METHODS FOR IMPROVING GENERALIZATION AND CONVERGENCE IN ARTIFICIAL NEURAL CLASSIFIERS

by

Joel David Hewlett

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
December 12, 2011

Keywords: Neural Networks, Training Algorithms, Learning Machines, Pattern Classification

Approved by

Bogdan M. Wilamowski, Chair, Associate Professor of Electrical Engineering
Thaddeus Roppel, Associate Professor of Electrical Engineering
Robert Dean, Assistant Professor of Electrical Engineering
Vitaly Vodyanoy, Professor of Veterinary Medicine

ABSTRACT

Artificial neural networks have proven to be quite powerful for solving nonlinear classification problems. However, the complex error surfaces encountered in such problems often contain local minima in which gradient based algorithms may become trapped, causing improper classification of the training data. As a result, the success of the training process depends largely on the initial weight set, which is generated at random. Furthermore, attempting to analytically determine a set of initial weights that will achieve convergence is not feasible since the shape of the error surface is generally unknown.

Another challenge which may be faced when using neural classifiers is poor generalization once additional data points are introduced. This can be especially problematic when dealing with training data that is poorly distributed, or in which the number of data points in each respective class is unbalanced. In such cases, proper classification may still be achieved, but the orientation of the separating plane and its corresponding margin of separation may be less than optimal.

In this dissertation, a set of methods designed to improve both the generalization and convergence rate for neural classifiers is presented. To improve generalization, a single neuron pseudo-inversion technique is presented that guarantees optimal separation and orientation of the separating plane with respect to the training data. This is done by iteratively reducing the size of the training set until a minimal set is reached. The final set represents those points which lie on the boundaries of the data classes. Finally, a quadratic program formulation of the margin of

separation is defined for the reduced data set, and an optimal separating plane is obtained. A method is then described by which the presented technique may be applied to non-linear classification by systematically optimizing each of the neurons in the network individually.

Next, a modified training technique is discussed, which significantly improves the success rate in gradient based searches. To do this, the proposed method monitors the state of the gradient search in order to determine if the algorithm has become trapped in a false minimum. Once entrapment is detected, a set of desired outputs are defined using the current outputs of the hidden layer neurons. The desired values of the remaining misclassified patterns are then inverted in an attempt to reconfigure the hidden layer mapping, and the hidden layer neurons are retrained one at a time. Linear separation is then attempted on the updated mapping using pseudo-inversion of the output neuron. The process is repeated until separation is achieved.

The second method is compared with other popular algorithms using a set of 8 nonlinear classification benchmarks, and the proposed method is shown to produce the highest success rate for all of the tested problems. Therefore, the proposed method does, in fact, achieve the desired, which is to improve the rate of convergence of the gradient search by overcoming the challenge presented by local minima. Furthermore, the resulting improvement is shown to have a relatively low cost in terms of the number of required iterations.

ACKNOWLEDGMENTS

I would like to thank my parents, Marvin and Patricia Hewlett, for their unwavering love and support throughout my time in graduate school. I have taken great comfort in knowing that whenever discouragement, doubt or uncertainty may arise, I always have someone to turn to for encouragement and sound advice. They are a true blessing, and I love them both dearly.

I would also like to thank Dr. Wilamowski for everything he has done on my behalf. Whenever I have needed guidance or assistance his door has always been open, and any questions I may have had, no matter how trivial, have always been treated with patience and sincerity. The knowledge I have gained from him has been an invaluable resource, and is certainly not limited to the subject engineering. I am truly grateful and fortunate to have had such a wonderful adviser, for I honestly don't believe I could have made it this far without him.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ANN	Artificial Neural Network
RBFN	Radial Basis Function Networks
SVM	Support Vector Machines
BNN	Biological Neural Network
EBP	Error Back-Propagation
LM	Levenberg-Marquardt
NBN	Neuron-by-Neuron
LMS	Least Mean Squares
ASPI	Active Set Pseudo-Inversion
QP	Quadratic Program
HLPI	Hidden Layer Pseudo-Inversion

Chapter 1

INTRODUCTION

There are a number of well developed strategies for solving classification problems. These include, but are not limited to, artificial neural networks (ANN)[1], radial basis function networks (RBFN)[2] and support vector machines (SVM)[3] . These methods are especially useful for solving problems which are non-linearly separable. Although the specifics of the various methods differ, they operate on the same basic principle known as Cover's theorem [4]. The theorem states that a classification problem which is not linearly separable in the input space may become separable when cast into a higher dimensional space. From this it is evident that the primary challenge when solving a non-linear classification problem is not in the separation itself, but rather in choosing a linearly separable mapping. While RBFN and SVM methods take direct advantage of this, ANN methods generally do not. Instead, the nonlinear mapping and separation are combined into a single optimization problem which is most often solved using a least squares gradient approach. Though this method has proven successful, it is not necessarily the most efficient solution.

Each neuron in an ANN can be categorized as either a hidden unit or an output unit. Each output unit serves as a linear classifier whose inputs are supplied by the hidden units that together form a nonlinear mapping from the input space. In addition, for a bridged or fully connected architecture[5], the mapped space is guaranteed to be of higher dimension than the input space. Therefore, it follows from Cover's theorem that the likelihood of achieving

separation in the output layer is much greater in this higher dimensional space than in the original input space. Viewing the network in this way gives rise to an interesting and useful observation: if the output layer is solely composed of linear classifiers, then the primary challenge for nonlinear classification must lie in the training of the hidden units. This fact suggests that training methods designed to focus more heavily on the hidden layers than on the output layer might represent a more efficient and effective approach, which is the goal of the work presented in this dissertation.

In Chapter 2, a general overview of ANNs is presented. This includes the biological inspiration for the artificial neuron, as well as the computational details of the artificial models. An overview of some common training methods is also included, with a short discussion and comparison of the first and second order search directions. In Chapter 3, a set of pseudo-inversion techniques for performing linear classification with a single neuron are discussed. The strengths and weaknesses of the various methods are discussed, and an example case is used to illustrate the performance characteristics of each. Optimal linear separation is also defined. Next, a two phase method is presented that makes direct use of Cover's theorem via a periodic search of the hidden space to achieve separation at the output. The method seeks to escape entrapment by individually retraining the hidden layer neurons using the pseudo-inversion techniques described in Chapter 3. The performance of the proposed method is also discussed, using the methods from Chapter 2 for comparison. Finally, a short conclusion is offered in Chapter 5.

Chapter 2

ARTIFICIAL NEURAL NETWORKS

An artificial neural network is a biologically inspired computational model composed of a collection of interconnected functional units known as neurons[1]. The behavior of a network is characterized by the strengths of the interconnections, which are represented by multiplicative constants known as weights. Together, the weights of a network form a multidimensional search space known as a weight space. Networks are designed to perform specific tasks by systematically searching the weight space for those values which produce the best fit for a desired data set; or in the case of dynamic networks, yield equilibrium points which act as a form of memory. This search process is commonly referred to as training and the search methods themselves are known as training algorithms.

There are a number of different classes of networks that are most often differentiated by either the characteristics of their neurons or the structure of their interconnections. In non-dynamic networks, where the weights define a nonlinear mapping from the input space to the output space, the interconnections between neurons are only made in the forward direction. These are known as feed-forward networks. Allowing connections in the reverse direction, e.g. feedback, may cause instability or high frequency oscillation in the network; however, damping such systems via integration can produce stable dynamic behaviors which have proven useful for some applications.

2.1 FUNDAMENTAL CONCEPTS

To better understand the details of the training techniques described later, a short overview of some of the underlying concepts of ANNs are introduced first.

2.1.1 BIOLOGICAL NEURON

Because artificial neurons are loosely modeled on the nerve cells of living organisms [6], a basic understanding of biological neural networks (BNN's) may offer a useful starting point for understanding their artificial counterpart. Many aspects of the operation of BNN's are still not fully understood, and remain the subject of ongoing research. Therefore, the discussion presented here is limited only to the basic operation of the nerve cells that make up these networks rather than the operation of the networks as a whole.

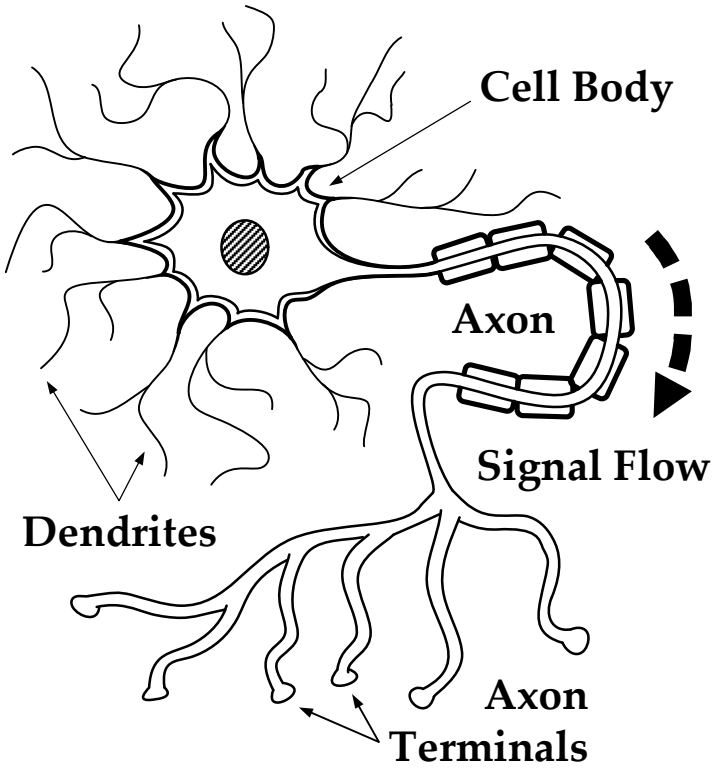


Figure 2.1: Schematic diagram of a biological neuron.

Biological neurons are elementary nerve cells which act as the fundamental building blocks of BNN's. A schematic diagram is shown in Figure 2.1. The typical neuron has three major parts:

1. Cell Body: The cell body is surrounded by a thin membrane which is capable holding an electrical potential, and the mechanisms which dictate the neurons responses to stimuli are housed inside with the nucleus.
2. Dendrites: The dendrites are thin branch-like fibers protruding from the cell body that form what is known as the dendritic tree. Electrical impulses from neighboring neurons are received by the dendrites, which act as inputs for the neuron.
3. Axon: The axon is a long cable-like projection that carries impulses generated in the cell body, and may extend up to hundreds or even thousands of times the diameter of the cell body. The axon's terminals act as the outputs of the neuron.

Each neuron receives impulses from neighboring cells via its dendrites, which build a cumulative potential on the cell membrane. Once the magnitude of this potential reaches a specific level, known as the threshold, the neuron fires its own impulse, which is then transmitted out through the axon terminals to be received by other neurons.

The strengths of a neuron's interconnections vary, making it more sensitive to some neurons than others. This variance on connection strength combined with the structure of the interconnections is what determines the behavior of the network as a whole. Since the cell structure remains unchanged of over the lifetime of the neuron, the learning process relies almost entirely on the manipulation of these interconnections.

2.1.2 ARTIFICIAL MODELS

Artificial neurons are loosely modeled after their biological counterparts[6]. In reality, the artificial neuron is little more than a multivariate function with weighted inputs. Still, despite their relative simplicity, ANNs have proven to be quite powerful[7], and like their biological analogs, their behavior is entirely dependent on the structure and strength of their interconnections. Because the neurons that make up the ANN are identical in form, they are highly modular systems, which is one of their most attractive aspects.

Although a number of different artificial models have been devised[1], most share the same general form, which is shown in Figure 2.2. The mathematical description for models of this form is

$$out = f(k \cdot net), \text{where}$$
$$net = w_b + \sum_{i=1}^{n} x_i w_i. \tag{2.1}$$

The values x_i are known as the neuron's inputs, and may be supplied by input pattern values or by the outputs of other neurons, depending on the unit's location in the network. Each input x_i is scaled by a corresponding weight denoted by w_i, and the threshold of the neuron is controlled by the biasing weight w_b. The weighted sum of all inputs is commonly referred to as the net value of the neuron, and is analogous to the cumulated potential on the cell membrane of the biological neuron. The function $f(\cdot)$ is known as the activation function, and plays the most important role in the behavior of the system.

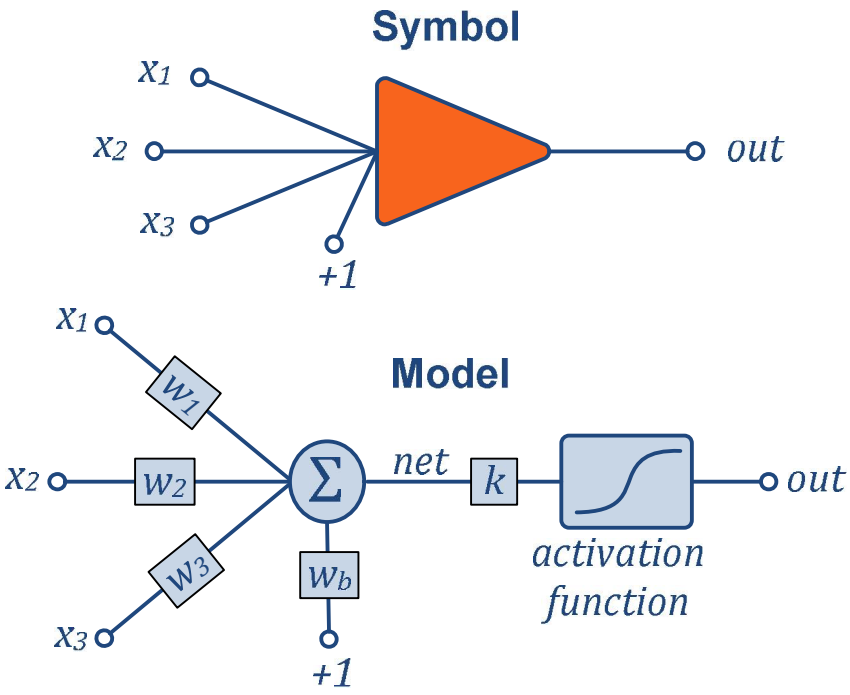


Figure 2.2: General form of the artificial neuron model.

ACTIVATION FUNCTIONS

The activation function is a mathematical mapping that describes the relationship between the weighted sum of the neuron's inputs and its output. A number of different functions have been proposed. A selection of these, shown in Figure 2.3, is discussed here.

The earliest of activation functions used the binary valued sign operator[8]. That is,

$$out = \text{sign}(net). \tag{2.2}$$

Activation functions of this type are referred to as "hard" activation functions due to their sharp discontinuity. One of the major drawbacks to using functions of this type is the fact that they are not differentiable. This lack of differentiability means that they cannot be trained using gradient based methods.

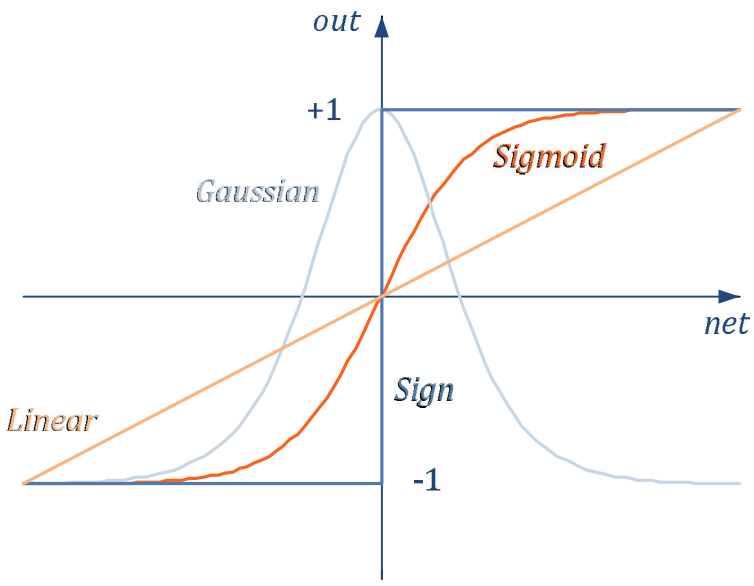


Figure 2.3: A comparison of common activation functions.

To overcome the training difficulties faced when using hard activation function, a number of differentiable functions were introduced. The simplest of these is the linear activation function. Technically speaking, for the general model given by (2.1), the linear activation function is not an activation function at all. Instead, the output value is equal only to the gain multiplied by the net value, with the activation function $f(\cdot)$ removed entirely. Therefore, for the linear model,

$$out = k \cdot net. \tag{2.3}$$

Although the linear model is continuously differentiable, it is not a suitable choice for solving problems binary desired outputs. Simply saturating the linear function at ± 1 will not work either since the resulting function would not be differentiable in the saturated region. To solve this problem, a sigmoidal function is used, which saturates asymptotically, allowing it to maintain its differentiability. The most common of these functions is the tangent hyperbolic,

$$out = \tanh(k \cdot net). \tag{2.4}$$

Another commonly used function is the Gaussian. The Gaussian activation function is most useful for non-linear classification and surface fitting problems. The general form of this function is

$$out = e^{-(k \cdot net)^2} \tag{2.5}$$

Gaussian functions are also the basis for an entire class of ANNs known as radial basis function networks, however the form of the RBF neuron model differs from that of (2.1) in the way that the net value is defined.

2.2 TRAINING

In the field of ANN's, the process of determining the proper set of weights for achieving a desired mapping is known as training. This task is commonly formulated as an optimization problem involving the minimization of the output error over the network's weights. That is,

$$\mathbf{w}^* = \min_{\mathbf{w}} E(\mathbf{w}). \tag{2.6}$$

where $\mathbf{w}$ is a vector containing the weights if the network, E is the total error of the network as a function of $\mathbf{w}$, and $\mathbf{w}^*$ is the weight vector that minimizes the value of $E(\mathbf{w})$. As a result, a number of optimization methods have been adapted for this purpose. In this section, some of the more notable methods are presented.

2.2.2 SINGLE NEURON TECHNIQUES

Some of the simplest and earliest training methods were developed for optimizing the weights of single neurons. However, despite their simplicity, many of these methods played an

influential role in the development the generalized algorithms that came later. The methods presented in this section form the foundation for many of the techniques discussed in later chapters.

DELTA LEARNING RULE

The delta learning rule is a steepest descent method used to train individual neurons for linearly separable classification problems[9]. As a steepest descent method, the delta learning rule operates using the gradient of the output error with respect to the weights. Here, the output error is defined as the one half the square of the difference between the desired output and the observed output for a given pattern. That is,

$$e_i = (d_i - o_i)^2, \tag{2.7}$$

where d_i is the desired output for input pattern i, and o_i is the observed output. The values of the observed outputs are calculated using

$$o_i = f(\mathbf{x}_i \cdot \mathbf{w}), \tag{2.8}$$

where $f(\cdot)$ is the neuron's activation function, $\mathbf{x}_i$ is i^{th} input vector, and $\mathbf{w}$ is the weight vector. To determine the gradient of the error, the partial derivatives of e_i with respect to the weights must be found. Therefore, substituting (2.8) into (2.7), the partial derivatives of the error are determined by

$$\frac{\partial e_i}{\partial w_j} = -(d_i - o_i) f_i' x_j, \tag{2.9}$$

where f_i' is the slope of the activation function for pattern i, w_j is the j^{th} element of the weight vector, and x_j is the corresponding input. Together, the partial derivatives computed using (2.9) form the gradient,

$$\nabla E_i = \begin{bmatrix} \frac{\partial e_i}{\partial w_1} \\ \frac{\partial e_i}{\partial w_2} \\ \vdots \\ \frac{\partial e_i}{\partial w_n} \end{bmatrix}, \tag{2.10}$$

where n is the total number of weights. Finally, using (2.10), a steepest descent update rule is defined,

$$\Delta \mathbf{w} = -\alpha \nabla E \tag{2.11}$$

The scalar α is known as the learning constant, and is used to control the step size.

PSEUDO-INVERSION FOR LINEAR ACTIVATION FUNCTIONS

Pseudo-inversion training is a regression technique used to solve for the weights of a single neuron[10]. Suppose a single neuron is to be used to classify a set of P training patterns with dimension N, and let x_{ij} be the j^{th} input of the i^{th} pattern, where $i = 1, \dots, P$ and $j = 1 \dots N$. The squared errors for the individual training patterns can then be written as

$$\begin{gathered} Err_1 = [d_1 - f(w_1 x_{11} + w_2 x_{12} + \cdots + w_N x_{1N})]^2 \\ Err_2 = [d_2 - f(w_1 x_{21} + w_2 x_{22} + \cdots + w_N x_{2N})]^2 \\ \vdots \\ Err_P = [d_P - f(w_1 x_{P1} + w_2 x_{P2} + \cdots + w_N x_{PN})]^2, \end{gathered} \tag{2.12}$$

where w_j are the weights associated with each of the neuron's N inputs. Defining the total error as the sum of the squared errors for all patterns yields

$$TE = \sum_{i=1}^{P} Err_i. \tag{2.13}$$

Using (2.13), the training process can be described as the following optimization problem:

$$\min_{w} \sum_{i=1}^{P} Err_i \,. \tag{2.14}$$

There is, however, an alternative formulation of the problem which allows for a simpler and more direct solution. The method makes use of the property that at the minimum of the error function, the components of the gradient must all be equal to 0. This leads to the following set of N equations and N unknowns:

$$\begin{aligned}
\frac{\partial(TE)}{\partial w_1} &= -2\sum_{i=1}^{P}[d_i - o_i]f'x_{i1} = 0 \\
\frac{\partial(TE)}{\partial w_2} &= -2\sum_{i=1}^{P}[d_i - o_i]f'x_{i2} = 0 \\
&\vdots \\
\frac{\partial(TE)}{\partial w_N} &= -2\sum_{i=1}^{P}[d_i - o_i]f'x_{iN} = 0,
\end{aligned} \tag{2.15}$$

where d_i and o_i are the respective values of the desired and actual outputs for input pattern i, and f' is the derivative of the activation function. Assuming the activation function is linear and has a gain of 1 helps to simplify things by making the f' terms in (2.15) equal to 1, and thereby allowing them to be ignored. Now, (2.15) can be rearranged to yield a set of linear equations in matrix form.

$$\begin{bmatrix}
\sum_{i=1}^{p} x_{i1}x_{i1} & \sum_{i=1}^{p} x_{i1}x_{i2} & \cdots & \sum_{i=1}^{p} x_{i1}x_{iN} \\
\sum_{i=1}^{p} x_{i2}x_{i1} & \sum_{i=1}^{p} x_{i2}x_{i2} & \cdots & \sum_{i=1}^{p} x_{i2}x_{iN} \\
\vdots & \vdots & \ddots & \vdots \\
\sum_{i=1}^{p} x_{iN}x_{i1} & \sum_{i=1}^{p} x_{iN}x_{i1} & \cdots & \sum_{i=1}^{p} x_{iN}x_{iN}
\end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix}
\sum_{i=1}^{p} d_i x_{i1} \\
\sum_{i=1}^{p} d_i x_{i2} \\
\vdots \\
\sum_{i=1}^{p} d_i x_{iN}
\end{bmatrix} \tag{2.16}$$

Defining $\mathbf{X}$ as the $P \times N$ matrix of input patterns, $\mathbf{w}$ as the $N \times 1$ vector of weights, and $\mathbf{d}$ as the $P \times 1$ vector of desired outputs, (2.16) can be solved for $\mathbf{w}$ and expressed in more concise matrix form as

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{d}. \tag{2.17}$$

2.2.3 Error Backpropagation

Error backpropagation (EBP) is a supervised learning method which is a multi-neuron generalization of the delta learning rule described in section 2.2.2 [11]. The method is primarily intended for feed-forward networks, and is one of the most widely used training algorithms. The method calculates the gradient of the error surface by using the chain rule for differentiation to exploit the structure of feed-forward networks. The resulting gradient is then used in a simple steepest descent update rule.

Since the algorithm utilizes the gradient of the error, it is only applicable for networks whose neurons have differentiable activation functions. Therefore, it cannot be used for networks with *hard* activation functions. Still, it can be used to design such networks by using sigmoid functions during training and then replacing them with the sign operator once training is complete.

Derivation

Before moving on to the derivation, some useful assumptions can be made. First, assume the network architecture is such that the hidden units are in layers with no cross layer connections, and the inputs of the neurons in a given layer are connected to the outputs of all neurons in the previous layer, with the exception of the input layer, whose inputs are the individual elements of the training patterns. Also assume that there is only one training pattern in

the data set, and that all neurons have sigmoid activation functions. This will greatly simplify the notation moving forward.

The following notation will be used:

$$\begin{aligned} x_{ij} &= i^{th} \text{ input to } j^{th} \text{ neuron}, \\ w_{ij} &= \text{weight for } x_{ij}, \\ net_j &= \text{net value } (\mathrm{x}_j \mathrm{w}_j) \text{ for neuron } j, \\ o_j &= \text{output for unit } j, \\ d_j &= \text{desired output for unit } j. \\ f_j' &= \text{activation slope for unit } j. \end{aligned}$$

In order to determine the direction of steepest descent, the gradients of the errors for each of the output units is required. Since the process is the same for each output, only one output is assumed for the purpose this derivation. Therefore, the goal is to determine the values of the partial derivatives of the total error with respect to the each of the weights. To begin, note that because net_j is a function of w_{ij} regardless of the location of unit j in the network, then by the chain rule,

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ij}} = \frac{\partial E}{\partial net_j} x_{ij} \tag{2.18}$$

Since the differential term on the right side is the same for all input weights of unit j, it can be denoted as δ_j for short.

Now, consider the case in which unit j is in the output layer. In this case, the total error is defined as

$$E = \frac{1}{2}\left(d_j - f\left(net_j\right)\right)^2 \tag{2.19}$$

Thus,

$$\begin{aligned} \delta_j &= \frac{\partial}{\partial net_j} E, \\ &= -(d_j - o_j) f_j'. \end{aligned} \tag{2.20}$$

Substituting this into (2.18) yields

$$\frac{\partial E}{\partial w_{ij}} = \delta_i x_{ij} = -x_{ij}(d_j - o_j) f_j'. \tag{2.21}$$

Next, the case in which neuron j is in the hidden layer must be considered. From here, the derivation follows a similar line of reasoning as before, only this time the two following observations should be made:

1. Since the direction of signal flow is from the input layer to the output, for hidden unit j, all units whose inputs are directly connected to the output of j can be referred to as being *downstream*. Using this terminology, for each neuron k downstream from j, net_k is a function of net_j.
2. For all units $l \neq j$ in the same layer as j, the contribution to the total error is independent of w_{ij}.

As before, the goal is to determine $\frac{\partial E}{\partial w_{ij}}$, only this time j is a hidden unit rather than an output unit. Notice that the hidden input weight w_{ij} only effects the value of net_j, and net_j only influences the value of o_j. Furthermore, o_j effects the values of net_k for all k downstream of j, all of which influence the total error E. So, once again applying the chain rule,

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}} &= \sum_{k \in downstream} \frac{\partial E}{\partial net_k} \cdot \frac{\partial net_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ij}}, \\ &= \sum_{k \in downstream} \frac{\partial E}{\partial net_k} \cdot \frac{\partial net_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j} \cdot x_{ij}. \end{aligned} \tag{2.22}$$

Again, as was the case for the output layer, all the terms in (2.22) except x_{ij} are the same for all input weights of unit j, and can be combined to form the single term δ_j. Also note that the term $\frac{\partial E}{\partial net_k} = \delta_k$, $\frac{\partial net_k}{\partial o_j} = w_{kj}$ and $\frac{\partial o_j}{\partial net_j} = f_j'$. Therefore, by substitution,

$$\begin{aligned} \delta_j &= \sum_{k \in downstream} \frac{\partial E}{\partial net_k} \cdot \frac{\partial net_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j}, \\ &= \sum_{k \in downstream} \delta_k w_{kj} f_j', \\ &= f_j' \cdot \left(\sum_{k \in downstream} \delta_k w_{kj} \right). \end{aligned} \tag{2.23}$$

Now, using the above notation and definitions, the backpropagation algorithm can be stated formally.

BACKPROPAGATION ALGORITHM

Let α be a learning constant which controls the step size, and let n_i, n_h and n_o be the number of inputs, hidden units and output units respectively. Also let the training data be denoted by the input vectors $\mathbf{in}_p$ and the desired output vectors $\mathbf{d}_p$, for $p = 1,2, \ldots, P$, where P is the number of training patterns in the data set. The training process is implemented by the following steps.

1. Apply the training pattern $\mathbf{in}_p$ to the input of the network and compute the corresponding output vector $\mathbf{o}_p$.
2. For each unit k in the output layer, calculate

$$\delta_k = -(d_k - o_k) f_k'.$$

3. For each hidden neuron h, compute the values of

$$\delta_h = f_h' \cdot \left(\sum_{k \in downstream} \delta_k w_{kh} \right).$$

4. For each w_{ij}, compute the corresponding partial derivative of the error using

$$\Delta w_{ij} = \delta_j x_{ij}. \tag{2.24}$$

5. Update the weights using the gradient descent rule

$$w_{ij}^{n+1} = w_{ij}^{n} + \alpha \Delta w_{ij} \tag{2.25}$$

6. Repeat steps 1-5 until all input patterns have been applied.
7. Repeat steps 1-7 until the total error becomes sufficiently low.

The method presented above uses an incremental update, meaning that the patterns are applied one at a time and the weights are updated for each pattern. There is an alternative method by which the weights are simultaneously updated for all patterns at the same time. This is known as a cumulative update. The derivation of the latter is not presented here, however, the backpropagation process is very similar, and the derivation follows the same basic logic as that of the incremental version.

2.2.4 LEVENBERG-MARQUARDT

The Levenberg-Marquardt (LM) algorithm is a second order algorithm whose update rule is a combination of both the Newton method and the method of steepest descent[12][13]. As with the backpropagation algorithm, the forward calculations are made first, after which the resulting outputs are used to define the error. The error is then propagated back through the network to determine the partial derivatives of the error with respect to the system's weights. The resulting partial derivatives are used to form the Jacobian matrix, which is in turn used to approximate the Hessian as well as calculate the gradient of the total error. An adjustable learning parameter μ is then used to control the contributions of the gradient and Newton directions to the direction of

the final step. A short summery and comparison of these two search directions is presented in the next section.

NEWTON AND GRADIENT SEARCH DIRECTIONS

The most obvious of all search directions is the gradient direction $-\nabla f_k$, also known as the direction of steepest descent. As the name implies, $-\nabla f_k$ represents the direction in which the objective function $f(x)$ decreases most rapidly, thereby making it a natural choice. Although it is sometimes referred to as the gradient direction, the steepest descent direction actually points in the opposite direction of the true gradient, which corresponds to the direction of greatest increase. The true gradient is, by definition, the vector of all first partial derivatives of the objective function or, more formally,

$$\nabla f(x) \triangleq \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}, \text{where } \mathbf{x} = (x_1, x_2, \dots, x_n) \tag{2.26}$$

In optimization, where the goal is to find the point at which a given function yields the lowest possible value, knowing the direction in which that function changes most rapidly is of obvious value, making the gradient a particularly useful tool.

Another commonly used search direction is the Newton direction[14]. Though it is more difficult to compute than the gradient, it provides significantly more information about the local behavior of the objective, which in turn yields a much higher rate of convergence. This is achieved through the use of a second-order model $m_k(\sigma_k)$ of the objective function, derived from its Taylor series approximation,

$$f(\mathbf{x}_k + \sigma_k) \cong f_k + \sigma_k^T \nabla f_k + \frac{1}{2}\sigma_k^T \nabla^2 f_k \sigma_k^T \triangleq m_k(\sigma_k). \tag{2.27}$$

From this, the Newton direction is defined as the vector σ_k which minimizes the quadratic model $m_k(\sigma_k)$. Assuming for now that $\nabla^2 f_k$ is positive definite, it is possible to solve for σ_k by setting (2.27) equal to zero. Doing so yields the following explicit formula for the Newton direction:

$$\sigma_k = -(\nabla^2 f_k)^{-1} \nabla f_k. \tag{2.28}$$

Whereas the steepest decent direction revolves around the computation of the gradient, the Newton direction relies primarily on the calculation of the Hessian $\nabla^2 f_k$, which is a matrix containing all second partial derivatives of the objective function. That is,

$$\nabla^2 f_k(\mathbf{x}) \triangleq \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}, \text{where } \mathbf{x} = (x_1, x_2, \dots, x_n) \tag{2.29}$$

As alluded to earlier, the computation of the Hessian can be a rather expensive operation, which is one of the Newton method's major drawbacks. Still, the benefits of this approach are generally considered to outweigh the costs.

In addition, a number of strategies have been devised which serve to reduce the computational requirement by replacing the true Hessian matrix with a close approximation that does not need to be fully reevaluated at each iteration. Search directions which operate in this way are commonly referred to as quasi-Newton methods. Some of the more common implementations include the Broyden, Fletcher, Goldfarb and Shanno (BFGS) algorithm[15], and the closely related Davidon, Fletcher and Powell (DFP) algorithm[16].

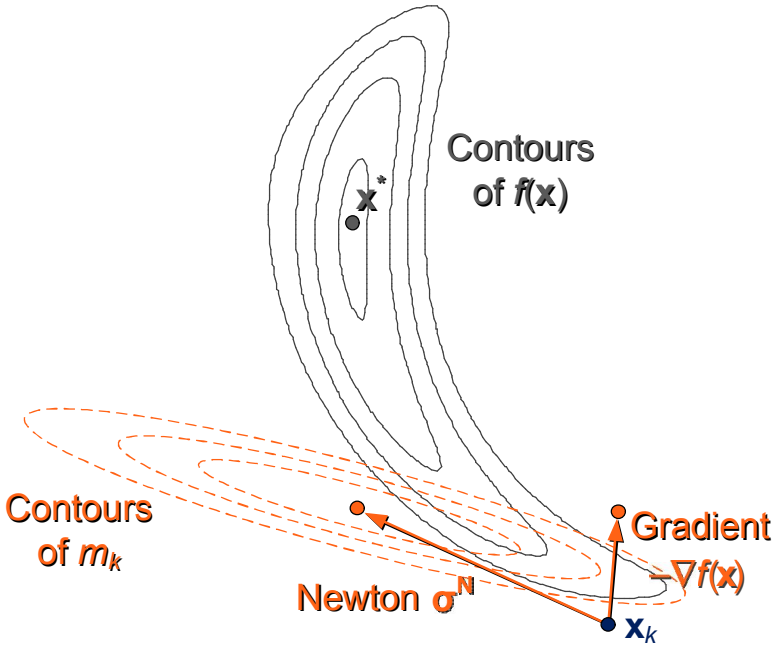


Figure 2.4: Comparison of the Newton and gradient search directions.

Another attractive attribute of many quasi-Newton methods is that they can be reformulated to operate on the inverse of the approximated Hessian instead of on the approximation itself, which alleviates the burden of performing costly matrix inversions when solving (2.28). This ability can prove especially advantageous on problems with high dimensionality.

A comparison of the two steps is shown in Figure 2.4. The lines shown in grey represent the contours of the objective function $f(x)$, while those in orange are the contours of the second order model m_k. The points $\mathbf{x}_k$ and $\mathbf{x}^*$ represent the location of the current iterate and the minimum of the objective respectively. The vector $\mathbf{s}^N$ corresponds to the Newton step, and $-\nabla f_k$ represents the gradient.

It is clear from Figure 2.4 that the additional information provided by the second order Newton approximation yields greater improvement in the objective. However, it is also evident that neither of the two search directions is optimal. Instead, the optimal direction appears to lie somewhere between the two. This would suggest that some combination of the two may produce the best result, which is exactly what the LM algorithm seeks to do by actively varying the learning parameter mentioned in the previous section.

ALGORITHM DESCRIPTION

The primary difference between the backward computations in LM and those of EBP is the necessity for determining the elements of the Jacobian rather than the total error gradient. Because the LM step is formulated as a least squares problem, the Jacobian is needed, which contains the first partial derivatives of the errors associated with the individual patterns rather than the total error used in EBP. Aside from this, the approach uses the chain rule of differentiation in the same manner as EBP. The resulting Jacobian matrix is of the form

$$\mathbf{J} = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_1} & \frac{\partial e_{11}}{\partial w_2} & \cdots & \frac{\partial e_{11}}{\partial w_n} \\ \frac{\partial e_{21}}{\partial w_1} & \frac{\partial e_{21}}{\partial w_2} & \cdots & \frac{\partial e_{21}}{\partial w_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{M1}}{\partial w_1} & \frac{\partial e_{M1}}{\partial w_2} & \cdots & \frac{\partial e_{M1}}{\partial w_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{1P}}{\partial w_1} & \frac{\partial e_{1P}}{\partial w_2} & \cdots & \frac{\partial e_{1P}}{\partial w_n} \\ \frac{\partial e_{2P}}{\partial w_1} & \frac{\partial e_{2P}}{\partial w_2} & \cdots & \frac{\partial e_{2P}}{\partial w_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{MP}}{\partial w_1} & \frac{\partial e_{MP}}{\partial w_2} & \cdots & \frac{\partial e_{MP}}{\partial w_n} \end{bmatrix}, \tag{2.30}$$

where e_{ij} is the error at the i^{th} output for the j^{th} input pattern, P is the number of training patterns, and M is the number of outputs.

Once the Jacobian matrix has been constructed, the only pieces missing are the gradient and the Hessian matrix, which contains the second partial derivates of the error with respect to the weights. In the case of the latter, it is possible to avoid the cost of computing these values directly by using a convenient approximation in place of the actual Hessian matrix. The convenience arises from the fact that the approximation can be computed using only the Jacobian matrix, which is already available. That is,

$$\mathbf{H} \cong \mathbf{J}^T\mathbf{J}\,. \tag{2.31}$$

Replacing the Hessian with (2.31) results in a quasi-Newton formulation of the LM algorithm.

Similarly, the gradient vector can also be calculated using the Jacobian. This is accomplished using the following formula:

$$\mathbf{g} = \mathbf{J}^T\mathbf{e}. \tag{2.32}$$

where $\mathbf{e}$ is a vector containing the error for each pattern at each output, and has length equal to the product of the number of patterns and the number of outputs.

In general, the LM step is computed in the following manner:

$$\Delta\mathbf{w} = -(\mathbf{H} + \mu\mathbf{I})^{-1}\mathbf{g}. \tag{2.33}$$

Therefore, substituting (2.31) and (2.32) into (2.33), the following formula is obtained for the weight update rule:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - (\mathbf{J}_k^T\mathbf{J}_k + \mu\mathbf{I})^{-1}\mathbf{J}_k^T\mathbf{e}_k \tag{2.34}$$

where $\mathbf{I}$ is the identity matrix of dimension $M + P$.

The learning parameter μ in (2.34) is used to control the contributions of the Newton and gradient search directions to the overall step. For smaller values of μ, the algorithm behaves more like a first order gradient descent method, whereas larger values cause the algorithm to behave more like the second order Newton method. During each iteration, the parameter is varied and the value which produces the greatest reduction in the objective is used in the final step. The pseudo-code in Figure 2.5 details the process used to select new values of μ.

```
E0 = totalerror(μ);
μ = μ*10;
E1 = totalerror(μ);
for i = 1:5
        if E1 <= E0
                μ = μ*10;
        else
                μ = μ/10;
        end
        E0 = E1;
        E1= totalerror(μ);
end
```

Figure 2.5: Pseudo-code for the learning parameter update.

2.2.5 NEURON-BY-NEURON METHOD

The neuron-by-neuron (NBN) method is a method for calculating the partial derivatives of the error with respect to the weights in arbitrarily connected feed-forward neural networks[17]. This is in contrast to the traditional backpropagation method which is only applicable for the standard multilayer perceptron architectures with no cross-layer connections. The ability to handle arbitrarily connected networks offers a significant advantage since the introduction of cross-layer connections can greatly reduce the number of neurons required to achieve specific nonlinear mappings and classifications, as well as diminishing the likelihood of over-fitting.

TOPOLOGY REPRESENTATION

The NBN method makes use of a netlist style notation for describing the network topology. The order of the listing is used to determine the order in which the reverse calculations are to be made. A simple example will be used to explain the syntax and format of the listing.

Take, for example, the network shown in Figure 2.6. Notice that each node in the network is individually numbered. There are 3 input nodes (1,2,3), 2 hidden nodes (4,5) and 2 output nodes (6,7). Also note that the lowest numbers are given to the input nodes, while the higher numbers are reserved for the output nodes. The hidden nodes are labeled with the interstitial numbers. The convention is that for any neuron in the network, the numbers associated with the nodes connected at its input are lower than those of its output. For this particular network, the topology would be described by the following listing:

N 4 model1 1 2 3

N 5 model2 1 2 3

N 6 model3 1 2 3 4

N 7 model4 1 2 3 5

Each line beginning with "N" describes the connections of a single neuron. The first number represents the output node of the neuron described by that line. Next, the neuron model is supplied. Each model has an activation function and gain value associated with it which describe how the neuron output is be computed. The remaining numbers represent the nodes connected to the neuron's input. The neurons should be arranged so that the output nodes are in ascending order when moving down the list.

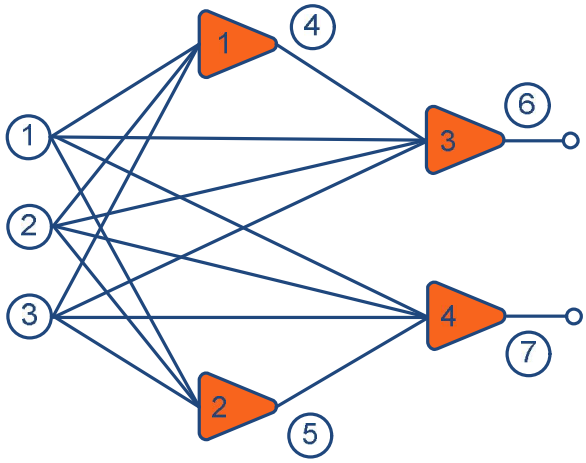


Figure 2.6: Example network for using NBN topology description.

FORWARD COMPUTATION

Before the partial derivatives of the error can be calculated, the values at all nodes must be known. Thus, the first phase of the NBN method involves making the forward computations for each input pattern, and determining the corresponding values at all nodes in the hidden and output layers. These values are then stored in matrix from, allowing them to be easily retrieved during the backpropagation phase. The basic process for the forward calculation phase is presented in pseudo code form in Figure 2.7.

```
for each pattern (p=1:P)
        for each neuron (n=1:N)
                net=0;
                for each input i
                        net=net+w(i)*node(p,ndx(i,n));
                end
                node(p,n)=act(net);
                der(p,n)=1-node(p,n)^2;
        end
        for each output (o=N-O:N)
                err(p,o)=dout(p,o)-node(p,o);
        end
end
```

Figure 2.7: Pseudo-code for the NBN forward calculation phase.

The variables and constants in Figure 2.7 are defined as follows:

- *P* is the number of input patterns
- *N* is the number of neurons
- *O* is the number of output nodes
- *node(p,n)* is a $P \times N$ 2-dimensional array containing the node values for all patterns at all nodes.
- *w(i)* is the weight associated with neuron input *i*
- *ndx(i,n)* is an integer array which stores the indices of the input nodes for each neuron.
- *der(p,n)* is a $P \times N$ 2-dimensional array containing the derivatives of the activation function of all neurons for all patterns. These values will be needed for the Jacobian and gradient calculations.
- *dout(p,o)* is a $P \times O$ 2-dimensional array containing the desired outputs for all input patterns at each node in the output layer.
- *err(p,o)* is a $P \times N$ 2-dimensional array containing the error associated with each pattern at each node in the output layer.

The forward calculation routine returns three arguments for each input pattern: the resulting error (*err*) at each output, the derivatives of the activation functions (*der*) for all neurons, and the output value (*node*) of each node in the network. Therefore, once the forward calculation routine is complete, all information needed for calculating the error gradient and Jacobian matrix is available.

JACOBIAN COMPUTATION

Since the error gradient can be calculated from the Jacobian using (2.32), the NBN routine for calculating it directly has been omitted, and instead, only the Jacobian routine is included in the present discussion. When used with first order methods this method of calculating the gradient does require more memory; nevertheless, for second order methods in which the Jacobian must be computed anyway, using (2.32) is actually more efficient because it removes the additional computation time required to calculate the gradient directly. The Pseudo-code for the Jacobian calculation is presented in Figure 2.8.

```
for each pattern (p=1:P)
        for each output (o=N-O:N)
                errnode(p,o,n)=err(p,o);
                for each neuron (n=N:-1:1)
                        delta(p,o,n)=der(p,n)*errnode(p,o,n);
                        for each input i
                                errnode(p,o,ndx(i,n))=
errnode(p,o,ndx(i,n))+delta(p,o,n)*w(i);
                        end
                end
        end
end
for each pattern (p=1:P)
        for each output (o=N-O:N)
                for each input i
                        J(n,i)=delta(n)*node(p,ndx(I,n));
                end
        end
end
```

Figure 2.8: Pseudo-code for the NBN Jacobian calculation.

The variables and constants in Figure 2.8 are defined in the same way as in Figure 2.7, with three additions:

- *errnode(p,o,n)* is a $P \times O \times N - 1$ array containing the sum of the error at the output of each neuron over all patterns.
- *delta(p,o,n)* is a $P \times O \times N - 1$ array containing the sum of the error at the inputs of each neuron over all patterns.
- *J(n,i)* is the $P \cdot O \times M$ Jacobian matrix, where *M* is the number of weights in the network.

2.3 USING PARITY-N PROBLEMS AS A BENCHMARK FOR PERFORMANCE

In this section, the classic parity-N problems are introduced and used as a performance metric for the comparison of the EBP and LM training. The algorithms are used to solve a set of six problems using cascade architectures of varying size. For each test, performance is measured based on the overall success rate and the average number of iterations required for convergence.

2.3.1 THE PARITY-N PROBLEM

The parity-N problems are N-dimensional binary classification problems, which are commonly used as performance benchmarks for neural classifiers[18][19]. The problems are notoriously difficult to solve due to their high degree of nonlinearity, and the difficulty is known to increase significantly with respect to dimensionality.

In the general parity-N case, the training patterns are comprised of all N-bit binary numbers from 0 to $2^N - 1$. The desired output for each pattern is assigned based on the number of input bits that have values equal to +1. The rules for this assignment are as follows:

- If the number of input bits equal to +1 is even, a desired output of -1 is assigned.
- If the number is odd, a desired output of +1 is specified instead.

In addition to the inherent difficulty, another attractive aspect of the parity-N problems is the existence of analytical solutions [20]. The benefit of the analytical solutions is that the minimum number of neurons required for a given architecture can be determined prior to testing, which provides a trustworthy baseline for comparison. Without this information, it would be impossible to determine if tests which produce a 0% convergence rate were the result of the failure of the algorithm or simply the nonexistence of a solution for the chosen network. Furthermore, the difficulty of the problem for a particular architecture is inversely related to the size of the network. Therefore, knowing the minimum number of neurons for a given network structure offers a worst case scenario for testing.

2.3.2 PARITY-2 EXAMPLE

Take, for example, the parity-2 problem. Since $N = 2$, the training data is defined as the set of 2-bit binary values from 0 to 3, and the 4 corresponding output values are chosen according to the rules defined in the previous section. The resulting truth table is shown in Table 2.1. One may also notice that for the 2-dimensional case, the parity problem is equivalent to the familiar XOR operator used in digital logic.

Table 2.1: Truth table for the parity-2 problem.

x_1	x_2	o
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1

For classification problems, the weights of a neuron can be viewed as defining a plane of separation for the input patterns. For a single neuron, this means that proper classification of a given data set depends on the linear separability of the data. For data which is not linearly

separable, additional neurons are required. In the case of the parity-2 problem, the separability can be determined visually by plotting the data on a 2-dimensional plane using the symbols × and o to represent desired outputs of $+1$ and -1 respectively, as shown in Figure 2.9.

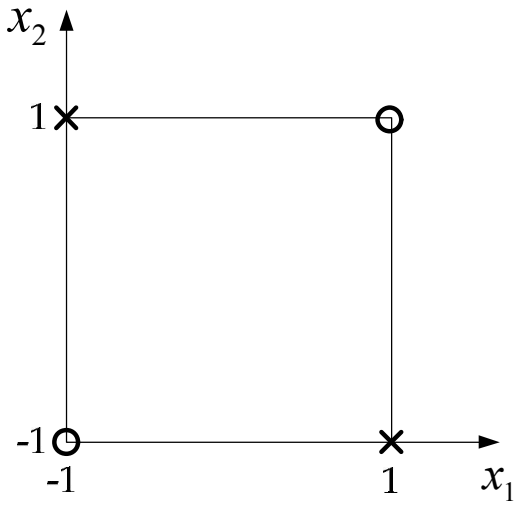


Figure 2.9: Graphical representation of the parity-2 problem.

It is evident from Figure 2.9 that no single plane of separation will produce the desired classification. This confirms that the data set in Table 2.1 is in fact not linearly separable. To separate the data, the input patterns must first be mapped into a higher dimensional space through the addition of a hidden layer. To do this, the two neuron cascade architecture shown in Figure 2.10 is chosen.

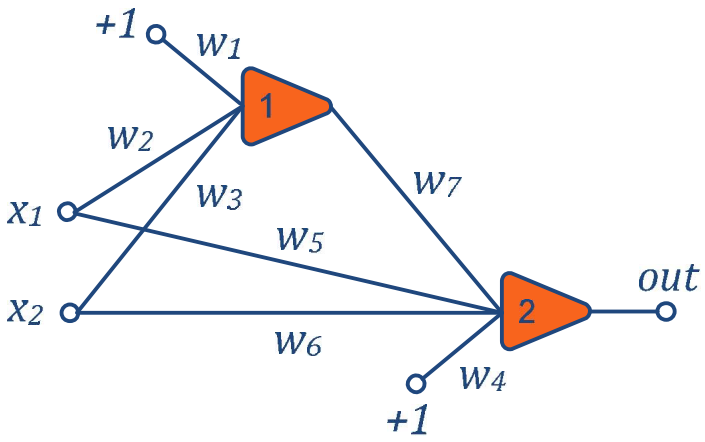


Figure 2.10: The two neuron cascade used to solve the parity-2 problem.

The next step is to determine a set of weight values for the hidden layer neuron that produce a separable mapping at the output layer. To do this, the separating plane shown in Figure 2.11 is chosen. From this, the value of w_1 can be computed using the slope intercept form equation

$$x_2 = -w_1 x_1 + \frac{w_3}{w_2}. \tag{2.35}$$

From Figure 2.11, it is apparent by inspection that the slope of this line is equal to -1. Therefore, form (2.35), w_1 must be equal to one. It is also clear from (2.35) that the plane passes through the points (-1,0) and (0,-1). Since the net values for the neuron are equal to 0 for all points which lie on the separating plane, plugging these values into the net equation and setting it equal to zero results in a set of two equations with two unknowns. That is,

$$\begin{aligned} 1 - 1w_2 + 0w_3 &= 0, \text{ and} \\ 1 + 0w_2 - 1w_3 &= 0. \end{aligned} \tag{2.36}$$

Solving the system in (2.36) for w_2 and w_3 yields the following weight vector for the hidden neuron:

$$\mathbf{w} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}. \tag{2.37}$$

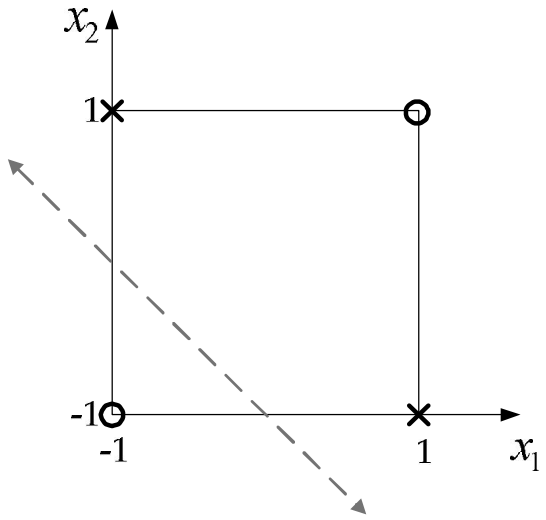


Figure 2.11: Hidden layer separating plane for the parity-2 problem.

Now, the output of the hidden neuron is calculated for each of the patterns in Table 2.2, and a new truth table is defined with respect to the output layer. The augmented data set is shown in 0.

Table 2.2: Truth table for the parity-2 problem.

x_1	x_2	x_3	o
-1	-1	-1	-1
-1	1	1	1
1	-1	1	1
1	1	1	-1

The data in Table 2.2 can be visualized in a similar manner as before, only this time three dimensions are required due to the addition of the hidden layer output values. This is illustrated in Figure 2.12.

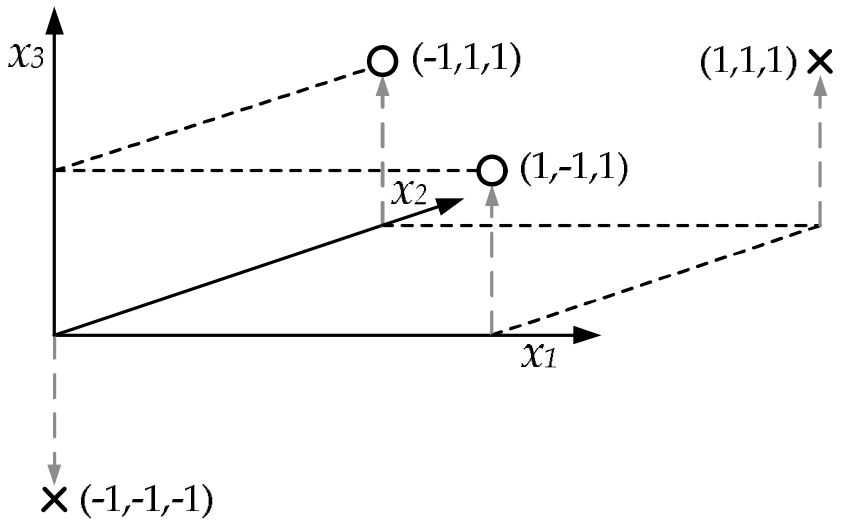


Figure 2.12: Graphical representation of the augmented XOR data set.

It should be apparent from Figure 2.12 that, using the augmented data set provided in Table 2.2, the input patterns can now be linearly separated by the remaining neuron in the output layer.

2.3.3 Comparison of Algorithm Performance

In this section, the performance of the previously discussed EBP and LM algorithms is compared using the set of parity problems from $N = 2$ to $N = 7$. Each problem was attempted using fully connected cascade architectures with sizes ranging from 2 to 6 neurons for parity 2 and 3, and 3 to 6 neurons for parity 4 through 7, resulting in a total of 26 test cases per algorithm. The performance for each test case was measured based on the overall success rate and average number iterations required for convergence after 100 attempts. The EBP and LM algorithms were allowed respective maximums of 100,000 and 10,000 iterations per training attempt, with the success rate defined as the number of attempts that reached an average total error of 0.001 per pattern within the allotted number of iterations.

The test results for the EBP algorithm are shown in Table 2.3. Training was done using the following parameters.

- Learning Constant (α): 0.7
- Activation Function: Tangent-Hyperbolic (Bipolar)
- Neuron gain: 0.1

It is evident from the table that difficulty presented by the parity problems is directly proportional to the problem dimensionality, and inversely proportional to the size of the network. Despite the increased success rate when using larger networks, in general, optimal architecture achieve better overall performance since they are less susceptible to over fitting. This behavior may not be evident when solving parity problems, however some applications, such as function approximation, this can become a significant issue. Therefore, the network size must be taken into account when assessing algorithm performance.

Table 2.3: Parity-N performance of the EBP algorithm.

		Problem					
		Parity-2	Parity-3	Parity-4	Parity-5	Parity-6	Parity-7
Neurons	2	100% 12,175	100% 6,060	N/A	N/A	N/A	N/A
	3	100% 6,428	100% 3,139	40% 46,882	49% 35,424	0% -	0% -
	4	100% 4,485	100% 2,009	89% 40,240	100% 25,148	13% 62,223	22% 55,309
	5	100% 2,886	100% 1,706	95% 34,665	100% 19,402	33% 55,773	50% 51,361
	6	100% 2,638	100% 1,530	99% 28,344	100% 13,130	100% 36,215	64% 45,151

Table 2.4: Parity-N performance of the LM algorithm.

		Problem					
		Parity-2	Parity-3	Parity-4	Parity-5	Parity-6	Parity-7
Neurons	2	100% 9	98% 9	N/A	N/A	N/A	N/A
	3	100% 9	99% 9	36% 20	51% 22	0% -	0% -
	4	100% 8	99% 8	81% 20	85% 19	29% 34	42% 35
	5	100% 8	98% 8	95% 19	93% 19	65% 32	71% 34
	6	100% 8	100% 8	98% 18	99% 18	84% 34	76% 33

The test results for the LM algorithm are shown in Table 2.4. For training, the following set of parameters was used:

- Initial Learning Parameter (μ_0): 0.1
- Activation Function: Tangent-Hyperbolic (Bipolar)
- Neuron Gain: 0.1

Comparing the results in Table 2.4 with those in Table 2.3, the superiority of the LM algorithm with respect to the speed of convergence is clear. This is not surprising since the second-order model used by the LM algorithm offers a more accurate description of the error surface. It is also clear, however, that the LM algorithm does not fare as well in terms of the success rate. The reason for this is that the LM algorithm's speed comes at a price. Because it converges so quickly, the neuron's are susceptible to being driven in to early saturation, before reaching the desired solution. Once saturation is reached, the derivatives of the activation function for the misclassified patterns become very small, despite having a relatively high error. Because the misclassified patterns make up a minority of the total set of input patterns, their

contribution to the step is not significant enough to overcome the influence of the correctly classified patterns. Therefore, the algorithm becomes trapped.

The EBP algorithm, on the other hand, benefits to some degree from its relatively slow rate of convergence since the algorithm is not as easily driven into saturation, which allows the misclassified patterns to have greater influence on the direction of the search.

Chapter 3

SINGLE NEURON TRAINING USING ITERATIVE PSEUDO-INVERSION TECHNIQUES

3.1 PSEUDO-INVERSION TRAINING FOR NONLINEAR ACTIVATION FUNCTIONS

The pseudo-inversion method described in section 2.2.2 was a regression technique used for neurons with linear activation functions. Here, an iterative method is presented that is capable of training neurons with nonlinear activation functions[10]. Two forms of the update rule are discussed. Although the two forms are mathematically equivalent, they differ computationally in terms of software implementation. The advantages and disadvantages of each formulation are discussed following the derivation.

3.1.1 DERIVATION

Let E_p, D_p and O_p be the neuron error, desired output and observed output for pattern respectively, where $p = 1, \dots, P$ is the pattern index, and let w_i and x_i be the weight and input values associated with the i^{th} input of the neuron. Next, define the error for pattern p as

$$E_p = D_p - O_p(net), \tag{3.1}$$

where $net = w_1 x_1 + w_2 x_2 + \cdots + w_N x_N$, and N is the dimension of the input. The derivative of this error with respect to the i^{th} weight can then be described as

$$\frac{dE_p}{dw_i} = \frac{dO_p}{dnet}\frac{dnet}{dw_i} = -f_p' x_{ip}, \tag{3.2}$$

where f_p' is the slope of the neuron's activation function as a function of the net value. Next, let the error in (3.1) be replaced by the first order Taylor approximation

$$E_p = E_{p0} + \frac{dE_p}{dw_1}\Delta w_1 + \frac{dE_p}{dw_2}\Delta w_2 + \cdots + \frac{dE_p}{dw_N}\Delta w_N. \tag{3.3}$$

Now, (3.2) and (3.3) can be combined to form the following overdetermined linear system of equations in matrix form:

$$\begin{bmatrix} f_1' x_{11} & f_1' x_{12} & \cdots & f_1' x_{1N} \\ f_2' x_{21} & f_2' x_{22} & \cdots & f_2' x_{2N} \\ \vdots & \vdots & & \vdots \\ f_P' x_{P1} & f_P' x_{P2} & \cdots & f_P' x_{PN} \end{bmatrix} \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_N \end{bmatrix} = \begin{bmatrix} d_1 - o_1 \\ d_2 - o_2 \\ \vdots \\ d_P - o_P \end{bmatrix}. \tag{3.4}$$

Solving (3.4) for $\Delta\mathbf{w}$ yields the least mean squares (LMS) formulation of the step,

$$\Delta\mathbf{w} = [(\mathbf{FX})^T \mathbf{FX}]^{-1} (\mathbf{FX})^T \mathbf{E}, \tag{3.5}$$

where $\mathbf{X}$ is the $P \times N$ matrix of input patterns, and

$$\mathbf{F} = \begin{bmatrix} f_1' & 0 & 0 & \cdots & 0 \\ 0 & f_2' & 0 & \cdots & 0 \\ 0 & 0 & f_3' & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & f_P' \end{bmatrix}.$$

Note that the term $[(\mathbf{FX})^T \mathbf{FX}]^{-1} (\mathbf{FX})^T$ in (3.5) is longhand for the pseudo-inverse of $\mathbf{FX}$, for which the method gets its name.

3.1.2 Improving Computational Efficiency

The computational requirements of the method derived in section 3.1.1 using a simple reformulation. Instead of multiplying the patterns in $\mathbf{X}$ by the corresponding values of f_p' on the

left side of (3.4), the values are divided through both sides of the equation, which produces the following algebraically equivalent formulation:

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1M} \\ x_{21} & x_{22} & \cdots & x_{2M} \\ \vdots & \vdots & & \vdots \\ x_{P1} & x_{P2} & \cdots & x_{PM} \end{bmatrix} \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_M \end{bmatrix} = \begin{bmatrix} \frac{d_1 - o_1}{f_1'} \\ \frac{d_2 - o_2}{f_2'} \\ \vdots \\ \frac{d_P - o_P}{f_P'} \end{bmatrix}. \tag{3.6}$$

Now, solving for $\Delta\mathbf{w}$ yields the reformulated update rule

$$\Delta\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{del} = \mathbf{Y}\,\mathbf{del}, \tag{3.7}$$

Where

$$\boldsymbol{Y} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T \text{ and } \mathbf{del} = \begin{bmatrix} \frac{d_1 - o_1}{f_1'} \\ \frac{d_2 - o_2}{f_2'} \\ \vdots \\ \frac{d_P - o_P}{f_P'} \end{bmatrix}.$$

The reason for reformulating the update rule in this way is that the value of $\mathbf{X}$ does not change from one iteration to the next since the values of the input patterns are constants. This means that $\mathbf{Y}$. the pseudo-inverse of $\mathbf{X}$, need only be computed once. Since matrix inversion is a computationally intensive operation, the formulation in (3.7) is much more efficient than (3.5), which requires inversion to be performed at each iteration due to the changes in $\mathbf{F}$.

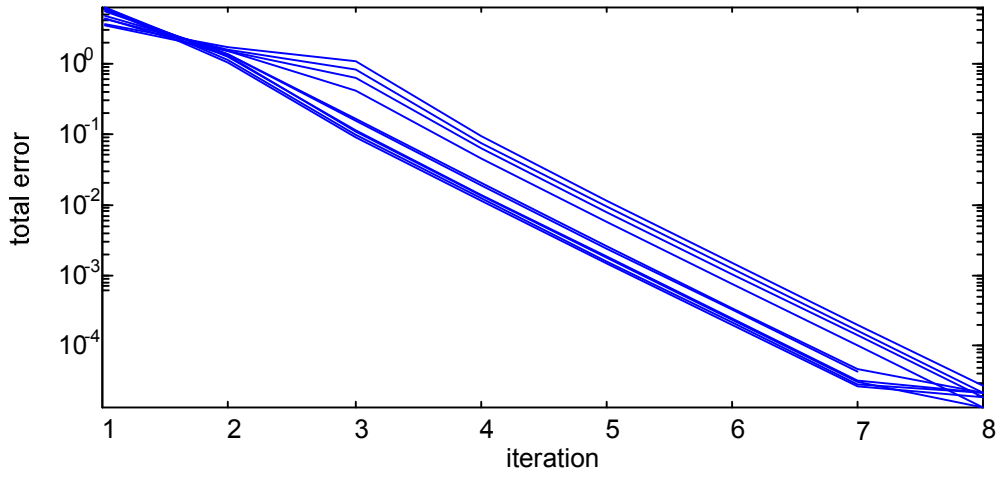
3.1.3 DISCUSSION

Although (3.5) and (3.7) appear algebraically equivalent, in practice, they differ in terms of performance. The disparity is a result of the fact that the underlying system of equations is overdetermined. Therefore, while they appear algebraically equivalent, they are in fact not. This

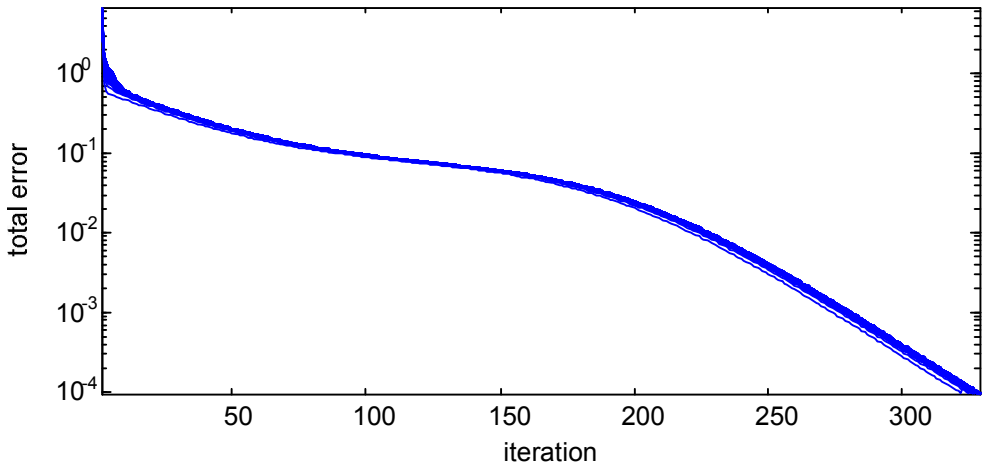
is due to the nature of the pseudo-inverse. In order for equivalence to be maintained, the following identity must hold:

$$(\mathbf{FX})^{-1} = \mathbf{X}^{-1}\mathbf{F}^{-1}.$$

Although the relation is valid whenever $\mathbf{F}$ and $\mathbf{X}$ are nonsingular square matrices, it does not hold in the case of pseudo-inversion. Therefore, the formulations given in (3.5) and (3.7) are not truly equivalent mathematically. Despite this fact, both methods do yield convergence; however, the rate of convergence for the method outlined in section 3.1.2 is significantly slower. This is evidenced by the comparison of the training error in Figure 3.1.



(a)



(b)

Figure 3.1: Ten nonlinear pseudo-inversion runs using (a) the iterative inversion method; and (b) the single inversion method.

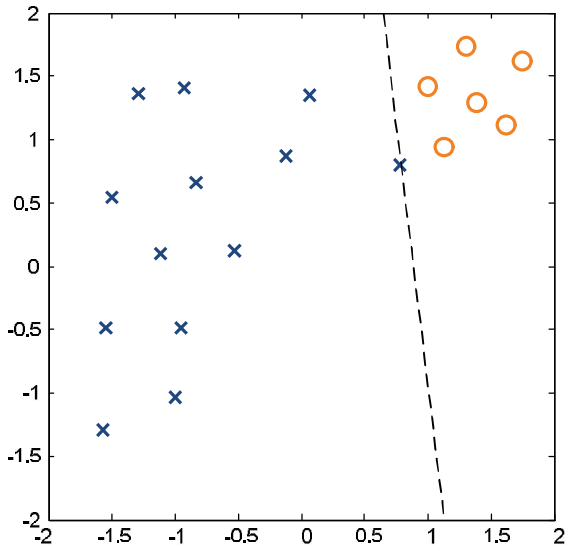


Figure 3.2: Separation of an unbalanced and poorly distributed data set using pseudo-inversion.

3.2 IMPROVED GENERALIZATION USING ACTIVE SET PSEUDO-INVERSION

When solving classification problems, the optimal orientation of the separating plane does not depend solely on the correct classification of the training data. It must also exhibit strong generalization. That is, it should also be capable of classifying additional data points with as high a degree of accuracy as possible given the information contained in the training set.

While the pseudo-inversion techniques outlined in section 3.1 are capable of reaching very low error levels in a relatively low number of iterations, closer inspection may reveal significantly less than optimal separating plane orientations when applied to training sets with unbalanced or poorly distributed data. Take, for instance, the data set shown in Figure 3.2. The separating plane represented by the dashed line was generated using the pseudo-inversion method described in Section 3.1.2 . Despite a total error of just 10^{-12}, it is clear from the figure that the placement of the separating plane is quite poor. This is due to the imbalance and poor

distribution of the training data. The larger number of data points on the left has resulted in a shift of the separating plane in that direction, and the poor distribution has had an undesirable influence on the plane's orientation.

Although it is true that the placement of the plane would improve if the training process were allowed to continue, the rate of the improvement would be quite slow due to the fact that the neuron is in saturation. The reason for this is that, in saturation, the derivatives of the activation function become quite small. Since the magnitude of the step is proportional to that of the derivatives, smaller derivatives mean smaller steps. To make matters worse, as the position of the separating plane improves, the total error continues to drop, causing the step size to do the same. Therefore, the rate of progress is inversely proportional to the number of iterations.

To overcome this issue, a modified pseudo-inversion training technique is presented that not only produces optimal or near optimal placement of the separating plane, but also achieves a higher rate of convergence. The method revolves around the use of a variably sized training set referred to as the *active set*.

The first major difference in the active set pseudo-inversion technique (ASPI) is that it uses the neuron's net values rather than the output values to determine the placement of the separating plane. To do this, a trick is employed which does not require any modification to the original update rule. All that is required is the replacement of the nonlinear activation function by a linear model with unity gain, which is equivalent to operating on the net value of the original model. Therefore, mathematically speaking, the update rule is of the same form as,

$$\Delta \mathbf{w} = \mathbf{F}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{E}, \tag{3.8}$$

where

$$\boldsymbol{F} = \begin{bmatrix} f_1' & 0 & \cdots & 0 \\ 0 & f_2' & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f_P' \end{bmatrix}, \boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1M} \\ x_{21} & x_{22} & \cdots & x_{2M} \\ \vdots & \vdots & & \vdots \\ x_{P1} & x_{P2} & \cdots & x_{PM} \end{bmatrix}, \Delta\mathbf{w} = \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_M \end{bmatrix}, \text{and } \mathbf{E} = \begin{bmatrix} d_1 - o_1 \\ d_2 - o_2 \\ \vdots \\ d_P - o_P \end{bmatrix}.$$

The only differences being that **t**he effect of the derivative values f_i' are ignored since the linear model has unity gain, and the output values of the original model are replaced with the linear model such that,

$$o_i = net_i = \mathbf{X}\mathbf{w}. \tag{3.9}$$

Therefore, ignoring **F** and substituting (3.9) into (3.8), the linear pseudo-inversion rule from section 2.2.2 is obtained. That is,

$$\begin{aligned} \mathbf{w} &= \mathbf{w} + (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{d} - \mathbf{X}\boldsymbol{w}), \\ &= \mathbf{w} + (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{d} - \mathbf{w}, \\ &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{d}. \end{aligned} \tag{3.10}$$

From this result, the relationship between the linear and nonlinear methods can be clearly seen.

Using the linear model offers two advantages. First, it restricts the magnitude of the weights, and second it removes the possibility of saturation. However, while this effectively removes the two primary restrictions faced by the nonlinear model, it also introduces a restriction of its own. Because the value of the output is no longer limited to the interval between plus and minus one, only those patterns which lie on or near the ± 1 contours of the plane defined by the net value are assigned the proper classification. The remaining patterns may contribute relatively large error values regardless of their position with respect to the separating plane, which adversely affects the placement of the separating plane in much the same way as the nonlinear model. This behavior is illustrated in Figure 3.3 using the data set from the previous example.

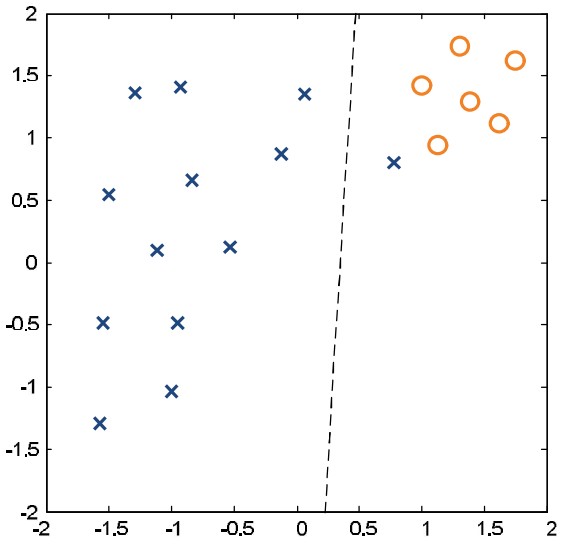


Figure 3.3: Separation using pseudo-inversion with linear activation function.

Fortunately, the undesirable behavior introduced by the linear model can be overcome by selectively reducing the number of training patterns at each iteration. The result is a variably sized subset referred to as the active set. During each subsequent iteration, only those patterns indexed by the active set A are used for training. The remaining patterns are simply ignored. To do this, an active input matrix is defined such that

$$\mathbf{X}_A = \begin{bmatrix} x_{a_1 1} & x_{a_1 2} & \cdots & x_{a_1 M} \\ x_{a_2 1} & x_{a_2 2} & \cdots & x_{a_2 M} \\ \vdots & \vdots & & \vdots \\ x_{a_n 1} & x_{a_n 2} & \cdots & x_{a_n M} \end{bmatrix} \tag{3.11}$$

for all $a_i \in A$ from $i = 1, \ldots, n$, where n is the number of indices contained in A.

The method used to determine which patterns should be included in the active set is quite simple. During each iteration, the active set is updated using the indices of all input patterns whose net values lie on the interval between ± 1. That is, they must lie within the region bounded by the ± 1 contours of the activation function. As it turns out, due to the LMS formulation of the

update rule, reduction of the size of the active set from one iteration to the next is guaranteed until a minimal set is reached, for which there exists a feasible separating plane such that all data points in the remaining set are collinear and run parallel to the plane. Thus, using the reduced dataset, an exact solution to the linear system in (3.10) can be found.

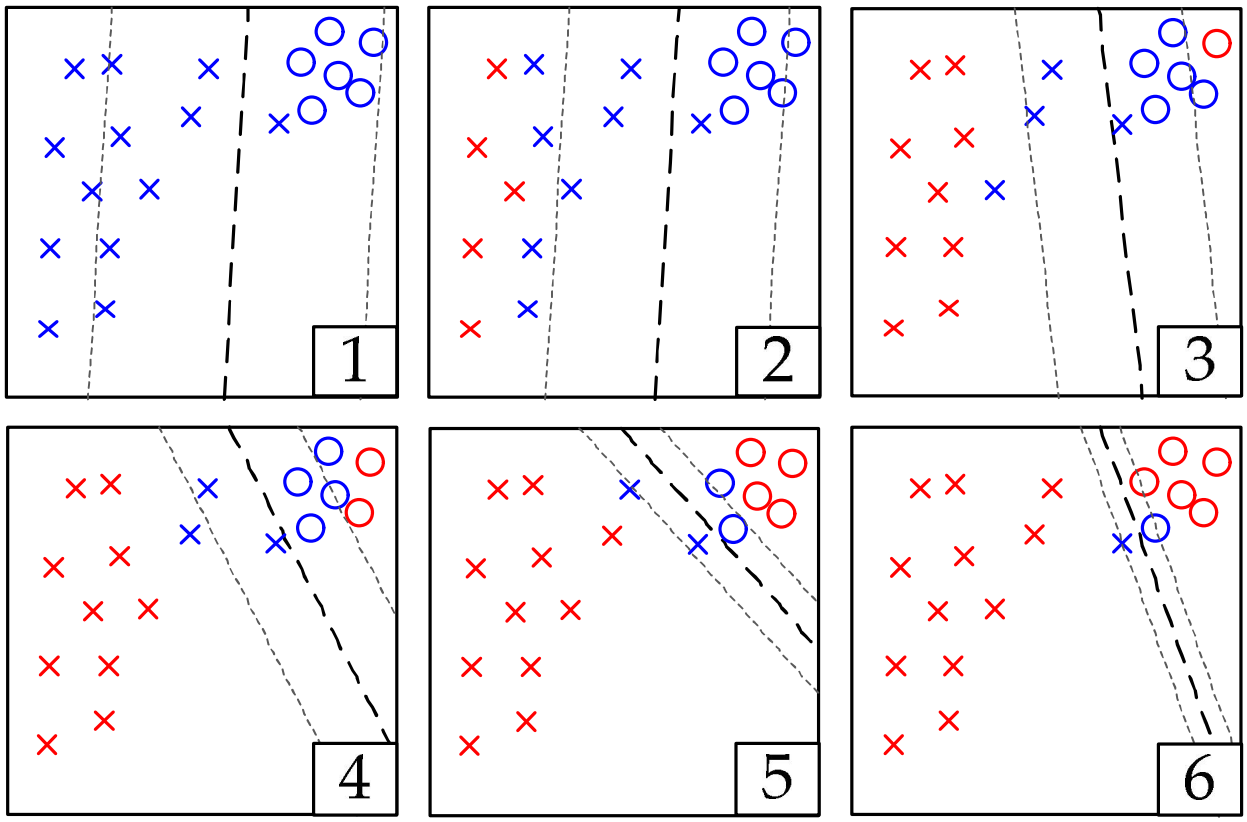


Figure 3.4: Eight iterations using ASPI. Patterns in the active set are shown in blue.

Six iterations of this process are shown in Figure 3.4, using the data set from the previous examples. The active set used at each iteration is represented by the blue data points. The remaining data points, shown in red, are ignored. The ± 1 contours used for determining the each subsequent set are represent by the gray dashed lines, and the current separating plane is shown

in black. At the sixth iteration, the two remaining blue points represent the minimal active set. It is important to note that for the minimal set, all active patterns lie on the ± 1 contours, which signifies that an exact solution to (3.10) has been found, and the training process is complete.

Once the proper separating plane has been determined, the resulting weight vector can then be scaled to produce as low a total error as may be desired. The reason for this is that the orientation of the separating plane for a neuron with a tangent hyperbolic activation function is entirely determined by the equation for its net value, which is obtained from the active set solution to (3.10). The only remaining parameter for the activation function is the gain, which controls the slope of the activation function in the linear region. For the tangent hyperbolic function, as the value of the gain is increased, the shape of the activation function approaches that of the sign operator. This means that once the separating plane is set, increasing the gain of the activation function effectively reduces the total error. This behavior is illustrated Figure 3.5. From the figure, it is clear that as the gain increases, the data points represented by the orange x and o converge toward their desired binary output values, thereby reducing the associated errors. Therefore, any arbitrary desired value for the maximum error per pattern can be set by simply scaling the weights.

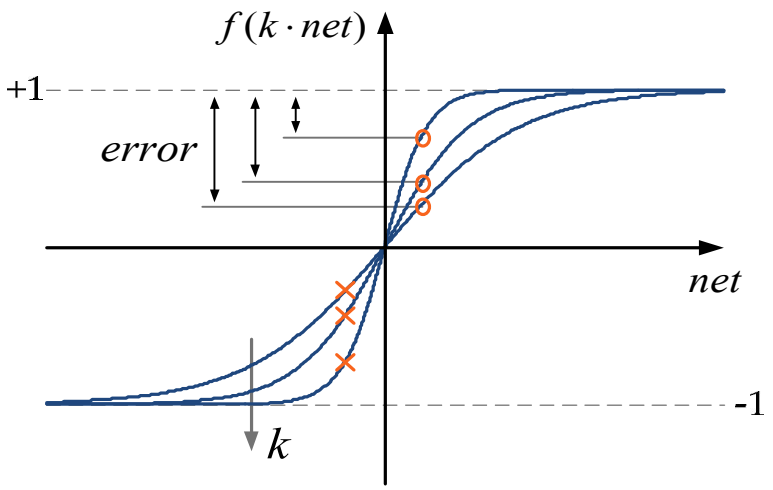


Figure 3.5: An illustration of error reduction as a function of gain.

The reason for this is straight forward. First, since the patterns in the active set are known to be those closest to the separating plane, they must also produce the largest error values. It follows, then, that by setting the errors for the active patterns equal to e_{max}, it is guaranteed that the errors for remaining patterns will be less strictly less than e_{max}. Therefore, a scalar value s is desired such that that

$$e_{max} = d_i - \tanh(k \cdot s \cdot net_i), \forall i \in A. \tag{3.12}$$

Conveniently, due to the fact that the separating plane resulting from (3.10) is equidistant from all patterns in A, satisfying the above equation for *any* pattern in a A is equivalent to satisfying it for *all* patterns in A. In addition, it is also known from (3.10) that the net value values for the active patterns are equal to their desired outputs. Therefore, (3.12) can be simplified by choosing any pattern i such that $d_i = net_i = 1$. This results in the equivalent relation

$$e_{max} = 1 - \tanh(k \cdot s) \tag{3.13}$$

Finally, solving (3.13) for s, the following equation for the weight scale factor is obtained:

$$s = \frac{\tanh^{-1}(1 - e_{max})}{k}. \tag{3.14}$$

In summary, not only is the active set method exceptionally fast, but it also offers three distinct advantages over the previous methods:

1. The placement of the separating plane is guaranteed to be equidistant from the nearest patterns in ether class.
2. By reducing the number of patterns used to compute the step during each subsequent iteration, the computational complexity is also reduced.
3. The maximum pattern error can be directly specified by the user.

3.3 MAXIMIZING THE MARGIN OF SEPARATION

While the active set method presented in the previous section offers a number of significant improvements, it still does not guarantee optimal separation in every sense. This is due to what is known as the margin of separation, which refers to the distance between the separating plane and the contour lines passing through the data points that comprise the optimal active set. For optimal separation, the orientation of the separating plane should achieve the largest possible margin of separation for which (3.10) remains satisfied. For a classification with an N-dimensional input space, the active set solution will, in fact, produce the maximum margin of separation, so long as the number of active patterns in either class is greater than or equal to N. However, if the number of active patterns in both classes is less than N, the maximum margin is no longer guaranteed. This is because for the latter case, there is more than one unique solution to (3.10). To state this formally, if A_+ and A_- are the active set patterns whose desired outputs are plus and minus 1 respectively, then

1. If the number of patterns in *either* A_+ *or* A_- is *greater than or equal to* N, then there is only *one* unique solution to (3.10).
2. If the number of patterns in *both* A_+ *and* A_- is *less than* N, then *more than one* unique solution exist.

To illustrate this, a two dimensional example is shown in Figure 3.6. The active set in Figure 3.6a, represented in blue, is an example of case one. As expected, the solution shown here is the only possible solution for which all points in A lie on the ± 1 contour lines. For the second case, represented by the example in Figure 3.6b, two possible solutions are shown. The latter case also illustrates the difference in the separating margin from one solution to the next. Clearly, solution number 2 has the largest margin of separation, and is the more preferable solution.

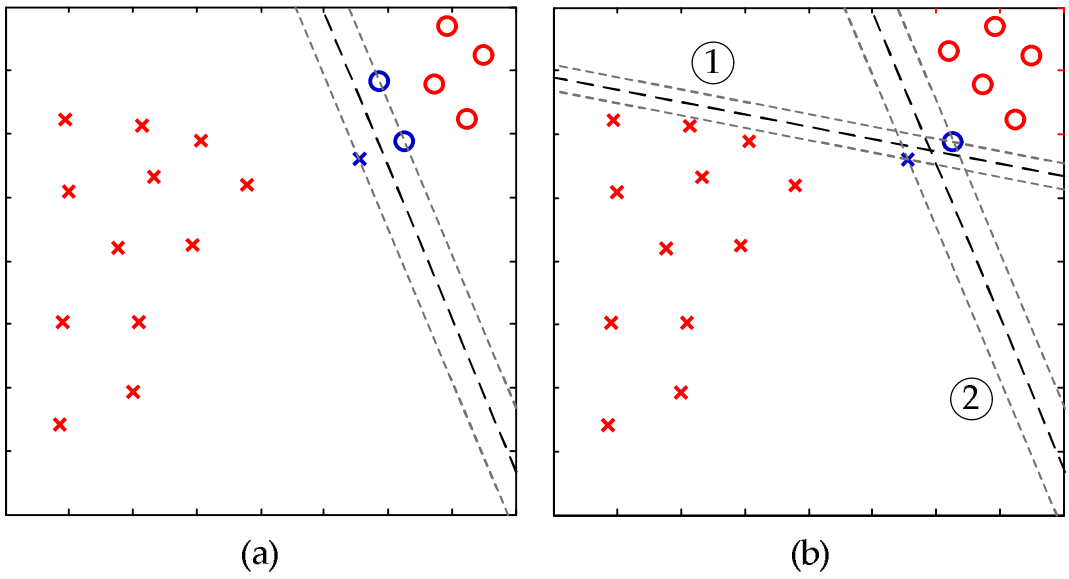


Figure 3.6: (a) If the number of patterns in either A_+ or A_- is greater than or equal to N, then only one unique solution can be found; (b) When the number of patterns in both A_+ and A_- is less than N, more than one unique solution exist.

In this section, a technique is presented that, when used in conjunction with the active set method, *guarantees* a maximal margin of separation. Therefore, using the proposed method, a truly optimal separation is obtained.

3.3.1 FORMULATING A MARGIN MAXIMIZING OBJECTIVE

The goal of the proposed method is to determine a separating plane which achieves a maximum margin of separation for the data points in the active set. To do this, a functional relationship between the separating margin and the weights must be found, beginning with the general description of the separating plane,

$$\mathbf{w} \cdot \mathbf{x} + b = 0, \tag{3.15}$$

where $\mathbf{w}$ is the weight vector, $\mathbf{x}$ is the input vector, and b is the biasing weight. Furthermore, in order that the patterns in $\mathbf{X}_A$ lie on the ± 1 contours, a set of equality constraints must also be defined,

$$\mathbf{w} \cdot \mathbf{X}_A + b \quad = \quad \mathbf{d}_A, \tag{3.16}$$

where $\mathbf{X}_A$ is the active pattern matrix and $\mathbf{d}_A$ is the corresponding vector of desired outputs. Thus, for two vectors $\mathbf{x}_1 \in \mathbf{X}_{A+}$ and $\mathbf{x}_2 \in \mathbf{X}_{A-}$ on opposite sides of the separating plane,

$$\begin{aligned} &\mathbf{w} \cdot (\mathbf{x}_1 - \mathbf{x}_2) = 2, \text{and} \\ &(\mathbf{x}_1 - \mathbf{x}_2) = 2\mathbf{w}. \end{aligned} \tag{3.17}$$

Using (3.17), the margin can be defined as the projection of $(\mathbf{x}_1 - \mathbf{x}_2)$ onto the vector normal to the separating plane, which is equivalent to the normalized weight vector $\hat{\mathbf{w}} = \mathbf{w}/\|\mathbf{w}\|$. Therefore,

$$\mu = \text{proj}_{\hat{\mathbf{w}}} (\mathbf{x}_1 - \mathbf{x}_2) = \frac{2}{\|\mathbf{w}\|}. \tag{3.18}$$

It follows. Then, that maximizing the margin described by (3.18) is equivalent to minimizing the following equality constrained quadratic:

$$\min_{\mathbf{w},b} \ \frac{1}{2}\|\mathbf{w}\|^2, \text{ subject to } \mathbf{w}\cdot\mathbf{X}_A + b = \mathbf{d}_A. \tag{3.19}$$

The formulation in (3.19) represents an equality constrained quadratic program (QP).

The Lagrangian of the equality constrained QP in (3.19) is defined as

$$L(\mathbf{w},b) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{P}\alpha_i\left[d_i\left((\mathbf{w}\cdot\mathbf{x}_i)+b\right)-1\right], \tag{3.20}$$

where P is the number of patterns and α_i are the Lagrange multipliers. Solving for the saddle points of the Lagrangian and substituting the result back into (3.20) yields the equivalent Wolfe dual formulation of the QP,

$$\min_{\alpha} \ \frac{1}{2}\sum_{i=1}^{P}\sum_{j=1}^{P}\alpha_i\alpha_j d_i d_j\left(\mathbf{x}_i\mathbf{x}_j\right) - \sum_{i=1}^{P}\alpha_i \text{ , subject to } \alpha_i \geq 0. \tag{3.21}$$

For the Lagrangian formulation of a constrained QP, the objective is defined as the gradient of the original objective minus the gradients of the constraints. Therefore, solutions are characterized by points for which the combined direction of the gradients of the active constraints points in the opposite direction of the gradient of the objective[14]. From a geometrical standpoint, this makes sense. If the gradients are in opposition, no direction exists by which the original objective function can be reduced while still satisfying the active set constraints. However, simply minimizing the difference in the gradients is not sufficient since their magnitudes may differ. To alleviate this problem, the Lagrange multipliers are used to scale the constraint gradients so that their combined magnitudes will be equal to that of the objective.

Normally, for inequality constrained problems, the influence of the inactive constraints is ignored by setting the corresponding multipliers to 0. If the Lagrange multiplier for any of the active constraints is found to be negative, then it is no longer a blocking constraint, and it is removed from the working set of constraints. This process is repeated until a solution is found for which all of the multipliers are greater than or equal to zero. However, for the set of points found using the active set method in the previous section, all of the data points yield strict equality constraints. This means that the all the resulting constraints will be active, and their corresponding Lagrange multipliers will be positive and non-zero. Therefore, the dual constraints on α_i in (3.21) may be ignored, resulting in an unconstrained minimization over α.

Quadratic programs of this form are encountered in a wide range of applications, and a number of efficient methods have developed for solving them.

3.4 IMPROVING GENERALIZATION IN FULL SCALE NETWORKS

Although the presented pseudo-inversion techniques are only directly applicable for training individual neurons, they can be used to optimize the weights found by nonlinear gradient methods using a simple retraining process. The resulting set of weights yield optimal generalization for the given set of selected features.

The first step in the process is to train the network using any standard gradient based search method. Once convergence is achieved, the gradient search process is terminated as usual. The outputs of all neurons are then recorded using the final weights of the gradient search, and a set of desired outputs is defined for each neuron by taking the signs of the measured values.

Next, the neurons are ordered such that the position of each neuron in the order is greater than that of all neurons connected to its input. In other words, for a given neuron, all neurons downstream have higher value in the order. This ensures that when it comes time to retrain each

neuron, the outputs of all neurons connected to it will be available to use as input patterns during training. After the ordering is determined, each individual neuron is retrained using the margin maximizing technique described in the previous section. In this way, the separating margin for each neuron is maximized, resulting in optimal generalization for the network as a whole. Since the ability to guarantee optimal separation is one of the most commonly cited factors when discussing the motivation for using SVM methods over ANN methods, this represents a significant step for neural classification.

Chapter 4

TRAINING NEURAL CLASSIFIERS USING A SEARCH OF THE HIDDEN SPACE

Generally speaking, neural networks are trained using the current output values to compute or approximate the first and second partial derivatives of the network error using backpropagation of the error through the network. This information is then used to determine a step direction which effectively reduces the total error of the network. Once an adequate step is found, all of the neurons are updated simultaneously. While this method has proven successful, the error functions it introduces can be rather complicated, making the training process slower and more susceptible to entrapment[21]. A faster and more efficient approach would be to train each neuron independently using the techniques described in the previous chapter; however this is not usually possible since the desired outputs are typically known only for those neurons that reside in the output layer. In this chapter, an approach is presented which seeks to determine these hidden outputs using a systematic realignment of the feature space. The result is a more robust algorithm which is not only able to escape entrapment in false minima, but also makes direct use of the information they provide in order to proceed.

4.1 OVERVIEW

If the desired outputs are known for all of the nodes in a given network, each neuron may be trained independently. This method of training is attractive for two reasons: one, because the training of an individual neuron requires relatively little computational effort, making convergence quite rapid, and two, because the error surface does not contain any false minima,

which eliminates the possibility of entrapment. Unfortunately, as previously stated, the desired outputs for the hidden layer neurons are not generally available a priori, and in most cases, the number of possible combinations makes a sequential search impractical. For some perspective, a problem with m input patterns and n hidden neurons would result in 2^{mn} possible combinations of hidden layer outputs. Thus, even a relatively small problem such as parity-7, which has 128 input patterns and 3 hidden neurons (using a bridged architecture) would have nearly 4×10^{115} possible choices! While some combinations could be ruled out immediately, the number of remaining choices would still be prohibitively large. The only way to proceed would be to somehow narrow the scope of the search. To do this, the proposed method uses entrapment in local minima, one of the weaknesses of gradient based approaches, to its advantage.

Entrapment occurs when an algorithm converges towards a solution which is not optimal. Unable to reduce the error any further, the process must be terminated. When this occurs, training is reinitiated using a randomly generated set of new weights, and the old weights are discarded. However, while the discarded weights may not produce the desired classification for all patterns, the number of misclassified patterns may be quite small. Therefore the corresponding hidden layer outputs might offer a useful starting point for a search of the feature space. If this information could be used to determine the desired outputs of the hidden layer neurons, then each neuron in the network could be trained independently, which is the goal of the proposed method.

4.2 ALGORITHM DESCRIPTION

The proposed method, referred to as the hidden layer pseudo-inversion algorithm (HLPI), begins by using a second order learning algorithm to train the network in the usual manner. Where it differs is primarily in the way it handles entrapment. If the algorithm fails to converge,

the second order process is halted, and the second phase of training begins. In the second phase, the values of the current hidden layer outputs are used as a starting point for determining the correct set of hidden layer outputs. To do this, the hidden neurons are systematically retrained in an attempt to reclassify the remaining misclassified patterns using pseudo-inversion. This process is performed using the following steps.

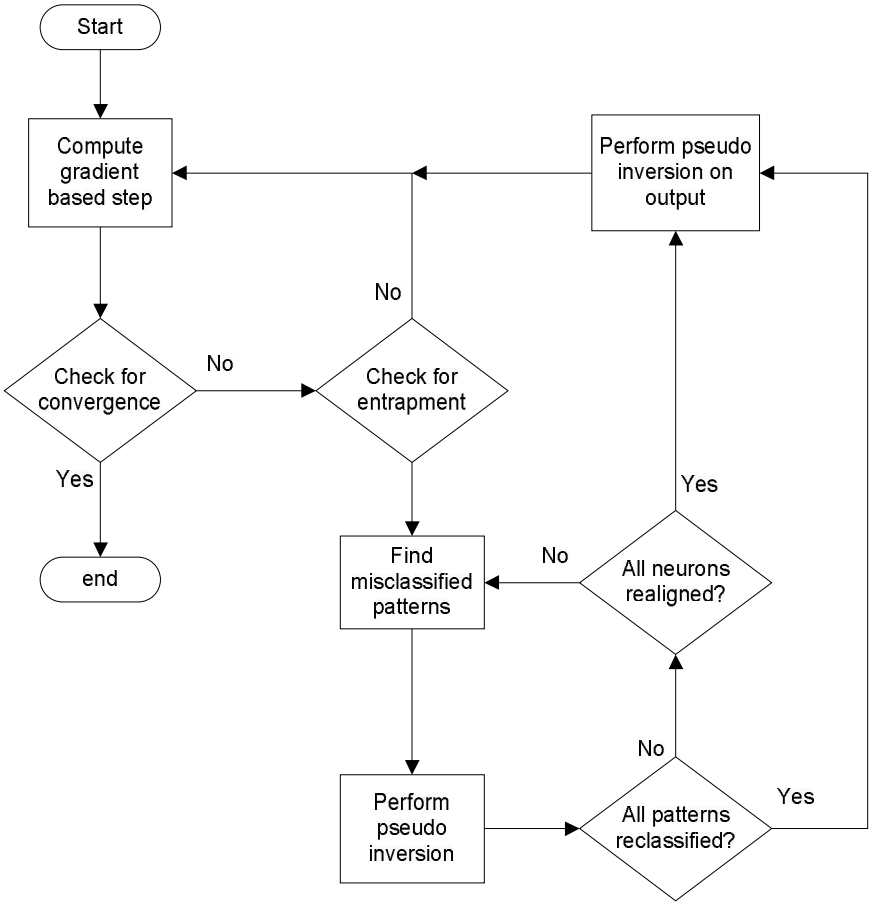


Figure 4.1: Flow chart of the HLPI algorithm.

Step 1: Train the network using the gradient method until the process converges. If all patterns are classified correctly, terminate training and return the final weights. If not, proceed to step 2.

Step 2: Find the hidden unit n for which the net values of the misclassified patterns are closest to 0. Define a set of desired outputs by taking the sign of the current outputs and then multiplying the outputs for any misclassified pattern by -1. That is, let the set of desired outputs do^n be defined as

$$do^n = \{\text{sign}(X_i^n)\} \cup \{-sign(X_j^n)\},$$

where X^n is the set of current outputs for hidden neuron n, i contains the indices of all correctly classified patterns, and j contains the indices of all misclassified patterns.

Step 3: Perform weighted pseudo-inversion (described in the next section). Incrementally scale the derivatives for the misclassified patterns and repeat until one or more of the misclassified patterns is reassigned.

Step 4: Let k be the indices of all correctly classified patterns whose outputs are unchanged and all misclassified patterns whose outputs were reassigned, and let l contain the indices of all remaining patterns. Find the neuron whose net values for the patterns denoted by l are closest to 0, and define a new set of desired outputs

$$do^n = \{\text{sign}(X_k^n)\} \cup \{-sign(X_l^n)\}.$$

Step 5: Repeat steps 2 and 3 until the number of indices in l reaches 0, or until all hidden units have been retrained.

Step 6: Perform pseudo-inversion for the output layer neurons using the updated outputs of the hidden units. If all patterns are classified correctly, terminate training and return the final weights. If not, repeat steps 1 through 6.

A flowchart of this process is depicted in Figure 4.1.

The pseudo-inversion technique used in step 3 is a weighted variation of the non-linear method presented in Section 3.1.2 . The details of this method are presented in the next section.

4.2.1 WEIGHTED PSEUDO-INVERSION TRAINING

In Chapter 3 a series of pseudo-inversion techniques were presented for optimizing the weights of a single neuron with a non-linear activation function. A variation of this technique is presented here, and is used during the realignment phase of the proposed learning algorithm.

The goal of the realignment process is to systematically adjust the feature detectors in the hidden space in order to achieve better separation in the output layer. This is accomplished by retraining the hidden layer neurons one at a time in an attempt to invert the signs of the current node values for all misclassified patterns, while preserving the signs of the remaining patterns. Thus, a desired output vector is defined using the current node values for all correctly classified patterns and the inverted values for all misclassified patterns. Unfortunately, there is no guarantee that the newly specified classification will be linearly separable. Therefore, in the event that linear separation is not achievable, a modified objective must be used. The primary goal in such cases remains the inversion of the node values of the misclassified patterns. However, the node values of the correctly classified patterns should not be ignored either since they play an important role in separation at the output. Therefore, simply performing pseudo-inversion is not sufficient since there is no guarantee that the misclassified patterns will be reassigned; and placing emphasis on the misclassified patterns by ignoring the remaining patterns is also not advisable since it may have an adverse effect on separability at the output layer. Clearly, a compromise must be made. An approach is needed that seeks to invert as many of the misclassified output values as possible while at the same time preserving as many of the correctly classified values as possible. To do this, an incremental weighting scheme is introduced that gradually increases the emphasis on the misclassified patterns until one or more of the corresponding node values is inverted. In this way, a subset of the misclassified patterns can be

inverted while maintaining the influence of the correctly classified patterns on the feature realignment process. The resulting update rule is given by

$$\Delta \mathbf{w} = \mathbf{F}_{\omega}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{E}, \tag{4.1}$$

where

$$\mathbf{F}_{\omega} = \begin{bmatrix} \omega_1 f_1^{'} & 0 & \cdots & 0 \\ 0 & \omega_2 f_2^{'} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \omega_p f_P^{'} \end{bmatrix}, \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1M} \\ x_{21} & x_{22} & \cdots & x_{2M} \\ \vdots & \vdots & & \vdots \\ x_{P1} & x_{P2} & \cdots & x_{PM} \end{bmatrix}, \Delta \mathbf{w} = \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_M \end{bmatrix},$$

$$\text{and } \mathbf{E} = \begin{bmatrix} d_1 - o_1 \\ d_2 - o_2 \\ \vdots \\ d_P - o_P \end{bmatrix}.$$

The values ω_i are the weight values used to scale the derivatives of the activation function for each pattern. The weight values are determined according to which patterns are to be inverted. For correctly classified patterns, the value of ω_i is set to 1. For all misclassified patterns, ω_i is set equal to the current value of the incrementing scale. Pseudo-code for the complete process is shown in Figure 4.2.

```
scale = 1;
reclassified = false;
while reclassified == false;
        for ite = 1to nite
                w = w + (F'*X)/E;
                out = tanh(k*X*w);
                E = dout – out;
                der = k*(1-out^2);
                for p = 1 to P
                        F'(p,p) = der(p);
                        if p ∈ ndx
                                F'(p,p) = F'(p,p)*scale;
                                if E(p) < margin
                                        reclassified = true;
                                end
                        end
                end
        end
        scale = scale + step;
end
```

Figure 4.2: Pseudo-code for the weighted pseudo-inversion technique.

The variables and constants in Figure 4.2 are defined as follows:

- *scale* stores the incremental weight values used to scale the derivatives of the activation function for all misclassified patterns.
- *reclassified* is a binary value that represents the state of the reclassification process. If any of the misclassified patterns has been reclassified, the value is true, otherwise the value is false.
- *w* is a vector containing the weights on the neuron's inputs.
- *F'* is a $P \times P$ diagonal matrix containing the scaled activation function derivatives for all patterns.

- X is a matrix whose rows contain the input patterns to the neuron.
- E is a $P \times 1$ vector containing the errors associated with each of the input patterns.
- *out* is a $P \times 1$ vector containing the output values for each of the input patterns.
- *dout* is a $P \times 1$ vector containing the desired output value for each input pattern.
- k is the gain of the neuron.
- *nite* is the number of pseudo-inversion updates performed between increments of the derivative scaling factor.
- *der* is a $P \times 1$ vector containing the derivative of the activation function for each of the input patterns.
- *margin* is a constant value which determines how close the outputs for the misclassified patterns must be to the desired values before they are considered reclassified.
- *step* is a constant that determines size of the increments in the derivative scaling factor.
- P is the number of input patterns

4.3 DETECTING ENTRAPMENT

The purpose of the proposed method is to improve the success rate of training by realigning the hidden layer neurons whenever the gradient search fails to converge. In order to do this effectively, a method is needed for determining when the algorithm becomes trapped. The proposed method tests for three conditions to determine the current state of the training process.

The first test checks for changes in the pattern separation at the output of the network over a preset interval. If, at the end of the specified number of iterations (γ), the separation of the training data is unchanged, it is likely that the algorithm has become stuck. A binary valued flag

($cond_c^k$) is used to track the current status of the classification at each iterate (k). Therefore, the test for this condition is defined by

$$cond_c^k = \begin{cases} 1 \text{ if } \sum_{i=1}^{P} sign\left(o_i^k\right) \cdot sign\left(o_i^{k-\gamma}\right) = P, \\ 0, \text{ otherwise.} \end{cases} \tag{4.2}$$

As an example, suppose a value of 20 was chosen for γ. This would mean that at the end of each iteration, the resulting output values would be compared with those generated 20 iterations before. If the values were found to be the same, then the $cond_c$ flag would be set equal to one.

The separation test alone is not sufficient for detecting entrapment. One reason for this is that once a correct separation has been reached, it may still require a number of iterations to reduce the total error to the desired level, during which time the separation test may return true. To avoid the possibility of false detection, a second condition is tested by comparing the current separation to that of the desired outputs. This is done by defining a second condition flag ($cond_s^k$), which is defined as

$$cond_s^k = \begin{cases} 0 \text{ if } \sum_{i=1}^{P} sign\left(o_i^k\right) \cdot d_i = P, \\ 1, \text{ otherwise.} \end{cases} \tag{4.3}$$

The final condition check for the percent decrease in the total error from one iteration to the next. This is important in the early stages of training when the convergence may be slow due to the low output values produced by the initial weights. In order to avoid false detection during this stage, the percent improvement in the total error is monitored. If the percent difference is less than the desired value (ϵ), the third flag ($cond_e^k$) is set. That is,

$$cond_e^k = \begin{cases} 1 \text{ if } \frac{TE_{k-\sigma} - TE_k}{TE_{k-\sigma}} < \varepsilon, \\ 0 \text{ otherwise,} \end{cases} \tag{4.4}$$

where TE_k is the total error for the k^{th} iterate, and σ is the size of the interval used in the comparison. For instance, if a value of $\sigma = 5$ was selected, the $cond_e$ condition would be determined by the percent difference in total error between the current iterate and the one generated five iterations before it.

Used together, these three conditions offer a reliable method for detecting entrapment in local minima. If all three flags came back true during the same iteration, the gradient search is terminated and the realignment process begins.

4.4 AVOIDING REDUNDANT FEATURE SELECTION

There are a number of conditions which may lead to false convergence, the majority of which can be difficult or even impossible to detect. There is, however, one such condition that can be detected quite easily. The condition is known as redundant feature selection, and occurs when two or more hidden units attempt to perform the same classification task[22]. Obviously, this is a problem when training networks with optimized architectures. For certain classes of problems, such as analog-to-digital conversion, this issue can be especially significant. A simple method for detecting and avoiding redundancy is presented in this section.

There are two means by which redundant hidden neurons can be detected. One is to compare the weights. The drawback to using this method is that different weights do not guarantee a different behavior. Weight vectors may differ a great deal in magnitude, and even orientation, and still produce a very similar separation of the test patterns. In order to make the test reliable, the neurons must be allowed to reach saturation, and their weights must be normalized. Allowing the neurons to reach saturation removes the possibility for early detection, and limits the network's flexibility once any similar neurons have been realigned. This lack of flexibility tends to draw the network back to the previous state.

The second method of detecting redundant hidden neurons is to compare the sign of the neurons' outputs. This offers a more reliable test and does not require the neurons to be in saturation. Although the dimensions of the neurons' output vectors are generally greater than that of the weight vector, there is no need to normalize the output vectors since the dot product of two identical outputs will be equal to the sum of either vector's elements. In addition, the computational requirements can be further reduced by only performing the test periodically, rather than after each iterate.

4.5 GRAPHICAL REPRESENTATION OF THE TRAINING PROCESS

During the development of the proposed training algorithm, it became apparent that to verify the method was operating correctly, a significant amount of data would have to be analyzed after each training attempt. To complicate matters further, in order to determine the state of the training process, much of the data generated in a given iteration needed to be compared with that from previous iterations. This process was time consuming and made debugging quite difficult. To solve this problem, a visual representation was devised that allows large amounts data to be combined into a single graph, and offers a concise and easy to read summary of the training process.

To do this, the outputs of each neuron are represented as bitmap images. Each pixel represents the output of a single neuron at a specific iteration. Therefore, the image generated for each neuron has dimensions equal to the number of patterns by the number of iterations. The color of each pixel signifies the sign of the output, with negative values drawn in blue and positive values in red. The intensities of the pixels are proportional to their respective values. Lighter pixels correspond to smaller values, and darker pixels are used to represent larger ones. Once the output images have been generated for all neurons in the network, they are stacked

vertically to form a combined image. The resulting composite contains nearly all of the information needed to evaluate the performance of the algorithm. A list of the available information is given below.

- Classification: The position of each pattern with respect to the separating plane can be determined by the sign of its corresponding output, which is denoted by the color of its representative pixel. Therefore, since each column of pixels represents one iteration, the classification of the input patterns at any given iteration can be determined by the colors of the pixels in that column.
- Movement of the separating plane: Since the pixels in each row represent the output values of a single pattern from one iteration to the next, if two consecutive pixels share a common row but are of different colors, then the position of the separating plane has changed.
- Entrapment: If the colors and intensities of the pixels in each row become static, then the position of the separating plane must also be static. Therefore, if the desired classification has not yet been satisfied, the algorithm is most likely trapped in a false minimum.
- Redundant feature detection: If the columns of two or more hidden layer neurons have matching pixel colors, they are performing the same task.
- Pattern saturation: If the intensity of the pixels in a given row becomes high, the output for the corresponding pattern has reached saturation. In general, as the number of saturated patterns becomes larger, the mobility of the separating plane is reduced.

Not only does the generated plot provide the information listed above, it also allows the data for all iterations to be evaluated simultaneously. This greatly simplifies the debugging and verification process.

An example image of the training of a two neuron cascade for parity-3 using EBP is shown in Figure 4.3. Looking at the total error alone, there is little if any noticeable change over the first 30 iterations. However, the output images tell a much different story. First, the low intensity of the pixels means that the patterns have not yet reached saturation. Therefore, there is still a great deal of mobility for the separating plane. This is confirmed by the changes in the pixel color, which signify changes in the orientation of the separating plane. Furthermore, the first 5 columns of the bitmap for the output neuron show that the algorithm is experiencing some oscillation. This would suggest that the selected learning constant may be too large, and could cause instability. It is also evident that despite the low change in error, by the 30^{th} iteration the hidden neuron has achieved a linearly separable mapping, after which an additional 6 iterations are required for separation of the augmented set of inputs. After 36 iterations the patterns have been correctly classified, and the remaining error is slowly reduced as the outputs are driven further into saturation. This can be seen by the gradual increase in the intensity of the remaining pixels.

Clearly, representing the training process in this way provides a great deal of additional insight. Furthermore, all of these observations can be determined directly from Figure 4.3, with no additional analysis required, making it an efficient and powerful tool for debugging and verification.

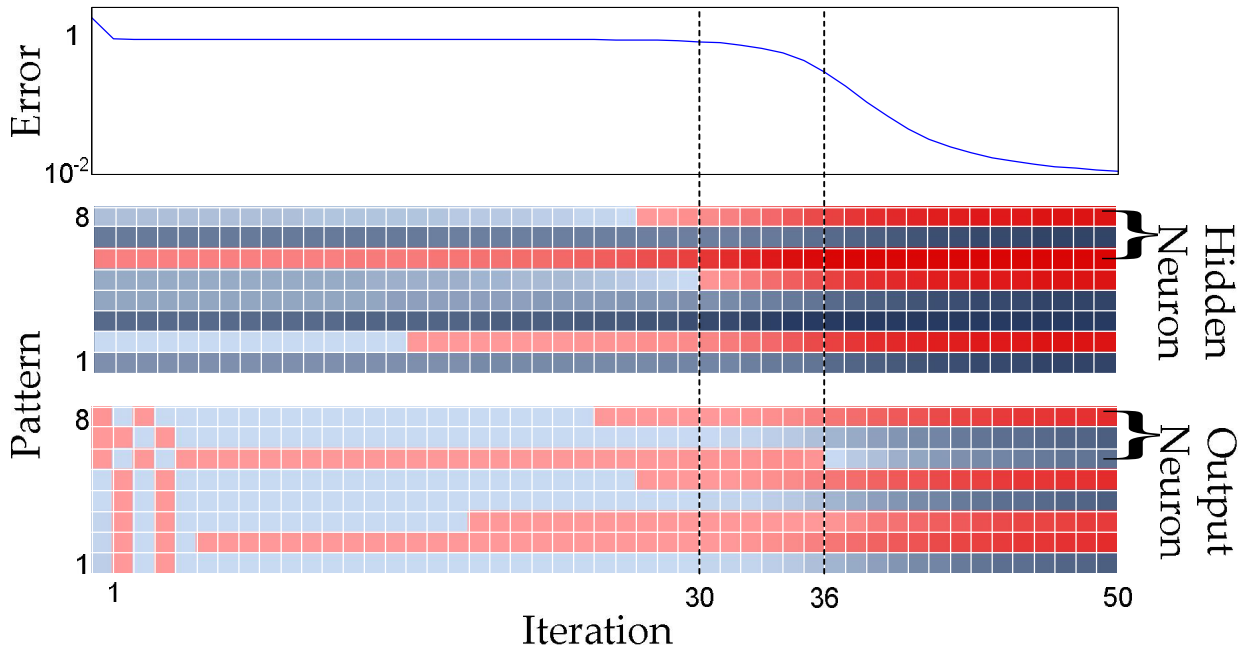


Figure 4.3: Output image for the parity-3 problem trained with EBP.

4.6 HLPI TRAINING EXAMPLE

In this section, a step-by-step description of the training process is presented, and the results are verified using the graphical representation described in section 4.5. For the purpose of this demonstration, the parity-4 problem is selected. The reason for this choice is that, despite being relatively difficult, the problem has only 16 training patterns and requires just 2 neurons in the hidden layer. Thus, the problem's manageable size makes it well suited for a detailed analysis. The output image for this problem, shown in Figure 4.4, is used as a reference for the remainder of this discussion. In addition to the node values and total error, the training summary in Figure 4.4also specifies which training patterns remain misclassified following each iteration. The output pixels of the misclassified patterns are outlined in green and crossed out.

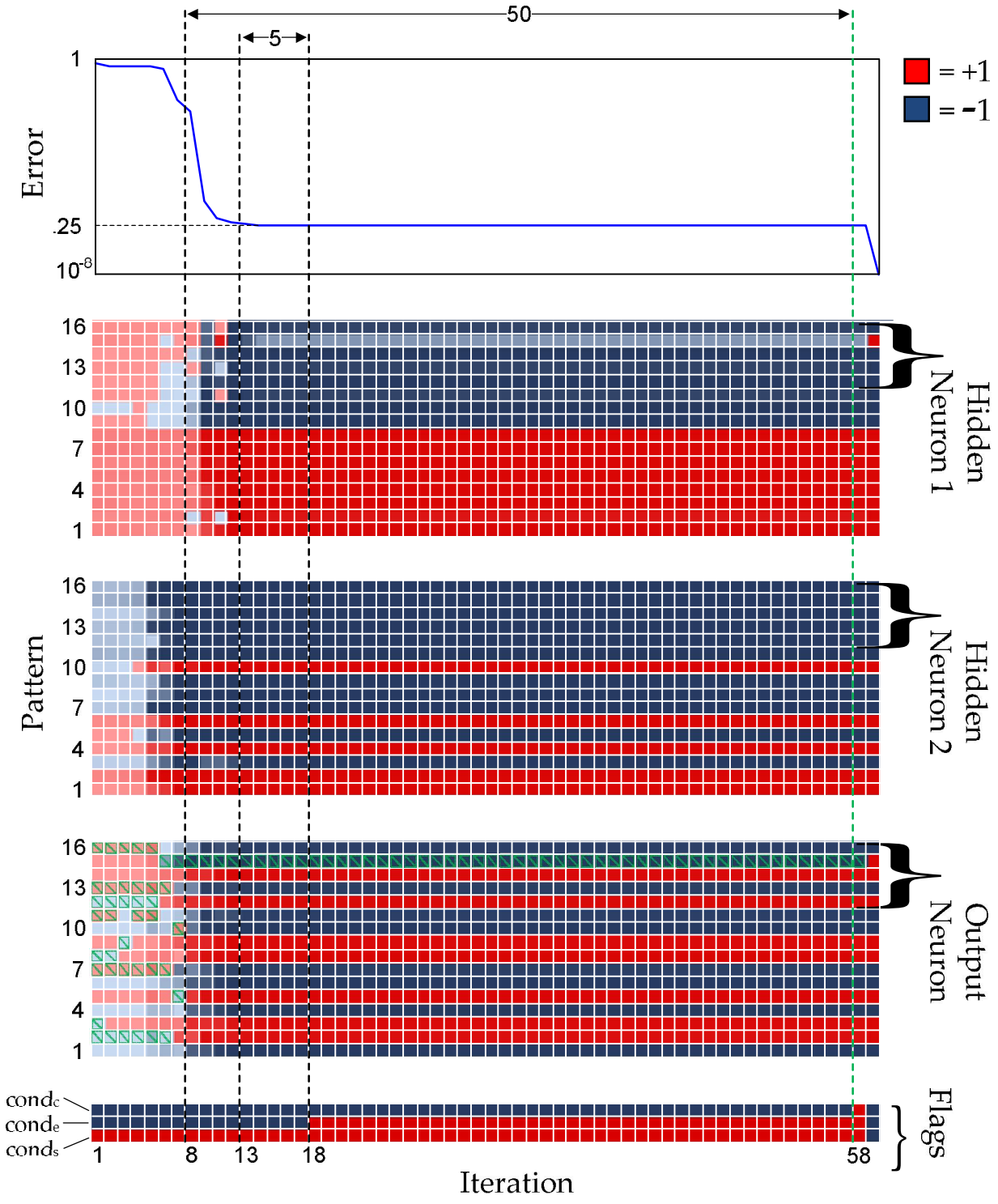


Figure 4.4: Output image for the parity-4 problem trained using the proposed method.

OVERVIEW

The training process is carried out in two phases. The first phase involves a gradient search using the LM algorithm, and comprises the first 58 iterations shown in Figure 4.4. After the 58th iteration, all 3 condition flags have been set, signifying that the algorithm has become trapped in a false minimum. At this time the second phase begins, during which a search of the hidden space is performed using pseudo-inversion of the hidden layer neurons. At the end of the pseudo-inversion process, linear separation is achieved resulting in the successful classification of the training patterns. The training process is then terminated, and the final weights are returned.

For this example, the parameters used in determining the state of the condition flags are chosen to be $\gamma = 50$, $\sigma = 5$ and $\epsilon = 10^{-4}$. For the network topology, a 3-neuron single layer bridged architecture, shown in Figure 4.5, is used. The activation functions for all three neurons are tangent hyperbolic, and have gains of $k = 0.1$.

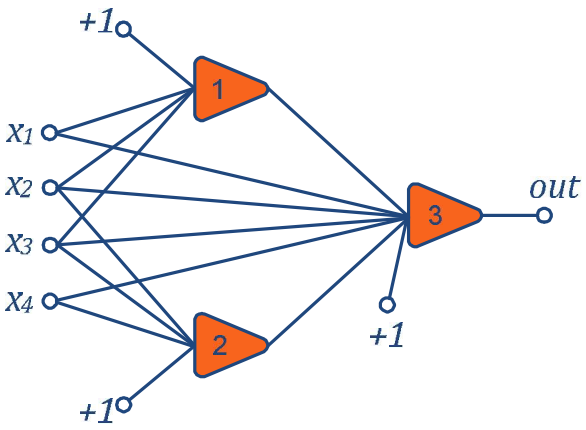


Figure 4.5: Single layer bridge architecture used to solve the parity-4 problem.

PHASE 1: GRADIENT SEARCH

The gradient search is initialized using a set of randomly selected starting weights with values between ± 1. The small magnitude of the initial weights helps ensure that the initial outputs are not in saturation, which allows more flexibility for the separating planes during the early stages of the training process. This is clearly seen when looking at the first 10 iterations of the output images in Figure 4.4. The low intensities of the pixels confirm that the outputs have not yet reached saturation, and the resulting flexibility is evidenced by the frequent changes in the coloration of the pixels from one iteration to the next. It is also clear that as the number of patterns in saturation increases, indicated by darkly shaded pixels, the changes in pixel color becomes less frequent, which indicates less movement in the separating plane.

As the first phase of training progresses, the condition flags are repeatedly updated in order to detect entrapment. The $cond_s$ flag is only cleared once separability is reached in the hidden space, so it is always set equal to 1 when training begins. The $cond_c$ and $cond_e$ flags, on the other hand, are always cleared when training starts. Since $\sigma = 5$, at least 5 iterations must be completed before the error condition can be tested. Therefore, at the beginning of training, the $cond_e$ flag is cleared, and remains cleared for at least the first 5 iterations. Looking at the error plot at the top of Figure 4.4, the total error reaches a fixed value of around 0.25 after 13 iterations. Five iterations later, the error condition test yields

$$\frac{TE_{k-5} - TE_k}{TE_{k-5}} = \frac{0.2524738 - 0.2524733}{0.2524738} = 1.98 \times 10^{-6} < 10^{-4},$$

and the $cond_e$ flag is set. Checking the value of the error condition for the 18th iteration in Figure 4.4 confirms that the flag has been updated as expected.

With the error and separation flags set, the only remaining condition is the classification condition. As with the error condition, the classification test cannot be performed until the

number of total iterations reaches the necessary value. In the latter case, this is determined by the γ parameter, which is equal to 50 for this example. Therefore, the $cond_c$ flag remains cleared for at least the first 50 iterations of the gradient search phase. Returning to the training summary in Figure 4.4, it is apparent that, by the 8th iteration, the classification of the data has become fixed. Since $\gamma = 50$, it is expected that if the classification remains unchanged for 50 iterations, the test for the classification condition should come back positive, and the $cond_c$ flag should then be set. Checking the state of $cond_c$ at the 58th iteration in Figure 4.4 confirms that the flag has been updated correctly.

PHASE 2: HIDDEN LAYER PSEUDO-INVERSION

At the 58th iteration, all three condition flags are set, which means that the gradient search has failed to converge. Thus, the first phase of training is terminated and the second phase begins. During the second phase of training, the hidden layer neurons are retrained using pseudo-inversion in an attempt to invert the output values for the misclassified patterns.

In order to maximize the probability of correctly reclassifying the misclassified patterns, the hidden neuron with the lowest net values for the designated patterns is chosen to be retrained first. The reason for this choice is that the lower net values mean that the misclassified patterns are closer to the separating plane. Therefore, reassigning these patterns will require less movement in the separating plane, which reduces the likelihood of unintentionally reclassifying any of the correctly classified patterns in the process.

For the example in Figure 4.4, it is clear that the only pattern which remains misclassified is pattern 15. Looking at the output pixels of the hidden layer neurons for the 15th pattern, it is apparent from the comparison of the pixel intensities that the output of the first hidden neuron is lower than that of the second. Since both neurons use the same activation function, the lower

output value also corresponds to lower net value. Therefore, hidden neuron 1 is chosen to be retrained first.

In order to retrain the selected neuron, a set of desired outputs must be defined. To do this, the sign of the current outputs are used, but with the output values of the misclassified patterns inverted. For the present example, this results in

$$o_{58} = \begin{bmatrix} -0.89832 \\ \mathbf{-0.41765} \\ -0.98549 \\ -0.89387 \\ -0.98369 \\ -0.88135 \\ -0.99776 \\ -0.98294 \\ +0.99775 \\ +0.99971 \\ +0.98361 \\ +0.99785 \\ +0.98542 \\ +0.99809 \\ +0.89783 \\ +0.98606 \end{bmatrix} \Longrightarrow \text{sign}(o_{58}) = \begin{bmatrix} -1 \\ \mathbf{-1} \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ +1 \\ +1 \\ +1 \\ +1 \\ +1 \\ +1 \\ +1 \\ +1 \end{bmatrix} \Longrightarrow d = \begin{bmatrix} -1 \\ \mathbf{+1} \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ +1 \\ +1 \\ +1 \\ +1 \\ +1 \\ +1 \\ +1 \\ +1 \end{bmatrix}.$$

Using the designated set of desired outputs, the first hidden neuron is retrained according to the weighted pseudo-inversion technique described in section 4.2.1 . An initial weighting value of one is used. Since the desired set of outputs is linearly separable, the misclassified pattern is reclassified after the first pseudo-inversion attempt, without incrementing the scaling factor. Furthermore, no unintentional reclassification of the remaining patterns has occurred, so there is no need to retrain the second hidden neuron. This marks the end of the hidden layer search. The resulting output can be seen in the last column of the output image for the first hidden neuron in Figure 4.4. Notice that all output assignments remain unchanged except for the inversion of the 15th pattern, which was the desired result.

Once pseudo inversion of the hidden layer neurons is complete, their weights are frozen and linear separation of the updated mapping is attempted using pseudo-inversion of the output neuron. Here, the margin maximizing active set method described in section 3.3 is adopted. The resulting classification, represented in Figure 4.4 by the last column of pixels in the bitmap of the output neuron, matches that of the desired classification. Therefore, training is complete, with a final mean squared error of 10^{-8}.

4.7 EXPERIMENTAL RESULTS

For verification, the proposed method was tested against the EBP and LM algorithms described in 0. The EBP algorithm was chosen because it remains the most widely used method for training feed-forward neural networks. The LM algorithm was chosen for its speed. All three algorithms were applied to a range of test problems, and performance was measured according to success rate and average number of iterations per successful run. Each algorithm was assigned a maximum number of iterations per run, and a maximum acceptable error was chosen for each problem. Success rate was calculated as the number of runs that reached the maximum allowable error within the allotted number of iterations.

The classification problems used for testing were chosen for their high degrees of nonlinearity. The chosen problems are listed below, with a brief description of each.

- **Parity-N**: The parity problems, described in section 2.3, were chosen for their high degree of nonlinearity and their reputation as benchmarks for nonlinear classification. Five cases were selected, ranging from parity-3 to parity-7.
- **N mod 2:** The N mod 2 problem is a single input single output problem that produces a binary mapping of the digits from $-N$ to N that determines if the input is even or odd.

The problem can also be viewed as a parity-N problem with the individual inputs replaced by their sum.

- **Checker-N:** The checker-N problem consists of an $10N \times 10N$ grid whose desired outputs form an alternating pattern like that of the colored squares on a chess board. Mathematically speaking, the desired outputs are assigned as

$$d(x, y) = 2[\lfloor x \rfloor + \lfloor y \rfloor \pmod 2] - 1,$$

where $x, y \in \{0, 0.1, 0.2, \ldots, N\}$ are the "rank" and "file" respectively, mod is the modulo operator, and $\lfloor \cdot \rfloor$ is the floor operator. Thus, for an integer N, the resulting "board" is comprised of N^2 1×1 "squares", each containing 100 points. The difficulty of the problem increases with N, as does the size of the data set, which grows at a rate of $100(2N + 1)$. A 3-dimensional example of the training surface for the checker-3 problem is shown in Figure 4.6.

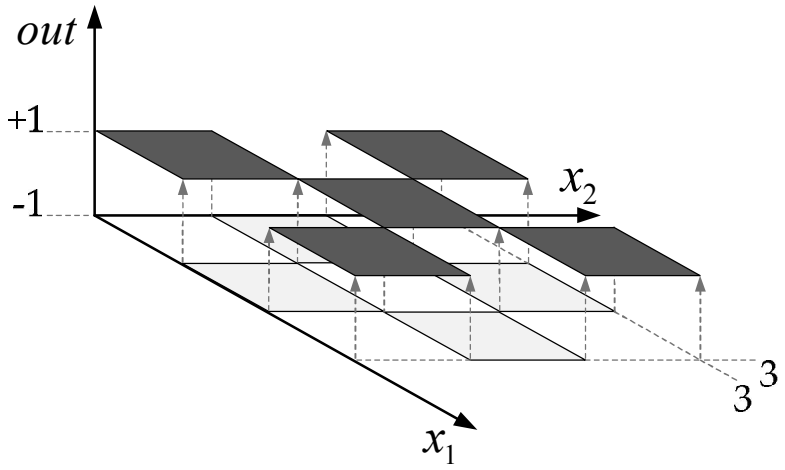


Figure 4.6: Training surface for the checker-3 problem.

Table 4.1: Performance comparison for the proposed method.

		Algorithm			Hidden Neurons
		EBP	LM	HLPI	
Problem	Parity-3	Success: **100%** Ave. ite: 6,060	Success: 98% Ave. ite: **9**	Success: **100%** Ave. ite: 10	1
	Parity-4	Success: 81% Ave.ite: 77,988	Success: 7% Ave. ite: **27**	Success: **100%** Ave. ite: 114	2
	Parity-5	Success: 83% Ave. ite: 103,999	Success: 4% Ave. ite: **34.25**	Success: **98%** Ave. ite: 150	2
	Parity-6	Success: 1% Ave. ite: 190,634	FAILED TO CONVERGE	Success: **98%** Ave. ite: **276**	3
	Parity-7	FAILED TO CONVERGE	FAILED TO CONVERGE	Success: **96%** Ave. ite: **291**	3
	15 mod 2	FAILED TO CONVERGE	FAILED TO CONVERGE	Success: **95%** Ave. ite: **2,553**	7
	Checker - 3	FAILED TO CONVERGE	Success: 11% Ave. ite: **111**	Success: **100%** Ave. ite: 596	4
	Checker - 4	FAILED TO CONVERGE	Success: 4% Ave. ite: **1,472**	Success: **75%** Ave. ite: 3,770	6

The training results for all 8 of the test cases are shown in Table 4.1. Single layer bridged architectures were used for all problems, and the number of hidden neurons used in each case is listed in the right hand column of Table 4.1. Numbers in bold font represent the best performers for each problem. The training parameters used by each algorithm are presented in Table 4.2.

The training results in Table 4.1 show that the HLPI method offers the highest rate of convergence for all of the tested problems, with a rate of 95% or better in all but one of the cases, and is the only one of the three algorithms to converge successfully for on all 8 problems. It is also evident that the average number of iterations required for the HLPI method is around 2 to 3 orders of magnitude lower than that of the EBP algorithm, and shows comparable performance to with the LM algorithm in that regard. Moreover, the comparison with the LM algorithm is somewhat misleading since only the successful training attempts are considered in the calculation of the average. Therefore, the significant portion of attempts in which the number of

iterations required by the LM algorithm exceeded the allowable value must also be taken into account when making a direct comparison. Furthermore, since the first phase of the HLPI method is equivalent to the LM method, the average number of iterations for those cases in which the LM algorithm was able to converge successfully should be identical for HLPI.

Most importantly, the test results indicate that the proposed method does, in fact, achieve the objective for which it was originally designed: to improve the rate of convergence of the gradient search by overcoming the challenge presented by local minim. Clearly, judging by the success rates in Table 4.1, this goal has been achieved. Furthermore, the resulting improvement has come at a relatively low cost in terms of the number of required iterations.

Table 4.2: Algorithm parameters.

		Problem							
		Parity-3	Parity-4	Parity-5	Parity-6	Parity-7	15 mod 2	Checker-3	Checker-4
EBP	Max ite.	150,000	150,000	150,000	250,000	250,000	250,000	250,000	250,000
	Max error	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
	Gain	0.1	0.1	0.1	0.1	0.1	0.1	0.025	0.025
	α	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7
LM	Max ite.	500	500	500	500	500	5000	1000	5000
	Max error	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
	Gain	0.1	0.1	0.1	0.1	0.1	0.1	0.025	0.025
	μ_0	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
HLPI	Max Ite.	500	500	500	500	500	5000	1000	5000
	Max error	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
	Gain	0.1	0.1	0.1	0.1	0.1	0.1	0.25	0.25
	μ_0	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	γ	50	50	50	50	50	30	100	100
	σ	10	10	10	10	10	20	20	20
	ϵ	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-5}	10^{-4}	10^{-4}

Chapter 5

CONCLUSION

In Chapter 3, three pseudo-inversion techniques were presented for use in training individual neurons. The first technique used an algebraic manipulation of a least squares formulation of the weight update rule to improve the computational efficiency of the training process. This was done by eliminating the need for matrix inversion at all but the first iteration, which significantly reduced the necessary computation time, especially for problems with large data sets. However, despite the method's efficiency, it was shown that the modified version was not truly equivalent to the original least squares method from which it was derived, which caused a reduction in the rate of convergence.

Next, an active set method was presented that successfully reduced both the computation time and the rate of convergence. The method used a linear approximation of the activation function to iteratively reduce the size of the training set until a minimal set of training patterns was reached. Once the optimal set of patterns was found, the tangent hyperbolic activation function was restored, and the total error was directly set by scaling the final weight vector. The method was also shown to be more robust, and was better able to handle problems in which the training data was poorly distributed or unbalanced.

Despite the decided improvement represent by the active set method, it was shown that the method did not to guarantee an optimal solution in terms of the margin of separation. To solve this problem, a third technique was developed that uses the solution from the active set method to obtain an equality constrained QP formulation of the separating margin. The solution of the resulting QP guarantees maximum separation.

At the end of Chapter 3, a method for using single neuron pseudo-inversion techniques to improve the generalization of full scale networks is discussed. This is done by first finding a solution using a gradient search. Once the separation is found, the signs of the current outputs for all hidden neurons are used to define their desired outputs. Each neuron is then retrained starting with the units nearest the input layer and moving towards the output. Using the margin maximizing QP method, the resulting weight set will achieve optimal separation for the feature set determined during the initial training process.

In Chapter 4, a modified training technique, known as hidden layer pseudo-inversion, was proposed that significantly improves the success rate in gradient based searches. The method is similar to the generalization technique in Chapter 3, in that it has two training phases consisting of a gradient search followed by a hidden layer pseudo-inversion phase. However, unlike the generalization technique, the proposed method does not require convergence before beginning the second training phase. Instead the method monitors the state of the algorithm in order to determine if the algorithm has become trapped in a false minimum. Once entrapment is detected, the current hidden layer outputs are used to define a set of desired outputs. The desired values of the remaining misclassified patterns are inverted in an attempt to reconfigure the hidden layer mapping. The hidden layer neurons are then retrained one at a time. Once a new mapping is found, linear separation is attempted using pseudo-inversion of the output neuron. The process is repeated until separation is achieved.

The HLPI method was compared with the popular EBP and LM algorithms using a set of 8 nonlinear classification benchmarks. The proposed method was shown to offer the highest success rate for all of the tested problems. It was also shown to require significantly fewer iterations than the EBP algorithm, and performed comparably with the LM algorithm. However,

the latter comparison was somewhat misleading due to the fact that only the successful training attempts were considered in this calculation. Therefore the significant portion of attempts in which the required number of iterations exceeded the number allowable was not considered. Furthermore, since the first phase of the HLPI method is equivalent to the LM method, the average number of iterations for those cases in which the LM algorithm is able to converge successfully should be identical.

BIBLIOGRAPHY

[1] J. M. Zurada, *Artificial Neural Systems*, 1st ed. Boston, MA: PWS Publishing Company, 1995.

[2] A. J. Eide, T. Lindblad, and G. Paillet, "The Radial-Basis-Function Type of Neural Network and Its Implementations in Hardware," in *Industrial Electronics Handbook*, 2nd ed.: CRC Press, 2011, vol. 5, ch. 12, pp. 1-12.

[3] K.P. Bennet and C. Campbell, "Support vector machines: hype or hallelujah?," *ACM SIGKDD Explorations Newsletter*, vol. 2, no. 2, pp. 1-13, Desember 2000.

[4] Thomas M. Cover, "Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition," *IEEE Transactions on Electronic Computers*, vol. EC-14, no. 3, pp. 326-334, June 1965.

[5] B. M. Wilamowski, "Neural network architectures and learning algorithms, ," *IEEE Industrial* , vol. 3, no. 5, pp. 56-63, 2009.

[6] R. Durbin, "On the correspondence between network models and the nervous system," in *The computing neuron*, R. Durbin, C. Miall, and G Mitchison, Eds. Boston, MA, USA: Adison-Wesley, 1989, ch. 1, pp. 1-10.

[7] H. T. Siegelmann and E. D. Sontag, "On The Computational Power Of Neural Nets," *JOURNAL OF COMPUTER AND SYSTEM SCIENCES*, vol. 50, no. 1, pp. 132-150, 1995.

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

1958.

[9] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing - Vol. 1: Foundations*.: MIT Press, 1987.

[10] T. J. Anderson and B. M. Wilamowski, "A Modified Regression Algorithm for Fast One Layer Neural Network Training," in *World Congress of Neural Networks*, Washington DC, USA, 1995, pp. 687-690.

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," in *Neurocomputing: foundations of research* , J. A. Anderson and E. Rosenfeld, Eds. Cambridge, MA: MIT Press, 1988, pp. 696-699.

[12] K. Levenberg, "A method for the solution of certain non-linear problems in leas squares," *Quarterly of Applied Mathematics*, no. 2, pp. 164-168, 1944.

[13] D. W. Marquardt, "An algorithm for least squares estimation of non-linear parameters," *SIAM Journal*, no. 11, pp. 431-441, 1963.

[14] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed., T. V. Mikosch, S. I. Resnick, and S. M. Robinson, Eds. New York, NY: Springer, 2006.

[15] C. G. Broyden, "The convergence of a class of double-rank minimization algorithms," *Journal of the Institute of Mathematics and Its Applications*, vol. 6, pp. 76-90, 1970.

[16] W. C. Davidon, "Variable metric method for minimization," *SIAM Journal on Optimization*, vol. 1, pp. 1-17, 1991.

[17] B. M. Wilamowski, N. J. Cotton, O. Kaynak, and G. Dunda, "Computing Gradient Vector and Jacobian Matrix in Arbitrarily Connected Neural Networks," *IEEE Trans. on Industrial Electronics*, vol. 55, no. 10, pp. 3784-3790, October 2008.

[18] B.M. Wilamowski, H Yu, and K T. Chung, "Parity-N Problems as a Vehicle to Compare Efficiency of Neural Network Architectures," in *Industrial Electronics Handbook, vol. 5 – Intelligent Systems*, 2nd ed.: CRC Press, 2011, vol. 5, ch. 10, pp. 1-8.

[19] M. E. Hohil, D. Liu, and S. H. Smith, "Solving the N-bit parity problem using neural networks," *Neural Networks*, vol. 12, no. 9, pp. 1321-1323, November 1999.

[20] B.M. Wilamowski and D. Hunter, "Solving Parity-n Problems with Feedforward Neural Networks," in *Proc. of the IJCNN'03 International Joint Conference on Neural Networks*, Portland, 2003, pp. 2546-2551.

[21] M. Gori and A. Tesi, "On the Problem of Local Minima in Backpropagation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 1, pp. 76-86, January 1992.

[22] T. D. Gedeon, "Indicators of hidden neuron functionality: the weight matrix versus neuron behaviour," in *ANNES '95 Proceedings of the 2nd New Zealand Two-Stream International Conference on Artificial Neural Networks and Expert Systems*, Washington DC, 1995, p. 26.

Appendix A

MATLAB CODE FOR THE PSEUDO-INVERSION TECHNIQUES

A.1 NLPI.m

```
function NLPI;
clear all; format compact;
clc; figure(1); clf;
warning('off');
global data input output;
clc; figure(1); clf;
input=[1 1; 1 -1; 1.5 1; 4 1];
output=[1; 1; -1; -1];
data.X = [1 1;1 -1];
data.Y = [1.5 1;4 1];
data.axs = [0 5 -2 2];

gain=0.1;
act_n=2;
ite_num=1000;
[m,n]=size(input)
X=cat(2,input,ones(m,1));
X
Y=X'*X\X';
for ii=1:1
    w=generate_weights(n+1);
    for iteration=1:ite_num
        [del,err(iteration),outs]=compute_del(input,output,w,act_n,gain);
        delta_w=Y*del;
        w=w+delta_w';
        w=1.2*w;
        if err(iteration)<1e-4 break; end;
    end
    figure(1);
    semilogy(err); hold on;
    drawnow;
    axis tight
    disp(['Total Error: ',num2str(err(end))]);
    disp(['Final Weights: ',num2str(w)]);
    xlabel('iteration')
    ylabel('total error')
end;
%%
function [weight] = generate_weights(n);
weight=zeros(1,n);
for i = 1:n             % number of weights
    ra = 2*rand(1)-1;          % generate random weights between -1 and 1
    while( ra == 0 )
```

```
        ra = 2*rand(1)-1;
    end;
    weight(i) = ra;
end;
%%compute delta using error backpropagation%%%%%%%%%%
function [del,err,outs]=compute_del(input,output,w,act_n,gain)
[m,n]=size(input);
err=0;
for p=1:m
    temp=input(p,:);
    net=w(n+1);
    for i=1:n
        net=temp(i)*w(i)+net;
    end
    [out,der]=actFuncDer(net,act_n,gain);
    outs(p)=out;
    del(p,1)=(output(p)-out)/(der+0.0001);
    err=err+(out-output(p))^2;
end
%%
function [out,der]=actFuncDer(net,act_n,gain)
switch act_n
    case 0, out = gain*net;  der = gain;
% linear neuron
    case 1, out = 1/(1+exp(-gain*net)); der = gain*(1-out)*out;
% unipolar neuron
    case 2, out = 2/(1+exp(-gain*net))-1; der = gain*(1-out*out)/2;
% bipolar neuron
    case 3, out = gain*net/(1+gain*abs(net));der = gain/(gain*abs(net)+1)^2;
% unipolar elliot neuron
    case 4, out = 2*gain*net/(1+gain*abs(net))-1; der =
2*gain/(gain*abs(net)+1)^2;      % bipolar elliot neuron
end;
```

A.2 ASPINV.m

```
function [w,err] = aspinv(inp,dout,gain,err_max);
[np,ni]=size(inp);
X=cat(2,inp,ones(np,1));
A = [1:length(dout)];
Xndx = find(dout==1);
w = 2*rand(ni+1,1)-1;
for ite=1:100
    Xa = X(A,:);
    w = Xa'*Xa\Xa'*dout(A);
    out = X*w;
    B = find((abs(out))<=1);
    if (length(B)>=2)
        cls0 = length(find(Xndx==B(1)));
        for i=2:length(B)
            cls = length(find(Xndx==B(i)));
            if cls0~=cls
                A = B;
                break
```

```
            end
        end
    end
    E = dout(A) - out(A);
    if (E'*E<1e-4)&(sign(out)==dout),
        break;
    end;
end
scale = atanh(1-err_max)/gain;
w = w*scale;
E = dout - tanh(gain*X*w);
err = E'*E/np;
return
```

A.3 PINVQP.m

```
function [w,err] = pinvqp(inp,dout,gain,err_max);
[np,ni]=size(inp);
X=cat(2,inp,ones(np,1));
A = [1:length(dout)];
Xndx = find(dout==1);
w = 2*rand(ni+1,1)-1;
for ite=1:100
    Xa = X(A,:);
    w = Xa'*Xa\Xa'*dout(A);
    out = X*w;
        E = dout(A) - out(A);
    if (E'*E<1e-4)&(sign(out)==dout),
        break;
    end;
    B = find((abs(out))<=1);
    if (length(B)>=2)
        cls0 = length(find(Xndx==B(1)));
        for i=2:length(B)
            cls = length(find(Xndx==B(i)));
            if cls0~=cls
                A = B;
                break
            end
        end
    end
end
y = dout(A);
out = inp(A,:);
% V ------------------ Formulate Wolfe Dual --------------------- V %
nump = length(y);
G = (out*out').*(y*y');
c = -ones(nump,1);
Ai = eye(nump);
Ae =  y';
Bi = zeros(nump,1);
Be = 0;

% ---------------------- Solve Wolfe Dual ----------------------- %
```

```
x = rand(nump,1);
[x] = QPsolve(G,c,Ai,Ae,Bi,Be,x);
wo = (y'.*x')*out
S = find(abs(x)>=1e-10);
nums = length(S);
s1 = 0;
for i = 1:nums
    s2 = 0;
    for j = 1:nums
        s2 = s2 + x(S(j))*y(S(j))*out(S(j),:)*out(S(i),:)';
    end
    s1 = s1 + y(S(i)) - s2;
end
b = 1/length(S)*s1;

w = [wo,b]';

scale = atanh(1-err_max)/gain;
w = w*scale;
E = dout - tanh(gain*X*w);
err = E'*E/np;
return
```

A.4 QPSOLVE.m

```
function [x] = QPsolve(G,c,Ai,Ae,Bi,Be,x)

% Initialize working set
W = find(Ai*x-Bi<=0);                                    % Find active
constraints for initial x

a = Ai(W,:);
a = [a;Ae];
b = Bi(W,:);
b = [b;Be];
% -------------------- Apply Algorithm -------------------- %
while(1)

numc = length(b);                                        % Number of
constraints in working set
% Solve for step direction (p):
g = G*x+c;                                               % Gradient of the
Objective
h = a*x-b;                                               % Vector of
constraint values

O = zeros(numc,numc);                                    % Zero matrix for
the KKT matrix
KKT = [G a';                                             % KKT Matrix
       a O];

gb = [g;
      h];
```

```
if cond(KKT)>1e5                                                    % Solve KKT
problem
    sol = (KKT+diag(.001*ones(1,length(KKT(1,:)))))\gb;
     1
else
    sol = KKT\gb;
     2
end
p = -sol(1:length(x));                                              % Step direction
if length(W)>0
    lam = sol(length(x)+1:end-1);                                   % Lagrange
Multipliers
else
    lam = 0;
end
% Test step direction:
if sum(abs(p))<=1e-10,

    ndx = find(lam<0);
    x = x + p;
    if isempty(ndx)
        break;
    else
        ndx = find(lam==min(lam(ndx)));
        W(ndx) = [];
        a = Ai(W,:);
        a = [a;Ae];
        b = Bi(W,:);
        b = [b;Be];
    end
else
    an = Ai;
    an(W,:) = [];
    bn = Bi;
    bn(W,:) = [];
    ndx = 1:length(Bi);
    ndx(W)=[];

    apgz = find(an*p>=0);
    an(apgz,:)=[];
    bn(apgz,:)=[];
    ndx(apgz)=[];
    blk = (bn-an*x)./(an*p);

    alpha = min([1,min(blk)]);
    x = x + alpha*p
    if alpha<1
        ndx = ndx(find(blk==min(blk)));
        W = sort([ndx,W]);
        a = Ai(W,:);
        a = [a;Ae];
        b = Bi(W,:);
        b = [b;Be];
    else
        W = W;
```

```
    end
end
end
% ------------------------------------------------------------ %
```

Appendix B

MATLAB CODE FOR THE HLPI ALGORITHM

B.1 NRA.m

```
function [ww, nodes, TERR, ite, dbug] = NRA(Nparam,Aparam,Tparam)
global CANCEL
global RESTART
unpackparam(Nparam);
unpackparam(Aparam);
unpackparam(Tparam);
dbug = [];

if length(ww) ~= length(topo),
    ww = (2*rand(size(topo))-1);
    ww = ww_init(gain,np,ni,no,nn,inp,dout,ww,iw,topo,act);
    nw=length(topo);
end;



if Tparam.op == 1
    disp(' ');
    disp('Initial Weights');
    dispf(0,ww', 8,4,'ww');
    disp(' ');
    disp('Training started...');

    [nodes,nets,ooo,der,E] =
cal_forward(gain,np,ni,no,nn,inp,dout,ww,iw,topo,act,Nparam);
    TER=E'*E/np; TERR(1)=TER;

    if strcmp(alg,'NRA_NBN.m')

addpath('C:\Users\Joel\Documents\MATLAB\my_libs\NNT\alg_files\NRA_files\');
        I=eye(nw); x=ww; x_new = x; jite=0; RESTART=2;
        mu1 = 0.01;
        flag = 0;
        count = 0;
        % Inititialize debugging output
        dbug = Nparam; dbug.wwi = ww; dbug.ww = []; dbug.rlnite = [];
dbug.rlntyp = []; dbug.nodes = []; dbug.der = []; dbug.ndxo = {}; dbug.ord =
[];
        % Entrapment test parameters
        nlast = 100;
        outlast = zeros(np,nlast);
```

```
        ncount = 50;
        errite = 50;
        derr = 1e-4;
        % Weight reset parameters
        sim_mes = 0.95; % similarity between neuron outputs. range: [0,1]
        test_freq = 10;  % Number of iterations between tests
        nite_init = 100;
        ite_last = 0;


        for ite=2:ite_max,
        if CANCEL == 1; break; end;
            run Jstandard
            x = x_new;
            gra = J'*E; %gradient
            JJ=J'*J;
            jw=0;
            while (1),
                x_new=x-((JJ+mu1*I)\gra)';
                ww=x_new';
                if (ite==1),    TERR(ite)=terr; break;  end;
                %calculate new performance
                [nodes,nets,ooo,der,E] =
cal_forward(gain,np,ni,no,nn,inp,dout,ww,iw,topo,act,Nparam);
                TER=E'*E/np; % evaluate the objective function
                TERR(ite)=TER;
                if TER<=TERR(ite-1)
                    if mu1>mu_min,  mu1=mu1/LM_scal;  end;
                    break;
                end;
                if mu1<mu_max, mu1=mu1*LM_scal;  end;
                jw=jw+1;
                if jw>5, break; end;
            end; % while (1),
                % Reset weights for redundant hidden neurons
                if (~rem(ite,test_freq))&&((ite-
ite_last)>(nite_init+2))&&(sum(sign(nodes(:,end))~=dout)<np/20)
                    nds = nodes(:,ni+1:end-no);
                    ndslen = repmat(1./sqrt(sum(nds.^2)),np,1);
                    nds = nds.*ndslen;
                    cmp = triu(nds'*nds,1);
                    [n1,n2] = find(abs(cmp)>=sim_mes);
                    pairs = [n1,n2];
                    [h1,h2] = find(abs(cmp)<sim_mes);
                    hndx = unique([h1,h2]);
                    if ~isempty(pairs)

                        while ~isempty(pairs)
                            hndx = unique(hndx);
                            rsndx = pairs(1,1);
                            pct = 0.6*rand+0.2;
                            if length(hndx>1)
                                h1 = ceil(rand*(length(hndx)));
                                mv1 = ww(iw(hndx(h1)):iw(hndx(h1)+1)-1);
                                hndx(h1) = [];
                                h2 = ceil(rand*(length(hndx)));
```

```
                                mv2= ww(iw(hndx(h2)):iw(hndx(h2)+1)-1);
                                hndx(end+1) = h1;
                            else
                               [mv1,mv2] =
find(abs(cmp)==max(max(abs(cmp))));
                                mv1 = ww(iw(mv1(1)):iw(mv1(1)+1)-1);
                                mv2 = ww(iw(mv2(1)):iw(mv2(1)+1)-1);
                            end
                            if rand>0.6;
                                ww(iw(rsndx):iw(rsndx+1)-1) = pct*mv1+(1-
pct)*mv2;
                            else
                                ww(iw(rsndx):iw(rsndx+1)-1) = (1-
2*rand(size(mv1)));
                            end
                            pairs(find(pairs==rsndx),:)=[];
                            hndx(end+1) = rsndx;
                        end
                    end
                    mu1 = 0.01;
                end

            if
((count>=ncount)&&isempty(find(sign(outlast(:,1))~=sign(nodes(:,end)))))||((i
te>errite+2)&&((TERR(ite-errite)-
TERR(ite))<derr*TERR(ite))&&TERR(ite)<=TERR(ite-
errite))&&(sum(sign(nodes(:,end))~=dout)<np/20)
                ite_last = ite;
                ndx = find(dout~=sign(nodes(:,end)));
                if ~isempty(ndx),
                    [ww,dbug.ndxo{end+1},dbug.ord(end+1,:)] =
realign(nodes,gain,ni,no,nn,ww,iw,topo,act,Nparam,ndx,flag);



                    nds = nodes(:,ni+1:end-no);
                    ndslen = repmat(1./sqrt(sum(nds.^2)),np,1);
                    nds = nds.*ndslen;
                    cmp = triu(nds'*nds,1);
                    [n1,n2] = find(abs(cmp)>=sim_mes);
                    pairs = [n1,n2];
                    [h1,h2] = find(abs(cmp)<sim_mes);
                    hndx = unique([h1,h2]);
                    if ~isempty(pairs)

                        while ~isempty(pairs)
                            hndx = unique(hndx);
                            rsndx = pairs(1,1);
                            pct = 0.6*rand+0.2;
                            if length(hndx>1)
                                h1 = ceil(rand*(length(hndx)));
                                mv1 = ww(iw(hndx(h1)):iw(hndx(h1)+1)-1);
                                hndx(h1) = [];
                                h2 = ceil(rand*(length(hndx)));
                                mv2= ww(iw(hndx(h2)):iw(hndx(h2)+1)-1);
                                hndx(end+1) = h1;
```

```
                            else
                               [mv1,mv2] =
find(abs(cmp)==max(max(abs(cmp))));
                                mv1 = ww(iw(mv1(1)):iw(mv1(1)+1)-1);
                                mv2 = ww(iw(mv2(1)):iw(mv2(1)+1)-1);
                            end
                            if rand>0.9;
                                ww(iw(rsndx):iw(rsndx+1)-1) = pct*mv1+(1-
pct)*mv2;
                            else
                                ww(iw(rsndx):iw(rsndx+1)-1) = (1-
2*rand(size(mv1)));
                            end
                            pairs(find(pairs==rsndx),:)=[];
                            hndx(end+1) = rsndx;
                        end
                    end

                    [nodes,nets,ooo,der,E] =
cal_forward(gain,np,ni,no,nn,inp,dout,ww,iw,topo,act,Nparam);
                    inpo = nodes(:,topo(iw(end-1)+1:iw(end)-1));
                    if act(end)==4,
                        funo = Nparam.fun{end};
                        dero = Nparam.der{end};
                    else
                        funo = 0;
                        dero = 0;
                    end
                    [wwo,outn,rln] =
pseudo_inv(inpo,dout,gain(end),act(end),funo,dero,[],10);
                    ww(iw(end-1):iw(end)-1) = wwo/sqrt(sum(wwo.^2));
                    dbug.rlntyp(end+1) = 1;

                    mu1 = 0.01;
                    x_new = ww';
                    dbug.rlnite(end+1) = ite;
                    [nodes,nets,ooo,der,E] =
cal_forward(gain,np,ni,no,nn,inp,dout,ww,iw,topo,act,Nparam);
                    if (E'*E/np>=TER),
                        ww(iw(end-1):iw(end)-1) = 1-2*rand(1,iw(end)-iw(end-
1));
                        dbug.rlntyp(end) = 0;
                    end
                    [nodes,nets,ooo,der,E] =
cal_forward(gain,np,ni,no,nn,inp,dout,ww,iw,topo,act,Nparam);
                    TERR(ite) = E'*E/np;
                end;

                if count>=ncount, count = 0; end;
            end;
            if nlast>1,
                outlast = [outlast(:,2:nlast),nodes(:,end)];
            else
                outlast = nodes(:,end);
            end
            count = count + 1;
```

```
            if rem(ite,pscale)==0; dispf(0,TER,18,12,'total error  ',ite);
end;
            if TER<er_max, RESTART=1; break; end;
            if RESTART==0, return; end;
        end;
    else
        ndir = which('NNT');
        buildalg(alg);
        [tok,ndir]=strtok(fliplr(ndir),'\');
        ndir = fliplr(ndir);
        eval(['run ',ndir,sprintf('alg_files\\%s.m',strtok(alg,['.',' ']))])
    end

elseif Tparam.op == 0
    disp(' ');
    disp('Network Weights');
    dispf(0,ww', 8,4,'ww');
    disp(' ');
    disp('Simulating...');
    ite = 1;
    [nodes,nets,ooo,der,E] =
cal_forward(gain,np,ni,no,nn,inp,dout,ww,iw,topo,act);
    TER=E'*E/np; TERR(1)=TER;
    CANCEL = 1;
end;

return;
```

B.2 PSEUDO_INV.m

```
function [ww,out,rln] = pseudo_inv(inp,dout,gain,act,fun,der,ndx,nite)
% PSEUDO_INV Perform pseudo-inversion in a single neuron.                    %
%   PSEUDO_INV(INP,DOUT,GAIN,ACT,FUN,DER,NDX,NITE) performs pseudo           %
%   inversion on a single neuron based until realignement of one or more of %
%   a specific set of patterns are properly classified. If realignement      %
%   does not occur, the derivative for the specified patterns are scaled     %
%   incrementally, and the process is repeated. Once one or more of the      %
%   desired patterns is successfully clasified, the process is terminated    %
%   and the resulting weights are returned.                                  %
%                                                                            %
%   INP  - [NPxNI]  - Matrix of neuron input patterns.                       %
%   DOUT -  [NPx1]  - Vector containing the desired outputs for the neuron.  %
%   GAIN -  [1xNN]  - Gains of the neuron.                                   %
%   ACT  - [scalar] - The activation function of the neuron.                 %
%   FUN  - [string] - Used for custom activation functions.                  %
%   DER  - [string] - Used for custom activation functions.                  %
%   NDX  -  [1xNM]  - Vector of indices for misclassified outputs.           %
%   NITE - [scalar] - Number of pseudo inversion iterations.                 %

%   Author(s): J. D. Hewlett, 2/16/11, revised                               %
%       $Revision: 1.3 $  $Date: 07-Mar-2011 11:26:52 $                      %
```

```
% --------------------------------------------------------------------------
%

nrln = 1; % Number of patterns which must be reclassified for termination.
d_scal = 1; % Initial value for derivative scaling.
rln_cnst = .6; % Minimum difference of actual and desired output for
realignment.
d_step = 0.25; % Increment size for derivative scaleing.
[np,ni]=size(inp);
inp=[ones(np,1),inp];    % Augmenting input bias.
inp0=inp;
inpi=pinv(inp);          % Find pseudo inverse of input matrix.
ww=(inpi*dout)';         % Find initial weights using regression
% Calculate outputs:
net=inp0*ww';
switch act
    case 1,
        out = tanh(gain*net);
        fp  = gain*(1-out.*out)+0.001;
    case 2,
        out = 1/(1+exp(-gain*net));
        fp  = gain*(1-out).*out+0.001;
    case 3,
        out = gain*net;
        fp  = gain;
    case 4,
        out = eval([fun,';']);
        fp  = eval([der,';']);
end;
fp(ndx) = fp(ndx)*d_scal; % scale derivatives
E=dout-out;
e = E'*E/np; % Determine total error.
while length(find(abs(E(ndx))<rln_cnst))<nrln % Repeat until realignment
occurs.
    % Perform iterative pseudo inversion:
    for ite=1:nite
        inp=diag(fp)*inp0;
        inpi=pinv(inp);
        dw=inpi*E; % Calculate step.
        ww=ww+dw'; % Update weights.
        % Calculate outputs:
        net=inp0*ww';
        switch act
            case 1,
                out = tanh(gain*net);
                fp  = gain*(1-out.*out)+0.001;
            case 2,
                out = 1/(1+exp(-gain*net));
                fp  = gain*(1-out).*out+0.001;
            case 3,
                out = gain*net;
                fp  = gain;
            case 4,
                out = eval([fun,';']);
                fp  = eval([der,';']);
        end;
        fp(ndx) = fp(ndx)*d_scal; % Scale derivatives.
```

```
        E=dout-out;
        e = E'*E/np; % Calculate total error.
        er(ite)=e;

    end;
if isempty(ndx), rln = []; return; end; % If realignment index is empty, end
process.
d_scal=d_scal+d_step; % Increment derivative scalar.
end
rln = find(abs(E(ndx))<rln_cnst); % Determine indices of realigned patterns.
```

B.3 Realign.m

```
function [ww,ndxo,ord] =
realign(nodes,gain,ni,no,nn,ww,iw,topo,act,Nparam,ndx,flag)
% REALIGN Realign hidden layer neuron.                                    %
%   REALIGN(NODES,GAIN,NI,NO,NN,WW,IW,TOPO,ACT,NPARAM,NDX,FLAG) realigns  %
%   hidden layer neuron classification based on which input patterns are  %
%   misclassified with respect to the desired output for the network.     %
%                                                                         %
%   First, the hidden neurons are Ranked based on the average output for  %
%   the misclassified patterns. Lower averages imply that the misclassified %
%                                                                         %
%   patterns are nearer to the separating plane, meaning they are more    %
%   likely to be reclassifiable. Pseudo inversion is then performed, and  %
%   the derivatives for the misclassified patterns are incrementaly scaled %
%   until reclassifcation occurs for one or more patterns.                %
%                                                                         %
%   Once reclassification is achieved for a neuron, a new index of patterns %
%   is formed containing those patterns which were not reclassified as well %
%   as any patterns which may have been incidentally reclassified in the  %
%   process. Then, the remaining neurons are reranked and the process is  %
%   repaeted until either the index is empty or all hidden units have been %
%   have been realigned.                                                  %
%                                                                         %
%   NODES  - [NPxNI+NN] - Matrix of network inputs and neuron outputs for %
%                         the current iteration.                          %
%   GAIN   -   [1xNN]   - Vector containing the gains of all neurons.     %
%   NI     -  [scalar]  - Number of inputs to the network.                %
%   NO     -  [scalar]  - Number of outputs from the network.             %
%   NN     -  [scalar]  - Number of neurons in the network.               %
%   WW     -   [1xNW]   - Vector containing the weights of the network.   %
%   IW     -  [1xNN+1]  - Index of biasing weights within WW.             %
%   TOPO   -   [1xNW]   - Vector defining the network topology.           %
%   ACT    -   [1xNN]   - Vector designating the activation function for  %
%                         each neuron.                                    %
%   Nparam -  [struct]  - Structure of additional network parameters.     %
%   NDX    -   [1xNM]   - Vector of indices for misclassified outputs.    %
%   FLAG   -  [scalar]  - Indicates if the realignment process is         %
%                         oscillating.                                    %

%   Author(s): J. D. Hewlett, 2/16/11, revised                            %
```

```
%       $Revision: 1.3 $  $Date: 07-Mar-2011 10:39:44 $                    %
% ------------------------------------------------------------------------ %

disp('Beginning realignment...');
disp(sprintf('Patterns misclassfied: %s', num2str(length(ndx))));
ndxo = ndx;
nite = 10; % Number of pseudo inversion iterations per attempt
nh = nn-no; % Number of hidden neurons
nds = nodes(ndx,ni+1:ni+nh); % Get the outputs of all hidden neurons.
[nds,ord] = sortrows((sum(abs(nds),1)/length(nds(:,1)))'); % Rank hidden
neurons.
if flag,
    ord = flipud(ord); % If oscillation occurs, reverse rank order.
end
% For each hidden neuron:
for ii = 1:nh,
    don = sign(nodes(:,ni+ord(ii))); % Desired output for current neuron.
    don(ndx) = -don(ndx); % Change sign for misclassified patterns.
    inpn = nodes(:,topo(iw(ord(ii))+1:iw(ord(ii)+1)-1)); % Get inputs for
current neuron.
    % Check for custom activation functions.
    if act(ord(ii))==4,
        fun = Nparam.fun{ord(ii)};
        der = Nparam.der{ord(ii)};
    else
        fun = 0;
        der = 0;
    end

    [wwn,outn,rln] = pseudo_inv(inpn,don,gain(ord(ii)),...
        act(ord(ii)),fun,der,ndx,nite); % Perform pseudo inversion for
current neuron.
    nodes(:,ni+ord(ii)) = outn; % Update neuron output values.
    ww(iw(ord(ii)):iw(ord(ii)+1)-1) = wwn; % Update neruon weights.
    ndx = find(don~=sign(outn)); % Generate new realignment index.
    disp(sprintf('Hidden Neuron #%s - Patterns realigned: %s',...
        num2str(ord(ii)),num2str(length(rln))));
    % If more than one neuron is remaining, update rank.
    if ii<nh-1
        nds = nodes(ndx,ni+ord(ii+1:end));
        [nds,ord1] = sortrows((sum(abs(nds),1)/length(nds(:,1)))');
        ord2 = ord(ii+1:end);
        ord(ii+1:end) = ord2(ord1);
    end
    % If realignment index is empty, terminate the process.
    if isempty(ndx),
        if ii<nh,
            ord(ii+1:nh) = 0; % For debugging purposes.
        end
        break;
    end;
    ndxo = [ndxo;0;ndx]; % For debugging purposes.
    if rand>0.9, break; end;
end
return
```