forecasting problem. The chapter begins with the determination of the forecast period, then explains the selection and preprocessing of training data. Next, the training process is reviewed and forecast results are presented.

4.1 Determination of Forecast Period

Hourly load, time-of-day, day of week, wind speed and humidity data were available for a one year period from a Midwestern utility with a summer peaking load. It was decided that the forecast would be performed for the morning of a single weekday to keep the training time to a minimum, while adequately testing the algorithm. The precedent for using separate ANNs for a.m. and p.m. forecasts was established in [3]. Load profiles for both winter and summer days were analyzed and it was found that the winter load profile had more detail than its summer counterpart. It was decided that the forecast should be performed for a winter day under the assumption that the ANN would have no problem with a summer day's forecast if it could catch the minutiae of the winter load.

4.2 Selection of Input Variables

Several methods have been employed historically in the selection of input variables. The set of input variables is system dependent and also varies with the type of forecast being made.[11]. A correlation analysis was done in that paper to determine the amount of historical load data required for a 24-hour forecast for two separate utilities. The amount of data required varied between the utilities but showed little seasonal variation. Some forecasters start with a large set of input variables to train the ANN, and then the ANN is pruned by eliminating the inputs that have low connection weights because the low weights indicate little relation between the variable and the load. Heuristics may also be used to determine some of the input variables. The knowledge-based load forecasting paper [16] gives a good explanation of the heuristic selection of input variables.

The amount of data selected for training the ANN reported in this thesis was kept to a minimum, as stated above. The forecast was to be made for a Thursday, and so the training data were taken from the two Thursdays that immediately preceded this day. The input variables included time-of-day, previous hour's load, and temperature for the forecast hour. It should be noted that the number of training patterns–24–and the number of network inputs–3–are about 15% and 10% respectively of the amount that would normally be used to train a network.

4.3 Preprocessing of Data

Before training the ANN, it was necessary to preprocess the training data to help maintain the numerical stability of the training process. If the input and output values are scattered across a large range of values (i.e., the temperature may be 12 degrees while the load is 1500 megawatts) some of the calculations that take place in the algorithm–the inversion required in the calculation of the projection matrix in particular–could end up handling numbers over a much broader range. This can introduce errors from roundoff and the finite limits on the machine precision. In order to overcome this problem, the data were scaled to keep them all in a small range of values.

There are two common practices for preprocessing time-of-day information for input to the ANN. The first method represents each hour by a discrete variable. For each training pattern, the variable for the corresponding hour is set to unity, while all the other time-of-day variables are held to zero. For a 12 hour forecast, 12 separate inputs to the ANN would be required. This *one-of-n* format offers very good discriminatory power between different hours of the day, but at the cost of severely increased training time because of the additional variables. The preprocessing used in this thesis is a simple scaling procedure that keeps the time-of-day as a single continuous variable and scales the data between some minimum and maximum values. The bounds used here were 0.1 and 0.9, and the scaling parameters were found by solving

$$0.9 = (\text{MAX} * \text{A}) + \text{B} \tag{4.1}$$

and

$$0.1 = (\text{min} * \text{A}) + \text{B} \tag{4.2}$$

simultaneously, where MAX is the largest value in the historical data, and min is the smallest measurement in this data. The other input variables, previous hour's load and forecast hour's temperature were scaled in the same manner as the time-of-day input.

4.4 Network Structure

Both networks trained were three-layer, feedforward types. The input layers each had three neurons with linear transfer functions. The ANN trained with the new algorithm had two hidden layer neurons with the PWL transfer function. The more traditional network also had two hidden layer neurons, but these had a sigmoidal transfer function. Each network had a single output neuron with a linear transfer function and the biases were connected to each hidden and output layer neuron.

4.5 Neural Network Training

The new training algorithm was coded into *Matlab*™. This code begins by setting up the pertinent matrices and vectors, and then finds an initial, interior feasible solution to the problem, as described in Section 3.4.5. It then performs the iterative algorithm for training the network. The number of overall iterations was set manually, as was the ratio of the number of iterations of the first linear program to the second program. The ratio of the two programs was set at 2:4, that is, the first LP went through two iterations before the hidden layer neuron outputs were extracted and the second LP began. After four iterations of the affine scaling routine on the second LP, the segment variables in the first LP were scaled and that LP began again. The 2:4 ratio resulted from a trial-and-error process that attempted to find the best solution possible without having the solution become infeasible. This process was set to be run three times, but while performing the algorithm on the first LP during the third loop, the solution became infeasible and the process stopped. The weights from the previous iteration, where the solution was feasible, were used to build and test the network.

The plot of overall absolute error -vs- number of iterations is shown in Figure 4.1. Note that in iterations 1-2 and 6-7 the error decreases significantly. These iteration numbers correspond to the IHLP. This program adjusted nine different weights while the HOLP adjusted only three. The difference in error reduction between the two LPs is probably due to this difference. The rate of error decrease slowed as predicted, with later iterations achieving less gain before infeasibility occurs.

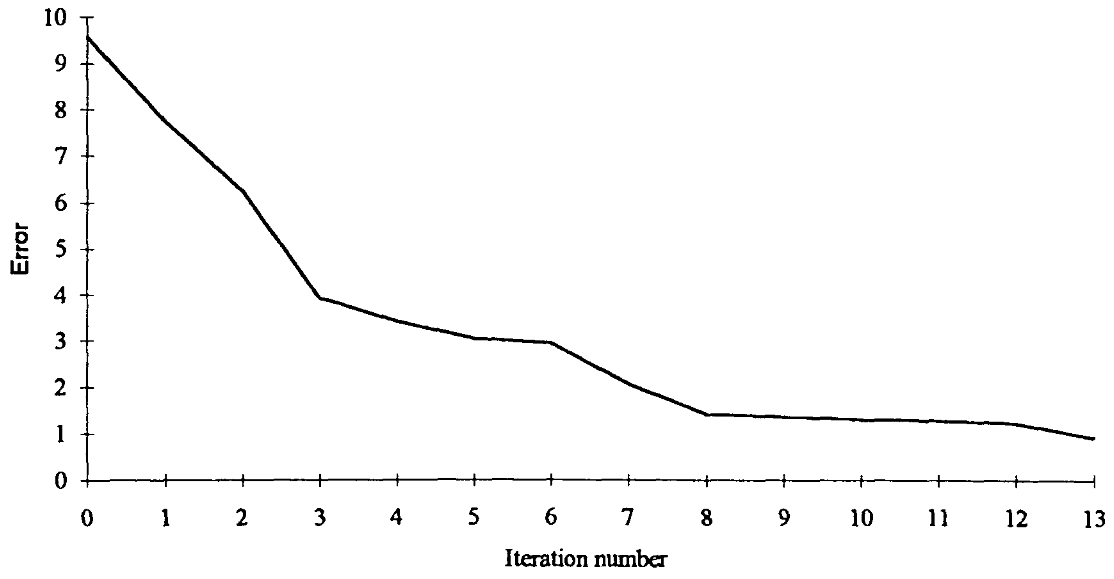

Figure 4.1. Error over the set of training patterns -vs- number of iterations

The step size parameter, α, became progressively smaller during the algorithm to keep the solutions inside the feasible region. In the first LP, α began at 0.8 and decreased by 0.02 each time through the entire loop. Because only two main loops were completed, this value ended up at 0.78. Similarly, this parameter in the second LP began at 0.9 and decreased by 0.05 with each full loop and ended up at 0.85.

The ANN trained with the back-propagation algorithm required 40,000 iterations of that algorithm to reach a point where error no longer decreased. Training time for the new ANN was 8 minutes, 30 seconds. Training time for the back-propagation ANN was 2 minutes, 45 seconds.

4.6 Results

The accuracy of the forecasts made by each network were adequate considering the minimal amount of training data used. The plot comparing the forecasts with the actual load is shown in Figure 4.2 Average error for the PWL network trained with the LP algorithm was 4.0%, with a peak error of 8.7%, while the average error for the back-propagation ANN was 2.6% with a peak error of 6.6%.

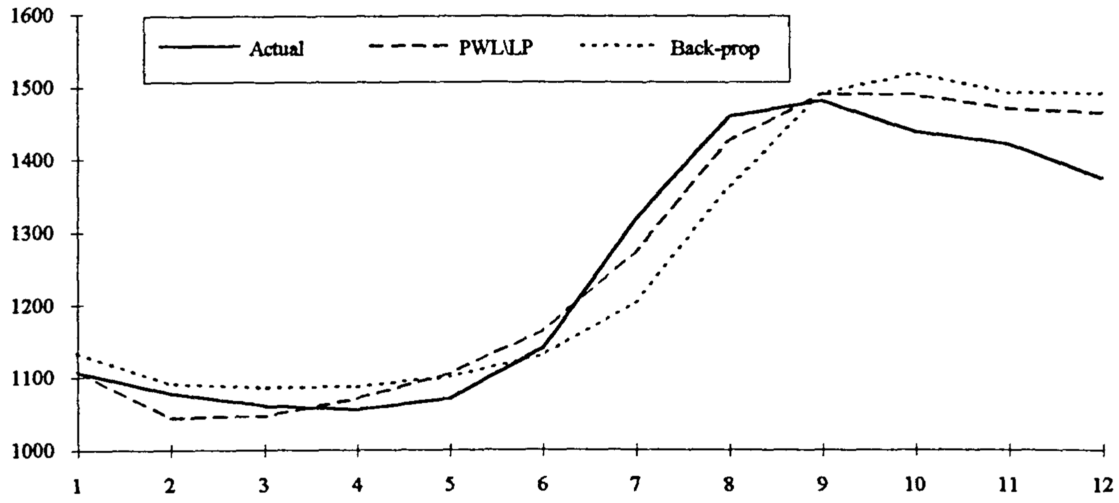

Figure 4.2. Plot comparing forecasts to actual loads

It is interesting to note in Figure 4.2 that both forecasts performed poorly during the latter part of the period, specifically for hours 9-12. In fact, during this time the forecasts both increase slightly when the load is decreasing. The cause of this is apparent when one compares the load profile for the forecast day to those of the training days. This is shown in Figure 4.3. In both the training days the load continues to rise until hour 11, while on the

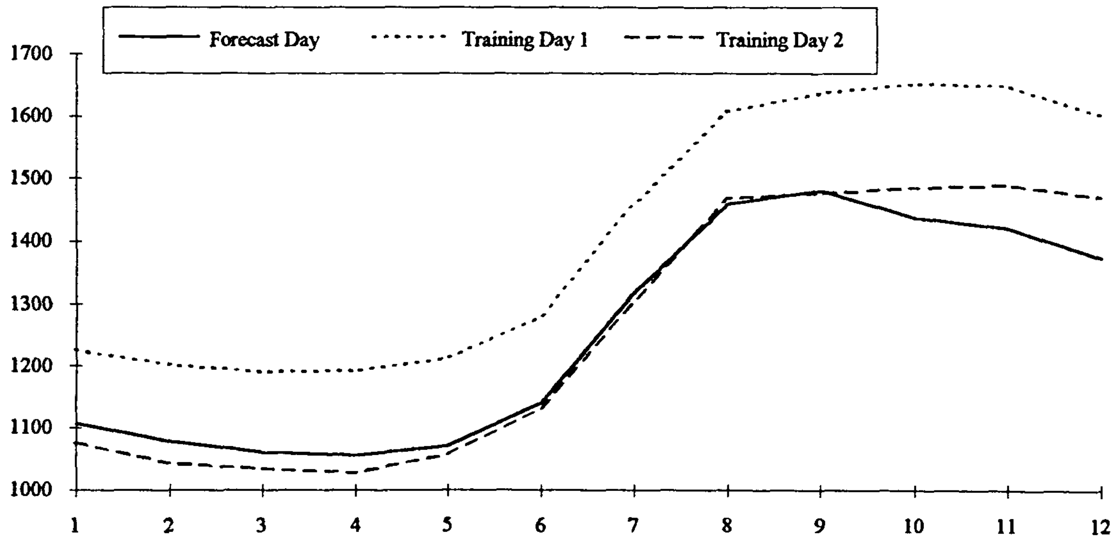

Figure 4.3. Training and forecast days' loads

forecast day the load begins to decrease at hour 9. This difference reinforces the value of additional training data for the networks, since neither network will respond properly to a phenomena it has not seen before at all.

CHAPTER 5. CONCLUSION AND SUGGESTIONS FOR FUTURE WORK

5.1 Conclusions

This work presented a modification to the affine scaling interior point algorithm for linear programming. The extension developed in this thesis allows the restricted basis entry to be used by the affine routine. This is a valuable extension that will allow the technique to work with a wider variety of problems, especially those that incorporate piecewise-linear functions. Also presented in this work was an artificial neural network with hidden layer neurons having piecewise-linear transfer functions to facilitate the formulation of the network training problem as a linear program, and a training algorithm for this network that uses the modified affine scaling algorithm. Finally, a power system short-term load forecast was performed using the new ANN and training algorithm.

The restricted basis entry method is implemented by using variable upper and lower bounds. A variable that is not allowed to rise far from its minimum is held there by setting the upper bound on the variable to a value just above the lower bound thus sandwiching variable. A variable that is to be held near its maximum is constrained in a similar manner by adjusting its lower bound to a value just below the upper bound.

The artificial neural network presented in this thesis is a feedforward type with linear transfer functions in both the input and output layers. The hidden layer has a piecewise-linear transfer function. Each segment of the PWL function may be represented as a separate variable in a linear program. The training of the ANN was formulated as a linear program with the goal of minimizing the absolute error over the training data. This training routine is actually drawn up as decomposed linear program. The first program is used to determine the weights between the input and hidden layer neurons. In this stage, the algorithm makes use of the new restricted basis entry approach in the affine scaling routine to step through the PWL segments. The outputs of the various hidden layer neurons are extracted from the first linear program–which finds the weights for the paths between the input and hidden layer neurons–and used in a second linear program to determine the weights for the connections between the hidden layer and the output layer. Information from this program is then extracted after several iterations and used in the first program. A unique scaling process is used to adjust the slopes of the PWL segments in the first program to simulate the weighting of connections from the hidden layer to the output layer.

Finally, the network and training algorithm are used to perform a short-term load forecast. This type of forecast is extremely valuable to electric utilities, because it can be used to schedule equipment and personnel in an economic fashion. The forecast performed was for a 12-hour period on a Thursday morning. Training data was taken from only two Thursdays prior to the forecast period, and consisted of temperature, time of day, and previous hour's load. The forecast was quite adequate considering so little data was used for training. For the PWL network and LP training algorithm, the average error was 4.0% and the peak error was 8.7%. This compares with 2.6% and 6.6% respectively, for a forecast performed by a network with sigmoidal hidden layer transfer functions that was trained by back-propagation. Training time for the LP algorithm for this network was 8 minutes, 30 seconds. The back-propagation algorithm required 2 minutes, 45 seconds to converge. The need for additional training data was illustrated when both ANNs made some significant errors because the latter portion of the forecast day load profile was different from the training days' profiles.

Convergence of the LP algorithm to a "good" solution prior to exiting the feasible region or failing because of numerical instability requires patience. Parameters such as step size, upper and lower bounds on free variables, the ratio of iterations between the two programs, and the number of overall iterations must be adjusted manually at present to obtain this type of solution. The restricted basis entry procedure presented in this thesis is operational but the tendency of the algorithm is to keep the solution on the first segment of the PWL function.

This may be overcome by keeping this segment short and keeping the limits on the upper and lower bounds for the output neuron bias terms small. Additionally, increasing the number of iterations of the first LP should force the PWL function values to increase.

5.2 Suggestions for Future Work

In its present state, the optimization technique presented here for training an ANN is powerful but requires some refinement to guarantee convergence to a solution with minimal error prior to the solution becoming unstable. This work should focus in part on the PWL transfer function used in the hidden layer neurons. There is no reason to suppose that the function used here is optimal. Possibly several different transfer functions should be used in the same network to see which ones tend to affect the solution the most. Additional work should be performed to determine algorithm parameters that encourage convergence of the program before an unstable or infeasible condition is reached. The initial feasible solution found in this work places the sum of the inputs to the hidden layer to a low value. The final solution to the training problem, though, will have sums at the hidden layer neurons spanning the range from near zero to near unity. With this in mind, it would make sense to start with the initial feasible solution distributed over a larger range. This should significantly reduce the number of iterations required to train the network and may help to obtain a better solution.

Because the amount of time required to train an ANN is one of the first criteria involved in selecting an ANN, incorporation of sparse matrix techniques to perform the affine scaling algorithm must be a suggestion for future work. The constraint matrix used here can be huge, but for a decent sized problem much fewer than one percent of the matrix consists of nonzero entries. In addition, the location of the nonzero elements in the constraint matrix and the cost vector may easily be known because of the problem formulation. Sparse matrix techniques could cut the computation time by at least 90% for a reasonable sized problem without affecting the accuracy of the solution.

The application of ANNs to the short-term load forecasting problem is becoming common. The key elements to a good forecast are selecting the appropriate training data in proper amounts. The ANN used here was trained with a minimal amount of data. Other variables that would traditionally have been included would be additional temperature and load data, and day of the week data for a week-long forecast. The dew point is one nontraditional variable that should be investigated for use in ANN training because it is a more accurate measure of human discomfort than temperature. Also, temperature and load data may be more valuable if, in addition to being included as measurements, the hourly temperature and load differences are included.

REFERENCES

[1] Chen, S.T., Yu, D.C., and A.R. Moghaddamjo. "Weather Sensitive Short-Term Load Forecasting Using Nonfully Connected Artificial Neural Network." Paper 91 SM 449-9 PWRS presented at the IEEE/PES 1991 Summer Meeting, San Diego, California, July, 1991.

[2] Cooper, L, and Steinberg, D. Methods and Applications of Linear Programming. Philadelphia. W.B. Saunders Company, 1974.

[3] Grady, W.M., Groce, L.A., Huebner, T.M., Lu, Q.C., and M.M. Crawford. "Enhancement, Implementation, and Performance of an Adaptive Short-Term Load Forecasting Algorithm." Paper 91 SM 448-1 PWRS, presented at the IEEE/PES 1991 Summer Meeting, San Diego, California, July, 1991.

[4] Hebb, D.O. The Organization of Behavior A neuropsychological Theory. New York: John Wiley and Sons, 1949.

[5] Hillier, F.S. and G.J. Lieberman. Introduction to Operations Research. Fifth edition. New York: McGraw Hill, 1990.

[6] Ho, K.L., Hsu, Y.Y., and Chien-Chuen Yang. "Short Term Load Forecasting using a Multilayer Neural Network with an Adaptive Learning Algorithm." Paper 91 SM 450-7 PWRS presented at the IEEE/PES 1991 Summer Meeting, San Diego, California, July, 1991.

[7] Hopfield, J.J. "Neural Networks and Physical Systems with Emergent Collective Computational Abilities." Proceedings of the National Academy of Science, U.S.A, 79, (1982): 2554-2558.

[8] Karmarkar, N. "A New Polynomial-Time Algorithm for Linear Programming." Combinatorica, 4, (1984): 373-395.

[9] Khanna, T. Foundations of Neural Networks. Reading, Massachusetts: Addison-Wesley, 1990.

[10] Kosko, B. Neural Networks and Fuzzy Systems: A Dynamical Systems Approach. Englewood Cliffs, New Jersey: Simon and Schuster, 1991.

[11] Lu, C.N., Wu, H.T., and S. Vemuri. "Neural Network Based Short Term Load Forecasting." Paper 92 WM 125-5 PWRS presented at the IEEE/PES 1992 Winter Meeting, New York, New York, January, 1992.

[12] Luenberger, D.G. Linear and Nonlinear Programming. Second edition. Reading, Massachusetts: Addison-Wesley, 1989.

[13] Mbamalu, G.A., and M.E. El-Hawary. "Load Forecasting via Suboptimal Seasonal Autoregressive Models and Iteratively Reweighted Least Squares Estimation." Paper 92 WM 133-9 PWRS presented at the IEEE/PES 1992 Winter Meeting, New York, New York, January, 1992.

[14] Park, D.C., El-Sharkawi, M.A, Marks, R.J. II, Atlas, L.E. and M.J. Damborg. "Electric Load Forecasting Using and Artificial Neural Network." IEEE Transactions on Power Systems, 6, no. 2, (May, 1991), 442-449.

[15] Peng, T.M., Hubele, N.F., and G.G. Karady. "Advancement in the Application of Neural Networks for Short-Term Load Forecasting." Paper 91 SM 451-5 PWRS presented at the IEEE/PES 1991 Summer Meeting, San Diego, California, July, 1991.

[16] Rahman, S., and O. Hazim. "A Generalized Knowledge-Based Short-Term Load-Forecasting Technique." Paper 92 WM 124-8 PWRS presented at the IEEE/PES 1992 Winter Meeting, New York, New York, January, 1992.

[17] Vanderbei, R.J, Meketon, M.S. and B.A. Freedman. "A Modification of Karmarkar's Linear Programming Algorithm." Algorithmica, 1, (1986): 395-407.

[18] Vanderbei, R.J. "Affine-Scaling for Linear Programs with Free Variables." Mathematical Programming, 43, (1989): 31-44.

[19] Watkins, D.S. Fundamentals of Matrix Computations. New York, New York: John Wiley and Sons, 1991.

[20] Welsh, D.E., and G.B. Sheble'. "Review of Artificial Neural Network Applications to Short-Term Power System Load Forecasting." Proceeding of the Twenty-Fourth Annual North American Power Symposium, Reno, Nevada, October, 1992.

ACKNOWLEDGMENTS

I would like to thank Dr. Gerald B. Sheble' for his support and encouragement in serving as my major professor, and Dr. John W. Lamont, and Dr. Richard K. Miller for the valuable comments they gave me as members of my graduate committee.

The Electric Power Research Center at Iowa State University, Power Technologies, Inc., and the National Science Foundation also have my appreciation for financially supporting my graduate work.